Bachelorarbeit

Entwicklung einer 3D-Animations-Bibiliothek für Java zur Darstellung von Flugobjekten und Flugtrajektorien

BACHELORARBEIT

für die Prüfung zum
Bachelor of Engineering

des Studienganges Informationstechnik
an der Dualen Hochschule Baden-Württemberg Mannheim

von:
Philipp Posovszky

Matrikelnummer: 2028068
Kurs: TIT 10 A NS

Bearbeitungszeitraum	01.07.2013 – 23.09.2013
Matrikelnummer, Kurs	2028068, TIT 10 ANS
Ausbildungsfirma	Deutsches Zentrum für Luft- und Raumfahrt e.V., Oberpfaffenhofen
Betreuer der Ausbildungsfirma	Dr. Erich Kemptner
Gutachter der Dualen Hochschule	Prof. Dr. Harald Kornmayer

Erklärung:

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und nur unter Benutzung angegebener Quellen und Hilfsmittel angefertigt habe.

Oberpfaffenhofen, den 19.09.2013 _____

Unterschrift des Verfassers

"*Knowledge speaks, but wisdom listens.*" *- Jimi Hendrix*

Kurzfassung

Das Projekt FaUSST behandelt die Erforschung von Verfahren und Technologien zur Entwicklung von „Unmanned combat air vehicles“ (UCAV). Das Institut für Hochfrequenztechnik und Radarsysteme des Deutschen Zentrums für Luft- und Raumfahrt (DLR) ist bei diesem Projekt mit dem Unterthema Radarsignaturreduktion von UCAV-Konfigurationen beteiligt. In der vorhandenen Software des Instituts ist eine Visualisierung von Radarzielen integriert, welche jedoch nicht mehr auf dem neuesten Stand der Technik ist. Sie soll daher durch eine leistungsfähige moderne Bibliothek ersetzt werden. Diese soll Funktionen für eine Visualisierung von 3D-Modellen, die Darstellung der Flugtrajektorie sowie die Visualisierung des gestreuten Radarsignals implementieren. Diese Funktionen werden unter anderem in Programmen des Projektes FaUSST benötigt. Dazu wurde zu Beginn der Arbeit eine geeignete Rendering Application Programming Interface API evaluiert. Im nächsten Schritt wurde eine modulare Architektur entworfen, welche das Austauschen der Rendering API ermöglicht. Anschließend wurde die entwickelte Bibliothek in die jeweiligen Programme integriert und getestet. Für die Zukunft ist angedacht, dass die entwickelte API als Grundlage für neue Softwareprodukte zur Visualisierung verwendet werden soll.

Abstract

The Microwaves and Radar Institute of DLR is involved in the project FaUSST, which studies processes and technologies for the development of unmanned combat air vehicles. In this project the department deals with radar signature reduction of UCAV configurations. In the existing software a visualization of radar targets is integrated, but cannot achieve all requirements in terms of efficiency and correctness. The goal is to develop a new modern library for Java, which implements following functions: a visualization of 3D models, a presentation of the flight trajectory as well as a visualization of the reflected radar signal. Therefore, in this thesis different rendering API's have been studied and evaluated. Furthermore, a modular software architecture has been designed and implemented. This architecture allows a flexible change of the rendering API. Finally the library will be tested and integrated. For the future it is considered that the developed API serves as basis for new software products for visualization.

Inhaltsverzeichnis

Abkürzungsverzeichnis

API	**A**pplication **P**rogramming **I**nterface
AWT	**A**bstract **W**indow **T**oolkit
CPACS	**C**ommon **P**arametric **A**ircraft **C**onfiguration **S**chema
CPU	**C**entral **P**rocessing **U**nit
DAG	**D**irected **a**cyclic **g**raph
DLR	**D**eutsches Zentrum für **L**uft- und **R**aumfahrt e.V.
FaUSST	**F**ortschrittliche **a**erodynamische **U**CAV **S**tabilitäts- und **S**teuerungs**t**echnologien
FPS	**F**rames **p**er **S**econd
GLU	Open**GL** **U**tility Library
GLSL	Open**GL** **S**hading **L**anguage
GPU	**G**raphical **P**rocessing **U**nit
GUI	**G**raphical **U**ser **I**nterface
IDE	**I**ntegrated **D**evelopment **E**nvironment
jME	**jM**onkey**E**ngine
JOGL	**J**ava Bindings for **O**pen**GL**
JVM	**J**ava **V**irtual **M**achine
LWJGL	**L**ight**we**ight **j**ava **g**aming **l**ibrary
OO	**O**bject **O**rientated
OpenGL	**Open G**raphics **L**ibrary
OpenAL	**Open A**udio **L**ibrary
OSGI	**OSGi Alliance** (früher **O**pen **S**ervices **G**ateway **i**nitiative)
Pun	**Pun**kt
Pan	**Pan**eel
RADAR	**Ra**dio **D**etection **A**nd **R**anging
RCS	**R**adar **C**ross **S**ection
SAR	**S**ynthetic **A**perture **R**adar
SWT	**S**tandard **W**idget **T**oolkit
SDK	**S**oftware **D**evelopment **K**it
UCAV	**U**nmanned **c**ombat **a**ir **v**ehicle
XML	E**x**tensible **M**arkup **L**anguage

Abbildungsverzeichnis

Diagramme

Tabellen

1. Motivation

Nach der Inbetriebnahme des Eurofighters Typhoon, auf längere Zeit keine umfangreichen Entwicklungsarbeiten für bemannte Kampfflugzeuge in Deutschland geplant. Vielmehr wird der Fokus künftig auf unbemannte Flugsysteme gerichtet. Diese sollen dazu in der Lage sein, autonome Luftunterstützung, Kampfeinsätze wie auch Luft- und Bodenaufklärung durchzuführen. [1] Das Institut für Hochfrequenztechnik und Radarsysteme des Deutschen Zentrums für Luft- und Raumfahrt e.V. (DLR) ist an dem Projekt FaUSST beteiligt, welches das Ziel hat, diesbzgl. Verfahren und Technologien zu entwickeln und überdies Planung und Beurteilung von UCAV Konfigurationen durchzuführen. Das Projekt FaUSST ist ein interdisziplinäres Projekt, es sind mehrere Institute des DLR aus ganz Deutschland daran beteiligt. Dazu zählen unteranderem das Institut für Aerodynamik und Strömungstechnik, das Institut für Antriebstechnik und das Institut für Hochfrequenztechnik und Radarsysteme. Dazu gesellt sich eine internationale Beteiligung in Form der NASA, sowie der Industrie im Rahmen der NATO Forschungsorganisation RTO.

Abbildung 1: Logo des Projekts FaUSST (Das F wird durch das DLR-F17-Modell repräsentiert)

Das Institut für Hochfrequenztechnik und Radarsysteme, Abteilung Aufklärung und Sicherheit, beschäftigt sich im Zuge des Projektes mit der Entdeckungs-wahrscheinlichkeit eines UCAVs durch ein oder mehrere Radarsysteme. Ziel ist es ein UCAV zu entwerfen, welches hohe Tarneigenschaften besitz. Um dies im Radarbereich

zu erreichen wurde bei beim UCAV-Design auf Seiten- und Höhenleitwerk verzichtet. Dadurch sinkt zwar die Radarsignatur, aber für die Aerodynamik entsteht dabei eine große Herausforderung bei der Flugzeugsteuerung.

„Grundsätzliche Fragen zur Stabilität und Steuerbarkeit eines solchen Nurflüglers werden am Beispiel der generischen Konfiguration DLR-F17 untersucht. Hierbei handelt es sich um einen gepfeilten Lambdaflügler modularen Aufbaus, der verschiedene Vorderkantengeometrien, Steuerklappen und durchströmte Triebwerkseinläufe zulässt."[2]

Die Radarsignatur wird mit der Hilfe des Simulationsprogramms SIGMA für jeden Punkt auf einer Flugtrajektorie des Flugobjektes berechnet. Danach wird in einem statistischen Analyseverfahren ausgewertet, ob die Radarstation (zukünftig eventuell mehrere) dazu in der Lage ist, das Flugzeug sicher zu detektieren. Mit dem Veranschaulichungswerkzeug Visualizer UCAV ist es möglich, sich die Ergebnisse aus dem Simulationsprogramm SIGMA und der statistischen Analyse in Diagrammen anzeigen zu lassen. Anhand dieser Ergebnisdiagramme können Flugmanöver ausgemacht werden bei denen die Entdeckungswahrscheinlichkeit stark ansteigt.

Um diese Visualisierung nun zu verbessern, wird eine 3D-Darstellung des Flugobjektes und der Flugtrajektorie benötigt. Dadurch ist es möglich, kritische Flugmanöver noch leichter zu ermitteln und die genaue Ursache für hohe Detektionswahrscheinlichkeitswerte auszumachen. Zudem existieren noch alte Softwareprodukte innerhalb der Fachgruppe, welche auch eine Visualisierung von 3D-Objekten basierend auf Java nutzen. Diese sind aber nicht mehr auf einem aktuellen Stand der Technik und aufgrund einer unübersichtlichen Softwarestruktur schwer zu erweitern und zu verbessern.

2. Einleitung

Innerhalb dieser Arbeit soll eine eigenständige Bibliothek zur Visualisierung für Java entworfen und implementiert werden. Diese Bibliothek soll es ermöglichen 3D-Objekte aus dem Geometriedatensatz im Punkt-/Paneelformat (Pun/Pan) darzustellen. Dazu soll noch die Möglichkeit bestehen eine Flugtrajektorie innerhalb eines 3D-Koordinatensystems zu zeichnen, inklusive eines Miniaturmodells des Flugzeugs auf der Trajektorie und weiteren Objekten (Beispielsweise die Radarstationen die versuchen das Flugzeug zu detektieren). Das Ziel am Ende ist es, eine Bibliothek zu erhalten die einfach zu warten ist und sich über den oben beschrieben Anwendungsfall hinaus benutzen lässt. Somit können alle momentan in der Abteilung eingesetzten Java basierenden Software-Visualisierungsmöglichkeiten vereinheitlicht werden.

Damit eine leichte Wartbarkeit und eine hohe Kompatibilität zu anderen Softwareprodukten möglich sind, wird der Schwerpunkt dieser Arbeit auf die Softwarearchitektur gelegt. Dazu sollen ein Konzept erarbeitet werden, welche es ermöglicht, über eine einfache Schnittstelle die Bibliothek zu steuern. Auch soll eine geeignete Wahl für eine API zur 3D Visualisierung getroffen werden.

Die Bibliothek soll dazu in der Lage sein einen Geometriedatensatz (priorisiert aus dem Pun/Pan-Datenformat) zu laden und als vollständiges 3D-Objekt zu Visualisieren. Dieses soll dann für einen gegebenen Sichtpunkts in die richtige Perspektive gedreht werden oder frei mit der Maus rotier-/skalier-/zoombar sein. Dazu kommt noch eine Visualisierungsmöglichkeit für Trajektorien im 3D-Raum. Das Objekt selbst soll sich entsprechend der Blickrichtung von der Radarstation aus perspektivisch korrekt ausrichten oder mit der Maus frei rotier-/skalier-/zoombar sein.

Es sollen die verfügbaren für Java Grafikbibliotheken auf ihre Eignung für dieses Projekt überprüft und miteinander verglichen werden. Nach einer grundlegenden Vorauswahl wurden folgende Bibliotheken (auch Engins genannt) ausgewählt und in dieser Arbeit auf ihre Eignung geprüft:

- jMonkey
- Lightweight Java Game Library
- Java 3D
- Java Bindings for OpenGL
- jPCT
- Ardor3D
- Aviatrix3D

Von diesen Engins soll die am besten geeignetste Engine innerhalb der Bibliothek eingesetzt werden. Dabei soll kein überproportionaler Änderungsaufwand für alte Programme entstehen. Die Grafikengine der Bibliothek soll nach Möglichkeit so eingebunden werden, dass sich diese leicht austauschen lässt. Somit soll für die Zukunft ein schneller Umstieg auf andere Grafikbibliotheken ermöglicht werden. Durch die Ausrichtung hin zu einer definierten Schnittstelle zur Grafikbibliothek wird gewährleistet, dass auch bei einem Stopp deren Weiterentwicklung schnell auf eine aktuelle, moderne Bibliothek gewechselt werden kann.

3. Technischer Hintergrund und Terminologie

3.1. Simulationsablauf einer Flugmission im Projekt FaUSST

Im Projekt FaUSST wird an neuen Technologien und Methoden zur Entwicklung von UCAV Konfigurationen geforscht. Jedes der beteiligten Institute trägt ein Teil zu der Entwicklung der entstehenden UCAV Konfiguration bei. Am Ende soll eine Vielzahl an Werkzeugen entstehen, die zu einer schnellen Entwicklung eines UCAVS beitragen. Das Institut für Hochfrequenztechnik und Radarsysteme mit der Abteilung Aufklärung und Sicherheit ermittelt im Zuge des Projektes die Entdeckungswahrscheinlichkeit eines UCAV's während einer Flugmission. Dabei dient zur Definition des Flugzeugs und der Mission das vom DLR spezifizierte CPACS-Dateiformat[1]. Diese Datei wird bei den Simulationen von einem Tool zum nächsten weitergegeben und anhaltend mit den Simulationsergebnissen erweitert. Durchgeführt werden alle Simulationen auf einem Server ohne ein Graphical User Interface (GUI) zu nutzen. Der komplette Ablauf aller Werkzeuge wird über die Client-/Server-Software **ModelCenter** gesteuert. In folgender Abbildung ist ein Ausschnitt aus der Prozessierungskette zu sehen, wie er innerhalb von **ModelCenter** abläuft.

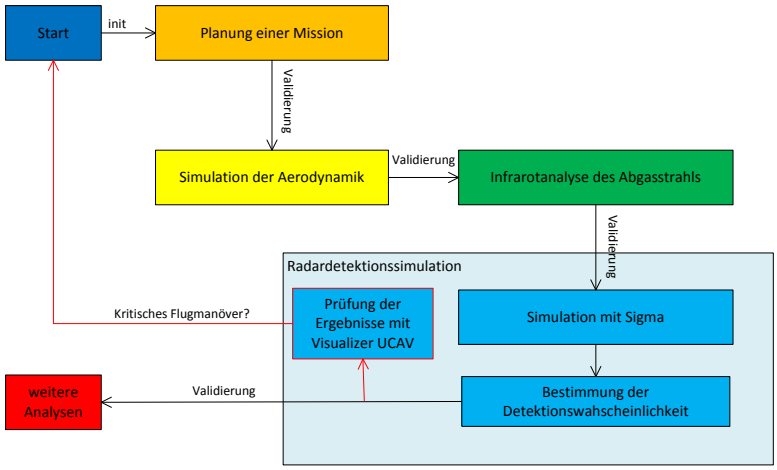

Diagramm 1: Ausschnitt der Prozessierungskette innerhalb von ModelCenter

[1] Link der CPACP-Projekt Homepage: Link: http://code.google.com/p/cpacs/

Das Programm ModelCenter wird für alle Simulationen verwendet, da die eigentlichen Programme bei den einzelnen Instituten bleiben. Aber jeder Mitarbeiter des Projekts kann die Prozessierungskette mit eigenen Beispielmissionen anstoßen kann, welche dann auf dem ModelCenter-Server ausgeführt wird.Am Ende der Prozessierungskette entsteht eine CPACS-Datei mit allen Simulationsergebnissen, welche mit beliebigen Tools (die XML-Dateien lesen) ausgewertet werden kann. Natürlich ist es für das Institut für Hochfrequenztechnik und Radarsysteme auch möglich, die Simulation der Entdeckungswahscheinlichkeiten auf einem lokalen Rechnersystem durchzuführen, ohne den Server mit ModelCenter zu verwenden. Eine Prüfung der Zwischenergebnisse mit Visualizer UCAV kann nur in einer Offline-Simulation, unabhängig vom Server erfolgen (roter Pfad im Diagramm), da dieses Programm die genauen Endergebnisse aus der Berechnung der Detektionswahrscheinlichkeit benötigt und nicht nur die Resultate aus der CPACS-Datei. In dieser sind derzeit nur der maximale RCS-Wert und die Detektionswahrscheinlichkeit abgelegt. Die Software Visualizer UCAV bietet momentan nur eine Auswertung der Plots und wird durch die Ergebnisse dieser Arbeit um eine 3D-Visualisierung erweiterbar sein.

3.2. Radardetektion von fluktuierenden Zielen

RADAR steht für „Radio Detection and Ranging“ (ursprünglich: Radio Aircraft Detection and Ranging) und beschreibt prinzipiell das aktive Verfahren zum Entdecken und Lokalisieren von Objekten und der Bestimmung von Objektparametern mit elektromagnetischen Wellen. Bei einem Radar wird mit der Hilfe eines Senders eine solche Welle erzeugt und abgestrahlt. Diese wird nach ihrer Wechselwirkung mit der Umgebung zur Empfangsantenne reflektiert/gestreut und im Radarempfänger ausgewertet. Durch eine anschließende Signalverarbeitung können abhängig vom Radarverfahren verschiedene Informationen bestimmt werden:

- Zielentfernung
- Zielrichtung
- Bewegungszustand des Ziels
- Zielklassifizierung
- Zielidentifizierung
- Zielabbildung

Im Gegensatz zum optischen Verfahren ist man zudem beim Radar relativ unabhängig von Wetter oder Tageszeit, da Wolken für das Radarsignal bedingt durchlässig sind. Die Radardetektion wird dabei heutzutage in vielen Bereichen des Lebens eingesetzt. Flugzeuge werden vom Kontrollturm mittels Radar dirigiert und überwacht, Schiffe wären ohne Radar im Nebel blind und Wetterprognosen werden ebenfalls mit Radardaten unterstützt. [3]

Speziell bei diesem Projekt geht es aber um die Entdeckung von Flugzeugen mittels eines Radars. Dazu muss ein Flugobjekt mittels eines Radargerätes erst einmal detektiert werden, das bedeutet das Flugobjekt muss sich klar aus dem Radarsignal abheben (das Signal-zu-Rausch-Verhältnis muss groß genug sein). Allgemein bekommt man als Radarsignal immer das Signal der kompletten Antennenblickrichtung (der Bodenreflexionsbeitrag wird auch Ground Clutter genannt) inklusive einem Signalrauschen zurück. Bei Flugzeugen entstehen bei der Verwendung von mm-Wellen starke Fluktuationen im RCS-Signal (Radar Cross Section; Radarrückstreuquerschnitt), wodurch keine konstanten Ausschläge im Radarsignal messbar sind.

„Das für eine Zielentdeckung mit vorgegebener Entdeckungswahrscheinlichkeit erforderliche Signal-zu-Rausch-Verhältnis wird durch die Fluktuation des Rückstreuquerschnittes σ eines Radarzieles erheblich vergrößert. […] Meist fluktuiert σ stark mit dem Aspektwinkel und der Sendefrequenz.“ [4]

Diese Problematik hat zu dem Programm **Trajekt** geführt (entwickelt in der Fachgruppe Radarsignaturen), welches anhand von Swerling-Modellen und einer stochastischen Analyse die Entdeckungswahrscheinlichkeit des Flugzeuges berechnet. Diese wird anhand von den berechneten RCS-Werten und technischen Daten des Radarsystems (Sendefrequenz, Position, Falschalarmwahrscheinlichkeit etc.) ermittelt. Da der mathematische Aufwand hinter der Berechnung der Entdeckungs-wahrscheinlichkeit zu groß ist, erfolgt nur eine kurze Umschreibung des Berechnungsvorgangs. Das Programm Trajekt berechnet die Einzelpuls-detektions¬wahrscheinlichkeit und greift dazu auf die klassischen Arbeiten von Marcum und Swerling zurück. Dabei werden nur die für die Einzelpulsdetektion relevanten Fälle eins und drei berücksichtigt (bei Spezialfällen noch die Log-Normalverteilung). Anhand der gegebenen Simulations¬daten werden anschließend mit dem Levenberg-Marquardt-Algorithmus die Parameter mit der geringsten

Quadratischen Abweichung ermittelt. Diese Parameter werden dann als Grundlage für die Berechnung der Entdeckungs¬wahrscheinlichkeit verwendet.

