

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, M.Sc.

Eingereicht von Tim Oelkers

8. September 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 Anforderungsanalyse	8
3.1 Use Case	8
3.2 Ist-Zustand	8
3.2.1 Mars Trek	9
3.2.2 Google Earth	11
3.3 Anforderungen	13
3.3.1 Funktional	14
3.3.2 Nicht-Funktional	16
4 Konzeption	18
4.1 Vorgehensmodell	18
4.2 Programmiersprache	18
4.2.1 Java	18
4.2.2 C++	19
4.2.3 JavaScript/TypeScript	20
4.2.4 Fazit	21
4.3 Bibliotheken	21
4.3.1 Webserver	21
4.3.2 Grafik/Mathematik	23
4.4 Architektur	24
4.5 User Interface	27
5 Implementierung	31
5.1 Build System	31
5.2 Visualisierung	32

Inhaltsverzeichnis

5.3	Kamerabewegung	33
5.4	Daten-Ladeprozess	34
5.4.1	Modell Generierung	34
5.4.2	Projektion	34
5.5	User Interface	34
5.6	Tests	34
6	Evaluation	35
7	Fazit	36

Abbildungsverzeichnis

2.1	Indexierung eines Raster-Modells	4
3.1	Ladezeiten der Mars Trek Anwendung bei 50 Mbit/s auf Mittelklasse-PC	9
3.2	Initiale Ansicht der Mars Trek Anwendung	9
3.3	Initiale Ansicht der Google Earth Anwendung	11
3.4	Initiale Mars Visualisierung der Google Earth Anwendung	12
4.1	Komponentendiagramm der SolarViewer Architektur	24
4.2	Aktivitätsdiagramm für die Erstellung der Welt auf Client-Seite	26
4.3	Mock Up der SolarViewer Anwendung	28

Liste der Algorithmen

1	Redundanzentfernung	7
---	-------------------------------	---

1 Einführung

1.1 Motivation

Der Mars als unser Nachbarplanet ist, aufgrund seines Klimas und der Nähe, der am Besten für eine Besiedelung geeignete, 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 Missionen schlugen 29 zumindest teilweise fehl¹.

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.

¹https://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[5]. 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[4, 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).

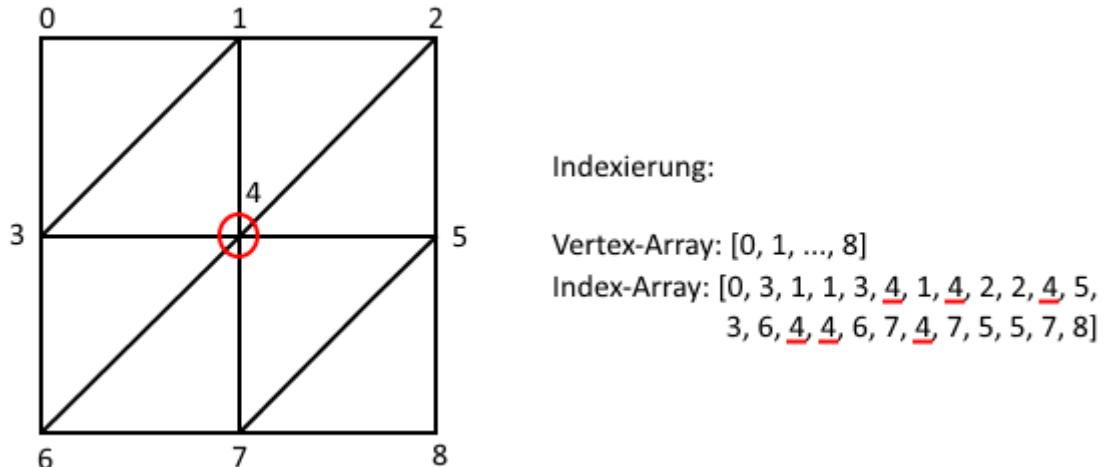


Abbildung 2.1: Indexierung eines Raster-Modells

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:

$$\text{Anzahl} = 6 * (\text{Höhe} - 1) * (\text{Breite} - 1) = 6.369.684.486$$

. 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[1]. 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 2.3.4 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 Datenpunkten, 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 gleichmäßigen 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

2 Grundlagen

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:

$$O(n) = O(1.061.683.200) = 2^{n/9*2/100} = 7.32 * 10^{710.218}$$

‘ Als machbare Alternative wurde hier ein Algorithmus aus der Klasse der greedy Algorithmen gewählt. Dabei wird in jedem Teilschritt des Algorithmus immer die aktuell beste Lösung, also das größte Teilraster, gewählt. Die gefundene Lösung ist damit zwar nicht unbedingt optimal, aber es wird ein lokales Minimum erreicht. Der größte Vorteil dabei ist, dass die Lösung mit fast linearer Komplexität gefunden werden kann. Die Komplexität ist leicht erhöht, da pro Wert immer eine nicht bestimmbarer Anzahl an nachfolgenden Werten überprüft werden muss. Diese können aber potentiell bei nachfolgenden Iterationen übersprungen werden, falls sie als redundant eingestuft wurden. Der Algorithmus (siehe Algorithmus 1) iteriert über alle Werte und sucht für jeden nicht besuchten Datenpunkt alle nachfolgenden horizontalen und vertikalen linearen Abhängigkeiten. Dabei werden jeweils die Breiten und Längen dieser Abhängigkeiten zurückgeliefert und auch alle Alternativen in Betracht gezogen. Es können zum Beispiel bei den Reihen jeweils 3 Reihen eine Breitenlänge von 4 haben oder 5 Reihen eine Breitenlänge von 3. Beides kann zum größtmöglichen Raster führen. Dies wird auch für die Spalten durchgeführt und die größtmögliche Überschneidung aus Spalten und Reihen wird als redundant markiert, solange sie die Mindestgröße von 3x3 erreicht hat.

Algorithmus 1 Redundanzentfernung

```

procedure FINDREDUNDANCIES(data, replacement, chunkSize)
  for all x, z in data do
    if Wert an Stelle x, z noch nicht besucht then
      rows = Finde alle Abhangigkeiten in den Reihen ab x, z (checkRows)
      columns = Finde alle Abhangigkeiten in der Spalten ab x, z
      raster = Finde das Raster mit dem groten Flacheninhalt in rows, columns
      if raster hat Mindestgroe 3x3 then
        Ersetze alle Werte in raster, die nicht Kanten sind, mit replacement
        Markiere alle Werte in raster als besucht
      end if
    end if
  end for
end procedure

procedure CHECKROWS(data, currentX, currentZ, chunkSize)
  for all z von currentZ bis Ende der Reihen oder Ende des Chunks do
    difference = data[z][x] - data[z][x + 1]
    for all x von currentX bis Ende der Spalte oder Ende des Chunks do
      if nachste Differenz == difference then
        Inkrementiere aktuelle Breite
      else if aktuelle Breite < 3 then
        return alle Breitenlangen und deren Reihen
      else if aktuelle Breite < letzte Breite then
        Speichere Breitenlange und Reihe
        letzte Breite = aktuelle Breite
        Weiter mit nachster Reihe
      else
        Weiter mit nachster Reihe
      end if
    end for
  end for
  return alle Breitenlangen und deren Reihen
end procedure

```

3 Anforderungsanalyse

3.1 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, dass 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 und realitätsnah repräsentieren. Eine weitere mögliche Zielgruppe sind Schüler und Studenten, welche die Visualisierung unter anderem für Forschungszwecke nutzen können. Das Jet Propulsion Laboratory (JPL), einem Institut der NASA, welches sich auf die Entwicklung von Raumsonden und Satelliten spezialisiert, nutzt intern eine Visualisierung namens Mars Trek (siehe Abschnitt 3.2.1) bereits um Landeziele für Marsmissionen zu finden und plant dies auch für die Auswahl zukünftiger bemannter Erkundungsmissionen zu nutzen^{3.1}. Des Weiteren könnte eine Visualisierung in den Forschungsbereichen Astronomie und insbesondere Geologie nützlich sein.

3.2 Ist-Zustand

Bevor Anforderungen definiert werden können, müssen alternative Visualisierungen in Betracht gezogen werden. Dieses Projekt soll sich dabei an gängigen Visualisierungen orientieren und versuchen deren Schwächen zu vermindern. Zu beachten ist, dass eine Limitierung auf kostenlose Anwendung vorgenommen, um einen besseren Vergleich für dieses Projekt zu erhalten. Eine bekannte Visualisierung des JPL nennt sich Mars Trek¹ und wurde im Jahr 2015 veröffentlicht. Es ist ein web-basiertes Tool um verschiedene Daten, gesammelt aus unterschiedlichen NASA Missionen, in einer interaktiven 3D Visualisierung zu visualisieren. Eine weitere Visualisierung versteckt sich in der Desktop Version von Google Earth². Google Earth ist wohl einer der populärsten Visualisierungen von Geodaten, insbesondere für die Erde. Und auch wenn es der Name nicht vermuten lässt, befindet sich in der Download-Version seit dem Jahr 2009 auch eine Visualisierung des Mars. Im folgenden werden die Alternativen beschrieben und deren Stärken und Schwächen hervorgehoben.

¹<https://trek.nasa.gov/mars/index.html>

²<https://www.google.com/earth/versions/#earth-pro>

3.2.1 Mars Trek

Das erste, was bei der Anwendung negativ auffällt, sind die langen Ladezeiten, die noch vor der eigentlichen Visualisierung zu sehen sind. Diese liegen beim ersten Laden der Seite im zweistelligen Sekundenbereich bis zur Sichtbarkeit der Visualisierung und ungefähr doppelt so lang bis zum vollständigen Laden der Seite (siehe Abbildung 3.1 für ein durchschnittliches Laden). Nachfolgende Ladezyklen sind dann dank Cachings deutlich schneller, die Ladezeiten sind aber für ein anfangs sehr gering aufgelöstes 2D Bild trotzdem sehr hoch. Zwar sind dadurch anfangs keine visuellen Artefakte zu sehen, diese werden dann beim Bewegen der Kamera jedoch sichtbar, sodass die Nützlichkeit in Frage gestellt werden kann.