3.3. Beschreibung der CPACS-Datei

Das Common Parametric Aircraft Configuration Schema (CPACS) ist ein vom DLR entwickelter Dateistandard, mit welchem sich verschiedenste Flugzeugkonfigurationen generisch beschreiben lassen. CPACS-Dateien werden innerhalb des DLR hauptsächlich in Projekten für die Flugzeuge- und Helikopterentwicklung, Motorendesigns und der Analyse von Einflüssen auf das Klima im Flugverkehr verwendet. Dieses Dateiformat ermöglicht es, große interdisziplinäre Projekte in diesem Fachgebiet zu stemmen, bei dem zahlreiche Ingenieure/Wissenschaftler ihre Ergebnisse und/oder Konfigurationen in die CPACS-Datei einbringen können. Eine CPACS-Datei ist im XML-Format aufgebaut und kann mit der vom DLR entwickelten TIXI-Bibliothek leicht ausgelesen und verändert werden (aber auch mit jedem anderen XML-Parser). Das Flugzeug wird in seinen kompletten Einzelteilen beschrieben, das heißt die Verbindungsstreben des Rumpfes, die Motorposition, Position der Leitwerke, Materialien und viele weitere Komponenten. Zusätzlich gibt es noch die Möglichkeit eigene Parameterblöcke für bestimmte fachspezifische Simulationen zu definieren. Für die Simulationen mit **SIGMA** und die anschließende stochastische Analyse wird dazu ein neue Parameterblock „<sigma> … </sigma>“ angelegt. Dieser beschreibt Einstellungen für **SIGMA** (z. B. Aktivieren/Deaktivieren von Programmmodulen wie Doppelreflexion, Kantenkorrektur usw.) und sowie für die stochastische Analyse die detaillierte Beschreibung der Radarstationen. Für diese Berechnung mit **SIGMA** wird jedoch nicht die komplette Flugzeugkonfiguration aus der CPACS-Datei benötigt, sondern nur das Polygonnetz der Aussenhaut. Dieses wird, sobald die Flugzeugstruktur beschrieben ist, mit der Softwarebibliothek TIGL erzeugt. Dabei wird in Abhängigkeit von der Radarfrequenz der Paneeldatensatz so generiert, dass dessen Abweichung von der tatsächlichen Oberfläche maximal $\frac{\lambda}{10}$ beträgt.

3.4. Beschreibung des Punkt/Paneel-Format

Die geometrischen Datensätze liegen in dem fachgruppenintern Punkt/Paneel-Datenformat (Pun-/Pan-Format). vor Diese wird in diesem Projekt aus der CPACS-Datei mit der Hilfe von einem Werkzeug zur Polygonnetzgenerierung erzeugt. Aus diesen zwei Dateien wird im Simulationsprogramm und der Visualisierung das 3D-Modells aufgebaut. 3D-Modelle werden im Allgemeinen durch Polygone beschrieben bzw. aus einem Polygonnetz aus Dreiecken oder ebenen Vierecken beschrieben. Mit solchen Polygonnetzen kann jedes beliebige Objekt beliebig genau angenähert werden, einfach durch die Erhöhung der Polygonanzahl. Grundsätzlich wäre es auch möglich Objekte als Non-Uniform Rational B-Splines zu beschreiben, was eine optimale Annährung des Objektes ermöglichen würde. Das Programm **SIGMA** kann jedoch ausschließlich ebene Oberflächenstücke verarbeiten, was eine hohe Rechengeschwindigkeit der Software zur Folge hat. **SIGMA** basiert auf der physikalisch-optischen Methode, auf die numerische Integration der Maxwell'schen Gleichungen zur Streufeldberechnung kann daher verzichtet werden, was zu einem deutlich geringerem Rechenaufwand führt.

Folgend zwei Beispieltabellen und eine Darstellung der entstandenen Geometrie. Anschließend noch die Beschreibung des Tabellenaufbaus.

#Kommentar			
1	0	0	0
2	0	2	0
3	1	1	1
4	2	0	0
5	2	2	0

Tabelle 1: Beispiel einer Pun-Datei

#Kommentar					
1	3	2	5	3	0
1	3	1	2	3	0
1	3	4	1	3	0
1	3	4	3	5	0
2	4	1	2	5	4

Tabelle 2: Beispiel einer Pan-Datei

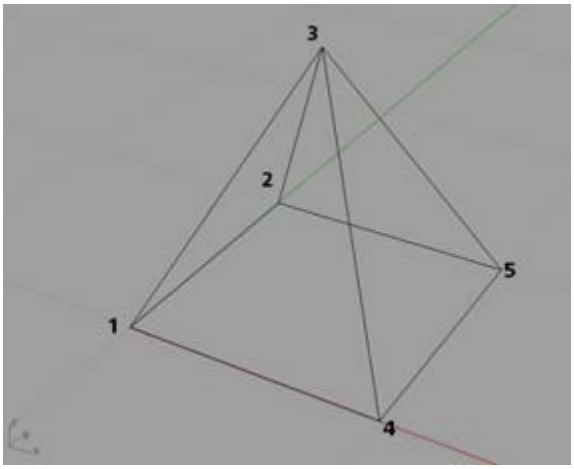

Abbildung 2: Erstellte Pyramide aus den Pun/Pan-Dateien

Dieser Datensatz besteht immer aus zwei Dateien. Die erste ist die Punktdatei (.pun). In jeder Zeile dieser Datei steht eine Punktnummer mit seinen zugehörigen X-, Y-, Z-Koordinaten. Die zweite, sogenannte Paneeldatei (.pan), enthält die Beschreibung der einzelnen Paneele. Pro Zeile stehen sechs Zahlen: die erste Zahl steht dabei für einen Materialindex, der auf das Material verweist, aus dem das Paneel besteht. Über diesen Index lässt sich somit das Material festlegen, welches das Rückstreuverhalten des Paneels gravierend beeinflussen kann. Die zweite Zahl gibt an ob es sich um ein Viereck (4) oder Dreieck (3) handelt. Die dritte bis sechste Zahl gibt die Nummer der Punkte an, aus denen das Paneel aufgebaut ist. Zu beachten ist bei der Erstellung von solchen Datensätzen, dass der Umlaufsinn der Paneele korrekt und das Polygonnetz geschlossen ist. [5]

3.5. Beschreibung von Szenegraphs

Innerhalb der meisten 3D-Engins werden Objekte über einen Szenegraph verwaltet. Dieser beschreibt eine komplette 3D-Szene wie sie später ausgegeben werden soll.

Dieser Graph ist dabei in einer baumähnlichen Struktur angeordnet, welche alle Informationen zu Rotationen, Transformation und Beleuchtung sowie die eigentlichen Geometriedaten beinhaltet. Die Szenengraphen (DAG) sind immer azyklisch als zusammenhängender Baum strukturiert, jeder Knoten (engl. node) hat genau einen Vorgänger (mit Ausnahme des Wurzel-Knotens).

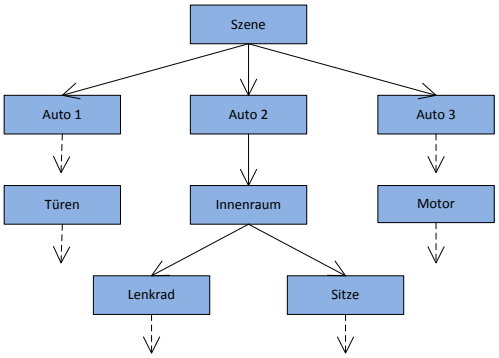

Diagramm 2: Szenengraph einer Szene mit mehreren Autos

Durch diese Struktur können Abhängigkeiten zwischen Objekten leicht umgesetzt werden. Im Diagramm 2 ist ein Szenengraph dargestellt. In der Szene befinden sich mehrere Autos, welche wiederum aus einem Innenraum, Türen, dem Motor und weiteren Bestandteilen bestehen. Diese Unterobjekte bestehen wiederum aus vielen anderen Objekten. Mittels der gegebenen 3D-Engine kann nun ein Objekt verschoben/rotiert werden und alle Unterobjekte erfahren dieselbe Veränderung. Abhängig von der Umsetzung innerhalb der 3D-Engine können auch weitere Eigenschaften in den Objekten abgelegt werden, welche dann auch an Unterobjekte weitervererbt werden (Beleuchtungsoptionen, Materialparameter, Animations-parameter, Rauminformationen). Zudem bietet der Szenegraph den Vorteil, dass die statischen Objekte zu einer Geometrie verschmolzen werden können und sich somit vorausberechnen lassen. Durch diesen Trick kann viel Rechenzeit bei statischen Szenen gewonnen werden. Zusammenfassend ergibt sich durch den Szenengraph eine einfache Möglichkeit komplette 3D-Szenen zu beschreiben und zu manipulieren. [6]

4. Grundlagen

Im folgenden Kapitel werden die Grundlagen der einzelnen Grafikbibliotheken erläutert. Zudem schließt das Kapitel mit einem Vergleich ab, anhand dessen entschieden wird welche Grafikbibliothek zum Einsatz kommt.

Mit dem Aufkommen von immer schnelleren GPU's und CPU's steht nun eine immense Rechenleistung zu Verfügung. Diese kann zur Visualisierung von Grafiken, 3D-Modellen und Animationen genutzt werden. Um die Grafikhardware zu abstrahieren gibt es unterschiedliche Grafikbibliotheken. Eine davon ist DirectX von Microsoft für die Windows Plattform, welche in unzähligen Softwareprojekten (hauptsächlich Spielen) eingesetzt wird. Mit diesen Bibliotheken ist es möglich über eine einheitliche Schnittstelle die Grafikkarte anzusteuern.

Innerhalb der Programmiersprache Java hat sich dabei hauptsächlich die Bibliothek OpenGL, aufgrund ihrer Plattformunabhängigkeit durchgesetzt. Für diese Bibliothek existieren drei Low-Level-API's, nämlich JOGL, LWJGL und Java 3D. Diese ermöglichen es mit Java auf die nativen Bibliotheken (OpenGL, DirectX) zuzugreifen. Dabei ist Java 3D ein Sonderfall, dazu später mehr. Im Diagramm 3 unten ist das Schichtenmodell für einen kleinen Auszug der Bibliotheken für Java gegeben. Die Low-Level-API's unterscheiden sich gegenüber den High-Level-API's in erster Line in der Modellorientierung. Die Low-Level-API's verwenden ein punktorientiertes Modell und die High-Level-API's ein objektorientiertes Modell. Das bedeutet in den Low-Level-API's wird alles direkt programmiert: wird beispielsweise irgendwo eine Farbe gesetzt so gilt diese solange bis eine neue Farbe gesetzt wird. 3D-Objekte werden dazu Polygon für Polygon aufgebaut. Bei den High-Level-API's hingegen wird alles weiter abstrahiert. Es werden Szenengraphen mit einer Unzahl von 3D-Objekten erstellt, welche dann von der API gezeichnet werden.

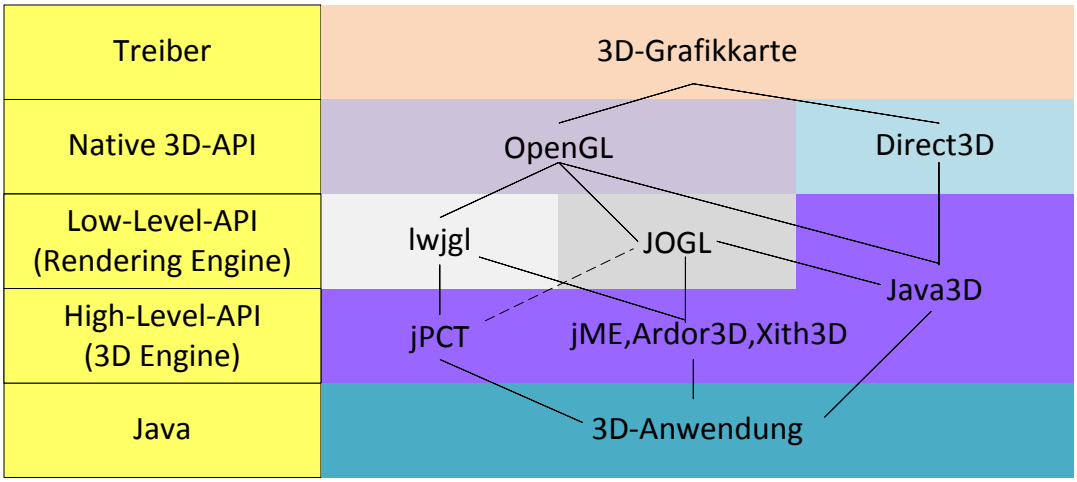

Diagramm 3: Zuordnung der Rendering und 3D-Engins als Schichtenmodell

In den folgenden Abschnitten wird auf OpenGL und DirectX eingegangen sowie diverse Bibliotheken für Java vorgestellt. Am Ende wird eine Bibliothek ermittelt, welche den folgenden Anforderungen (nach Prioritäten sortiert) am besten entspricht:

- Einbindbarkeit in Java Swing/AWT
- Unterstützung von aktuellen Grafikstandards (OpenGL 4.2, OpenGL ES 1.0, DirectX 11)
- Einfache definierte Schnittstellen
- Zukunftssicherheit, stetige Weiterentwicklung der Bibliothek
- Einfache Handhabung der API
- Einfaches Erzeugen eigener Modelle

Auf die tieferen Softwarestrukturen der einzelnen API's wird nicht eingegangen, da dies den Umfang dieser Arbeit sprengen würde. Das Ziel ist es die einzelnen API's zu beschreiben und am Ende die Unterschiede herauszustellen, anhand welcher entschieden werden kann ob diese für den Einsatz in diesem Projekt geeignet sind.

4.1. DirectX

DirectX ist die von Microsoft entwickelte API für multimediaintensive Anwendungen. DirectX ist in der Version 11.1 verfügbar (26.10.2012) und ist unter der MS-EULA Lizenz veröffentlicht.

Mit dieser Schnittstelle ist es möglich Grafiken in Programm einzubinden, Peripherieeingaben entgegen zu nehmen und Audioeffekte auszugeben. Jedoch ist diese API auf die Windows und Xbox Plattform beschränkt und bietet somit keine Plattformunabhängigkeit. Momentan werden die meisten aufwendigen Multimediaanwendungen in Windows mit DirectX entwickelt, da diese vollständig dokumentiert ist, viel Support bietet und eine stetige Weiterentwicklung erfährt.

Innerhalb von DirectX stehen viele Komponenten zu Verfügung, die wichtigste dabei ist Direct3D. Mithilfe dieser Komponente ist es möglich die Grafikkarte anzusprechen und Modelle darzustellen, zu manipulieren und zu beleuchten. Weiter gibt es noch unzählige Möglichkeiten zur Audioausgabe, Videoschnitt und eine breite Unterstützung von verschiedenen 3D-Formaten.

4.2. OpenGL

Für eine plattform-und programmiersprachenunabhängige API wurde von der Khronos Group die Open Game Library (OpenGL) in der Programmiersprache C entwickelt. OpenGL liegt momentan in der Version 4.3 vor (6.8.2012). OpenGL bildet das Gegenstück zu der Direct3D Komponente von DirectX.

Die OpenGL-API erlaubt es, wie bei DirectX, 2D- sowie 3D-Grafiken darzustellen und zu manipulieren. Dazu stehen im OpenGL-Standard ca. 250 Befehle zur Verfügung. Mit diesen Befehlen können Objekte verformt, verschoben, texturiert und beleuchtet werden. Durch den breiten Funktionsumfang an Zeichenfunktionen ist es möglich beliebige Objekte darzustellen und zu bewegen/animieren. Dabei ist OpenGL, wie Direct3D, als Zustandsmaschine umgesetzt, das heißt, Parameter werden immer beibehalten und nur explizit durch den Programmierer verändert. Somit können Parameter, wie z.B. die globalen Lichtquellen, einmalig für das komplette Rendering gesetzt werden.

Die Ansteuerung der Soundkarte oder von Peripheriegeräten ist nicht in OpenGL implementiert. Für Anwendungen, die eine Audioausgabe unterstützen, muss zum Beispiel die Open Audio Library (OpenAL) zusätzlich zu OpenGL eingebunden werden.

4.3. Rendering-Engins

Unter Rendering-Engins versteht man in diesem Bezug Bibliotheken, die die Funktionalität der Standard Bibliotheken (OpenGL, DirectX) direkt für Java zur Verfügung stellen. Somit kann innerhalb von Java die Bibliothek auf dieselbe Weise wie in nativen Code verwendet werden. Man nennt solche Bibliotheken dementsprechend Low-Level-API's, da diese keine weitere Abstraktion der eigentlichen Bibliothek vornehmen. Im folgenden Abschnitt werden die Java Bindings for OpenGL (JOGL) und Lightweight Java Game Library (LWJGL) vorgestellt.

4.3.1. Java Bindings for OpenGL

Java Bindings for OpenGL (JOGL) ermöglicht es innerhalb von Java auf 2D und 3D Ausgaben zurückzugreifen. Das Projekt wurde von Sun Microsystems und SGI als Open-Source-Projekt im Jahr 2003 ins Leben gerufen. Ein Hauptgrund für diese Entwicklung war es Java für Spieleentwickler attraktiver zu machen. Mit JOGL ist es möglich auf den vollen Funktionsumfang von OpenGL und dessen Erweiterungen zuzugreifen. JOGL unterstützt OpenGL bis zur Version 4.3, somit wird der aktuelle OpenGL Standard vollständig unterstützt. Dadurch kann die JOGL Engine auch auf Android-Geräten eingesetzt werden. OpenGL selbst liegt in der Version 1.1.1 (22.5.2008) vor. Jedoch steht die Veröffentlichung von JOGL Version 2.0 kurz bevor, diese ist schon als Release Candidate in der Version 2.0.2 und Revision 12 (25.6.2013) verfügbar.

Mittels JOGL können in speziellen Java Canvas 2D, oder Canvas 3D, Objekte gezeichnet werden (ein Canvas ist das elektronische Zeichenbrett, also der Darstellungsbereich auf dem Monitor). Dabei wird die komplette Steuerung genauso wie in OpenGL selbst umgesetzt. Rotation, Translation und Skalierung der Objekte werden direkt über Funktionen aufgerufen. Diese verändern dementsprechend immer die Model- oder Projektionsmatrix der Rotation/Translation. JOGL ist mit Swing/AWT Komponenten kombinierbar und lässt sich somit in jedes Swing/AWT-gestützte GUI einbinden.

4.3.2. Lightweight Java Game Library

Lightweight Java Game Library ist das eigentliche Gegenstück zu DirectX in der Java Welt. LWJGL verfügt, wie DirectX, über Komponenten zur Audioausgabe und für die Eingabe von Peripheriegeräten. Die API ist momentan in der Version 2.9.0 (21.4.2013) verfügbar und unterstützt OpenGL bis zu Version 4.1, OpenAL bis Version 1.1, OpenCL bis Version 1.0 und ist unter der BSD-Lizenz veröffentlicht.

Das Ziel von LWJGL ist es die Entwicklung von Spielen dadurch zu erleichtern, dass wichtige Bibliotheken zusammengefasst werden. So ist zum Beispiel die Einbindung von OpenAL, für die Audioausgabe, möglich.

Bei der Entwicklung von LWJGL wurde bewusst auf einige Funktionalitäten von OpenGL etc. verzichtet. So wurde GLU nur teilweise implementiert, einige andere Funktionen die nicht mit genug Performanz in Java ausgeführt werden können, wurden entfernt (glColor3fv). Die Bibliothek wurde nach dem Prinzip des Minimalismus entwickelt, das bedeutet die API wurde so klein wie möglich definiert. Durch dieses Prinzip ist eine einfache Portabilität möglich und die Übersichtlichkeit über die möglichen CodeKonstruktionen ist gegeben. [7]

4.4. 3D-Engins

3D-Engins bilden die nächste Abstraktionsstufe der beiden Low-Level-API's JOGL und LWJGL. Zu den sogenannten High-Level-API's gehört, auch der Sonderfall Java 3D. Die High-Level-API'S erweitern die Low-Level-API's noch um viele weitere Möglichkeiten, wie z.B. die einfache Darstellung von Szenengraphen, Kollisionskontrolle, Video- oder Audioeinbindung. Meist geschieht dies über Verwendung von weiteren Bibliotheken wie z.B. OpenAL für die Wiedergabe von Musik. Im den folgenden Abschnitten wird auf die oben erwähnten API's eingegangen. Theoretisch besteht auch die Möglichkeit API's aus anderen Programmiersprachen in Java zu verwenden, diese würde aber die Komplexität des Projektes stark erhöhen und den Einstieg für zukünftige Entwickler sehr stark erschweren. Alle folgenden API's setzen zum Managen von 3D-Objekten auf das zentrale Konzept des Szenengraphen-Modells auf, welches in Kapitel 3.5 Seite 20 kurz beschriebenen wird.