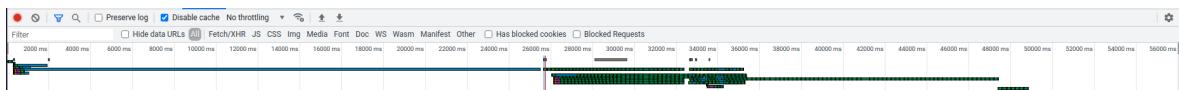


Abbildung 3.1: Ladezeiten der Mars Trek Anwendung bei 50 Mbit/s auf Mittelklasse-PC

Initial wird man von der Anwendung mit einer 2D Ansicht des Mars mit Daten der Viking Missionen begrüßt (siehe Abbildung 3.2). Zusätzlich wird dem Nutzer ein Tutorial angeboten, was auf eine relativ komplexe Anwendung schließen lässt. Die Anwendung sieht auf den ersten Blick allerdings sehr übersichtlich aus und die einzelnen Menüs lenken nicht von der eigentlichen Visualisierung ab.

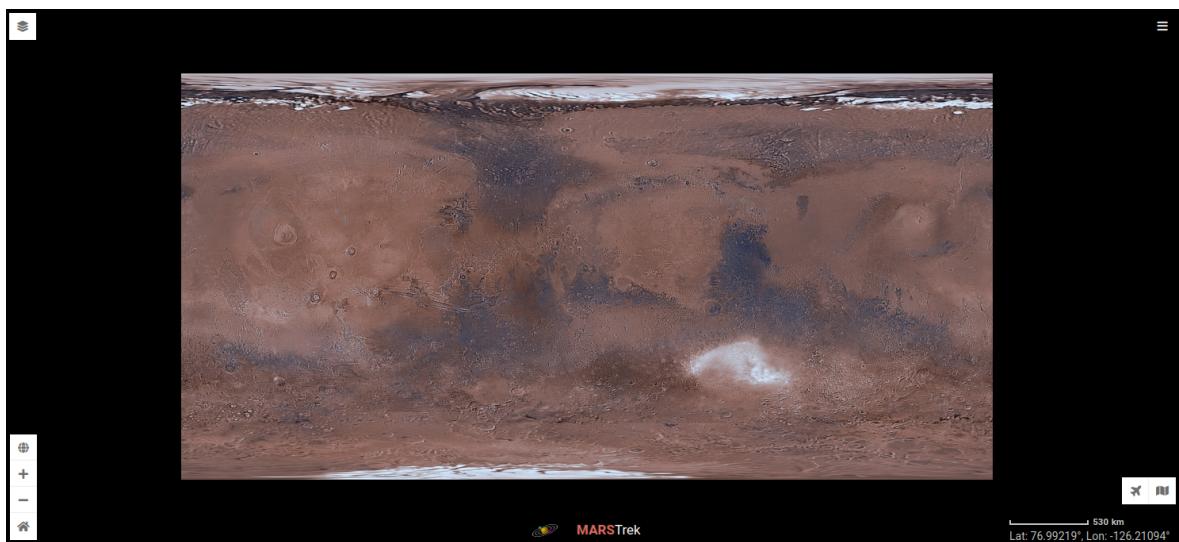


Abbildung 3.2: Initiale Ansicht der Mars Trek Anwendung

Dabei befindet sich in der oberen linken Ecke ein Menü, mit dem man die verwendeten Daten auswählen kann. Unter anderem können auch MOLA Daten gewählt werden, bei denen die Höhenwerte als Farbwerte kodiert sind, ein Punkt, der auch mit diesem Projekt genutzt werden soll. Negativ fällt hier auf, dass die UI hier sehr überladen ist. Es stehen über 2000 verschiedene Datensätze zur Auswahl und die

3 Anforderungsanalyse

Filter nach Mission, Messinstrument, Koordinaten oder nach verschiedenen (Sub-)Kategorien ist für den Laien schwer verständlich. Auch sind nicht alle Daten für alle Flächen verfügbar, sodass teilweise nicht das gewünschte Resultat zustande kommt und sich verschiedene Ebenen nicht vollständig überlappen. Hier ist zwar ein Zeichentool vorhanden, mit dem man Ausschnitte auf der Karte definieren kann, die als weiterer Filter genutzt werden können, dies ist aber auch nicht intuitiv verständlich.

In der unteren linken Ecke befindet sich ein Menü, mit dem man unter anderem die Projektion wechseln kann. Hier stehen eine Projektion der beiden Pole, eine globale 2D Ansicht und eine Projektion als Kugel zur Auswahl. Hier könnte das Wechseln noch intuitiver gemacht werden, da die Ansicht als 3D Modell das Highlight der Anwendung sein sollte und die derzeitige Lösung sehr versteckt ist. Des Weiteren lässt sich in dem Menü der Zoom in kleinen Schritten vergrößern und verkleinern, was durch ein Scrollen des Mausrads deutlich leichter fällt. Auch existiert dort ein Button, um die Visualisierung wieder auf den Startzustand zurückzusetzen. Wenn die 3D Ansicht ausgewählt wurde, ist zusätzlich ein Button eingeblendet, mit dem man zwischen einer statischen Kameraansicht und einer frei bewegbaren Kamera wählen kann. Die statische Kamera kann genutzt werden, um in einem festen Abstand um den Planeten zu rotieren und die dynamische Kamera bewegt sich entlang der Blickrichtung und lässt sich durch die bekannten Steuerungstasten WASD oder Pfeiltasten steuern. Dadurch lassen sich bestimmte Orte viel genauer ansteuern, was ein deutlicher Pluspunkt ist.

In der oberen rechten Ecke befindet sich ein allgemeines Menü, in dem verschiedene nützliche Tools vorhanden sind, unter anderem um Distanzen und Höhenprofile zu messen, eine 3D Datei zu erstellen oder die Winkel zur Sonne über die Zeit zu berechnen. Auch diese Tools sind für den durchschnittlichen Nutzer sicherlich nicht besonders nützlich. Auch sind diese Tools nicht in der 3D Ansicht vorhanden, was aus der reinen Machbarkeit keinen Sinn ergibt. Des Weiteren sind hier allgemeine Informationen über die Anwendung, die Entwickler, Release Notes oder Systemanforderungen zu finden. Letzte belegen, dass die Anwendung mit WebGL programmiert wurde und dass die 3D Ansicht einen relativ neuen Browser benötigt.

In der unteren rechten Ecke existiert zum einen eine Möglichkeit, zu bestimmten Koordinaten zu springen. Zum anderen befindet sich dort eine Übersicht, welche den Datensatz auf den einzelnen Ebenen anzeigt. Auch kann hier die Sichtbarkeit einzelner Ebenen angepasst werden, falls diese sich überlappen sollen. Abschließend befindet sich dort eine Skala, die die Längen in Kilometern einer Länge auf dem Bildschirm zuordnet und die Koordinaten des Punktes unter dem Mauszeiger. Hier wird allerdings die Angabe der Höhe vermisst, welche sich wunderbar in das Menü integrieren würde. Generell scheint es keine Möglichkeit zu geben, sich mehr Informationen zu einem bestimmten Punkt anzeigen zu lassen, was für bestimmte geografische Features durchaus sinnvoll sein könnte.

Alles in allem ist die Anwendung ein sehr gutes Beispiel und das generelle Design und viele Features können übernommen werden. Allerdings soll gerade im Hinblick auf eine breitere Zielgruppe die Komplexität verringert werden, indem einige Features bewusst nicht übernommen werden. Auch sollte die User Experience (UX) in einigen

Menüs überdacht werden. Idealerweise sollte auch auf die langen Ladezeiten verzichtet werden.

3.2.2 Google Earth

Ein Negativpunkt bei dieser Anwendung ist die Tatsache, dass die Web-Version eine Visualisierung des Mars nicht unterstützt. Ein Download ist immer ein weiterer Schritt, den viele Nutzer, welche schnell eine Visualisierung betrachten möchten, nicht unbedingt gehen möchten. Auch hier sollte das Web die technischen Möglichkeiten geben, alle existierenden Features umzusetzen. Und auch bei dieser Anwendung wird man als erstes mit einem sehr textlastigem Tutorial und anschließend von einer sehr umfangreichen Anwendung, mit Blick auf die Erde, begrüßt (siehe Abbildung 3.3).

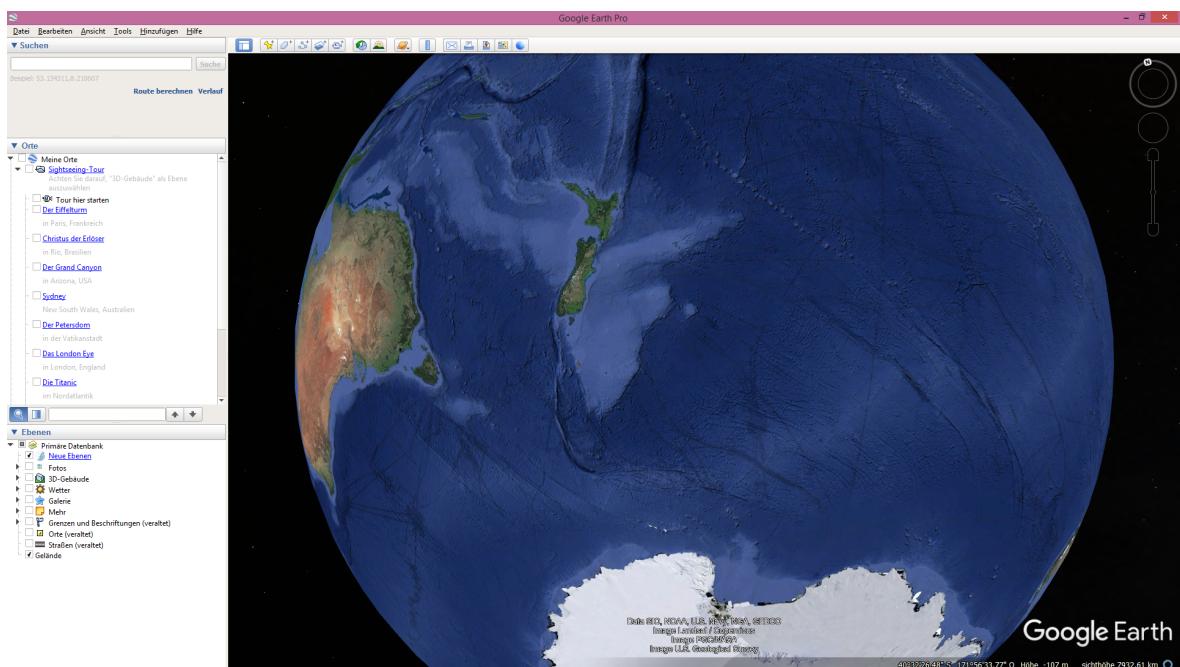


Abbildung 3.3: Initiale Ansicht der Google Earth Anwendung

Hier ist nicht auf Anhieb ersichtlich, wie man die Visualisierung auf den Mars umstellen kann. Es befindet sich die Standard-Windows Menüleiste in der oberen linken Ecke, doch dort sind die Optionen sehr begrenzt. Zusätzlich befindet sich darunter eine weitere Menüleiste, die einige Funktionen aus der eigentlichen Menüleiste besonders dupliziert und so besonders hervorhebt. In den Menüs befinden sich unter anderem verschiedene Tools wie Entfernungsmesser, GPS oder auch ein Flugsimulator, mit dem man die Oberfläche in sehr kreativer Weise erkunden kann. Dies ist ein großer Pluspunkt, allerdings können die wenigsten ein Flugzeug intuitiv steuern und natürlich überschreitet dies den geplanten Projektrahmen bei weitem. Des Weiteren können verschiedene Elemente in der Visualisierung an- und ausgeschaltet werden, unter anderem eine realistische Beleuchtung per Sonne zu verschiedenen Tageszeiten. Auch eine realistische Wasseroberfläche und Atmosphäre,