4.4.1. Java 3D

Java 3D wurde ursprünglich von Sun Microsystems, HP und SGI entwickelt und ist seit Mitte 2004 als Open Source freigegeben und wird seither von einer großen Community weiterentwickelt. Das Ziel der ersten Entwicklung war es eine pattformübergreifende und effiziente API zur Darstellung von Grafiken im Internet zu schaffen. Java 3D befindet sich momentan in der Version 1.5.2 (2008).

Java 3D ist ein Sonderfall unter den für Java verfügbaren Renderer-Engins. Die API greift direkt auf die native OpenGL Funktionalitäten zu und benutzt seit der Version 1.5 auch Teilaspekte aus JOGL. Zudem ist sie die einzige API, die den direkten Zugriff von der DirectX-API auf Direct3D ermöglicht. Des Weiteren abstrahiert Java 3D die Funktionen der nativen Bibliotheken ebenso weit wie die folgenden 3D-Engins. Im Schichtenmodell im Diagramm 3 auf Seite 22 ist gut zu erkennen wie Java 3D den Bereich der Low-Level-API und High-Level-API abdeckt.

Die API ist durchgängig konsistent nach OO-Richtlinien entwickelt und hat somit eine etwas komplizierte Struktur als API's, die näher an OpenGL oder LWJGL entwickelt wurden. Durch diese Strukturierung dauert es immer eine Zeitlang bis neue Hardwarefunktionen in die Bibliothek eingepflegt werden. Dadurch befindet sich der Entwicklungsstand dieser API immer ein wenig im Rückstand zu anderen. Vermutlich ist dies auch dem Qualitätsmanagement hinter dem Projekt geschuldet, da diese API im Java Community Process entwickelt wird und somit gleichwertige Kodier-Richtlinien wie die Programmiersprache Java selbst verfolgt. Die API ist im Prinzip für jegliche Art der Visualisierung ausgelegt. Dazu zählt beispielsweise die Darstellung von Objekten innerhalb einer Internetseite oder von komplexeren Simulationen.

Als Zusatzfunktionen unterstütz Java 3D die Kollisionserkennung, Multithread Szenengraph Strukturen, Verwalten von Eingabegeräten, 3D-Sound und es ermöglicht den Zugriff auf JOGL-, OpenGL- und Direct3D-Renderer (plattformabhängig). [8]

Innerhalb von Java 3D existieren drei verschiedene Renderer Möglichkeiten: der Retained, Compiled-Retained und Immediate Mode. Jeder der einzelnen Modi erlaubt einen gewissen Grad der Automatisierung beim Rendern. Der Immediate-Mode ist ein sehr hardwarenaher Modus. Punkte, Dreiecke etc. werden direkt in Listen gespeichert

und können jeder Zeit manipuliert werden. Im Retained-Mode wird dagegen ein Szenengraph verwendet, welcher ebenfalls voll manipulierbar ist. Im Compiled-Retained-Mode wird ebenfalls auf einen Szenengraph zurückgegriffen, jedoch lässt sich der Objektzugriffe (Zugriff auf Knoten) beschränken. Dadurch ist es Java 3D möglich Optimierungen durchzuführen und Objekte, die als nicht Veränderbar deklariert sind, vorauszuberechnen. Damit ergibt sich ein enormer Leistungsgewinn bei statischen Szenen. Jedoch wird der Objektverwaltungsaufwand für Programmierer im Hintergrund auch aufwendiger. Es muss vorher festgelegt werden, welches Objekt statisch ist und welches veränderbar. Bei Simulationen ist es auch oft der Fall, dass keine statischen Objekte existieren und somit die komplexere Objektverwaltung ein Mehraufwand für den Entwickler bedeutet. [9]

4.4.2. jMonkeyEngine

Die jMonkeyEngine (jME) wurde von Erlend Sogge Heggen, Skye Book, Kirill Vainer und Normen Hansen entwickelt. Die API liegt zurzeit in der Version 2.1 (9.3.1011) vor und ist unter der BSD-Lizenz veröffentlicht. Die API ermöglicht Die Nutzen der beiden Low-Level-API'S LWJGL und JOGL. Als Standard Renderer wird auf LWJGL gesetzt.

Als zentrales Konzept basiert die JMonkeyEngine wie alle andere High-Level-API's auf dem Szenengraphen-Modell. Jedoch wurde in jME das Konzept noch erweitert, im Gegensatz zu den anderen 3D-API's. In jME ist es möglich auch weitere Objekteigenschaften an den Szenengraphen anzuhängen, beispielsweise einen Geschwindigkeitsvektor oder die Masse des Objektes. Diese werden mit an den Szenengraphen gehängt und an jeden weiteren angehängten Knoten weitervererbt.

Als Zusatzfunktionen bringt jME eine eigene Physic-Engine mit, ein Partikelsystem, reflektierendes Wasser, Schattendarstellung, Pakete zur Netzwerkverwaltung, eine automatische Landschaftsgenerierung und eine Verwaltung von Eingabegeräten um nur einige zu nennen. Des Weiteren bringt die Engine ein eigenes SDK, basierend auf der NetBeans Plattform mit. Diese SDK ermöglicht es einfach mit der jME zu interagieren. So ist es z.B. möglich 3D-Modelle direkt zu manipulieren, Bewegungs-pfade zu erstellen, Animationen zu generieren und Terrain über einen Map-Editor zu

erzeugen. Das SDK ist auf OSGi aufgebaut und kann somit einfach um neue Funktionen ergänzt werden.

Anhand der Funktionalitäten ist zu erkennen, dass die jME komplett für die Verwendung zu Spieleentwicklung entwickelt wurde. Ein Nutzen als reines Visualisierungswerkzeug ist dennoch möglich, aber bei der Entwicklung muss man mit viel Overhead zurechtkommen. Jedoch steckt hinter dem jME Projekt eine sehr aktive Community, an die man sich bei Problemen wenden kann. Allerdings beschäftigt sich diese Community zum Großteil nur mit der Entwicklung von Spielen und bei komplexeren Fragen zur Umsetzung von Simulationen wird man vermutlich kaum Hilfe erwarten können. [10]

4.4.3. jPCT

Java Perspective Correct Texture mapping kurz jPCT ist eine weitere 3D-Engine zur Darstellung und Animation von Objekten. Da der Name nicht direkt auf den Verwendungszweck hindeutet, wird das Projekt vom Entwickler selbst nur noch „jPCT 3D engine – the free 3D solution for Java and Android“ genannt. Die API liegt in der Version 1.27 (24.07.2013) vor und ist ab der Java Version 1.4 voll kompatibel. Eine Verwendung unter älteren Java Versionen ist auch möglich, dabei kann aber nicht auf die Hardwarebeschleunigung zurückgegriffen werden Die API ist unter einer freien Lizenz verwendbar, der Source Code ist jedoch nicht offengelegt.

Im Grunde ist diese API dem Konzept von LWJGL gefolgt eine leichtgewichtige API zu schreiben. Die API ist somit einfach strukturiert und komplizierte Code-Konstruktionen werden vermieden. Wie LWJGL ist diese API im Prinzip für Spieleentwicklung ausgelegt, kann aber auch für beliebige Visualisierungen verwendet werden. Wie die Engine JOGL kann auch diese API für die Entwicklung von Android-Apps eingesetzt werden, da OpenGL ES 1.0 für Android unterstützt wird.

Zu den wichtigsten Zusatzfunktionalitäten zählt eine Kollisionserkennung mit verschiedenen Verfahren, Verwalten von diversen Eingabegeräten, Kompilieren der zu zeichnenden Objekte, um eine erhöhte Performance zu gewinnen (nachträgliches Manipulieren der Objekteeigenschaften ist dann nichtmehr möglich), eine einfache

Schattenberechnungsroutine und die Unterstützung von unbegrenzt vielen Lichtquellen.

Die jPCT-API kann mit drei verschiedenen Renderer verwendet werden. Zum einen dem Software-Renderer für Java Anwendungen ab der Version 1.1. Diese unterstützt jedoch nicht alle Funktionalitäten der Hardware, z.B. keine Unterstützung von multi-texturing. Der Software-Renderer ist zudem am einfachsten mit Swing/AWT zu kombinieren und ermöglicht es dem Programm auch auf Computer-/Android- und sonstigen Java Plattformen ohne Grafikkarte zu laufen. Die anderen beiden Renderer sind als Hardware-Renderer umgesetzt und unterstützen deshalb die volle Funktionalität der Hardware. Zum einen ist das der OpenGL-Renderer, welcher die Grafiken in einem nativen OpenGL-Fenster ausgibt. Das Verwenden eines nativen OpenGL-Fensters schließt eine Verwendung von Swing/AWT Elementen aus. Jedoch können innerhalb dieses nativen Fensters Swing-Objekte gezeichnet und verwendet werden. Als weiterer Renderer existiert der AWTGL-Renderer. Dieser basiert auf dem **AWTGLCanvas** aus der LWJGL-API. Das **AWTGLCanvas** ist voll in Swing/AWT GUI's Implementierungsfähig. Der AWTGLRenderer unterstützt von Haus aus Multithreading und die Verwendung von mehreren CPU Kernen. Bei dem OpenGL Renderer muss Multithreading explizit aktiviert werden. Durch dieses Aktivieren muss beim OpenGL Renderer sichergestellt werden, dass alle OpenGL-/Fenster-/Tastatur-/Mausbezogenen Code im Renderer-Thread ablaufen. Da dieses nicht immer von Haus aus gegeben ist, ist es standardmäßig deaktiviert. Die jPCT-API kann zudem mit JOGL(ab der Version 1.18) statt mit LWJGL verwendet werden. [11]

4.4.4. Ardor3D

Die Ardor3D Engine ist ein direkter Ableger der jME-API und ist in der Version 0.8 (01.11.2012) verfügbar. Ardor3D ist unter der Zlib-Lizenz veröffentlicht, somit ist der Source Code öffentlich verfügbar. Unterstützt werden alle gängigen OpenGL Versionen, Java wird erst ab der Version 1.6 unterstütz. Im Gegensatz zu der jME-API arbeitet Ardor3D standardmäßig mit JOGL. Ardor3D sieht sich selbst als Engine für Echtzeit-3D-Spiele und Echtzeit-Simulationen.

Unterstützt werden folgende Zusatzfunktionen, Verwalten von Eingabegeräten, Schattenberechnung, Partikelsystem, Multithreading, Kollisionserkennung und der Unterstützung von stereoskopischer Bildausgabe. Die Einbindung von Ardor3D in eine AWT/SWT GUI ist problemlos möglich.

Hinter Ardor3D steht ein aktives Entwicklerteam, jedoch fehlt es an einer breiten Community. Im Entwicklerforum können dennoch Fragen zu Problemen gestellt werden, welche auch sehr schnell direkt von den Entwicklern beantwortet werden. Beispiele zu Implementierungen finden sich zahlreich innerhalb der gegebenen Demo. Zum Großteil kann auch auf die jME-Demos und die Community zurückgegriffen werden.

4.4.5. Aviatrix3D

Die Aviatrix3D-Engine ist ebenfalls ein direkter Ableger von jME. Die Engine liegt aktuell in der Version 2.2 (13.9.2011) vor und benötigt mindeste Java 1.5. Veröffentlicht ist die Engine unter der LGPL-Lizenz und ist somit ein Open Source Projekt mit der Möglichkeit den Quellcode einzusehen. Unterstützt wird das Projekt von Ardor Labs.

Im Gegensatz zu den anderen Engins spezialisiert sich Aviatrix3D nicht auf die Unterstützung der Entwicklung von Computerspielen, sondern auf die Visualisierung von Daten im professionellen Bereich. Standardfunktionalitäten wie die Sound-Unterstützung sind somit, im Gegensatz zu den anderen API's, nicht Teil der Bibliothek. Auch beim Rendering Modus macht die Aviatrix3D-API Abstriche. Es existiert nur der Retained-Modus (Erklärung siehe 4.4.1 Java 3D, Seite 26). Es wird zudem nur OpenGL mit JOGL als Java-OpenGL-Wrapper unterstützt. Im Ganzen ist Aviatrix3D die leichtgewichtigste API, da viele für die Visualisierung unnötige Funktionen nicht implementiert sind. Der Entwickler selbst wirbt mit „Simple minimalistic unambiguous API design".

Eine Einbindung in AWT/SWT Anwendungen ist problemlos möglich. Die API selbst ist sehr gut und vollständig Dokumentiert und erste Programmiertutorials sind auch

vorhanden. Die Community hinter Aviatrix3D ist allerding sehr klein und ein aktives Supportforum auf der Seite des Entwicklers ist nicht zu finden.

Als Zusatzfunktionen bietet die API als die Möglichkeit direkt OpenGL-Befehle abzusetzen, was es erlaubt die API in gewissen Situationen zu Umgehen. Des Weiteren wird Multithreading und Multicores unterstütz. [12]

4.5. Vergleich der verschiedenen Java API's für grafische Visualisierung

Im folgenden Abschnitt werden die verschiedenen API's auf ihre Eignung für die Umsetzung des Projektes überprüft. Dabei werden hauptsächlich Probleme herausgestellt, welche gegen eine Verwendung der API sprechen und sich bei ersten Testimplementierungen nicht trivial lösen ließen.

Die jMonkey Engine 3.0 ist aktuell die am meisten genutzte Engine, wenn es um Visualisierung und Spieleentwicklung innerhalb von Java geht. Der Source Code von jME ist voll zugänglich und erweiterbar. Hinter jME steht eine große Community und es existiert ein großer Anzahl an Beispielen und Anleitungen. Zudem bietet jME eine breite Basis an integrierten Funktionalitäten, wie Schattenberechnung, Kollisionserkennung, Eingabegeräteunterstützung und vieles mehr. Die einzige Problematik bei der Arbeit mit jME in Kombination mit Swing ist die Thread-Sicherheit. Durch Manipulationen des Szenengraphen durch Swing-Komponenten kann der Render-Thread abstürzen. Deshalb muss ein zuverlässiges Thread Management Modell implementiert werden oder mit dem Aufruf von Callable sichergestellt werden, dass die Manipulationen des Szenengraphen zum richtigen Zeitpunkt durchgeführt werden. Diese Problematik lässt sich fast immer durch eine geschickte Implementierung lösen. Eine weitere unschöne Eigenschaft von jME ist der umständliche Objektaufbau von 3D-Modellen. Diese werden über mehrere einzelne eindimensionale Arrays aufgebaut, welche das Erstellen von Objekten aus dem Pun-/Pan-Format (oder anderen Formaten) unnötig erschwert. Zusätzlich gibt es ein weiteres unlösbares Problem in der jME Engine: es ist derzeit nicht möglich einzelnen Polygonen eigene Farben zuzuweisen. Dadurch ist ein Einfärben einzelner Polygone nur über den Einsatz von komplexen Shader-Technik oder einer dynamischen

Texturgenerierung möglich. Da diese Funktionalität aber benötigt wird, um die Stärke der Reflektion des Radarsignals zu visualisieren, ist der Einsatz von jME in diesem Projekt nicht möglich.

Als weitere mögliche Engins stehen Ardor3D API und Aviatrix3D zu Verfügung. Diese sind beide weiter spezialisierte Ableger von jMonkey und legen den Fokus nicht auf die reine Spieleentwicklung. Ardor3D legt den Fokus auf Spiele und Simulationen und verwendet dazu JOGL in der aktuellen Version. Aviatrix3D legt den Fokus auf Visualisierung und verwendet ebenfalls JOGL in der aktuellen Version. Eine große Problematik bei diesen beiden API's ist, dass viele Beispiele auf den offiziellen Entwicklerseiten nicht aktuell sind und somit ein Einstieg in diese API schwierig ist. Zudem stehen hinter beiden Projekten keine so großen Communities wie bei jME selbst. Auch sind die Problematiken die bei jME aufgetreten sind, hier nicht vollständig gelöst. Das Färben von Polygonen ist weiterhin nicht möglich.

Als weitere API existiert noch Java 3D, welche den Einsatz aller gängigen Visualisierungsbibliotheken vereinheitlicht. Der Einsatz von OpenGL, LWJGL und DirectX ist möglich. Früher wurde Java 3D direkt von Sun entwickelt, seit 2004 handelt es sich bei Java 3D um ein Open Source Projekt, wodurch die Weiterentwicklung ins Stocken geriet. Mittlerweile ist Java 3D ein Projekt des Java Community Process und wird stetig weiterentwickelt. Trotz dessen ist die aktuellste Version 1.5.2 aus dem Jahr 2008 und somit schon mehrere Jahre alt. Das Zeichnen von Objekten, deren Polygone eine unterschiedliche Farbe aufweisen, ist bei Java 3D möglich, jedoch muss zum Ändern der Farbe das komplette Objekt neu initialisiert werden, was zu Performanceproblemen bei schwacher Hardware und detaillierten Objekten führen kann. Die Erstellung von Objekten ist im Gegensatz zu jME ein wenig vereinfacht und erlaubt es Objekte zumindest teilweise nach dem OO-Prinzip zu erstellen.

Im Gegensatz zu Java 3D bietet jPCT die Möglichkeit, dass durch wenige Änderung von LWJGL auf OpenGL gewechselt werden kann. Dies ermöglicht ein einfaches Umstellen der zugrundliegenden Low-Level-API und erlaubt es immer mit der modernsten Technologie zu arbeiten. Java 3D bietet hier noch den Zugriff auf DirectX. Da DirectX zu einer Plattformabhängigkeit führt, wird dies nicht als Vorteil betrachtet. Des Weiteren ist jPCT eine sehr leichtgewichtige API. Dies bedeutet es wird auf viele Zusatzfunktionalität verzichtet, welche hauptsächlich in der Spielebranche benötigt

werden. Wichtige Funktionen wie Kollisionserkennung und Schattenwurf sind dennoch enthalten und erlauben es diese für zukünftige Anwendungsfälle zu verwenden, ohne es wie bei JOGL/LWJGL komplett selber Implementieren zu müssen. Ein großer Vorteil von jPCT ist, dass Polygone sehr einfach über den integrierten TextureManager mit einer beliebigen Farbe versehen werden können. Zusätzlich ist es in jPCT möglich 3D-Objekte über andere Objekte zu erstellen (SimpleVector). Trotz der Leichtgewichtigkeit der API sind einige für dieses Projekt unnötige Funktionalitäten enthalten, wie beispielsweise die Soundausgabe. Hinter jPCT steht eine nicht so große Community wie bei jME, jedoch sind in diesem Projekt sehr aktive Entwickler beteiligt. Bei Problemen wird einem meist direkt vom Entwickler geholfen und gefundene Fehler in der API werden schnell beseitigt oder zusätzliche Funktionalitäten ergänzt (sofern diese sinnvoll sind). Im Ganzen ist das Handhaben von jPCT durchweg einfach gestaltet und unlösbare Probleme, die andere API's besitzen, sind in den ersten Tests nicht aufgetreten. Jedoch sind, ähnlich wie mit jME, Multithreadingprobleme aufgetreten, welche aber schnell durch eine klare Thread-Trennung und Synchronisation beseitigt werden konnten.