3 Anforderungsanalyse

zusammen mit dem realgetreuen Sternenhimmel im Hintergrund, verstärken den Realitätsgrad in sehr großer Weise. Des Weiteren sind Ansichten wie ein Gitternetz mit Breiten- und Längenangaben und eine Minimap in der Ecke sehr hilfreich bei der Navigation. Weitere Menüpunkte dienen zum Wechseln zwischen Google Maps oder der Web-Version. Ein Wechsel von einer Download-Version zum Browser ist hierbei fragwürdig, vor allem da alle Funktionen der Web-Version auch in dieser Version vorhanden sind. Ein etwas kurioses Problem verbirgt sich hinter dem Menüpunkt "Regionating durchführen". Zum einen ist nicht verständlich, was man mit dem sich öffnenden Fenster verwirklichen kann, zum anderen lässt sich dieses Fenster einfach nicht mehr schließen, da die entsprechenden Buttons deaktiviert wurden. Warum so ein Usability-Problem bei einer so bekannten Anwendung einer so bekannten Firma beim Testen nicht entdeckt wurde ist nicht verständlich.

Schlussendlich findet man den Button zum Wechseln des Planeten unter den Menüpunkten "Ansicht" und dann "Erkunden", eine Wahl, die auf den ersten Blick nicht verständlich erscheint. Sobald man es geschafft hat, die Anwendung auf die Mars Visualisierung umzuschalten, fällt auf, dass gar keine 3D Höhendaten angezeigt werden, sondern Kamerabilder verschiedenster Marsmissionen als Textur auf eine Kugel projiziert wurden. Hierbei fällt auf, dass nicht immer die gleichen Daten für alle Flächen vorhanden sind und so ein Mix aus verschiedenen Datenquellen genutzt wird. Dies führt zu einem etwas schlechterem visuellen Eindruck (siehe Abbildung 3.4).

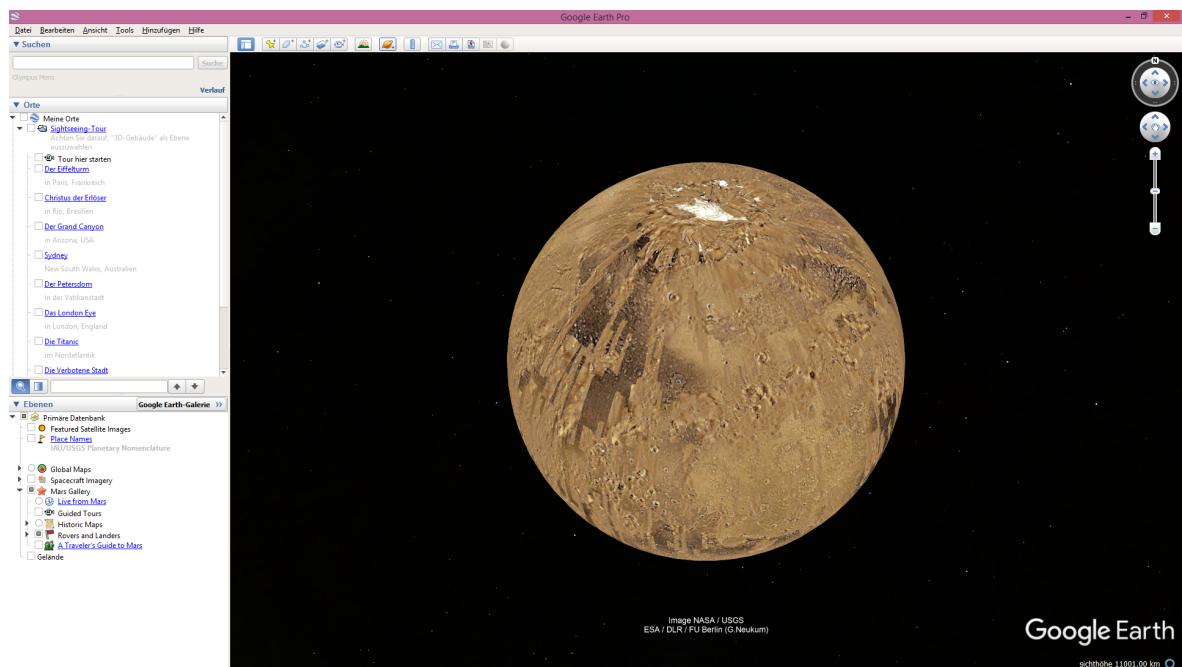


Abbildung 3.4: Initiale Mars Visualisierung der Google Earth Anwendung

Auch diese Anwendung nutzt das Konzept von Ebenen um verschiedene Daten auf dem Globus gleichzeitig anzuzeigen. Hier wird der Fokus aber mehr auf verschiedene Dinge gelegt, welche sich in vollständig in die Visualisierung integrieren (z.B. Ortsnamen, Mars Rover oder Fotos bestimmter Marsmissionen). Hier findet man

dann auch die Checkbox "Gelände", welche eine echte 3D Höhendarstellung ermöglicht. Der wahrscheinlichste Grund, warum dies standardmäßig deaktiviert ist, ist, dass weiterhin Satellitenbilder auf dem Globus angezeigt werden und diese durch eine 3D Darstellung verzerrt werden. Dies verschlechtert die Kantenbildung zwischen benachbarten Satellitenbildern noch weiter. Eine Möglichkeit die Bodendarstellung auf eine weniger realistische Darstellung zu wechseln ist nicht vorhanden. Ein weiteres Problem ist das Panel Orte auf der linken Seite. Es ist definitiv für die Erde ausgelegt und erlaubt es dem Benutzer zu verschiedenen Sehenswürdigkeiten wie dem Eifelturm oder dem Petersdom zu springen, ein Feature, was auf dem Mars völlig fehl am Platz ist. Immerhin ist die Suchleiste in der oberen linken Ecke funktional und man kann zu bestimmten geographischen Feature wie dem Olympus Mons springen.

Auf der anderen Seite befindet sich verschiedene Möglichkeiten die Kamera zu bewegen. Zum einen ermöglichen sie die Rotation um den stationären Mars in x- und y-Richtung. Dies ist ebenfalls mit den Pfeiltasten und WASD und durch Ziehen der Maus in eine bestimmte Richtung möglich, was eine deutlich intuitive Steuerung darstellt. Zum anderen ermöglicht es die Rotation des Globus um die z-Achse (Rolling), was nachfolgende Interaktionen sehr verwirrend macht. Eine feste Positionierung des Nordpols nach oben ist eine allgemein bekannte Darstellung (auch bei physischen Globen) und ein Ändern dieser Darstellung sollte den wenigsten Nutzern von Vorteil sein. Des Weiteren ist auch hier eine Möglichkeit gegeben, den Zoom per Mausklick anzupassen. Da dies per Mausrad jedoch deutlich einfacher ist, ist auch hier der Nutzen in Frage gestellt. Eine frei im Raum bewegliche Kamera ist nicht vorhanden (mit Ausnahme des Flugsimulators), sodass bestimmte Orte teilweise recht umständlich erreicht werden können. In der unteren rechten Ecke befinden sich noch die Koordinaten der Position des Mauszeiger auf dem Planeten und eine Angabe, in welcher Höhe sich die Kamera über dem Erdboden befindet. Dies ist eine sehr nützliche Information um Abstände einordnen zu können. Allerdings wird auch hier die Höhenangabe des Punktes auf dem Planeten nicht angezeigt. Auch das übergroßes Logo (15 der Gesamtbreite) in der Ecke ist etwas ablenkend und hat keinen funktionalen Nutzen.

Alles in allem ist auch diese Anwendung sehr komplex und für Laien nicht als Visualisierung des Mars erkenntlich. Es kommt auch das Gefühl auf, als ob diese Anwendung ursprünglich nicht für die Visualisierung weiterer Planeten gedacht ist, ein Punkt, der wegen des Namens sehr wahrscheinlich scheint. Die Benutzerführung sollte verbessert werden und doppelte Wege ein bestimmtes Ziel aus Nutzersicht zu erreichen vermieden werden. Ein großer Pluspunkt dieser Anwendung ist der unglaublich hohe Realitätsgrad mit hochauflösten Bildern bei hohen Zoomfaktoren und den verschiedenen Möglichkeiten Wetter, Sonne, Wasser und ähnliche Dinge physikalisch korrekt in die Visualisierung einfließen zu lassen. Dieser Realitätsgrad wird jedoch durch den Mix aus verschiedenen Satellitenbildern stark gestört.

3.3 Anforderungen

Auf Grund des Prototypen-Charakters dieses Projekts besteht eine geringere Priorisierung bei den funktionalen und nicht-funktionalen Anforderungen. Es sollen möglichst viele verschiedenen Möglichkeiten zur Datenreduzierung (siehe Abschnitt

2.3 implementiert und evaluiert werden. Die Anwendung soll dennoch einen Nutzen erfüllen um die Evaluation so realistisch wie möglich zu gestalten. Auch hilft dies dabei, die Korrektheit der Implementierungen zu verifizieren, wenn eine vollständige Visualisierung mit verschiedenen Features vorhanden ist. Um die konkreten Anforderungen zu ermitteln, wurden diverse Personen aus der Zielgruppe nach ihren Wünschen befragt. Aus diesen Wünschen wurden die Features priorisiert, die von den meisten Personen gewünscht wurden. Des Weiteren wurden die Stärken der bisher analysierten Visualisierungen in Betracht gezogen und in die Anforderungen integriert. Bestimmte Features, welche im gegebenem Projektzeitraum nicht umsetzbar waren, wurden dabei im voraus verworfen.