Anstelle von jPCT ist es auch möglich die von allen High-Level-API's verwendeten Bibliotheken JOGL und LWJGL zu verwenden. Zu den Funktionen der einzelnen Bibliotheken kann hauptsächlich gesagt werden, dass die Low-Level-API JOGL und LWJGL den größten Freiheitsgrad in der Entwicklung erlauben. Man kann alles perfekt an das gewünscht Produkt ohne zu viel Overhead anpassen. Der Nachteil dabei ist, dass man alle Funktionalitäten welche von anderen API's mitgebracht wird, selbst implementieren muss. Dies bedeutet einem hohen Zeitaufwand bei der Entwicklung und beim Testen der Software, welcher bei einem Projekt wie diesem nicht zu bewältigen ist. Theoretisch wäre es möglich mit der Verwendung von JOGL/LWJGL eine höhere Performance zu erzielen und ein Einblick in den Source Code zu erhalten (bei jPCT ist der Quellcode nicht offengelegt). Positiv ist zudem, dass unabhängig von der High-Level-API freigestellt ist, ob man JOGL oder LWJGL einsetzt. Abhängig von der Architektur ist man dann aber an diese API gebunden. Die jPCT-API bietet diesen Vorteil ebenso. Durch einen reinen Austausch der LWJGL-Bibliothek durch die JOGL Bibliothek (und ergänzen einer weiteren Jar-Datei) ist ein Wechsel zwischen den beiden Low-Level-API's möglich. Da des Weiteren eine erhöhte Performance bei diesem Projekt keinen Vorteil bringt (es werden keine hochkomplexen Grafikszenen mit mehr als 1 Million Polygonen erstellt) und man auch ohne Einblick in den Source

Code mit der API arbeiten kann, wird davon abgesehen eine direkte Implementierung von JOGL oder LWJGL vor zu nehmen.

Abzuschließend ist zu sagen, dass jPCT am besten für diese Projekt geeignet ist und die erste Implementierung der Bibliothek mit dieser API durchgeführt wird. Denkbar ist auch der Einsatz von Java 3D. Bevorzugt wird jedoch jPCT, da durch die Leichtgewichtigkeit der jPCT-API die Entwicklung schneller voran geht. Dabei ist aber in der Architektur darauf zu achten, dass die Abhängigkeit von jPCT auf ein Package zu beschränken und nur über eine definierte Schnittstelle auf die Funktionalitäten der jPCT-API zuzugreifen. Somit bleibt die Möglichkeit erhalten dieses Package auszutauschen und später noch durch andere Implementierungen von anderen API's zu ersetzen.

5. Architektur des Bibliothek Modells

Im folgenden Abschnitt wird die Architektur der Bibliothek festgelegt und auf die umgesetzten Design Konzepte, sowie die allgemeinen Anforderungen an die Bibliothek eingegangen. Als Grundlage für diese Bibliothek dient die im vorherigen Kapitel ausgewählte jPCT-API. Diese soll möglichst entkoppelt eingebunden werden um eine einfache Erweiterung zu anderen API's zu ermöglichen ggfs. durch den Einsatz von OSGi.

5.1. Festlegung der Anforderungen

Zur Festlegung der Anforderungen wird unter anderem die Software **Visual Control** und **Visualizer UCAV** herangezogen. Diese zwei Softwareprodukte definieren den aktuell üblichen Einsatz von Visualisierungen von 3D Objekten innerhalb der Fachgruppe. Die aus den Softwareprodukten extrahierten Anforderungen werden in drei Kategorien eingeteilt. Funktionale Anforderungen welche zwingend von dem Endprodukt unterstützt werden müssen. Nichtfunktionale Anforderungen welche über die funktionalen Anforderungen hinaus erfüllt werden müssen. Sowie Wunschanforderungen, Anforderungen die hilfreich sind aber nicht für den Einsatz des Softwareprodukts notwendig.

Aus der Analyse der Software **Visual Control** ergeben sich folgende funktionale Anforderungen: Als grundlegende Funktion muss die Software die Möglichkeit bieten 3D-Modelle aus dem Pun-/Pan-Format einzulesen und zu visualisieren. Das Einlesen von 3D-Modellen soll dabei so gestaltet werden, dass in zukünftigen Versionen noch weitere Dateiformate als Grundlage für 3D-Modelle dienen können. Die Einlese-Routine soll dabei automatisch erkennen um was für ein Dateiformat es sich handelt und das entsprechende Objekt zum Erstellen eines 3D-Models laden. Zur Darstellung des gerenderten Bildes soll ein **JPanel** dienen, welches sich einfach in ein GUI einbinden lässt.

Innerhalb von **Visual Control** muss bis jetzt immer nur ein Objekt gleichzeitig geladen werden. Dies ist dem Umstand geschuldet, dass in den Simulationssoftwares des Instituts auch immer nur ein 3D-Model berechnet wird (welches aber durchaus aus

mehreren einzelnen Teilen bestehen kann). Für den Einsatz in anderen Projekten ist es sinnvoll das Laden von mehreren 3D-Modellen zu unterstützen. Zusätzlich zum Laden der Modelle, müssen diese auch manipulierbar sein, dass bedeutet das Rotieren um eine beliebige Rotationsachse, sowie das Verschieben der einzelnen Modelle muss möglich sein. Weiter gibt es in **Visual Control** die Möglichkeit das Objekt entweder nach den verschiedenen Materialeigenschaften einzufärben oder die Farbgebung nach dem Einfallswinkel des Radarsignales zu gestalten. Diese Funktion soll beibehalten werden und so gestaltet werden, dass die Farben für das Material und die Grundfarben für die Farbgebung nach Einfallswinkel frei wählbar sind. Die Perspektive der Kamera, welche den gerade dargestellten Bildausschnitt repräsentiert, soll ebenfalls frei konfigurierbar sein. Eine einfach umgesetzte Zoommöglichkeit soll ebenfalls bereitgestellt werden. Dazu soll es noch möglich sein ein 3D-Objekt zu animieren. Dabei soll Hauptsächlich das Schwenken der Messanlage um das Objekt simuliert werden. Dass bedeutet im Anwendungsfall von **Visual Control** findet immer eine Rotation um die Y-Achse oder X-Achse statt. In der neuen Software soll zum einen die Rotationsachse frei wählbar sein und zudem soll die Animation am eingestellten Start-Winkel beginnen und auch am eingestellten Stopp-Winkel aufhören. In der aktuellen **Visual Control** Version wird der Stopp-Winkel nie für die Animation verwendet, es entsteht immer eine unendliche Rotation.

In Abbildung 3 ist die Ansicht eines 3D-Objektes in der alten **Visual Control** Version zu sehen. Deutlich zu erkennen sind hier einige Schwachstellen in der Darstellung von 3D-Objekten mit einer hohen Panel-Anzahl (ca. 14.000) zu erkennen.

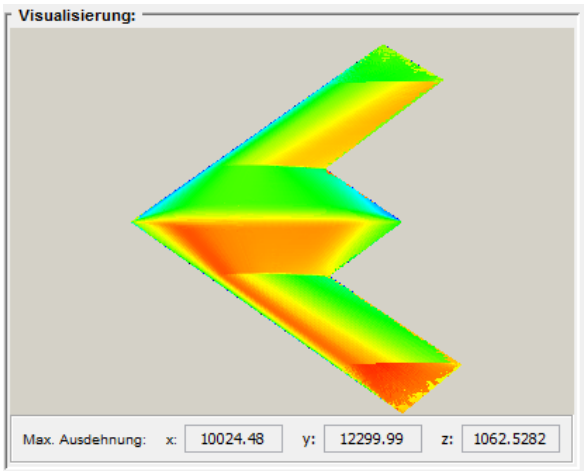

Abbildung 3: Darstellung eines UCAV in einer alten Visual Control Version

An den Flugzeugkanten entsteht durch ein fehlendes Antialiasing (Kantenglättung) Treppeneffekte. Dazu treten noch Fehler in der Überdeckungsberechnung auf, jeweils an dem oberen und unteren Ende des Flügels zu sehen. Diese Überdeckungsfehler sind auf einen falsch justierten Z-Buffer zurückzuführen. Der Z-Buffer wurde so konfiguriert, dass er nicht genug Tiefenstufen gibt und somit nah beieinanderliegende Polygone in die gleiche Tiefenstufe einsortiert werden. Die Software kann dann nichtmehr entscheiden welches der beiden Polygone vorne liegt und stellt diese durcheinander da. Diese Probleme müssen zwingend in der neuen Software behoben werden. Als weiteren Schwachpunkt ist anzumerken, dass die alte Software in ihrer Codestruktur sehr unübersichtlich ist. Beispielsweise existiert viel doppelter Code (ca. 40%), der durch eine ordentliche Restrukturierung hätte vermieden werden können. Auch existiert innerhalb der Software für ähnliche Funktionalitäten beinah identische Klasse und viele Komponenten der Programlogik sind in den grafischen Komponenten implementiert. Dadurch ist beispielsweise keine Umsetzung des Model-View-Presenter Entwurfsmuster möglich. Diese Schwachpunkte sollen zwingend bei der neuen Bibliothek vermieden werden.

Aus dem Programm **Visualizer UCAV** ergeben sich zudem noch folgende weitere funktionale Anforderungen. Zur Standarddarstellung eines Objektes, wird noch eine Darstellung der Flugtrajektorie in einem 3D-Koordinatensystem benötigt. Dieses Koordinatensystem soll frei drehbar (über Maus oder Tastatur) sein. In dieser Darstellung sollen zudem die in der CPACS-Datei definierten Radarstationen dargestellt werden. Dabei wäre es wünschenswert, dass die Möglichkeit besteht eine Radarstation farblich als Ausgewählt hervorzuheben. Zudem ist eine Synchronisation von mehreren Ansichten notwendig, diese dient dazu ein Objekt aus mehreren Perspektiven gleichzeitig darzustellen. Hierzu soll ein eigenes Panel implementiert werden, welches ein Layout für verschiedene Panelkonfigurationen (2 x 2 Panels, 1 x 3 Panels etc.) bereitstellt. Somit soll beispielsweise ermöglicht werden ein Objekt in der Frontalansicht, Seitenansicht und Draufsicht in drei nebeneinanderliegenden Panels zu visualisieren. Diese Ansichten sollen miteinander synchron über eine Schnittstelle steuerbar sein. Daraus ergibt sich für den Benutzer den Vorteil, dass er die Daten seine Anwendung nur an einer Stelle an die Bibliothek weitergeben muss.

Als Wunsch Anforderungen ergeben sich aus **Visual Control** und **Visualizer UCAV** eine Darstellung einer Farbskala, die angibt welcher Winkel mit welcher Farbe dargestellt

wird. Zudem Funktionen um ein Screenshot der aktuellen Szenen abzuspeichern und ein Video der aktuellen Animation zu erzeugen.

Als nichtfunktionale Anforderungen werden eine flüssige Performance der Animation und eine saubere Darstellung (hohes Antialiasing) gefordert. Des Weiteren soll die Software so konzipiert werden, dass eine Wartung und Erweiterung bzw. ein Austausch der Grafikbibliothek einfach möglich ist und auch von neu angelernten Java Entwicklern durchgeführt werden kann. Eine ausführliche Programmdokumentation sowie eine Anleitung zur Nutzung der entstehende API, mit Programmierbeispielen, sollen zu einem leichten Einstieg beitragen.

5.2. Entwerfen der Architektur

Aus den nicht funktionalen Anforderungen geht hervor, dass die Software eine flüssige Performance und eine saubere Darstellung ermöglichen soll. Diese Funktionen werden bereits durch die richtige Verwendung von jPCT erfüllt. Die richtige Verwendung bedeutet dabei, dass auf eine klare Thread-Trennung geachtet werden muss, um ein blockieren des Render-Threads zu verhindern. Andere Anforderungen, wie eine einfache Möglichkeit die Software zu erweitern und Bibliotheken auszutauschen, lassen sich über die Architektur umsetzen.

Die Architektur soll möglichst zentralisiert umgesetzt werden. Hierdurch ergibt sich für den Benutzer der Software eine einfache Übersicht der Möglichkeiten im Umgang mit der API. Es soll möglich sein über wenige Grundbefehle ein Fenster zu erzeugen, in dem Objekte geladen und manipuliert werden können. Natürlich soll es dem Benutzer aber auch möglich sein, von diesem zentralen Konzept unabhängig zu sein und eigene Implementierungen vorzunehmen. Damit diese eigenen Implementierungen auf einer kompatiblen Codebasis entstehen, müssen für diese Komponenten Schnittstellen und abstrakte Klassen bereitgestellt werden. Dabei bieten abstrakte Klassen den Vorteil, dass die grundlegende Implementierung von vielen Funktionen vorgegeben werden kann. Bei Schnittstellen hingegen werden nur die benötigten Funktionen vorgegeben. Die Implementierung der Funktionen bleibt dem Programmierer selbst überlassen. Natürlich ist auch bei abstrakten Klassen die Möglichkeit gegeben die Funktion zu überschreiben und nach eigenen Vorstellungen neu zu implementieren.

Bei der Entwicklung der Architektur und Package Struktur soll darauf geachtet werden, dass externe API's (jPCT) nicht über mehrere Paket Hinweg verwendet werden, also eine möglichst geringe zyklomatische Paket-Komplexität angestrebt wird. Daraus resultierend kann man dieses Paket später leichter vom Hauptprogramm abtrennen und durch ein anderes Paket ersetzen. Dazu später mehr im Kapitel Austauschbarkeit der Bibliotheken (Kapitel 5.2 d. Seite 45).

Als grundlegende Paketstruktur wird eine möglichst feine Gliederung gewählt, diese senkt jedoch die gesamte Übersicht über die API. Durch die feine Strukturierung ergibt sich aber eine klare Aufteilung von einzelnen Klassen die miteinander arbeiten. So gibt es ein Paket „model.data" in dem alle Klassen enthalten sind welche Informationen speichern, so z.B. eine Klasse **Motion** mit den Unterklassen **Movement** und **Rotation**. Diese Klassen stellen die aktuelle Rotation/Bewegung eines 3D-Modells da. Hierzu zählen auch eigene definierte **Exceptiones** und **Enumerations**, sowie diverse statische Klassen, die für mathematische Berechnungen verwendet werden. Des Weiteren existiert ein Paket „views" in dem alle Klassen zur Darstellung des gerenderten Bildes enthalten sind. Diese Klasse beinhaltet keine Logik und dient ausschließlich zur Visualisierung. Die Klassen die für die Ausführung der Geschäftslogik verantwortlich sind, befinden sich alle in dem Paket „model.modules" und teilt sich (Abbildung 4) in verschiedene Unterpakete auf.

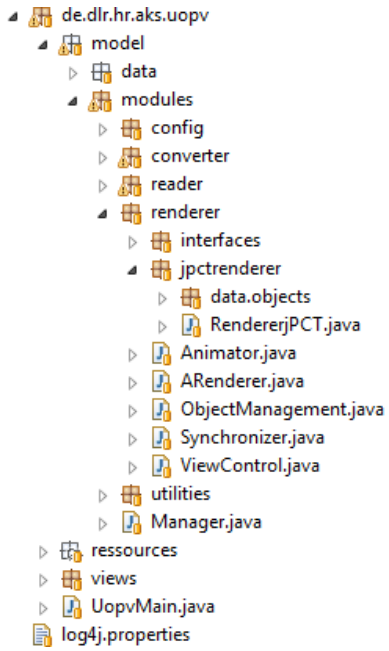

Abbildung 4: Auszug aus der Paket Struktur

Als zentraler Punkt der Software dient die Klasse **Manager**. Diese soll die direkte Interaktion zur Objektgenerierung des Nutzers auf einer Klasse beschränken. Innerhalb der Klasse stehen Fabrik-Methoden bereit, welche ein Panel zum Anzeigen von Visualisierung, zur Synchronisation von Objekten und ein Layout Panel zu Verfügung stellt. Auch sollen alle Klassen die zur Steuerung der Renderer-Vorgänge vom **Manager** erzeugt werden. Durch diese Zentralisierung kann ein Nutzer sofort erkennen welche Möglichkeiten ihm mit dieser Bibliothek gegeben sind.

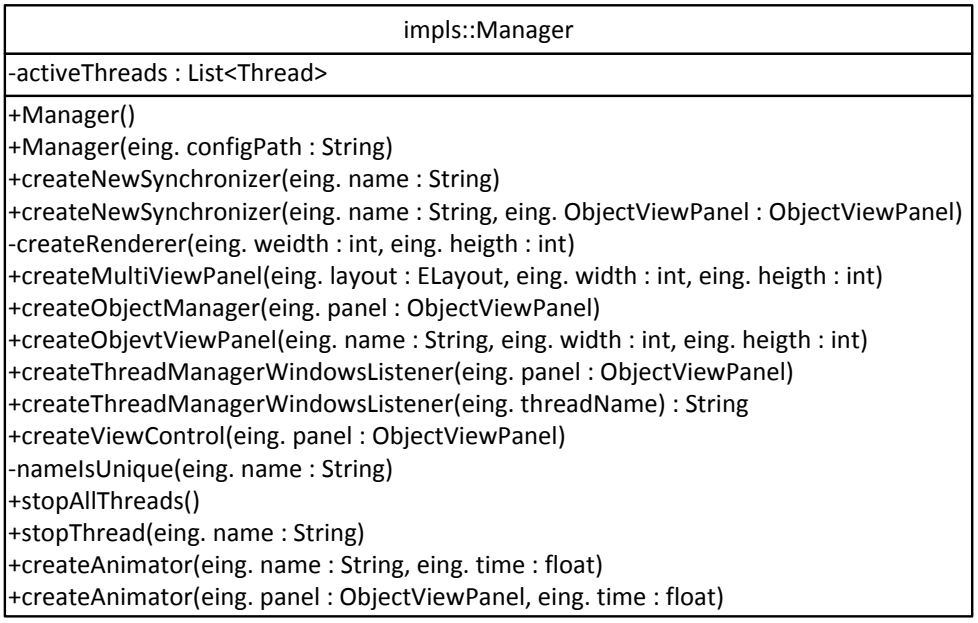

Diagramm 4: Manager, umgesetzt mit dem Fabrik-Entwurfsmuster, kümmert sich zusätzlich um die komplette Threadverwaltung.

Der **Manager** ermöglicht es vorkonfigurierte Objekte zu Visualisierung von 3D-Datensätzen zu laden. Die Konfiguration erfolgt dabei über eine zentrale Konfigurationsdatei, welche beim Initialisieren des Managers geladen wird. Diese Konfiguration lässt sich entweder im laufenden Betrieb über ein **Config-**Objekt verändern oder nach dem Erzeugen dieser Konfigurationsdatei mit einem Editor dauerhaft anpassen.

Im Gegensatz zu der üblichen Vorgehensweise ein Beispielprogramm direkt in die API zu integrieren wird dies bei dieser Bibliothek nicht der Fall sein. Dies hat den Grund, dass ein unnötiges Aufblähen der Datenmenge innerhalb der Bibliothek vermieden

werden soll. So müssten für Programmbeispiele eigene 3D-Datensätze und Texturen bereitgestellt werden und fest in die Bibliothek integriert werden. Es wird ein unabhängiges Projekt mit einer separaten Anleitung geben. Dieses Programm wird ersten Programmierbeispielen zur Verwendung der verschiedenen Layouts, Synchronisation der Ansichten, Steuerung von Lichtquellen und das Verwalten von Texturen beinhalten.

In den folgenden Unterabschnitten wird die Struktur der einzelnen Hauptkomponenten der Bibliothek herausgearbeitet. Dazu zählen das Laden der 3D-Modell, die darstellenden Komponenten, sowie die Schnittstellen und das Näherungsverfahren zur Darstellung des Einfallswinkels von Radarsignalen. Des Weiteren wird in einem Abschnitt noch kurz auf die Austauschbarkeit der jPCT Bibliothek eingegangen.

5.2.1. Laden von verschiedenen 3D-Modellen

Beim Laden von 3D-Modellen soll im aktuellen Stand der Entwicklung nur das institutseigene Pun-/Pan-Format als Grundlage dienen. Da die Software aber in Zukunft auch für weitere Formate kompatibel gehalten werden soll, reicht es nicht eine Klasse zu schreiben welche alle benötigten Daten aus dem Pun-/Pan-Format liest. Es muss eine Komponente entwickelt werden, welche es möglich macht verschiedene Einlese-Methoden bereitzustellen und diese in zukünftigen Versionen zu erweitern.

Um eine solche Kompatibilität und leichte Erweiterbarkeit zu gewährleisten wird mit dem Fabrik-Entwurfsmuster, kombiniert mit Reflektionen, gearbeitet. Das Fabrik-Entwurfsmuster zählt zu den Erzeugermustern und erzeugt ein Objekt, ähnlich einem direktem Konstruktor Aufrufes. Innerhalb der Fabrik lässt sich dabei vom Entwickler festlegen wie ein Objekt erzeugt werden soll. In diesem Fall geschieht das über das Reflektion-Konzept.

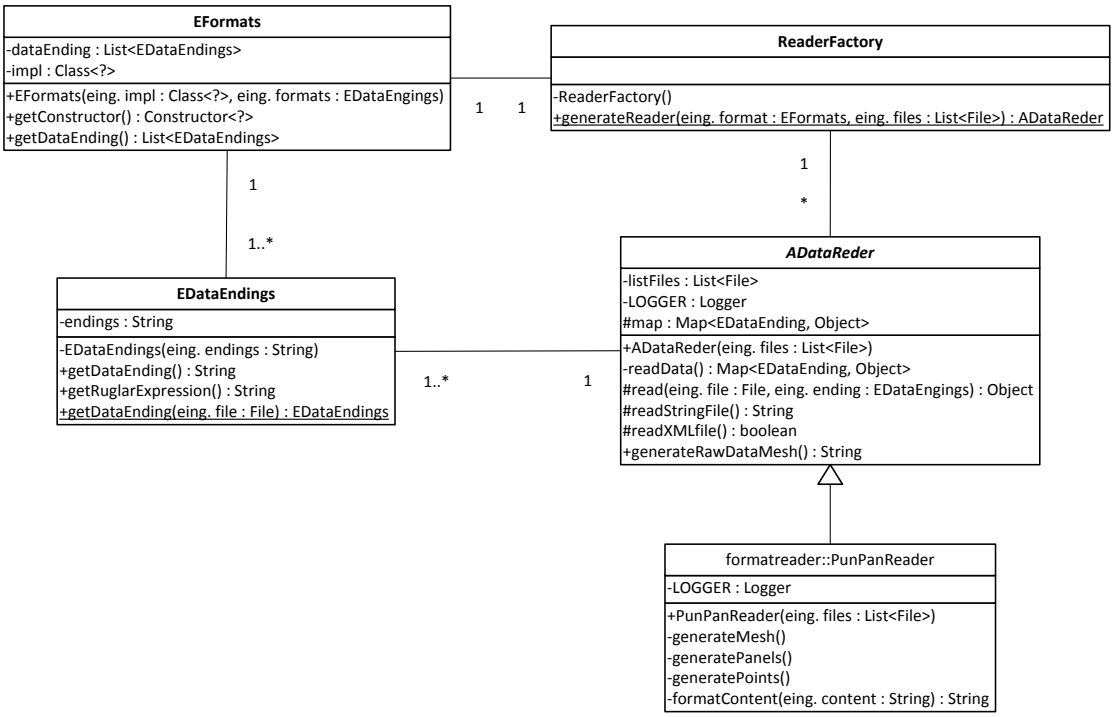

Diagramm 5: UML-Klassendiagramm zur ReaderFactory

Die Reflektion ist eine besondere Anweisung in Java, die es erlaubt auf die Metadaten einer Klasse zuzugreifen. Es erlaubt das Modifizieren von Klassen und Objekten im Speicher der JVM. Meist wird dieses Konzept eingesetzt um Debugger oder ähnliche Programm zu entwickeln. In diesem Fall wird die Reflektion dazu eingesetzt den Konstruktor einer Klasse zu ermitteln. Dazu wird in der Enumeration **EFormat** für jeden Reader eine Enumeration angelegt, welche die Klassennamen und die unterstützen Dateiformate enthält. Wird nun die Methode *generatReader(EFormat format, List<File> files)* aufgerufen, holt sich die Methode über **EFormat** den Klassennamen und ermittelt den Konstruktor dieser Klasse und erzeugt anschließend das entsprechende Objekt. Im Anhang II.a befindet sich ein Sequenzdiagramm, welches das Erzeugen eines **PunPanReader** darstellt. Durch dieses Konzept kann nun jederzeit einfach ein weiterer Reader implementiert werden. Dieser wird von **ADataReader** abgeleitet und in **EFormat** eingetragen. Danach kann dieser jederzeit über die **ReaderFactory** abgerufen werden. Der Entwickler muss dann selbst nur noch die Entscheidung treffen welchen Reader er verwenden will. Mit diesem Konzept kann auch auf eine einfache Weise eine automatische Erkennung des richtigen Readers umgesetzt werden.

5.2.2. Entwurf des Renderer

Die erforderlichen Komponenten für die Darstellung gliedern sich in zwei Bereiche. Zum einen die View-Komponenten und zum anderen die Geschäftslogik im Hintergrund. Die View-Komponenten dienen ausschließlich zur Darstellung. Interaktionen zum Steuern des Render-Vorgangs sollen über spezielle Steuerungs-klassen ermöglicht werden. Mit diesem Vorgehen wird eine klare Trennung von Visualisierung und Logik erreicht. Welches es ermöglicht die API in eine Software einzubinden welche nach dem Model-View-Presenter-Entwurfsmuster (oder ähnlichen) aufgebaut ist.

Die Hauptkomponente ist dabei die abstrakte Klasse **ARenderer**. Die Klasse **RenderjPCT** ist von der Klasse **ARenderer** abgeleitet. Durch die Kombination von dem Konzept einer abstrakten Klasse und Schnittstellen können Methoden bereits im **ARenderer** ausformuliert werden, um einen Grundablauf des Render-Vorgangs zu definieren. So wird in **ARenderer** die „Rendering Loop" erzeugt, die das anzuzeigende Bild aktualisiert. Dazu werden noch weitere abstrakte Methoden definiert, wie die geschützte abstrakte Methode *update()* und *rendern().* Durch den Einsatz der Schnittstelle können die Methoden definiert werden, welche später öffentlich zugänglich sind. Diese Schnittstelle ermöglicht somit die Festlegung einer einheitlichen Kommunikation mit der Implementierten Renderer-Klasse. Zu der detaillierten Implementierung später mehr.

Die Konfiguration der einzelnen Komponenten wird über eine Konfigurationsklasse **Config** gesteuert. Mit dieser Klasse können neue Konfigurationen aus XML-Dateien eingelesen werden. Diese müssen nach der in der Anleitung stehenden Definition aufgebaut sein. Es können jedoch auch direkt Werte gesetzt werden, ohne eine XML-Datei zu laden. Da es sich um eine Bibliothek handelt, ist es dem Benutzer freigestellt eine eigene XML-Datei zur Konfiguration zu verwenden. Alle Renderer-Komponenten greifen auf diese **Config**-Klasse zu und laden alle neu gesetzten Einstellungen dynamisch nach. Dies erfolgt mit der Hilfe des Observer-Entwurfsmusters. Jeder Renderer wird beim Erstellen über die Fabrikmethode als Observer der Konfiguration eingetragen. Sobald nun eine Änderung in der Konfiguration stattfindet, werden alle Observer benachrichtigt und sind in der Lage die Einstellungen zu aktualisieren.

Zusätzlich gibt es zu jeder Rendering API noch diverse „statische 3D-Objekte“. Dazu zählt das Koordinatensystem welches das Schnittstelle *ICoordinateSystem* implementiert. Diese Schnittstelle stellt alle Funktionen bereit die zur Konfiguration und zeichnen des Koordinatensystems notwendig sind. Zudem existiert eine Klasse welche eine 3D-Modell einer statischen **RadarStation** erzeugt. Selbstverständlich ist es aber auch möglich eine Radarstation aus einem Pun-/Pan-Modell zu laden.

Zur Darstellung von 3D-Objekten und deren Animationen wird ein **JPanels** als Basis eingesetzt. Dieses **JPanel** wird mit einem **Canvas**-Objekt zum Zeichnen der 3D-Objekte gefüllt. Mit dem **JPanel** ist eine Einbindung in jedes Swing GUI möglich. Das **ObjectViewPanel**, welches von **JPanel** erbt, wird über die zentrale **Manager** Klasse erzeugt. Wieder wird hierbei auf das Fabrik-Entwurfsmuster zurückgegriffen, welches eine definierte Objekterzeugung über einen indirekten Aufruf ermöglicht. Dadurch wird bei der Objekterzeugung gewährleistet, dass immer ein richtig konfigurierter Renderer für das jeweilige **ObjectViewPanel** initialisiert wird. Dazu zählt beispielsweise die Anmeldung als Observer bei der Konfiguration. Allgemein wird davon ausgegangen, dass immer nur eine Art von Renderer verwendet wird, es ist nicht vorgesehen mehrere verschiedenen Rendering API’s parallel laufen zu lassen. Über das **ObjectViewPanel** lässt sich dann eine Darstellung des Flugpfades sowie eine Visualisierung des 3D-Objektes umsetzen.

Die Steuerung des Render-Vorgangs erfolgt nicht über Aufrufe im **ObjevtViewPanels.** Hierfür existieren die Klassen **ObjectManagement**, **ViewControl** und **Synchronizer.** Die Klasse **ObjectManagement** dient zur Verwaltung der Objekte, dazu zählen auch Lichter und Linien. Die Klasse **ViewControl** dient zur Steuerung der Kamera und besonderen Kommandos, wie das Erzeugen von Screenshots. Die **Synchronizer**-Klasse unterstützt die gleichen Funktionen wie die **ObjectManagement**-Klasse setzt diese nur anders um. Die ersten beiden Klassen reichen die Aufrufe prinzipiell nur durch, der **Synchronizer** gewährleistet dagegen, dass alle Fenster die dem **Synchronizer** zugeordnet sind aktualisiert werden. Dazu ist es Notwendig zu gewährleisten, dass beim Hinzufügen von Objekten dessen ID auch korrekt zurückgegeben wird. Notwendig ist dies, da nicht gewährleistet werden kann, dass jeder Rendere immer dieselbe ID beim Hinzufügen von Objekten zurückgibt. Um dies zu ermöglichen wird im **Synchronizer** eine Zuweisungsliste geführt (gleiches gilt für Lichtquellen-ID’s und Linien-ID’s).

5.2.3. Schnittstellenbeschreibungen der Bibliothek

In dieser Bibliothek gibt es drei Hauptschnittstellen. Eine interne Schnittstelle *IARenderer* welche dazu genutzt wird die jeweilige Rendering API anzusprechen. Sowie drei Schnittstelle die für den Benutzer der API zur Interaktion mit selbiger gedacht sind. Jede dieser Schnittstellen wird von dem im vorherigen Kapitel beschriebenen zugehörigen Klassen implementiert (**ObjectManagement**, **ViewControl**, **Synchronizer**). Folgend die **ObjectManagement** Klasse welche die Schnittstelle *IObjectManagement* implementiert. Die Klassendiagramme zu den anderen Klassen finden sich im Anhang.

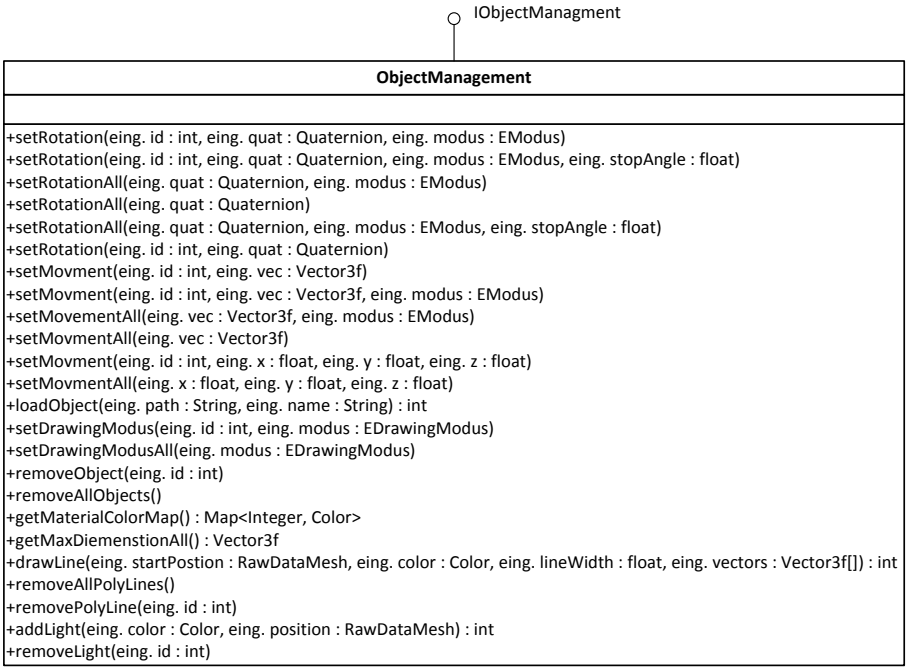

Diagramm 6: ObjectManagement-Klasse welche die Schnittstelle IObjectManagement implementiert

Mit dieser Klasse ist es dem Benutzer möglich alle Objekte die über den Manager erzeugt werden auf eine definierte Weise zu steuern. Die Steuerungsobjekte welche diese Schnittstellen implementieren werden von der **Manager**-Klasse erstellt. Dem Nutzer der Bibliothek steht es frei, ein eigenes Objekt diese Schnittstelle implementieren zu lassen. Die jeweiligen zuständigen Renderer kann er sich über die Methode *getRenderer(name)* von der Manager-Klasse liefern lassen.

5.2.4. Austauschbarkeit der Bibliothek

Durch die Schnittstelle *IARenderer* und die Klasse **ARenderer** wird gewährleistet, dass bei einer weiteren Implementierung einer weiteren Rendering-API diese ohne große Umstände an den aktuellen Programmablauf angepasst werden kann. Für die Benutzung der Bibliothek ergeben sich keine Änderungen da die Kommunikation mit der Bibliothek weiterhin über die Schnittstellen *IObjectManagement, IViewControl* und die **Manager**-Klasse abläuft.

Theoretisch ist es an dieser Stelle möglich eine voll modularisierte Architektur zu wählen, die mittels OSGi verwaltet wird. Dies würde zu einem Hauptmodul, mehreren Mathemodulen sowie diverser unabhängigen Renderer-Modulen führen. Da aber während der Arbeit aktuelle nur ein Renderer implementiert wird und dieser nur unter äußersten Umständen durch einen anderen wieder ersetzt wird, wird in diesem Fall auf OSGi verzichtet. Da der Vorteil von OSGi kommt erst bei Projekten mit einer großen Anzahl von Modulen zum Tragen. Gute Beispiele für solche Projekte ist Eclipse, dessen Architektur basiert auf OSGi und ermöglicht ein einfaches Management von Erweiterungen, wie sie auch in diesem Projekt zum Einsatz kommen (EGit-Plug-In, SonarQube-Plug-In etc.). Jedoch wurden alle Vorbereitungen getroffen um OSGi schnell umsetzen zu können. Dazu zählt unterwandern, dass Klassen in Paketen nur über definierte Schnittstellen angesprochen werden.

5.2.5. Näherungsverfahren zur Darstellung von Radarsignalen

Zur Darstellung des Beitrags den ein einzelnen Paneel zur gesamt Reflexion beiträgt ist eine Näherungsverfahren notwendig. Ein kurzer Umriss der Grundlagen zu Radarsignalen ist im Kapitel 3.2 Seite 15 zu finden. Kurz gesagt, die Stärke des reflektierten Radarsignales ist stark von dem Einfallswinkel abhängig. Es verhält sich näherungsweise wie einen Lichtstrahl der auf einen Spiegel trifft. Der Spiegel ist im Fall von Radarsignalen (Elektromagnetischen Wellen) ein leitfähiges Material. Wie in der Optik kann man nun für das Radarsignal die Annahme treffen, dass bei einem Einfallswinkel von 90° das Signal maximal zum Sender reflektiert wird. Bei allen anderen Winkeln entsteht eine Streuung des Signales. Andere Effekte, wie z.B. Dopplerreflexionen können bei dieser Näherung vernachlässigt werden. Die Näherung

soll nur ein erstes Indiz dafür liefern, welcher Teil des Objektes einen hohen RCS-Wert liefert. Dieser Wert ist zudem noch stark von der Größe des Paneels abhängig, welcher bei dieser Berechnung ebenfalls vernachlässig wird.

Die jPCT-API erlaubt es einzelne Paneele eines 3D-Modells anzusprechen und ermöglicht eine automatische Berechnung der Normale eines Paneels abhängig von der aktuellen Position und Rotation. Mit dieser Normale und der Blickrichtung der Radarstation, kann der Einfallswinkel berechnet werden.

$$\cos\lambda = \frac{\overrightarrow{a}\,\overrightarrow{b}}{\left|\overrightarrow{a}\right|\left|\overrightarrow{b}\right|}$$

Nach λ umgestellt erhält man nun den Winkel. Ist dieser Winkel 0°, dann liegen die Vektoren parallel zueinander. Dies bedeutet, dass der Einfallswinkel 90° Beträgt. Wird nun der sin anstelle des $\cos$ verwendet, erhalten wir direkt die 90°. Die Paneele werden dann abhängig von ihrer Winkelstellung zur Radarstation verschieden eingefärbt (Standardmäßig: 0° Schwarz, 45° Grün und 90° Rot). Die Winkel die zwischen Standardfarben liegen werden linear interpoliert. Der Interpolationsfaktor lässt sich über die Konfiguration ändern, ebenso wie die Standardfarben.

6. Implementierung der Bibliothek

Der folgende Abschnitt behandelt die endgültige Implementierung der Bibliothek. Dabei wird nur auf die Kernkomponenten eingegangen. Zudem werden verschiedene Probleme die während der Implementierung aufgetreten sind, erörtert und deren Lösung geschildert.

6.1. Implementierung des jPCTRenderer

Wie im Kapitel 5.2 beschrieben, baut der eigentliche Renderer auf einer abstrakten Klasse **ARender** auf. Diese Klasse dient zur Steuerung des Render-Threads und implementiert die Schnittstellen *IARenderer, IConfigObserver.*

Zuerst zu der Klasse **ARenderer**. Diese dient als Ausgangsbasis für alle Renderer, die zukünftig implementiert werden. Da der Rendering-Thread unabhängig vom GUI-Thread sein muss, wird die Schnittstelle *Runnable* implementiert. Klassen, welche diese Schnittstelle implementieren lassen sich als eigenständiger Thread starten. Zusätzlich implementiert diese Klasse auch die Schnittstellen *IARenderer* und *IConfigObserver*. Die **ARenderer-**Klasse kümmert sich somit um den allgemeinen Ablauf des Render-Vorgangs.

Als nächstes zur eigentlichen Implementierung des Renderers. Der **JPCTRenderer** erbt von der Klasse **ARenderer** und implementiert alle abstrakten Methoden aus der Elternklasse und den Schnittstellen. Die meisten Funktionen dienen dazu den Szenengraphen zu verändern. Beispielsweise löscht die Methode *resetObject(obj)* Objekte vom Szenengraphen. Die abstrakte Methode *initalize() dient z*um Initialisieren des Renderers, *update()* übernimmt das Abarbeiten der Bewegungsqueue und Einfärben von Objekten. Die eigentliche Darstellung des Bildes wird von der Methode *rendern()* übernommen. Folgend das Klassendiagramm zur Implementierung des Renderers.