3.3.1 Funktional

Die wichtigste funktionale Anforderung ist eine realistische Darstellung des Mars mit den beschriebenen MOLA Daten. Dabei soll eine reine 3D Darstellung gewählt werden, bei der die Höhenwerte realistisch den Globus verformen, das Anzeigen einer Textur der Oberfläche auf einer 3D Kugel soll nicht genutzt werden. Der Hauptgrund dafür ist die realistischere Darstellung, besonders bei hohen Zoomstufen. Des Weiteren hilft es dabei, die Datenreduzierung besser zu evaluieren, da das Generieren einer Textur aus den Daten eine einmalige Tätigkeit darstellt und auch keinem Zwang der Datenreduzierung unterliegt. Es kann unabhängig von der Laufzeit beliebig lange dauern und die Größe von einzelnen Texturen ist zwar durch die OpenGL Implementation limitiert, sie kann aber leicht in einzelne kleinere Texturen aufgeteilt werden. Die Gesamtdatenmenge ist dabei nicht vergleichbar mit der Datenmenge von Modellen mit individuellen Höhenwerten. Ein nicht repräsentativer Test auf eigener Hardware zeigte, dass ein Aufteilen der Textur in 4 Abschnitte bereits ausreichen würde um sie in höchster Auflösung anzeigen zu können. Und Techniken wie das frustum- oder occlusion-culling, die zur Laufzeit die Datenmenge reduzieren, sind dabei nicht notwendig. Neben der Projektion der Daten in eine Kugelform soll auch die flache Darstellung ermöglicht werden, bei der allerdings dennoch 3D Koordinaten genutzt werden. Dies soll, anders als bei der MarsTrek Anwendung, allerdings nicht der Standard sein und vom Benutzer bewusst ausgewählt werden müssen. Ein möglicher Nutzen von einer flachen Darstellung ist die bessere Einschätzung von Entfernung und geraden Linien, was auf einer Kugel oft nicht intuitiv ist.

Für die Darstellung der Oberfläche soll eine schematische Darstellung genutzt werden, bei der keine weiteren Datenquellen notwendig sind. Die Wahl fiel dabei auf eine topographische Darstellung, bekannt aus Atlanten, bei der der Farbwert abhängig vom Höhenwert ist. Der Farbwert wird dabei aus einem vordefiniertem Farbbereich linear 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. Die verwendete Farbskala muss dem Benutzer auch mitgeteilt werden, eine entsprechende Übersicht welche Farbe zu welchem Höhenwert gehört, muss also angezeigt werden. Dabei sollen die Ränder der Skala vom Benutzer konfigurierbar sein. Der Nutzer soll also effektiv bestimmen können, welcher minimale und maximale Höhenwert für die lineare Interpolation genutzt wird. Welche Farben genutzt werden, muss noch

3 Anforderungsanalyse

evaluiert werden, der Nutzer soll diese aber nicht ändern können, da dafür kein Use-Case gefunden wurde und es die Komplexität unnötig erhöht. Standardmäßig soll ein minimaler und maximaler Wert gefunden werden, der eine möglichst gleichmäßige Farbverteilung ermöglicht. Besondere Extreme, wie zum Beispiel der Olympus Mons, sollten dabei außen vor gelassen werden, da es die Skala sonst stark in eine Richtung verzerrt und die Farbverteilung so stark verringert.

Für die Interaktion mit dem Mars sollen verschiedene Möglichkeiten genutzt werden, die möglichst intuitiv sind und keiner textuellen Erklärung benötigen. Zum einen betrifft das die Steuerung der Kamera. Dies ist ein wichtiger Aspekt im Design des Prototypen und es 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. Zum einen soll hier eine stationäre Kamera implementiert werden, die in einem festen Abstand um den Globus rotiert. Die Rotation soll dabei mit der Maus erfolgen und auch das Ändern des Abstands, höchstwahrscheinlich mit dem Mausrad, ermöglichen. Zum anderen soll eine frei bewegliche Kamera implementiert werden, die mit der Tastatur frei in alle 6 Richtungen gesteuert werden kann. In dieser Konfiguration soll man sich mit der Maus umschauen können und sich dann entsprechend der Blickrichtung bewegen können. Diese Kamera muss in der flachen Projektion auch erzwungen werden, da hier natürlich eine Rotation um den Globus nicht möglich ist. Eine weitere wichtige Interaktion soll das Erhalten von Informationen zu bestimmten Orten darstellen. Dabei soll der Benutzer auf einen Punkt auf dem Mars klicken können und dann zumindest die Koordinaten und den exakten Höhenwert angezeigt bekommen. Hier sollte idealerweise der Höhenwert in verschiedenen, vom Benutzer konfigurierbaren Einheiten angezeigt werden um die Zielgruppe noch weiter zu erhöhen. Standardmäßig sollen Meter oder Kilometer als Einheit genutzt werden.

Das Anzeigen von weiteren Informationen ist fakultativ, hier könnte man zum Beispiel auf benannte geologischen Features zurückgreifen, welche von der *Working Group for Planetary System Nomenclature* bereitgestellt werden. Diese werden in einer Datenbank bereitgestellt, welche unter anderem deren Position und Durchmesser beinhaltet, sodass eine Integration machbar wäre³. Weitere fakultative Anforderungen sind Features aus den analysierten Alternativen, die als hilfreich bei der Navigation eingestuft wurden. Hierzu zählt zum Beispiel das Gitternetz mit Längen- und Breitenangaben, sowie die Möglichkeit zu bestimmten Koordinaten zu springen. Auch wenn bei Darstellung der Oberfläche dieser Anwendung nicht realitätsnah werden soll, könnte man den Realitätsgrad dennoch deutlich erhöhen. Besonders die Google Earth Anwendung hat dies in sehr guter Weise gezeigt und Features wie eine korrekte Simulation der Sonne mit Licht- und Schatteneffekten zu bestimmten Tageszeiten oder einen realistischen Sternenhimmel im Hintergrund verstärkten die Immersion stark. Eine weitere fakultative Anforderung ist die Integration weiter Datenquellen. Diese Anwendung soll, wenn möglich, möglichst unabhängig von der Datenquelle sein und eine Integration weiterer Datenquellen, inklusive Auswahl durch den Benutzer, ermöglichen. Gute Alternativen sind hier unter anderem Höhendaten des Mondes (z.B. LOLA mit 8GB Rohdaten) oder der Erde (z.B. SRTM mit über 100Gb Rohdaten). Das Konzept von Ebenen, was in den analysierten Alternativen zum Einsatz kam, soll hier explizit nicht verwendet werden,

³<https://planetarynames.wr.usgs.gov/AdvancedSearch>

da es in allen Fällen als zu kompliziert und wenig vorteilhaft eingestuft wurde. Der Nutzer soll immer nur Daten eines einzigen Datensatzes angezeigt bekommen.

3.3.2 Nicht-Funktional

Die nicht-funktionalen Anforderungen richten sich hauptsächlich an die Qualität der Visualisierung und die Performance. Hier ist es schwer, die formulierten Wünsche in konkrete Anforderungen umzuwandeln, da hier oft subjektive Dinge eine Rolle spielen. Auch lassen sich bestimmte Dinge wie die Performance einfach nicht in der gewünschten Qualität (z.B. in der Form x Frames Per Second (FPS) auf Hardware y) definieren. Insbesondere die Hardwareanforderungen können dabei auf Grund der Hardwarevielfalt nicht genau definiert werden. Eine Kernaussage ist, dass die Anwendung möglichst immer die höchste Datenauflösung anzeigt, die möglich ist. Wie bereits beschrieben (siehe Abschnitt 2.3.1) ist ein dauerhaftes Anzeigen der höchstmöglichen Datenauflösung auf normaler Hardware einfach nicht umsetzbar. Daher soll die Auflösung immer genau so hoch gewählt werden, dass eine dauerhafte Framerate von ungefähr 30 FPS erreicht wird. Die Framerate sollte möglichst genau getroffen werden, eine Abweichung darunter schadet der User Experience (UX), eine Abweichung darüber erlaubt eine Verbesserung der Datenauflösung. Hier ist also ein sehr starkes Ringen zwischen Performance und Qualität vorhanden. Der Wert wurde, anstatt der üblichen 60 FPS, nur auf ungefähr 30 FPS festgelegt, da die Visualisierung keine starken Bewegungen und somit keine flüssigen Übergänge benötigt. Der Nutzer soll zwar die Kamera bewegen können, allerdings liegt der Fokus dabei immer noch auf der Visualisierung an sich.

Ein weiterer Aspekt ist, dass keine dedizierten Ladezeiten vorhanden sein sollen, in denen die Visualisierung nicht zu sehen ist. Dies wird häufig genutzt, um visuelle Ladeartefakte zu verstecken. Diese können zum Beispiel auftreten, wenn neue Abschnitte geladen werden müssen, nachdem der Benutzer die Kamerasicht verändert hat. Idealerweise soll auch auch ohne Ladezeiten der Nutzer nichts von der Trennung in Abschnitte mitbekommen und immer den Mars als Ganzes sehen. Dies kann zum Beispiel erreicht werden, in dem die Bewegung der Kamera antizipiert wird und Abschnitte, welche als nächstes geladen werden müssten, bereits vorgeladen werden.

Wichtig ist, dass die bisherigen nicht-funktionalen Anforderungen immer abhängig von der Hardware des Nutzer sind, da die Visualisierung auf Client-Seite durchgeführt wird. Es kann also kein einheitliches Performance-Konzept geben, welches für alle Nutzer geeignet ist. Zum einen können natürlich einzelne Parameter im voraus für verschiedene Performance-Klassen per Hand definiert werden. Da dafür allerdings eine umfassende Evaluation auf unterschiedlicher Hardware notwendig ist, ist der Zeitaufwand sehr groß. Eine Alternative ist es, zur Laufzeit bestimmte Parameter an die aktuellen Hardware anzupassen. Ein Problem bei beiden Ansätzen ist auch das Erfassen der aktuellen Hardware an sich. Auch ohne sich bereits auf eine Sprache oder Plattform festgelegt zu haben, gibt es meist keinen einheitlichen Weg diese zur Laufzeit zu erfassen. Natürlich könnte man den Nutzer nach der gewünschten Qualität fragen. Dieser Ansatz wird in der Realität sehr oft durchgeführt, allerdings soll die Zielgruppe kein Spezialwissen benötigen und Kenntnisse über die aktuelle Hardware sind nicht überall vorhanden. Ein Ansatz wäre es sich nicht auf die

3 Anforderungsanalyse

Hardware an sich zu konzentrieren, sondern zur Laufzeit die Auswirkungen zu messen und sich dementsprechend anzupassen. Man könnte zum Beispiel Parameter wie die Framerate als Indikator nutzen und dann zur Laufzeit auf Schwankungen reagieren. Alles in allem soll dieses Projekt als Prototyp dienen und ein Finetuning aller Parameter würde den Projektrahmen sprengen. Daher sind die bisher beschriebenen Anforderungen fakultativ und eher als ungefähre Richtwerte zu verstehen.