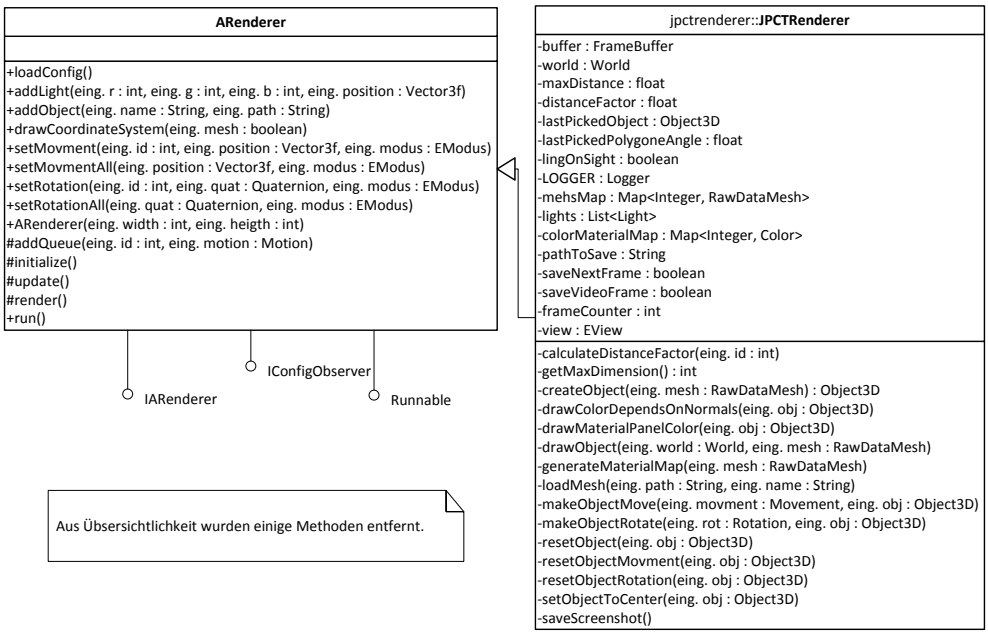

Diagramm 7: Klassendiagramm des Renderers in der ersten Version

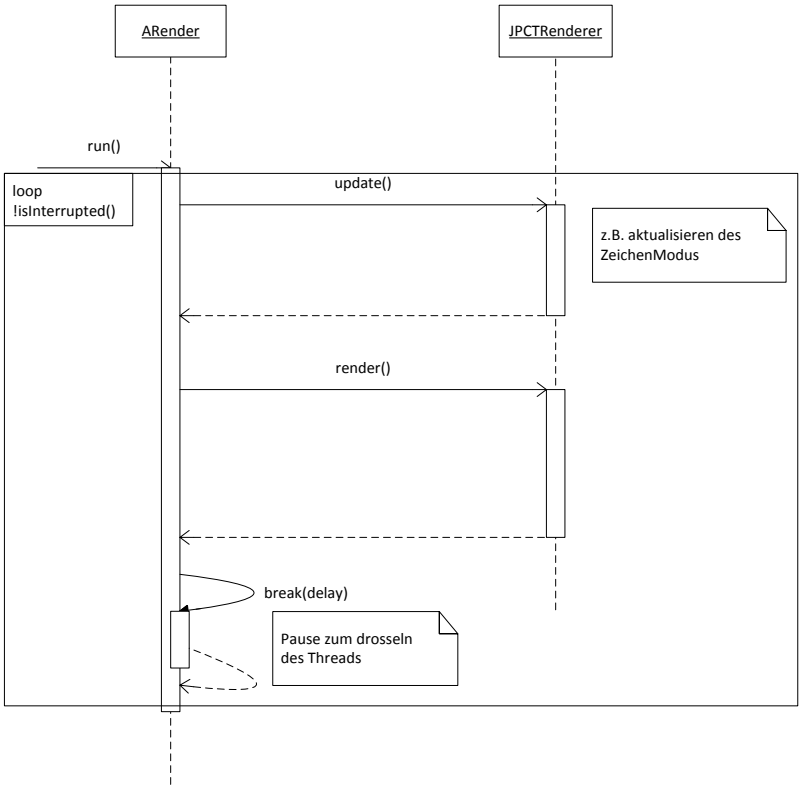

Diagramm 8: Steuerung des Renderers: Prozess wird durch einen *interrupt()* beendet.

Im oberen Sequenzdiagramm ist der Ablauf der *run()*-Methode beschrieben, welche sich um die Aktualisierung des Szenengraphen kümmert und diesen dann auf dem Bildschirm ausgibt. In der ersten Implementierung kam es zu starken Leistungseinbrüchen des Gesamtsystems beim Initialisieren von mehreren Render-Klassen. Zur Verringerung der Leistungsaufnahme des Render-Thread wurde dieser um die Methode *break(delay)* ergänzt.

Mit der *break(delay)*-Methode wurde die Ausführung auf maximal 40 Zyklen pro Sekunde beschränkt. Am Beginn des Schleifendurchlaufs wird die Startzeit gespeichert und am Ende mit der Laufzeit verglichen. Sobald der Schleifendurchlauf eine definierte Zeit unterschreitet, wird die Methode break(delay) aufgerufen. Diese Methode hält den Thread für die in *delay* definierte Zeit an.

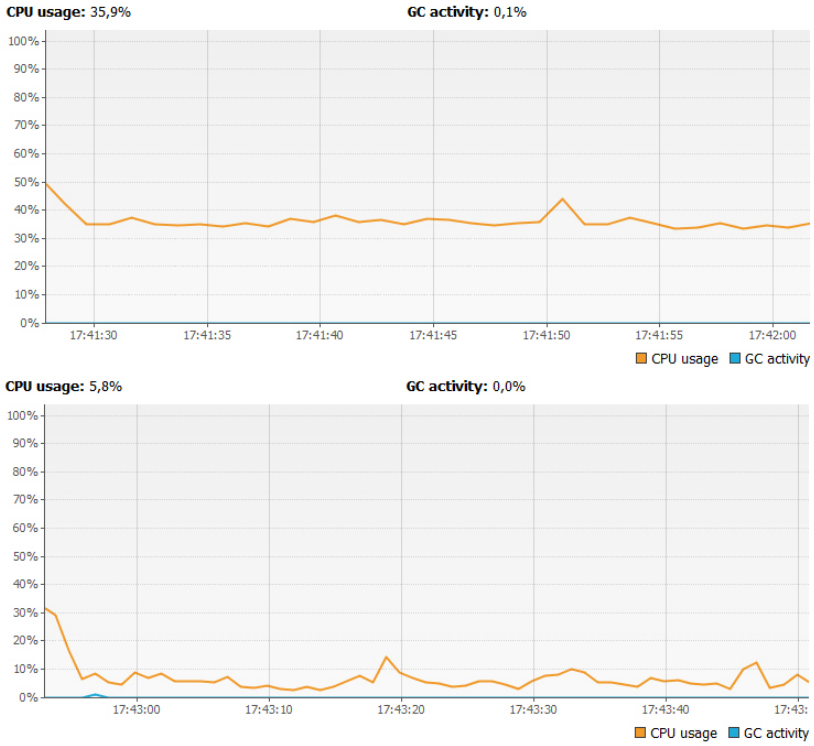

Diagramm 9: Mit VisualVM gemessener Auslastungsunterschied: erster Implementierung oben, neue Implementierung unten. (Differenz im Mittel ca. 20%)

Die Schleifendurchläufe entsprechen grob der Anzahl an Bildern die ausgegeben werden („Frames Per Second" (FPS)). Das menschliche Auge ist nicht dazu in der Lage mehr als ca. 30 FPS wahrzunehmen, weshalb eine Begrenzung auf 40 FPS eine ausreichend flüssige Darstellung ermöglicht. Bei einem ersten Test hat sich gezeigt,

dass durch die Begrenzung der Schleifenzyklen die CPU-Auslastung im Mittel um 20%, bei einem extrem detaillierten Objekt (DLR-F17) gesunken ist. Diese Begrenzung lässt sich über die **Config**-Klasse heben oder senken. So ist es möglich Objekte, wie z.B. den DLR-F17 auch nur mit 20 FPS anzeigen zu lassen.

Auf einzelne Funktionen der Render-Klasse wird in den folgenden Kapiteln noch näher eingegangen. Eine ausführliche Dokumentation der einzelnen Methoden ist in der Dokumentation zu finden.

6.2. Implementierung der Bewegungs-/Animationssteuerung

Im Zuge der Einführung der Queue wurden Verschiebungen und Rotationen in eigene Objekte ausgelagert. Es wird dabei zwischen Verschiebung (**Translation**) und Rotationen (**Rotation**) unterschieden, siehe Diagramm 10. Zusammengefasst werden beide Klassen unter der Elternklasse **Motion** (Bewegung)**.** Diese Klasse beinhaltet eine Variable für einen Vektor und einen Modus. Bei einer Rotation repräsentiert der Vektor die Rotationsachse, bei einer Verschiebung den Richtungsvektor. Abhängig vom Modus wird eine Rotation oder Verschiebung dann absolut oder relativ umgesetzt. Erzeugt wird die Rotationen/Verschiebung in den Methoden *setTranslation(…)* oder *setRotation(…).* Diese erzeugen jeweils das passende Objekt und hängen es an die Warteschlange an.

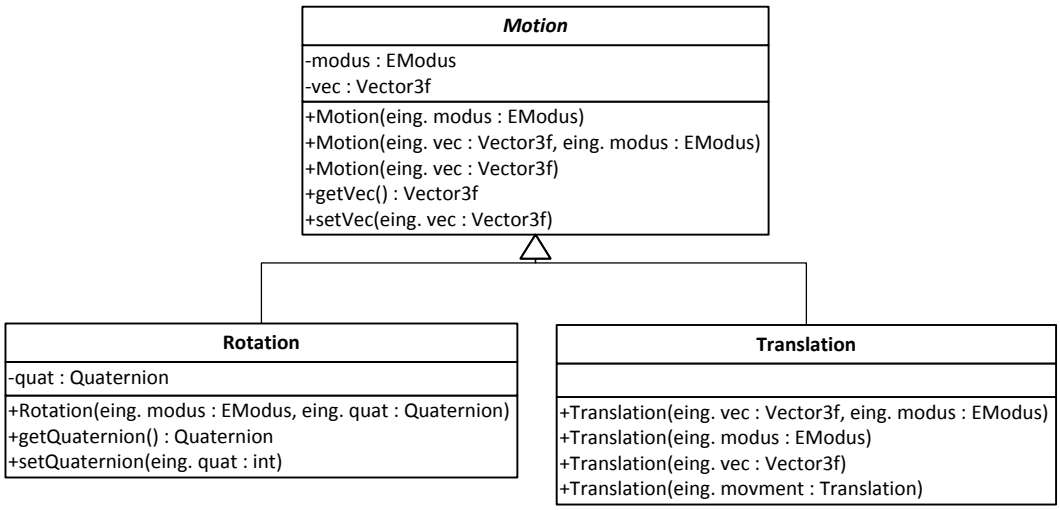

Diagramm 10: Klassen welche zur Repräsentation von Bewegungen verwendet werden

Der Renderer ruft bei jedem Durchgang die *update()*-Methode auf. In dieser Methode wird ein Element aus der Queue genommen und die jeweilige Bewegungen/Rotation wird ausgeführt (für jedes Element). Durch eine hohe Anzahl an Änderungen pro Sekunde konnten nun sehr lange Queues entstehen. Der Renderer war dann nicht mehr in der Lage dazu die Warteschlange schnell abzuarbeiten. Das zu bewegende Objekt reagiert folglich verzögert in der Visualisierung (die Bewegung erfolgt Zeitversetzt). Naheliegend war es nun die Geschwindigkeit der Abarbeitung zu erhöhen. Dieses sorgt wieder für eine schnellere Bewegung. Dabei ist aber der Effekt aufgetreten, dass die Bewegungen ruckartig ausgeführt wurden. Daraufhin wurde das Konzept der Queue wieder verworfen. Es war nicht möglich eine schnelle und flüssige Bewegung mit dieser Implementierung umzusetzen. Um nun doch die Threadsicherheit gewährleisten zu können wird das Java Konstrukt *synchronized* eingesetzt. Bewegungen werden wieder direkt am Szenengraphen durchgeführt und der Szenengraph wird mit *synchronized* davor geschützt das mehrere Threads gleichzeitig auf diesen zugreifen. Dabei muss aber gewährleistet sein, dass ein Thread sich nicht dauerhaft in diesem kritischen Abschnitt befindet, ansonsten kann der Render-Thread nie auf das Objekt zugreifen und es entsteht wieder eine Blockierung. Das Diagramm 7 unterscheidet sich zur finalen Version nur dadurch, dass die Methode *addQueue()* entfernt wurde.

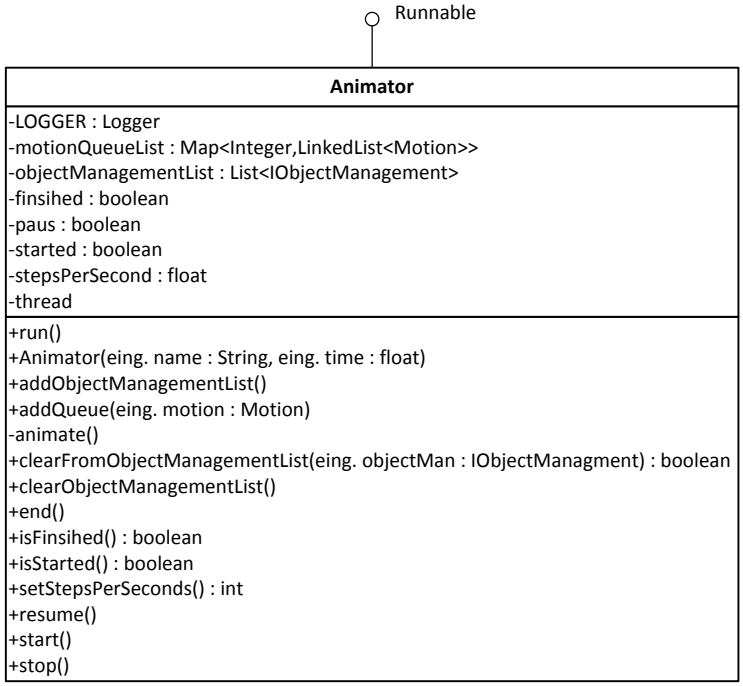

Diagramm 11: Animator-Klasse, führt eine Animation anhand der gegeben Bewegungen aus.

Animationen werden mit der Hilfsklasse **Animator** ausgeführt. Diese Klasse wird entweder vom Benutzer oder wieder über den **Manager** erzeugt. Innerhalb der Klasse befindet sich eine Liste von **Motion**-Objekten, die in einer bestimmten Zeit ausgeführt werden sollen. Dabei wird bei der Initialisierung angegeben wie lange die gesamte Animation dauern soll und welcher **Renderer/Synchronizer** die Animation ausführen soll. Danach kann die Animation über die Methodenaufrufe *start(), stop(), resume()* gesteuert werden.

6.3. Implementierung des Näherungsverfahren zur Darstellung von Radarsignalen und Materialarten

Das in dem Kapitel 5.2.5 beschriebene Näherungsverfahren wird in der Klasse-**RenderJPCT** umgesetzt. Abhängig von dem gesetzten Zeichenmodus werden entweder die unterschiedlichen Materialien farblich hervorgehoben oder der Einfallswinkel des Radarsignals. Bei beiden Varianten wird mit dem integrierten **TextureManager** die Farbe jedes Polygons berechnet und angepasst. In Abbildung 5 ist das Ergebnis dieser Berechnung für den Radarsignal-Modus zu sehen.

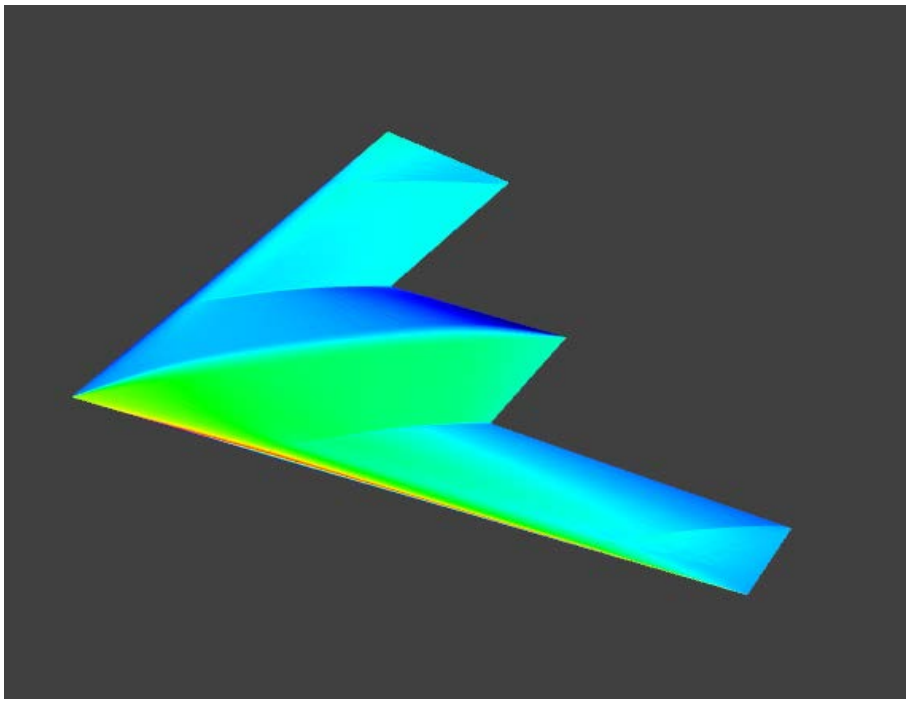

Abbildung 5: Darstellung des DLR-F17 mit dem neuen Renderer

In Abbildung 6 ist die Materialdarstellung mit eingeblendetem Drahtgittermodell gezeigt.

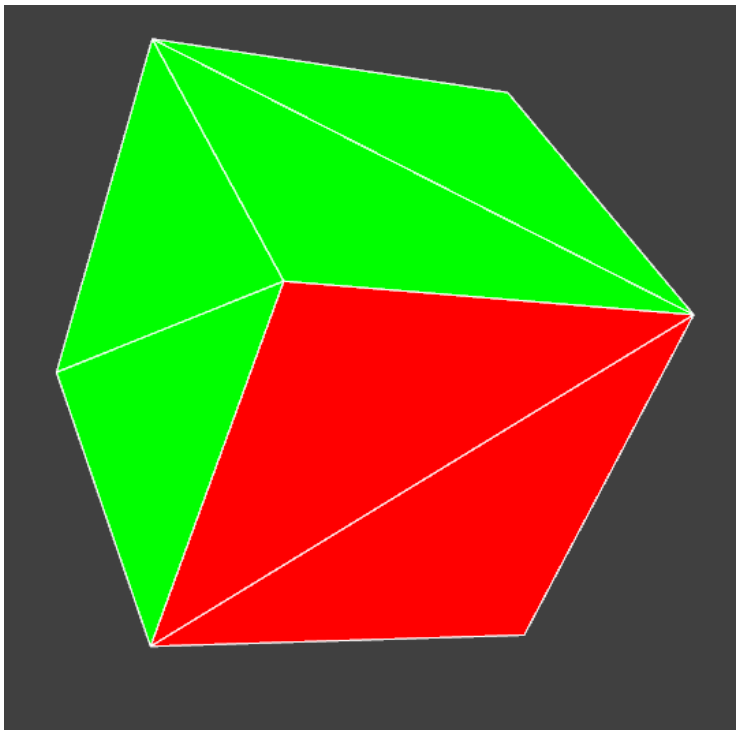

Abbildung 6: Zweite Darstellungsvariante mit eingeschaltetem Drahtgittermodell. (ein Würfel der aus zwei Materialien besteht).

Problematisch waren diese Varianten bei detaillierten Objekten wie dem DLR-F17. Durch das Berechnen der Farbe von jedem einzelnen Polygon ist die Bildwiederholungsrate stark gesunken. Alleine das Berechnen der Polygonfarben hat $40ms$ benötigt. Zusammen mit den restlichen Methodenaufrufen pro Zyklus hat sich eine Verzögerung von $140ms$ ergeben. Das führte zu einer Bildwiederholungsrate unter 7 FPS. In der neuen jPCT Version 1.27 wurde ein neuer **TexturManagers** implementiert. Durch diese neue Implementierung sollte die Ausführungsgeschwindigkeit steigen. Es war nur eine Verbesserung von *25%* Prozent messbar. Der **TexturManager** ist anscheinend nicht für diese Art der Verwendung ausgelegt. Da dieses Problem aber zu nicht flüssigen Bildwiederholungsraten führt, muss ein anderer Visualisierungsweg gewählt werden. Die einzige Alternative dafür ist, die Implementierung eines Shaders, welcher anhand von einer Lichtquelle den Einfallswinkel berechnet und daraus die Polygonfarbe bestimmt.

Shader sind kleine Recheneinheiten auf der Grafikkarte. Diese können in OpenGL mit der Programmiersprache OpenGL Shading Language (GLSL) programmiert werden. Insgesamt gibt es vier Typen von Shadern:

- Vertex-Shader – dient zur Transformation der Eckpunkte
- Tesselation-Shader – Aufsplitten von Polygonen
- Geometry-Shader – Veränderung der Geometrie
- Fragment-Shader – Setzen der Texturen und sonstige Farbberechnung

Für diesen Anwendungsfall werden nur der Vertex-Shader und der Fragment-Shader benötigt. Der Vertex-Shader berechnet die aktuelle Position der einzelnen Vertices (Eckpunkte) im Projektionsraum und ermittelt die Vertex-Normalen. Die Vertex-Normale wird aus dem Mittelwert der Normalen aller an ein Vertex angrenzenden Flächen berechnet. Danach wird der Richtungsvektor zwischen der Lichtquelle und dem Vertex bestimmt. Abhängig von dem zugrunde liegenden Beleuchtungsmodell werden entweder die Vertex-Normalen oder die Normalen des Polygons ermittelt. Bei dem Beleuchtungsmodell Flat-Shading wird die Normale des Polygons verwendet, beim Phong-Shading die Vertex-Normale. In Abbildung 7 ist der daraus resultierende Effekt beim Beleuchten mit einer Lichtquelle dargestellt. Beim Flat-Shading entstehen harte Kanten zwischen den Flächen, während beim Phong-Shading eine glatte Oberfläche entsteht.

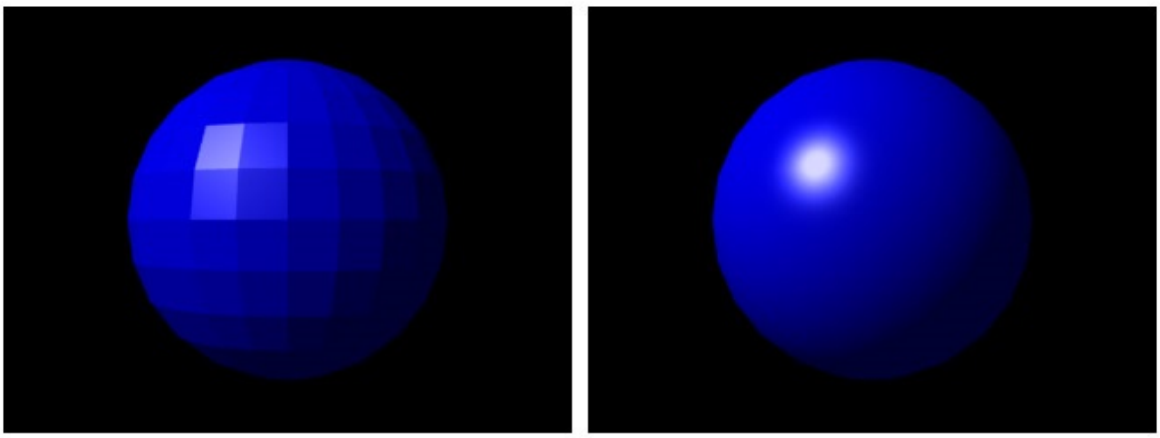

Abbildung 7: Geometrie einer Kugel: links Flat-Shading, rechts Phong-Shading [13]

Der Vertex-Shader, welcher für dieses Projekt programmiert wurde, bringt nun das Vertex in das Weltkoordinatensystem. Dazu wird die aktuelle Rotationsmatrix mit der Punktkoordinate multipliziert. Anschließend wird noch der Richtungsvektor der Lichtquelle („Radarstation“) zum Vertex gebildet und die Vertex-Normale des Eckpunktes abgespeichert. Die Vertex-Normale und der Richtungsvektor werden dann dem Fragment-Shader übergeben. Dieser berechnet den Einfallswinkel anhand der im Kapitel 5.2.5 beschriebenen Formel. Anschließend wird der Farbwert berechnet. Alle Farbwerte werden anhand der sechs Farben (schwarz, blau, cyan, grün, gelb und rot) berechnet, zwischen den Farben wird linear interpoliert.

Da jPCT standardmäßig das Beleuchtungsmodell Phong-Shading einsetzt, entsteht bei „harten“ Kanten eine falsche Objekteinfärbung. In der folgenden Abbildung sind ein VW-Golf mit vielen abgerundeten Kanten und ein Würfel mit harten Kanten jeweils im Phon-Shading und Flat-Shading dargestellt.

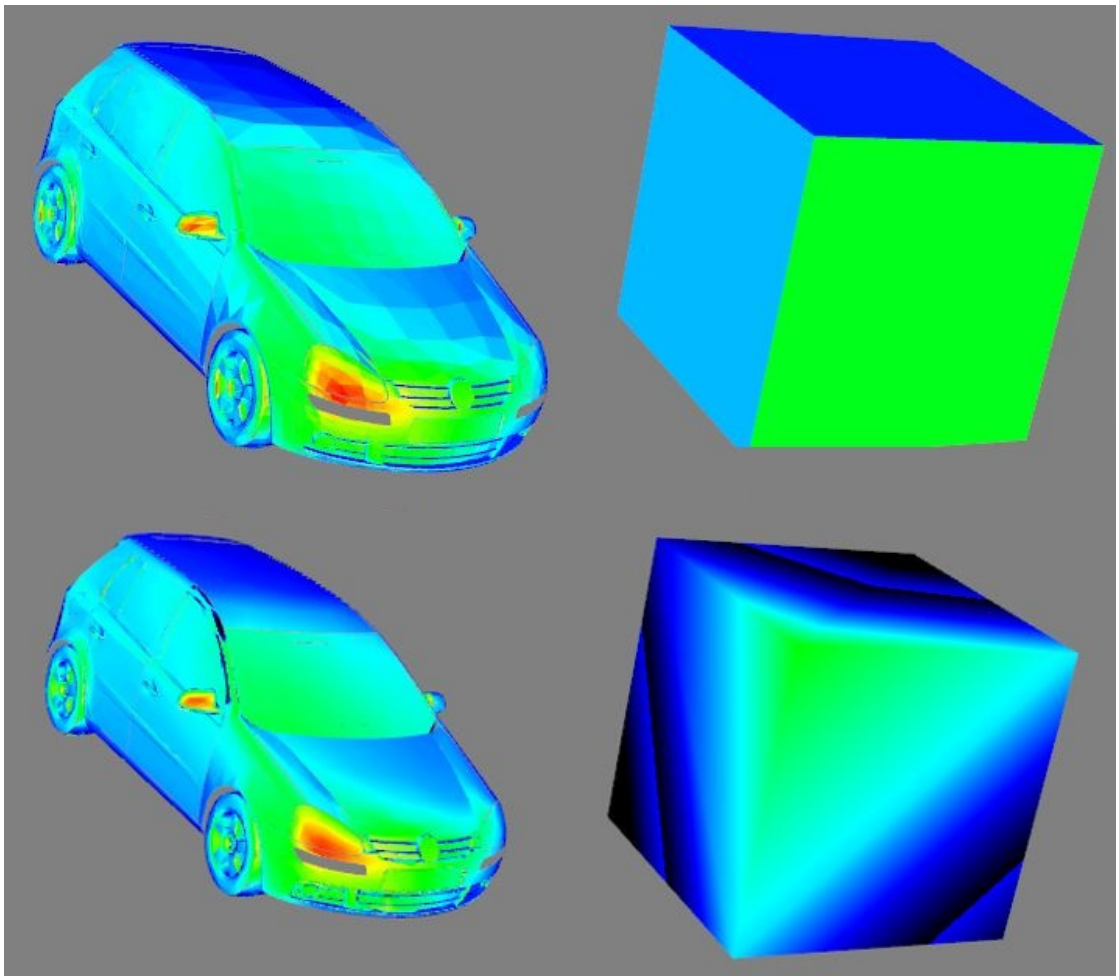

Abbildung 8: Oben Flat-Shading, unten Phong-Shading

Da eine ebene Fläche unabhängig von den angrenzenden Kanten eingefärbt werden soll, ist der Einsatz von Flat-Shading zu bevorzugen. Bei Objekten wie dem Pkw führt jedoch das Phong-Shading zu realistischeren Resultaten. Daher wurde entschieden, ein Aktiveren/Deaktivieren des Phong-Shadings zu implementieren.

Das Flat-Shading lässt sich in jPCT über einen Umweg aktivieren. Um dies zu erreichen, muss der Datensatz mit den gewünschten Normalen für das Flat-Shading in das Wavefront-Format konvertiert werden. In diesem Format kann eine manuelle Zuweisung der Normalen erfolgen. Dieses Format lässt sich dann von jPCT laden und beinhaltet alle Daten für eine korrekte Shader Darstellung. Danach muss nur noch das Flag *useNormalsFromObj* gesetzt werden. Die Wavefront-Datei wird dabei nur temporär abgespeichert und nach der Verwendung sofort gelöscht.

Der große Vorteil von Shadern ist, dass diese voll parallel auf der Grafikkarte ausgeführt werden. Durch den Einsatz von Shadern liegt die Bildwiederholungsrate bei weit über 500 FPS und wird über die Begrenzung in der ARenderer Klasse auf 50 FPS heruntergeregelt. Der Programmcode zu den Shadern befindet sich im Anhang.

6.4. Implementierung der Konfiguration

Für eine Anwenderfreundliche API ist es zwingend erforderlich, dass diese voll Konfigurierbar ist. Dies geschieht in den meisten Fällen über eine Konfigurationsdatei mit allen Parametern im XML-Format. Bei der Verwendung dieser Bibliothek soll es dem Benutzer auch möglich sein eine eigene Konfigurationsdatei zu erstellen. Um wieder einen einfachen Umgang mit der Bibliothek zu gewährleisten, wird aber auf ein zwingendes Vorhandensein einer Konfigurationsdatei verzichtet.

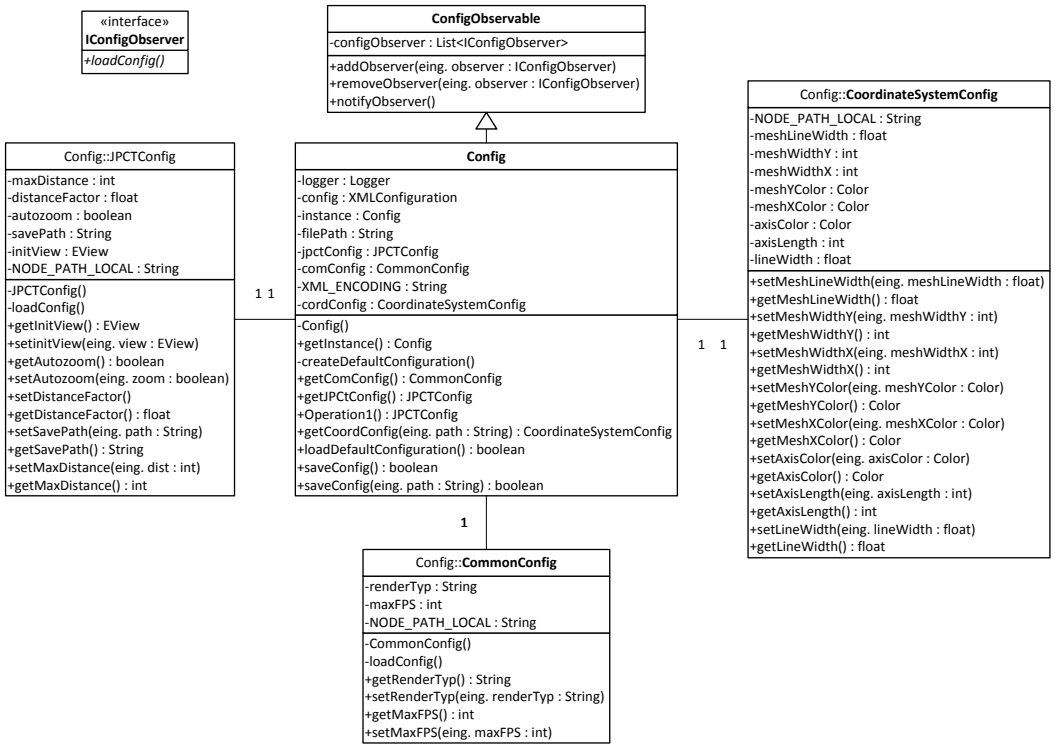

Diagramm 12: Config-Klasse mit allen Inneren-Klassen und der Observer-Schnittstelle

Im Diagramm 12 ist das passende Klassendiagramm zum Aufbau der Konfiguration innerhalb der Bibliothek dargestellt. Die Klasse **Config** dient als zentrales Objekt, ähnlich der Klasse-Manager. Diese Klasse ist als Singleton implementiert und ist über den statischen Aufruf *Config.getInstance()* von jedem anderen Objekt aufrufbar. Dabei wird von dieser Klasse während der ganzen Laufzeit nur ein Objekt erstellt. Zur Threadsicherheit muss der Methodenaufruf synchronisiert werden. Falls sonst mehrere Threads parallel auf diese Methode zugreifen besteht die Möglichkeit, dass mehrere Instanzen des Objektes erzeugt werden. Zudem implementiert diese Klasse das Observer-Entwurfsmuster. Dies dient dazu alle instanziierten Renderer, welche das

Interface *IConfigObserver* implementiert haben, über neugeladene Konfigurationen zu informieren. Dazu wird der Renderer beim Erstellen im Manager als Observer im **Config**-Objekt angemeldet.

6.5. Qualitätsüberwachung des Programmcodes

Im Zuge dieses Projektes wurde ein hoher Wert auf die Qualität der entstehenden API gelegt. Mit Hilfe diverser Werkzeuge ist es möglich einen Quellcode auf Regelkonformität zu prüfen. Dazu wird der Quellcode einer statischen Code-Analyse unterzogen. Dafür hat sich in der Java Welt die Plattform SonarQube (ehemals Sonar) durchgesetzt. Diese ermöglicht eine Analyse eines Quellcodes und stellt diese anschaulich auf einer Webpage dar. Diese lässt sich freigestalten und um unzählige Plug-Ins erweitern, auf die hier nicht näher eingegangen wird.

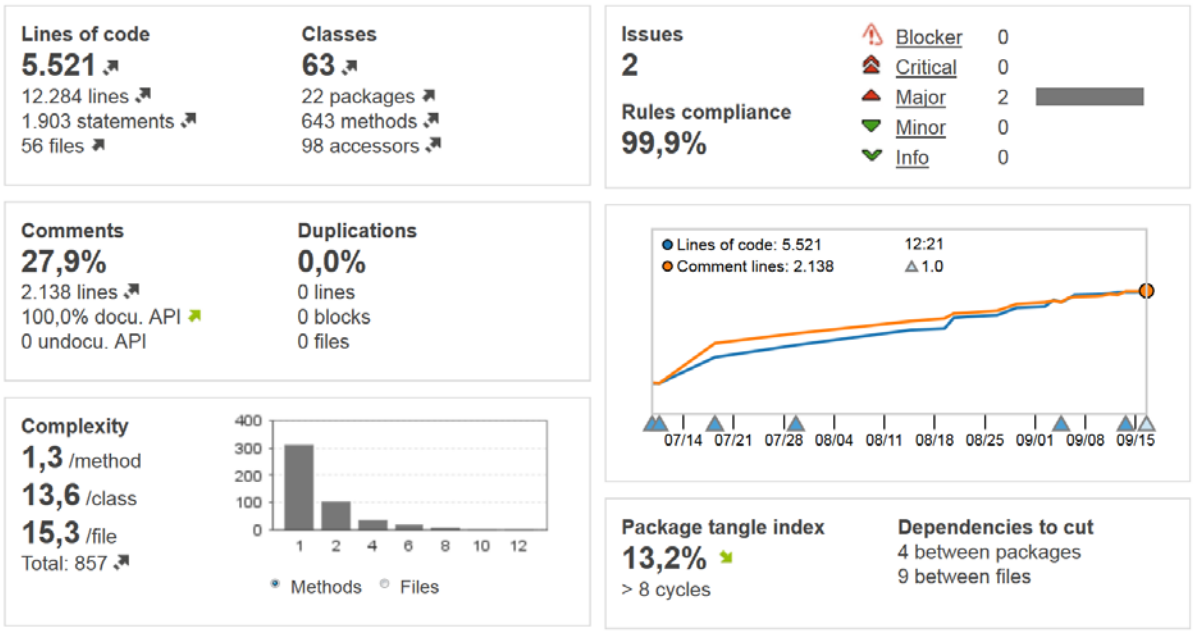

Abbildung 9: Dashboard von SonarQube zum Projekt

Dabei ist SonarQube nicht auf die Programmiersprache Java begrenzt sondern unterstützt durch diverse Plug-Ins diverse andere Programmiersprachen wie Groovy, PHP, Flex, C#, Python etc.

SonaqQube lässt sich auf zwei verschiedene Arten in den Entwicklungsprozess einbinden. Es kann einerseits lokal auf einem Entwicklungsrechner zur Prüfung des

Quellcodes ausgeführt werden. Als weitere Möglichkeit lässt sich SonarQube in einen Build-Server für die kontinuierliche Integration einbinden. Für dieses Entwicklungsprojekt wurde die erste Variante gewählt, da keine Ressourcen und keine Notwendigkeit für das Aufsetzen eines Build-Servers vorhanden sind.

Die stochastische Analyse in SonarQube läuft grundlegend in drei Schritten ab. Der erste Schritt ist die eigentliche Analyse des Quellcodes anhand verschiedener Qualitätsmerkmale. Es wird unteranderem die Komplexität einzelner Funktionen gemessen. Dazu werden die Anzahl der Verzweigungen innerhalb einer Methode gezählt (in diesem Projekt wird ab einer Methoden-Komplexität von 12 eine Warnung ausgegeben). Auch wird die Anzahl der codezeilen ermittelt, wie viele davon doppelt sind, an welcher Stelle potenzielle Nullpointer-Exceptions auftreten könnten und ob die Dokumentationsrichtlinien eingehalten wurden. Zusätzlich werden noch viele weitere Kodierrichtlinien überprüft, die sich innerhalb der Software frei konfigurieren lassen. Standard Java-Richtlinien, welche in diesem Projekt aktiviert sind, sind beispielsweise die Verbote von öffentlichen und geschützten Variablen. SonarQube fordert, bei entsprechend aktiver Regel, dass ein Zugriff auf solche Variablen nur über Getter/Setter-Methoden möglich ist. SonarQube bringt dabei die offiziellen Kodier-Richtlinien von Java bereits als Regelwerk mit.

Es hat sich als sinnvoll erwiesen nicht alle Regeln von SonarQube zu aktivieren. So ist die Regel „Magic Number“ zu restriktiv voreingestellt. Diese gibt an, wenn irgendwo im Programmcode eine Zahl ohne jeglichen Bezug auftaucht. Ein Umsetzen dieser Regel kann zu einer drastischen Zunahme an Variablen in einfachen mathematischen Klassen führen. Deshalb wurde die Zahlen wie z.B. 2, 3, 4 und 100 von der „Magie Number“ Regel ausgenommen.

Durch den modularen Aufbau der Plattform erlaubt es Sonar zudem diverse andere Plug-Ins zum Qualitätsmanagement einzubinden. So kann mit dem Tool FindBugs automatisch nach potentiellen Bugs gesucht werden (z.B. nicht freigegebene Ressourcen) oder es können mit Cobertura die Testabläufe automatisiert werden. [14]

Alle Ergebnisse der Code-Analyse werden in eine Datenbank eingetragen, dadurch ist ein einfacher Überblick über die laufende Entwicklung der Softwarequalität des Projektes möglich. Zudem wird jede Veränderung dem entsprechenden Entwickler zu

gewiesen, womit schnell der zuständige Entwickler für die Fehlerbehebung ausgemacht werden kann. In diesem Projekt dies ohne Bedeutung, da es nur einen Entwickler gibt.

Für das Qualitätsmanagement wurde während der Implementierungsphase ein Zeitfenster von 15 Minuten eingeplant (Empfehlung von SonarQube). In dieser Zeit wurde eine Code-Analyse durchgeführt und die Ergebnisse überprüft. Anschließend an die Prüfung der Ergebnisse wurden die Verletzungen der Kodierrichtlinien und gefundenen Fehler aus der Software beseitigt. Ein großer Vorteil, der sich aus dieser Vorgehensweise bei dem Qualitätsmanagement ergibt, ist dass man als Entwickler irgendwann die Kodierichtlinien verinnerlicht hat. Ein weiterer Vorteil ist, dass mögliche Fehlerursachen innerhalb des Codes frühzeitig erkannt werden, was zu einem weniger fehleranfälligen Endprodukt führt.

7. Integrationstests in verschiedenen Programmen

Die Integration der Bibliothek wird in die im vorherigen Kapiteln erwähnten Programme durchgeführt. Dabei soll überprüft werden mit welchem Aufwand sich die Bibliothek in ein bestehendes Softwareprodukt integrieren lässt, ob sich alle Funktionen richtig verhalten und welche Vorteile/Nachteile durch diese API entstehen.