Eine Anforderung, auf die großen Wert gelegt werden soll, ist die Skalierbarkeit der Anwendung. Das Ziel dieses Projekts ist die Evaluation von verschiedenen Methoden zur Datenreduzierung. Dies soll idealerweise nicht von den verwendeten Daten abhängen und muss universell funktionieren. Die gesamte Anwendung, mit Ausnahme von Modulen die spezifische Datenformate interpretieren, muss also mit dieser Anforderung gestaltet werden. Ein Kernaspekt dieser Anforderung ist eine strikte Begrenzung von Arbeitsspeicher (RAM) und Grafikkartenspeicher (VRAM). Diese beiden Performancekriterien sind gut geeignet, die Qualität der Datenreduzierung zu beurteilen. Wichtig ist auch hier, dass der verwendete Datensatz keinen Einfluss auf die Anforderung hat, die gleichen Anforderungen müssen auch für einen Datensatz gelten, der mehrere hunderte Male größer ist. Als konkrete obere Grenze wurde Richtwerte an Mittelklasse Computer verwendet und bestehen aus 8 GB für den RAM und 4 GB für den VRAM. Eine Konsequenz, die sich aus diesen Anforderung ergibt ist, dass die Quelldaten niemals vollständig in den Arbeitsspeicher geladen werden dürfen (auch wenn es für MOLA Daten mit 2 GB ausreichen würde).

4 Konzeption

4.1 Vorgehensmodell

4.2 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. Sie bestimmt schlussendlich über die Zielgruppe und hat einen entscheidenden Einfluss auf die Reichweite des Projekts. 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 sollten¹. 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.

4.2.1 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

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

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 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 werden². 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 Beliebtheit³ 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.

4.2.2 C++

C++ ist eine objektorientierte Sprache mit niedrigerem Abstraktionsniveau als Java. Sie besitzt keine automatische Speicher verwaltung (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 Betriebssystem (bzw. sogar für jede CPU) ein eigenes Kompilat erzeugt werden muss. Auch das Web und Smartphones 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 ist⁴. 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 die Nutzung einer nativen Sprache die Arbeit mit OpenGL. Die Funktionsdefinitionen von OpenGL sind zwar sprachenunabhängig, jedoch ursprünglich in C definiert und so direkt in C++ nutzbar, ohne dass weitere Bindings integriert werden müssen.

²<https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/nio/ByteBuffer.html>

³seit 2001 in den Top 3 der beliebtesten Sprachen: <https://www.tiobe.com/tiobe-index/java/>

⁴<https://github.com/search?q=openGL&type=Repositories>

4.2.3 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 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 transpiliert⁵, 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 (UIs). Mit Hilfe von HTML und CSS lassen sich diese deskriptiv beschreiben und es existieren dem Nutzer bekannte Elemente für die Benutzereingabe (z.B. Button und Checkbox). Das führt zu einer sehr guten Trennung von UI und Logik, was im Clean Code Standard als sehr wünschenswert angesehen wird⁶. Trotzdem bleibt die Anbindung der Programmlogik an die UI über JavaScript sehr einfach. Das Programm kann ohne weiteres auf bestimmte UI Elemente zugreifen und so Konfigurationen auslesen und Werte an den Nutzer zurück übermitteln. Ein weiterer Vorteil ist, dass das OpenGL-Kontext und Fenster-Management direkt vom Browser übernommen wird. Normalerweise sind dort weitere Bibliotheken notwendig (sehr bekannt ist zum Beispiel GLFW), da diese Dinge abhängig vom Betriebssystem sind und das Anpassen und Vereinen aller Variationen in einer API ein immenser Aufwand ist. Auch das Erfassen von Benutzereingaben über Maus und Tastatur wird durch den Browser ermöglicht, was normalerweise eine Bibliothek ermöglichen müsste.

Allerdings ist auch JavaScript keine native Sprache und es existieren theoretisch die gleichen Nachteile mit der isolierten Umgebung wie bei Java. Dies wird im Browser allerdings durch die Verwendung von WebGL abgeschwächt, da viele dieser Probleme aus Programmierseite eliminiert werden. Die gesamte Kommunikation mit der OpenGL Implementation ist transparent für den Programmierer und manuelle Anpassungen sind nicht notwendig. Allerdings ist auch hier ein gewisser Performance Overhead gegenüber einer nativeren Sprache zu erwarten. Das größte Problem ergibt sich aus der Kernanforderung der hohen Skalierbarkeit und im speziellen der Arbeitsspeicheranforderung. Es verbietet ein Laden der Datenquelle in den Arbeitsspeicher. Die Daten müssen also an einem anderen Ort gespeichert werden, wobei sich hier natürlich das Dateisystem anbietet. Da man in einem Browser allerdings kein Zugriff auf das Dateisystem des Benutzers hat und der Nutzer auch nicht dazu aufgefordert werden soll, diese Daten selbst bereit zu stellen, muss eine andere Option gefunden werden. Es ist jedenfalls zwingend erforderlich, die Daten in kleineren Subsets anzufragen, sobald diese innerhalb der Anwendung benötigt werden. Eine dieser Optionen sind sogenannte *Range Requests*[2], mit denen man einen Teil einer Ressource auf einem Server anfragen kann. Mit diesem Ansatz gibt es jedoch einige Probleme. Zum einen ist dieser RFC noch relativ neu und wurde von vielen externen Servern, die die Daten hosten könnten, noch nicht implementiert.

⁵im Gegensatz zum Kompilieren findet keine Verringerung des Abstraktionslevels statt

⁶auch oft durch Design Patterns wie MVC realisiert

Insbesondere das Anfragen von mehreren Ranges pro Requests stellte sich als sehr problematisch dar, ist jedoch hilfreich für das Anfragen von rechteckigen Ausschnitten aus dem Datensatz. Zum anderen ist für die Größenordnung der Range ausschließlich Bytes vorgesehen. Da die MOLA Daten jedoch in einem Bildformat vorliegen und effektiv das Erfragen von Pixeln notwendig ist, müsste hier eine Umrechnung von Bytes zu Pixeln erfolgen. Dies ist jedoch bei einem Containerformat wie TIFF nicht ohne weiteres möglich, da die Daten keinem einheitlichen Format folgen. Es könnten in einem ersten Schritt die Metadaten ausgelesen werden, dies ist jedoch sehr komplex und alle Eventualitäten müssten implementiert werden. Die Alternative dazu wäre ein Hosten der Daten auf einem eigenen Server, der in einer selbst implementierten API die Daten für bestimmte Teilabschnitte bereitstellt und intern die Daten in einem beliebigen Format speichern kann. Dies erfordert die Implementierung eines Backends, was die Komplexität im Gegensatz zu einer reinen Frontend-Anwendung stark erhöht.

4.2.4 Fazit

Trotz der Nachteile wurde das Web als Zielplattform ausgewählt. Die Hauptursachen dafür sind die hohe Verfügbarkeit und die einfache UI Gestaltung. Es verringert die Einstiegshürde eines Downloads, was gerade bei einer Visualisierung sehr vorteilhaft ist. Als konkrete Sprache wurde TypeScript gewählt, da die eine starke Typisierung das Arbeiten mit einer unbekannten API stark erleichtert. Um dem Problem der Bereitstellung der Daten zu begegnen, soll ein kleiner Webserver implementiert werden, der die Daten bereitstellt. Für die Sprache des Backends soll Java genutzt werden. Die Sprache wird in der Backend Entwicklung sehr häufig genutzt und durch das riesige Ecosystem sind viele Ressourcen, unter anderem Bibliotheken für den Webserver, vorhanden. Des Weiteren ist im JDK die ImageIO-API vorhanden, welche ohne weiteres TIFF Bilddateien auf sehr arbeitsspeicherschonende Weise, zum Beispiel durch Datei-basierte Zwischenspeicher, laden kann. Die ImageIO-API ist des Weiteren sehr gut geeignet, da sie Möglichkeiten bereitstellt, rechteckige Ausschnitte aus einem Bild zu laden und die Daten in einem Array-basierten Format zurückzuliefern. Außerdem sollte durch die eigene Expertise die Entwicklungszeit stark verkürzt werden.

4.3 Bibliotheken

4.3.1 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 oder die Unterstützung von Websockets. Auch das

4 Konzeption

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 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 parallelen, nicht lange blockierenden Weise abgearbeitet werden können, sollte natürlich selbstverständlich sein, um einen Mehrbenutzerbetrieb auf vernünftige Weise zu ermöglichen. Die Möglichkeit bestimmte Requests zu bestimmten Endpoints zu routen soll auch positiv bewertet werden, da es die Entwicklung vereinfacht, ist jedoch nicht zwingend notwendig.

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 JDK internen Klassen die Anforderungen umzusetzen. Dazu zählt zum einen die direkte Verwendung von Sockets. Sie bieten ein sehr einfaches Weg, Daten über ein Netzwerk zu empfangen. Allerdings besitzen sie eine zu geringe Abstraktionsschicht, implementieren natürlich keinen HTTP Standard und sind blockierend, sodass Dinge wie eine separate Bearbeitung des Requests auf unterschiedlichen Threads selbst implementiert werden müssen. Auch muss das Routing von Requests, insbesondere für verschiedene HTTP Methoden oder Parameter, selbst implementiert werden. Die benötigten Dinge des HTTP Standards müssten also effektiv selbst implementiert werden. 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. Die wahrscheinlich bekannteste Alternative ist Spring Boot, ein Webserver, welcher dem Prinzip von "Convention over Configuration" folgt und die Entwicklung fast ausschließlich durch Java Annotations ermöglicht. Er im Enterprise Umfeld mit Abstand der meist verwendete Webserver und ist daher sehr verlässlich und leicht zu entwickeln. Er erfüllt selbstverständlich alle Anforderungen, allerdings ist der Featureumfang für dieses Projekt viel zu groß. Er ist Teil des Spring Frameworks und kommt daher mit einer

Vielzahl an Konzepten wie der Dependency Injection einher und besitzt Features für Health Checks oder Metriken, welche nicht benötigt werden. Eine weitere gute Alternative ist NanoHttpd, welcher einen sehr minimalen Webserver darstellt. Es implementiert den HTTP 1.1 Standard und versucht den Featureumfang gering zu halten. Features wie Authorisierung oder Rate Limitierung sind also mit Absicht nicht implementiert, was den Webserver natürlich sehr schnell macht. Eine Unterstützung von HTTPS ist vorhanden. Er besitzt eine moderne API mit stark funktionalem Stil, allerdings ist das Projekt an sich leicht veraltet und einige wichtige Issues wurden noch nicht behoben. Schlussendlich wurde sich für den Undertow Webserver entschieden. Er ist ein immer noch aktiv entwickelnder Webserver, der von JBoss verwaltet wird. Es wird als OpenSource Projekt mit einer Apache 2.0 Lizenz entwickelt und unterstützt den HTTP 2.0 Standard. Die Abhängigkeit an sich ist sehr modular gehalten und man kann selbst entscheiden, welche Features man für sein Projekt einbindet. Die API ist auch hier sehr modern gehalten und als Entwickler definiert man eine Kette von Handlern (Chain-Of-Responsibility Pattern), die per Filter definieren können auf welche Requests sie reagieren. Er erfüllt also alle Anforderungen perfekt und ist gleichzeitig modern und besitzt keine unnötige Abhängigkeiten.

4.3.2 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. 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 Dinge neu implementiert werden, die schon tausende Male implementiert wurden. Des Weiteren ist die Mathematik teilweise sehr kompliziert und Fehler hätten sich dadurch leicht einschleichen können. Insbesondere bei den Schnittberechnungen im frustum- und occlusion culling sind existierende Algorithmen sehr hilfreich. Auch Definitionen von bekannten mathematischen Typen wie Vektoren oder Matrizen sind notwendig und müssen erneut implementiert werden.

Bei der Wahl des Frameworks soll ein kleines Framework bevorzugt werden, besonders ein Framework mit Fokus auf die Spieleentwicklung ist nicht notwendig. Insbesondere Features wie Physics, Audio oder Animationen sind in vielen Frameworks enthalten, sind aber für dieses Projekt nicht notwendig. Das Framework muss die Definition von eigenen Shadern ermöglichen, inklusive der Übergabe von selbst definierten Werten, und darf der Erstellung prozeduraler Modelle nicht im Wege stehen. Des Weiteren soll das Rendering auf eine selbst bereitgestellte Canvas funktionieren, damit diese gut in die UI integriert werden kann. Auch soll eine perspektivische Kamera

Implementation vorhanden sein, welche sich auf eine einfache Weise im Raum bewegen lässt. Da für dieses Projekt TypeScript verwendet werden soll, ist das Vorhandensein von Typendefinitionen absolut notwendig. Ein Pluspunkt ist es, wenn diese Definitionen von den Entwicklern selbst bereitgestellt werden und nicht von deren Nutzern. Außerdem wird sehr viel Wert auf eine gute Dokumentation gelegt, da jedes Grafikframework, im Gegensatz zu Webservern, eine eigene Art besitzt, das Rendering zu implementieren. Hier sollten idealerweise auch Beispiele vorhanden sein, da diese meist besser geeignet sind eine unbekannte API zu lernen. Die Wahl fiel relativ schnell auf ein Framework namens three.js. Es ist ein sehr aktiv entwickeltes Framework mit sehr vielen Beispielen und einer guten Dokumentation. Es erfüllt alle Anforderungen und besitzt gleichzeitig immer noch ein geringes Abstraktionsniveau, was den Nutzer nicht in ein bestimmtes Designmuster zwängt.

4.4 Architektur

Die Architektur der Anwendung soll sich in das Frontend und das Backend gliedern und kann in Abbildung 4.1 betrachtet werden. Die Architektur soll vor allem eigenständige Komponente voneinander trennen. Auch muss darauf geachtet werden, dass die Architektur offen für weitere Datensätze bleibt.

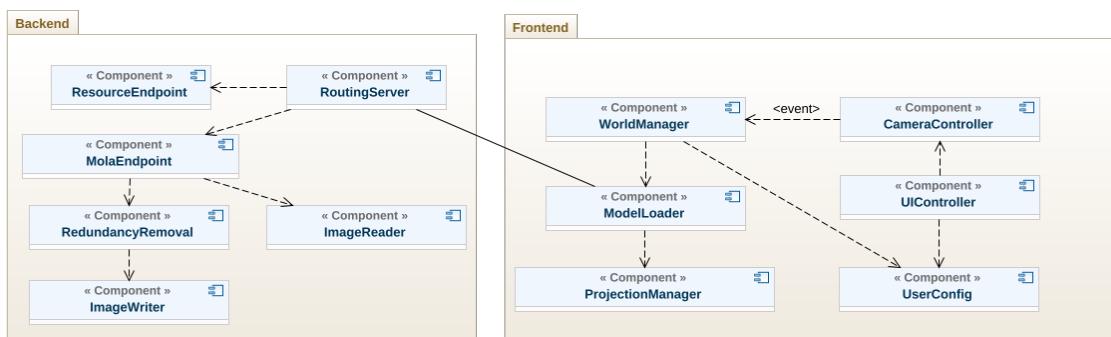


Abbildung 4.1: Komponentendiagramm der SolarViewer Architektur

Das Backend hat eine sehr einfache Struktur und besitzt einen einzigen Einstiegspunkt, den *RoutingServer*. Er ist dafür verantwortlich die eingehenden Requests an Hand ihrer URLs den einzelnen Endpoints zuzuordnen. Diese könnten in der Theorie auf verschiedenen Servern angesiedelt sein um ein Load Balancing zu ermöglichen. Dies ist für den Prototyp jedoch nicht erforderlich. Der *ResourceEndpoint* ist dafür zuständig, alle statischen Ressourcen wie die HTML-, CSS- oder JS-Dateien auszuliefern und bekommt alle Requests delegiert, welche vom *RoutingServer* keinem anderen Endpoint zugewiesen werden können. Der *MolaEndpoint* ist für die Auslieferung der MOLA Daten zuständig. Er muss unter anderem die Dimensionen des zu ladenden Abschnitts übergeben bekommen und muss diesen Ausschnitt dann vom Dateisystem laden. Er ist des weiteren auf eine Komponente angewiesen, welche den Datensatz, wie in Abschnitt 2.3.4 beschrieben, von Redundanzen befreit. Diese Komponente muss dann in der Lage sein, die Daten

4 Konzeption

wieder in das ursprüngliche Format zu schreiben, wofür die *ImageWriter* Komponente zuständig sein soll. Die modifizierten Daten können dann abschließend mit dem *ImageReader* gelesen und an das Frontend zurückgeliefert werden.

Das Frontend versucht durch eine lose Kopplung über Events die UI von der Logik zu trennen und orientiert sich am MVC-Pattern. Jegliche Aktionen des Benutzer, zum Beispiel ein Klick auf einen bestimmten Button, laden als erstes im *UIController*. Er ist für die UI-Logik, unter anderem auch die Validation von User Eingaben, zuständig und schreibt alle benutzerspezifischen Konfigurationen in die *UserConfig* Komponente. Diese Komponente besitzt keine Logik und kann direkt von anderen Komponenten genutzt werden um ihre Aufgaben zu spezifizieren. Der *CameraController* bekommt alle Benutzeraktionen weitergeleitet, die von der UI nicht interpretiert werden konnten. Diese werden dann genutzt, um die Kamera zu bewegen oder zu rotieren. Diese Aktionen sorgen dann für die Generierung eines Events, welche das Laden der einzelnen Abschnitte initiiert. Verantwortlich dafür ist der *WorldManager*. Er berechnet die Abchnitte die ge- oder entladen werden müssen, kümmert sich um den Detailgrad der Abschnitte und koordiniert den Ladevorgang. Auch besitzt er natürlich eine interne Repräsentation der aktuellen Abschnitte. Für das eigentliche Generieren eines Abschnitts ist dann der *ModelLoader* zuständig. Er fragt den Server über einen HTTP Aufruf nach den Höhendaten und generiert daraus ein 3D Modell. Dabei nutzt er den *ProjectionManager*, welche die Werte in verschiedene Projektion transformieren kann.

In Abbildung 4.2 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.

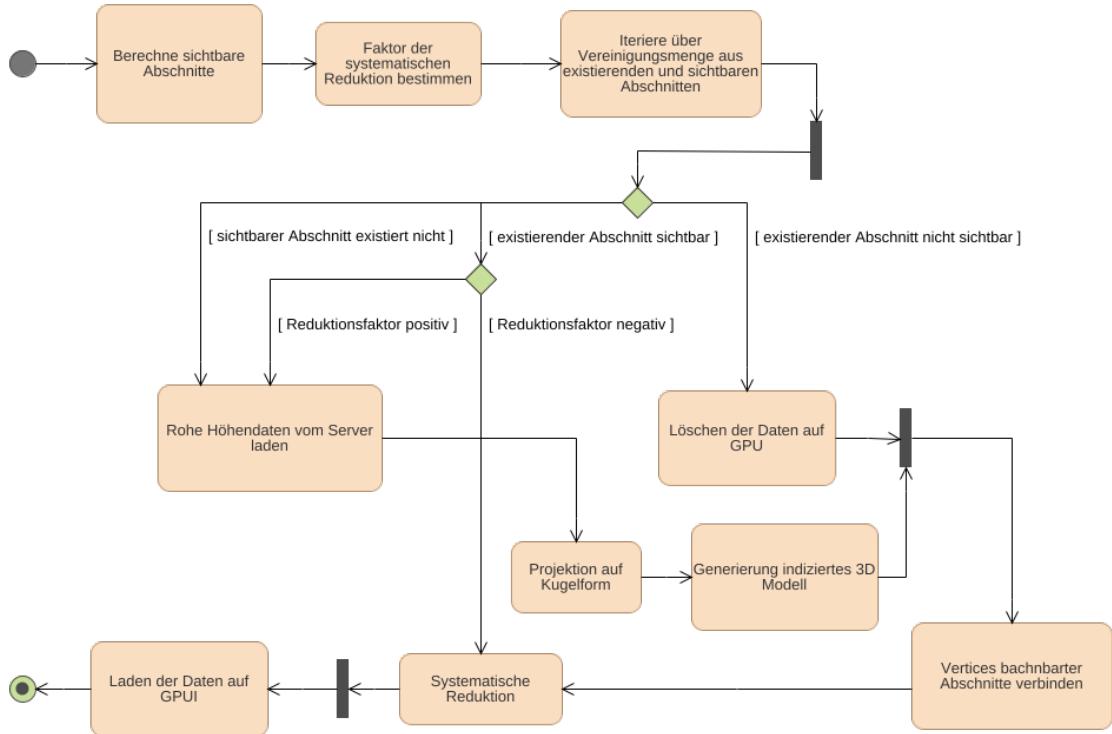


Abbildung 4.2: Aktivitätsdiagramm für die Erstellung der Welt auf Client-Seite

Als erstes muss eine Liste aller sichtbaren Abschnitte angelegt werden. Dabei kommt das beschriebene frustum- und occlusion culling zur Anwendung. Da eine verlustfreie Reduktion der Datenmenge nicht ausreicht, muss eine systematische Reduktion stattfinden. Wie der Faktor schlussendlich berechnet wird, muss noch evaluiert werden, allerdings muss er von der Anzahl der sichtbaren Abschnitte abhängen und die Datenmenge auf einem möglichst konstanten Niveau halten. Als nächstes muss über die Vereinigungsmenge aller aktuell existierender und sichtbarer Abschnitte iteriert werden. Sollte der Abschnitt aktuell existieren, aber nicht mehr sichtbar sein, so muss er 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. Sollte der Abschnitt aktuell existieren und auch weiterhin sichtbar sein, so muss geschaut werden, ob der Abschnitt aktuell den gleichen Reduktionsfaktor besitzt wie der berechnete. Ist dies der Fall, muss an dem Abschnitt nichts verändert werden. Sollte er größer sein, als der aktuelle Faktor, dann kann die Reduktion auf Client-Seite stattfinden, da nur eine bestimme Anzahl an Vertices entfernt werden muss. Sollte er jedoch kleiner sein, so müssen die Daten erneut vom Server angefragt werden, da die Rohdaten natürlich nicht mit dem Modell gespeichert werden. Auch hierbei muss darauf geachtet werden, die aktuellen Daten zu löschen, sobald der neue Abschnitt vollständig geladen wurde. Hier muss geschaut werden, dass keine Ladeartefakte beim Tauschen des Abschnitts auftreten. Der letzte Fall in der Ladeschleife ist, dass ein sichtbarer Abschnitt nicht existiert. Hier muss dann wie beschrieben als erstes der Server nach den Daten gefragt werden. Da Netzwerkoperationen in der Regel immer relativ lange dauern, ist hier eine

Parallelisierung geplant. Dies ist möglich, da in den nachfolgenden Aktionen keine Interaktion mit OpenGL notwendig ist. Als nächstes müssen die Daten in eine Kugelform projiziert werden, sollte dies vom Benutzer gewünscht werden. Auch muss aus den 1D Höhendaten jetzt ein vollständig indiziertes 3D Modell erzeugt werden. Ein Problem, dass bei der Benutzung von Abschnitten mit unterschiedlichen Modellen auftritt ist, dass diese eigentlich keine Vertices teilen. Die Polygone am Rand eines Abschnitts enden und verbinden sich nicht mit den Vertices des nächsten Abschnitts. Dies führt dazu, dass benachbarte Abschnitte eine sichtbare Lücke zwischeneinander besitzen. Um dies zu reparieren, muss als erstes darauf gewartet werden, dass alle parallel ladende Abschnitte fertig geladen wurden. Auch das Löschen muss zu diesem Zeitpunkt fertig sein, da klar sein muss, welcher Abschnitt welche Nachbarn besitzt. Dann müssen die Vertices der Kanten aller benachbarten Abschnitte in den eigenen Abschnitt kopiert werden. Erst wenn die Vertices eines Abschnitts vollständig definiert sind, kann die systematische Reduktion mit dem berechneten Faktor durchgeführt werden. Sobald dieser Ladevorgang für alle Abschnitte durchlaufen wurde, müssen die Daten erneut auf die GPU geladen werden. Hier ist erneut eine Synchronisation notwendig, da alle Befehle an OpenGL immer von einem Thread erfolgen müssen. Dies ist in der Regel der Thread, welcher den Grafikkontext initial angelegt hat. Man kann diesen Kontext zwar auch zwischen Threads wechseln, allerdings ist dies einer der teuersten Operationen in OpenGL und würde die Vorteile der Parallelisierung zunichte machen. Ob dies im Kontext von WebWorkers eines Browsers überhaupt funktioniert, ist auch fragwürdig, da diese in der Regel keinen Zugriff auf das DOM und somit auf die Canvas besitzen.

4.5 User Interface

Im Laufe der Planungsphase wurde anschließend ein Mock Up angelegt, welches in Abbildung 4.3 betrachtet werden kann. Wichtig beim Design war, dass nur das grundlegende Konzept geplant werden soll und spezielle Details außen vor gelassen werden. Diese können dann während der Implementierungsphase evaluiert werden. Dies ist zum einen dem Grund geschuldet, dass einige Unsicherheiten darin bestehen, welche Informationen der Implementation vom Benutzer zur Verfügung gestellt werden müssen. Zum anderen gibt es einige fakultative Anforderungen, bei denen noch nicht klar ist, ob sie schlussendlich umgesetzt werden.

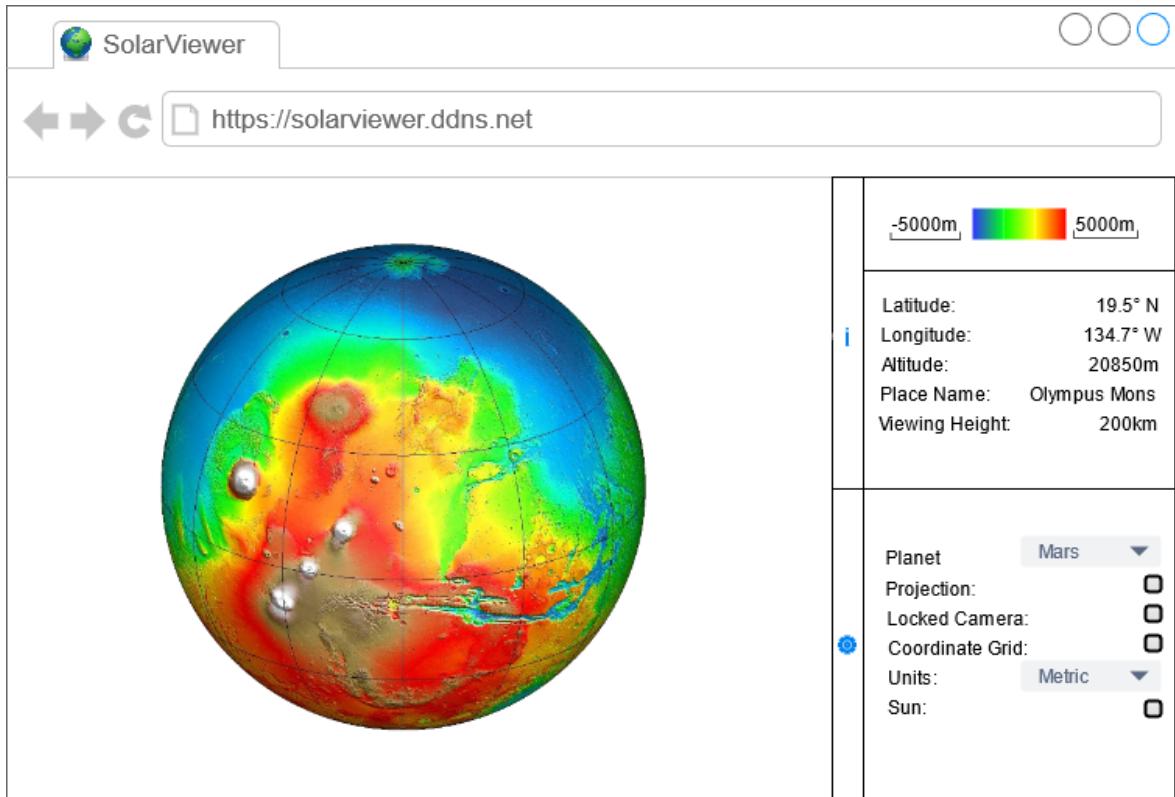


Abbildung 4.3: Mock Up der SolarViewer Anwendung

Das Design orientiert grundsätzlich sich an den analysierten Visualisierungen (siehe 3.2). Dabei soll die Visualisierung im Mittelpunkt stehen und eine kleine UI die Navigation unterstützen und einige nützliche Informationen anzeigen. Das Hauptkriterium für die UI ist der Ansatz, dass sie die Visualisierung in keiner Weise beeinträchtigen darf. Der User soll selbst entscheiden können, welche Informationen er benötigt. Dies soll über ausfahrbare Panels realisiert werden, die von einer Seite der Anwendung ein- und ausgeblendet werden können. Dabei sollen die unterschiedlichen Panels unabhängig voneinander agieren können und möglichst ähnliche Arten von Informationen beinhalten. Fürs erste soll die UI daher in mindestens drei Bereiche gegliedert werden: Visualisierung, Informationen und Konfigurationen. Hier könnte eine Aufteilung in weitere Panels, potentiell auf der anderen Seite der UI, sinnvoll sein, je nachdem welche Anforderungen umgesetzt werden können. Alle Panels sollen über ein kleines, deskriptives Icon am Rand identifiziert werden können. Hier wurde sich an gängigen Icons orientiert und ein Zahnrad für die Konfiguration und das bekannte Informationsicon für den Informationsbereich entschieden. Die Panels sollen initial eingeklappt sein um dem User die Dimensionen der Buttons zum Öffnen der Panels zu vermitteln. Ein Klick auf die eigentlichen Panels soll keinen Effekt haben, um versehentliches Schließen der Panels zu verhindern.

Das Informationspanel soll als erstes eine Skala zeigen, die die verwendeten Farben einem Höhenwert zuordnet. Da dies natürlich eine zentrale Information ist, ohne die die Visualisierung nicht verstanden werden kann, wurde als Alternative überlegt, sie dauerhaft im Hauptteil anzeigen zu lassen. Damit würden dem Benutzer einige unnötige Klicks erspart werden, allerdings steht es im Widerspruch zum

4 Konzeption

Kernkriterium, dass die Visualisierung nicht eingeschränkt werden darf. Bei besonders hohen Zoomstufen würde sich die Skala unweigerlich mit der Visualisierung überlagern und dies soll vermieden werden. Auch könnte der Nutzer diese Information dann nicht ausblenden, was ein Kernaspekt des Designs ist. Hier ist eine weitere Nutzer-Evaluation notwendig um dies zu bestätigen. Ein weiteres Problem ist die Schwierigkeit den Farben jetzt Höhenwerten zuzuordnen. Idealerweise muss dem Nutzer auf einen Blick klar sein, welche Farbe zu welchem Höhenwert gehört. Es ist jedoch aus Platzgründen schwer, mehrere Höhenangaben in die Skala aufzunehmen. Es wurde also entschieden, nur den minimalen und maximalen Höhenwert an den Rändern anzuzeigen. Hier muss vorausgesetzt werden, dass der Nutzer die Linearität versteht und sich die Höhenwerte zwischen den Grenzen erschließen kann. Auch dies muss im weiteren Verlauf mit Test-Nutzern evaluiert werden. Die Grenzen sollen laut den Anforderungen auch änderbar sein, daher muss dem Nutzer dies auch signalisiert werden. Hier wurde sich für ein Design ähnlich dem von Links entschieden, das zu einem Klick auf die Zahlen einladen soll.

Weitere Informationen auf dem Panel sollen die Koordinaten, die Höhenangabe und potentiell der Name des Ortes sein. Im Gegensatz zu den restlichen Informationen hängt es davon ab, auf welche Stelle der Nutzer auf dem Globus klickt. Hier wurde schon im voraus die fehlende Erklärung als Problem identifiziert. Es gibt kein Hinweis, der den Nutzer zu einem Klick auf den Globus verleitet. Sollte er als erstes das Informationspanel öffnen, so würden dort keine Informationen angezeigt werden können und der Nutzer könnte dies als Fehler werten. Auch ist unklar, was bei Klicks neben den Globus passieren sollte. Eine Möglichkeit wäre es, den Mauscursor zu einem bekannten, zu einem Klick verleitenden Cursor zu verändern, solange er sich über dem Globus befindet. Des Weiteren könnte man an Stelle von leeren Werten eine Textnachricht anzeigen, die den Nutzer zum Klick auf den Globus auffordert. Eine sehr gute Alternative nutzt die Google Earth Anwendung. Bei ihr kann man einen kleinen Marker per Drag&Drop an eine bestimmte Stelle ziehen, was sehr intuitiv ist, aber gleichzeitig wohl auch zu komplex für diese Anwendung. Im Endeffekt wird wohl auch Zeitgründen eine Textnachricht zum Einsatz kommen müssen, auch wenn Text aus Gründen der User Experience immer vermieden werden sollte. Des Weiteren soll noch die Höhe des Betrachters über der Erdoberfläche angezeigt werden, dies erfordert allerdings keine besonderen Nutzerinteraktionen und stellt daher kein Problem dar. Eine Designentscheidung bei all diesen Werten ist die Genauigkeit der angezeigten Werte. Dies ist natürlich abhängig von der schlussendlichen Nutzung der Anwendung und kann zwischen wissenschaftlicher und privater Nutzung stark schwanken. Idealerweise sollte die Genauigkeit deshalb konfigurierbar sein. Als Standard wurde hier eine eher geringere Genauigkeit gewählt, die allerdings abhängig von dem verwendeten Wert ist. Der Werte der Betrachterhöhe sollte auf Kilometer, die Erdbodenhöhe auf Meter und die Koordinaten auf eine Nachkommastelle gerundet werden.

Das Konfigurationspanel soll als erstes eine Möglichkeit beinhalten, den angezeigten Datensatz, bzw. den Planeten, zu ändern. Auch wenn die Anwendung fürs erste nur den Mars visualisieren soll, soll der Visualisierung weiterer Planeten nichts im Wege stehen und dies muss schon bei der ursprünglichen UI Gestaltung beachtet werden. Des Weiteren soll der Nutzer die Projektion ein- und ausschalten können. Andere Alternativen nutzen hier noch komplexere Auswahlmöglichkeiten (z.B.

4 Konzeption

unterschiedliche Kugelprojektion oder Projektionen der Pole), dies soll mit einer reinen Checkbox allerdings vereinfacht werden. Auch muss laut Anforderungen die Möglichkeit geschaffen werden, zwischen einer statischen und frei beweglichen Kamera zu wechseln. Ob hier ein besserer Weg gefunden werden kann, idealerweise ohne jedes Mal die Konfiguration zu öffnen, muss im Projektverlauf überlegt werden. Des Weiteren soll der Benutzer die Möglichkeit besitzen, die Einheiten der Höhenwerte zu ändern. Dabei soll zumindest die Wahl zwischen metrischen und imperialen ermöglicht werden. Falls im Projektverlauf keine weiteren Messsysteme gefunden werden können, kann dies auch als Checkbox realisiert werden. Des Weiteren sind in dem Panel weitere Optionen möglich, im konkreten Fall wurde Optionen zum ein- und ausschalten eines realistischen Beleuchtungsmodells und der Koordinatenlinien erdacht. Dies hängt natürlich stark von den schlussendlich implementierten Funktionen ab. Falls das Hinzufügen von Optionen aus Platzgründen schwierig wird, so muss das Informationspanel auf die andere Seite der Fensters verschoben werden.

Die Sprache der Anwendung soll Englisch sein, da so wieder die größere Zielgruppe erreicht werden kann. Insbesondere im akademischen Bereich und im Bereich der Raumfahrt sollte dies auch kein Problem darstellen, da hier Englischkenntnisse weit verbreitet sind. Die Wahl des Logos ist noch nicht final, es soll etwas universelles darstellen, was die Leute mit verschiedenen Planeten des Sonnensystems, nicht nur mit einem Planeten, in Verbindung bringen. Fürs erste wurde dafür ein kleines Erdlogo geplant. Für die Icons soll der Anbieter FontAwesome genutzt werden. Er stellt in einer freien Lizenz über 1600 verschiedene Icons bereit, welche als CSS Datei, gehostet auf FontAwesome Servern, in die Webseite eingebunden werden. Konkrete Icons können dann über CSS Klassen definiert werden, wobei für jedes Icon auch verschiedene Variationen bereitgestellt werden (solide, leichtgewichtig, mehrfarbig, etc.). Auch die Größen lassen sich ohne Verlust der Auflösung stark variieren und die Farben sind frei wählbar.

5 Implementierung

5.1 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 wird Maven als System für das Backend und npm als System für das Frontend genutzt. 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 wird das frontend-maven-plugin¹ genutzt, 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>

5.2 Visualisierung

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 sollte², 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 genutzt werden kann. Da die Ausgabe des Shaders 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) reichen.

²siehe [3], Kapitel 14, Abschnitt Avoid Dynamic Branching, S. 273

5.3 Kamerabewegung

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 bewusst 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. Stattdessen werden die Kameras in der Render-Schleife, begrenzt durch die Bildwiederholungsrate (vSync), so lange in der gleichen Konfiguration geupdated, bis ein entsprechendes keyup-Event der gleichen Taste registriert wurde.

Die zweite Implementierung ist die stationäre Kamera, welche sich um den Globus in einem festen Abstand bewegt. Auch hier wurde die Rotation durch das Dragging der Maus implementiert. Da beide Kameras die selben Steuerungsmöglichkeiten besitzen sollte dies für den User schnell verständlich sein. Der Abstand zum Globus kann dabei mit dem Mausrad verändert werden, was auch sehr intuitiv sein sollte. Beim ersten Ansatz der Implementierung wurde die Kamera zum Rotierungspunkt (Pivot) bewegt, um den entsprechenden Winkel rotiert und anschließend im lokalen Koordinatensystem entlang des ursprünglichen Bewegungsvektors zurück bewegt. Ein wesentlich einfacher und genauerer Ansatz wurde dann in der Form des Scene-Graphs der THREE Bibliothek gefunden. Dabei wird eine Hierarchie an Objekten in Form einer Baumstruktur definiert und alle Transformationen eines Objektes werden automatisch an dessen Kinder weitergegeben. Wenn jetzt die Kamera als Kind des Pivot definiert wird, dann kann dieser normal rotiert werden und die Kamera rotiert

automatisch im selben Winkel und Abstand mit. Für das Zoomen wurde das wheel-Event genutzt, welches Informationen darüber enthält, wie stark das Mausrad rotiert wurde. Hier gibt es die unterschiedlichen Einheiten Pixel, Zeilen und Seiten, welche auf die Navigation einer normaleren Seite mit Text ausgelegt sind. In konkreten Fall wurde nur die Scrollrichtung ermittelt und die Kamera mit einem festen Betrag entlang der lokalen z-Achse verschoben. Zusätzlich wurden minimale und maximale Abstände definiert und von der Implementation beachtet, damit die Kamera nicht über den Nullpunkt hinaus scrollt und somit effektiv die Scrollrichtung ändert.

Beide Implementationen updaten nach Erkennen jeglichen User-Inputs die View-Matrix und senden dann ein spezielles Event aus, dass den Ladeprozess in Gang setzt. Hier fiel auch ohne viel Testen auf, dass ein Anstoßen des Ladeprozesses bei jeder Interaktion performancetechnisch nicht durchführbar ist, da der Browser die Events in einer viel zu hohen Frequenz ausliefert. Es musste also eine Art Limitierung implementiert werden, die die Events auf eine maximale Rate beschränkt. Wichtig dabei war der Punkt, dass auch das letzte Event immer zugestellt werden musste. Es konnte also nicht einfach ein Zähler hochgezählt werden, der die vergangene Zeit inkrementiert und beim Erreichen des Limits das Events weiter propagiert, da natürlich jedes Event das potentiell letzte sein könnte. Stattdessen wurde die setTimeout-Funktion des Browsers genutzt, welche das Event am Ende der Wartezeit propagiert. Nachfolgende Events löschen dann den jeweils aktuellen Timeout und stellen sich selbst als Wartender in die virtuelle Warteschlange. Die Limitierung wurde im konkreten Fall auf 1 Event pro Sekunde gesetzt, was ein guter Kompromiss zwischen Performance und ausreichend dynamischem Ladens der Welt darstellt. Dieser feste Wert sollte idealerweise während der Laufzeit an die aktuelle Hardware angepasst werden um User Experience auf guter Hardware noch zu verbessern. Da dies jedoch nur eine fakultative Anwendung darstellt und das Finetuning zu viel Zeit in Anspruch genommen hätte, wurde es nicht implementiert.

5.4 Daten-Ladeprozess

5.4.1 Modell Generierung

5.4.2 Projektion

5.5 User Interface

5.6 Tests

6 Evaluation

7 Fazit

Literatur

- [1] Yue-Hong Chou, Pin-Shuo Liu und Raymond J. Dezzani. „Terrain complexity and reduction of topographic data“. In: *Journal of Geographical Systems* (1999). DOI: [10.1007/s101090050011](https://doi.org/10.1007/s101090050011).
- [2] R. Fielding, Y. Lafon und J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Range Requests*. 2014. URL: <https://datatracker.ietf.org/doc/html/rfc7233>.
- [3] Kyle Halladay. *Practical Shader Development*. 2019. DOI: [10.1007/978-1-4842-4457-9](https://doi.org/10.1007/978-1-4842-4457-9).
- [4] John Kessenich, Dave Baldwin und Randi Rost. *The OpenGL® Shading Language, Version 4.60.7*. 2019. URL: <https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.60.pdf>.
- [5] 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.