Die Integration in **Visual Control** war ohne Probleme möglich. Dazu mussten nur die alten Render-Klassen entfernt werden und deren Aufrufe im Programm durch die API ersetzt werden. Im Gesamten ist das Einbinden der Bibliothek ein sehr geringer Integrationsaufwand. Durch die neue API konnte an vielen Stellen Programmcode gelöscht werden. Bei der vollständigen Integration werden die „Lines of Code" um ca. 20% sinken. Eine abschließende vollständige Integration der API konnte aus Zeitgründen noch nicht durchgeführt werden. Es hat sich herausgestellt das es sinnvoller ist die GUI von **Visual Control** im Zuge der Integration ebenfalls zu überarbeiten. Dies bedeutet jedoch ein immenser mehr Aufwand und wird deshalb erst in naher Zukunft durchgeführt.

Durch die Verwendung der neuen API ergeben sich für den Benutzer des Programmes folgende Vorteile: eine verbesserte Optik der dargestellten Objekte, deutlich bessere und flüssigere Animationen von Objekten, qualitativ höherwertigere Screenshots und mehr Konfigurationsmöglichkeiten. Die Steuerung der Software blieb unverändert. Die einzige kleine Änderung war das Entfernen von nicht mehr benötigten Funktionen, wie das „manuelle Rendern".

Für den Programmierer der Software ergibt sich die Möglichkeit die Software besser zu strukturieren und das Rendering einfacher zu steuern. Es können nun in neuen und alten Visualisierungsprogrammen wesentlich schneller Funktionen implementiert werden, da die API viele grundlegende Funktionen mit sich bringt. Zudem ist ein Umsetzen des Model View Control Entwurfsmusters nun leichter umsetzbar. Dieses Entwurfsmuster erlaubt eine Trennung von Visualisierung (View), Datenhaltung (Model) und der eigentlichen Programmlogik (Control). Dabei wird jegliche Datenverarbeitung und Benutzereingabe von den Model und Control Komponenten

ausgeführt. Die View dient nur zur Visualisierung der Ergebnisse und dem Entgegennehmen von Benutzereingaben.

Eine Integration in das Programm **Visualizer UCAV** konnte aus Zeitgründen nichtmehr durchgeführt werden. Jedoch wurde in einer Beispielsoftware alle benötigen Funktionen für **Visualizer UCAV** getestet. Hier wird das **MultiViewPanel** implementiert und eine kleine Animation der Flugtrajektorie durchgeführt. Des Weiteren ist es möglich das Objekt zu rotieren und den Kamerablickwinkel zu verändern (über Maus, Tastatur). Bei dem visualisiertem Beispiel (Siehe Abbildung 10) wird angenommen, dass sich die Radarstation in unendlicher Entfernung auf der Z-Achse befindet.

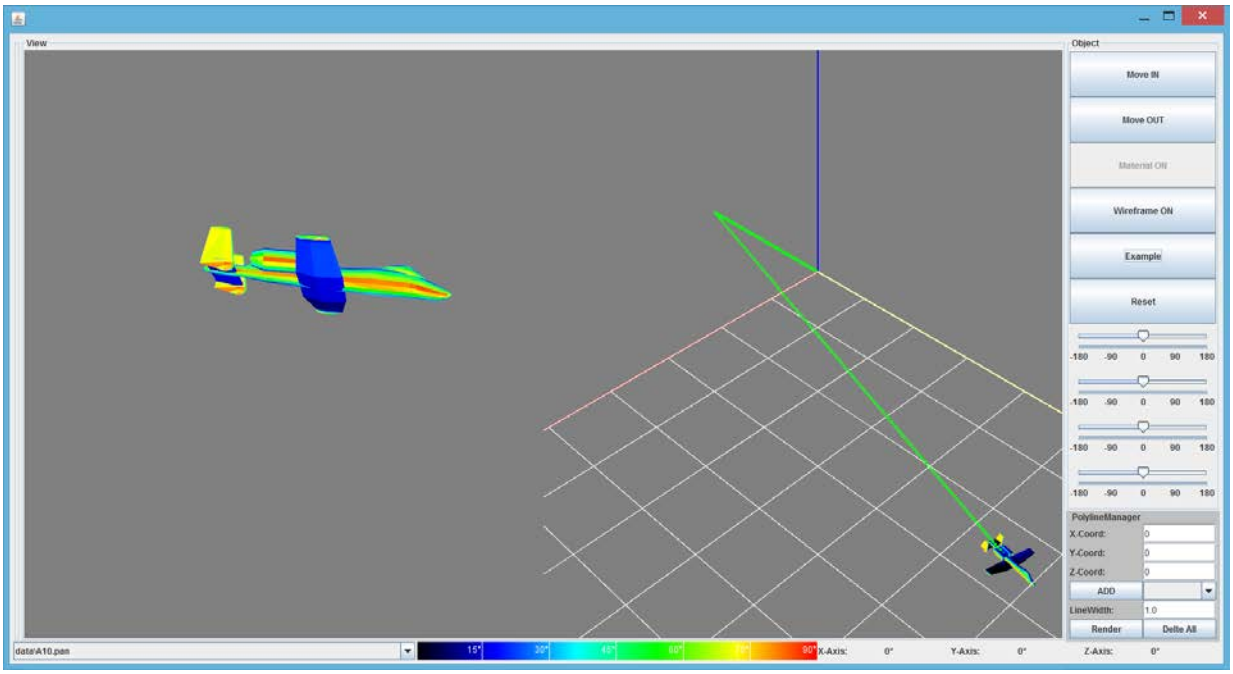

Abbildung 10: Demo-Programm zum Testen der Funktionen für Visualizer UCAV und als Programmierbeispiel für die Anleitung.(Rot = Z-Achse, Gelb = X Achse, Blaue = Y-Achse)

8. Fazit und Ausblick

Durch die neu entwickelte API ist es nun möglich alte Programme zu aktualisieren und zukünftige neue Programme zur Visualisierung darauf aufzubauen. Dabei steht für den Entwickler eine einfache API zur Verfügung, welche es erlaubt eine Visualisierung mit wenigen Handgriffen in eine GUI einzubauen. Durch ein modulares-Konzept ist die API einfach erweiterbar und Kernkomponenten, wie z.B. die Rendering-Engine, lassen sich einfach austauschen. Es konnten alle funktionalen sowie nicht funktionalen Anforderungen während der Entwicklung erfüllt werden. Die Funktion der Videogenerierung konnte in der aktuellen finalen Version nicht mehr realisiert werden, sie wird in der nächsten Version nachgereicht.

Das Überwachen des Programmcodes hatte diverse Verbesserungen an der Software zur Folge. Der Zeitverlust durch diese Qualitätskontrolle ist sehr gering und die erhöhte Qualität hat einen immensen Mehrwert für die Software. Die Anwendung dieser Qualitätsüberwachung ist sehr zu empfehlen und wird in zukünftigen Projekten weiterhin eingesetzt werden.

Im Nachhinein gesehen war die Wahl der Rendering-API nicht optimal. Die Performance-Probleme, die beim Einfärben von Polygonen aufgetreten sind, wurden unterschätzt. Durch die Lösung des Problems mit der Programmierung von Shadern ist der große Vorteil von jPCT, nämlich das simple Verwalten der Polygonfarben, verloren gegangen. So wäre es von vornherein möglich gewesen für die erste Implementierung Java3D oder die jMonkeyEngine einzusetzen. Um solche Probleme zukünftig zu vermeiden, sollten ausführlichere Tests der API vor dem eigentlichen Start der Entwicklung stattfinden. Andere Probleme, wie eine ungünstige Kapselung von Threads und Leistungseinbrüche des Gesamtsystems durch zu hohe Bildaktualisierungsraten, konnten erfolgreich beseitigt werden.

Für die zukünftige Weiterentwicklung der API ist es denkbar eine Verwaltung der verschiedenen Komponenten mit OSGi umzusetzen. Der Grundstein für eine solche Integration wurde durch das Abstrahieren der Render-Klasse und die strikte Interaktion über Schnittstellen bereits gelegt. Andere Komponenten, wie die Mathe-Klassen, könnten dann auch in eigene Module ausgelagert werden. Des Weiteren ist

für die Zukunft vorgesehen diese API noch in das Programm **Visualizer UCAV** zu integrieren. Ein Demoprogramm zum Testen der Funktionen für **Visualizer UCAV** wurde während dieser Arbeit bereits geschrieben. Das Testen der Funktionen war erfolgreich und es sollte kein Problem darstellen die Integration in **Visualizer UCAV** durchzuführen. Dieses Demoprogramm dient auch als Programmierbeispiel um den Umgang mit der API zu erlernen.

I. Literatur- und Quellenverzeichnis

1. DLR Projekt UCAV, 2010
   Besucht am 10.07.2013
   URL: http://www.dlr.de/as/desktopdefault.aspx/tabid-3574/5578_read-8080/

2. DLR, Forschungsbilanz und wirtschaftliche Entwicklung 2011/2012, Seite 13, FaUSST
   Besucht am 30.09.2013
   URL:
   http://www.dlr.de/dlr/PortalData/1/Resources/documents/2012_1/22914_DLR-FuwE_2012_online.pdf

3. Prof. Dr.-Ing. Bernhard Huder, Einführung in die Radartechnik, Leipzig 1999, Seite 1-3

4. Albrecht Ludloff, Praxiswissen Radar und Radarsignalverarbeitung, 2. Auflage, Braunschweig, Wiesbaden, 1998, Kapitel 3-1

5. Philipp Posovszky, Erweiterung des Skripts zur Überprüfung von Geometriedatensätzen in Rhinoceros®, Oberpfaffenhofen 27.09.2011

6. Sebastian Schäfer, Szenengraphen, SR Heidelberg
   Besucht am 25.07.2013
   URL: http://clever.gdv.informatik.uni-frankfurt.de/~schaefer/GEM11/V/V-Szenegraphen.pdf

7. LWJGL-Projekt,
   Besucht am 23.07.2013
   URL: http://www.lwjgl.org/

8. Hans Bacher,Java 3D – A Scene graph API, Technische Universität München,
   Besucht am 24.07.2013
   URL: http://wwwwestermann.in.tum.de/Teaching/SS2004/ProSem /Workouts/bacher/Folien.pdf

9. Mona Gerus, Vergleichende Untersuchung von szenengraph-basierten Renderingsystemen, Studienarbeit, Universität Koblenz, Institut für Computervisualisitk, Oktober 2003
Besucht am 24.07.2013
URL: http://www.uni-koblenz.de/~cg/Studienarbeiten/monagerus.pdf

10. jME-Projekt
Besucht am 25.07.2013
URL: http://jmonkeyengine.org/engine/

11. jPCT-Projekt,
Besucht am 24.07.2013
URL: htttp://www.jpct.net/

12. Aviatrix3D-Features List
Besucht am 25.07.2013
URL: http://aviatrix3d.j3d.org/features.html

13. Bild Phong-Shading –Flat –Shading
Heruntergeladen am 17.09.2013
URL: http://upload.wikimedia.org/wikipedia/commons/8/84/Phong-shading-sample.jpg

14. Brian Chaplin, Using Sonar to Bake a Quality Feedback Loop into the Build Cycle
Besucht am 30.08.2013
URL: http://www.theserverside.com/tutorial/Use-Sonar-to-Develop-a-Quality-Feedback-Loop-into-the-Build-Cycle

II. Anhang

a. Sequenzdiagramm – Initialisieren eines DatenReaders

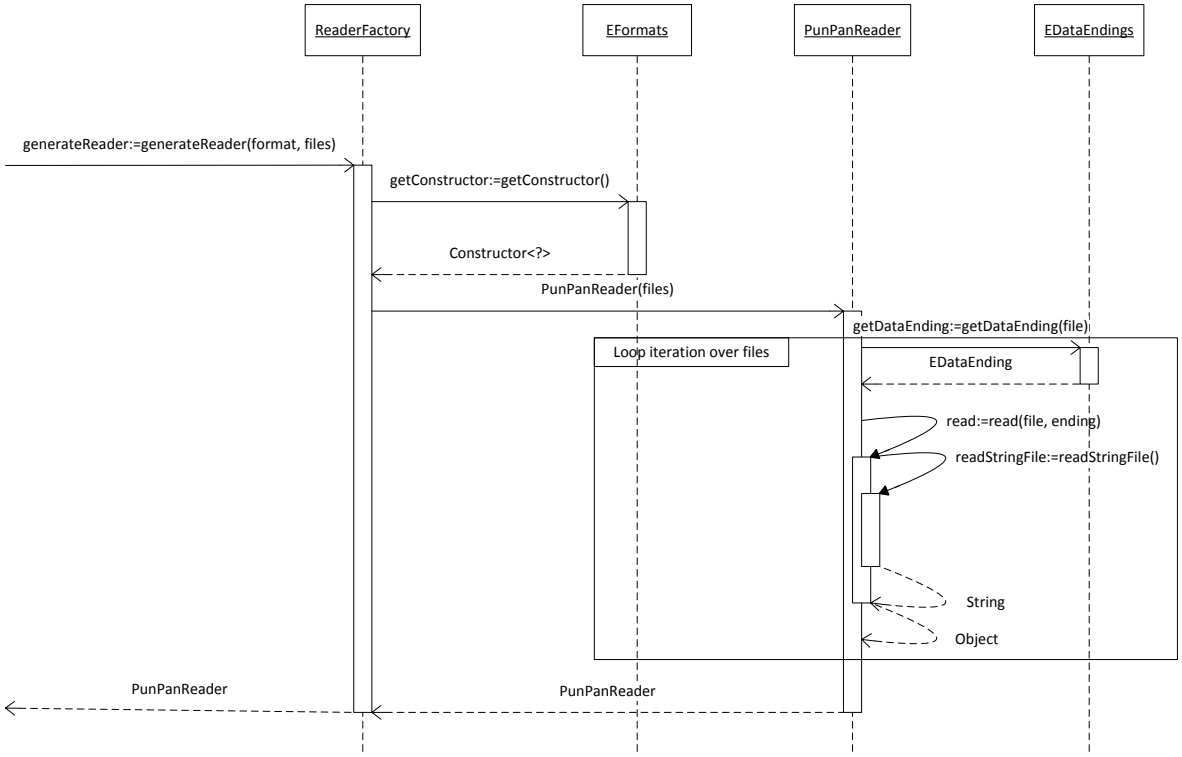

Diagramm 13: Sequenzdiagramm zur Erstellung des PunPanReader Objektes zum Einlesen der Geometrie

b. Klassendiagramm Synchronizer

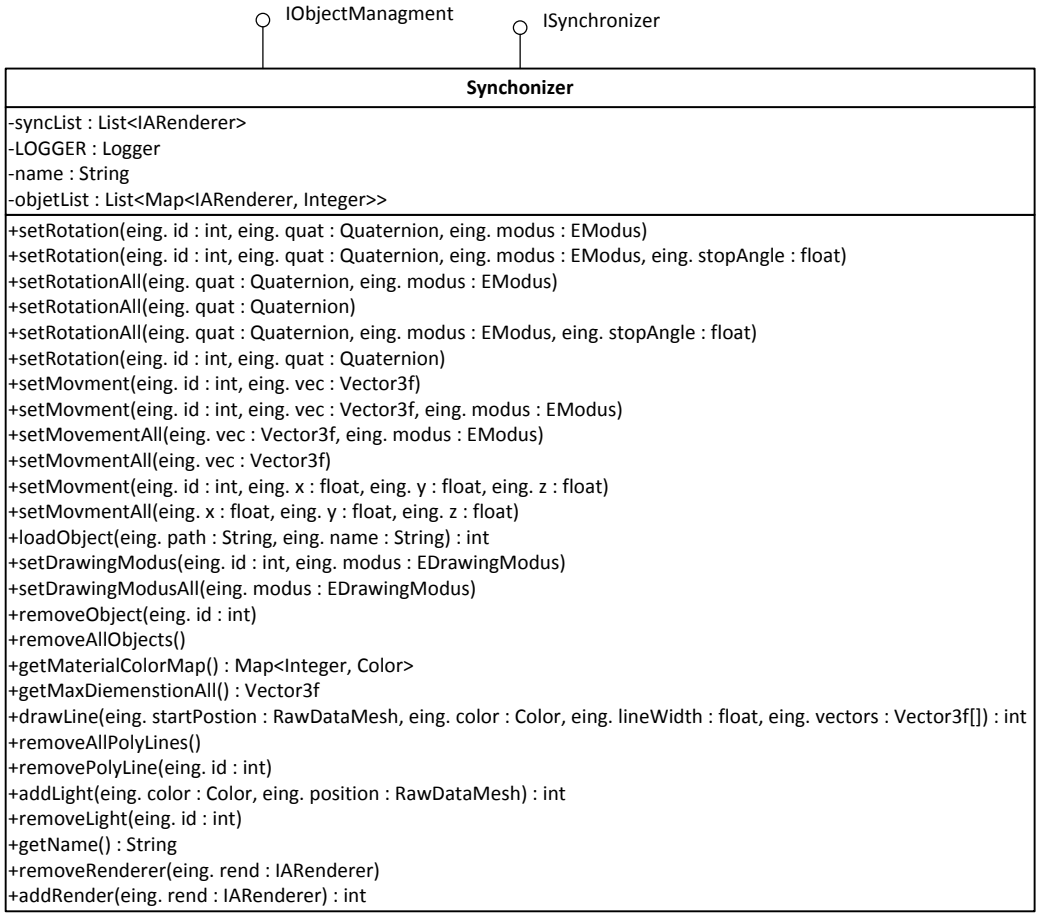

Diagramm 14: Klassendiagramm zum Synchronizer

c. Klassendiagramm ViewControl

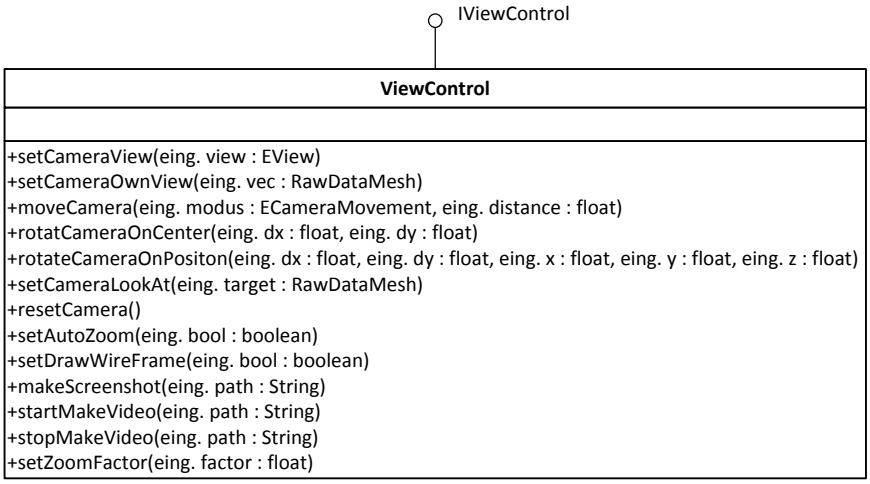

Diagramm 15: Klassendiagramm zu ViewControl

d. Programmcode Vertex-Shader

```
uniform int lightSourceNr;
varying vec3 normalized_normal;
varying vec3 normalized_vertex_to_light_vector;
void main()
{
    // Transforming The Vertex
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

    // Transforming The Normal To ModelView-Space
    vec3 normal = gl_NormalMatrix * gl_Normal;

    // Transforming the Vertex Position to ModelView-Space
    vec4 vertex_in_modelview_space = gl_ModelViewMatrix * gl_Vertex;

    // Calculating the Vector from the Vertex Position to the Light
   vec3 vertex_to_light_vector =
vec3(gl_LightSource[lightSourceNr].position-vertex_in_modelview_space);

    // Scaling The Input Vector To Length 1
    normalized_normal = normalize(normal);
    normalized_vertex_to_light_vector = normalize(vertex_to_light_vector);

 }
```

a. Programmcode Fragment-Shader

```
uniform vec3 colors[6];
uniform int interpolation;
varying vec3 normalized_normal;
varying vec3 normalized_vertex_to_light_vector;
        void main()
        {
        //Calculate the incident Angle
             float     intensity =  degrees(asin(dot(normalized_normal,
normalized_vertex_to_light_vector)))/90.0;
             float x=float(colors.length-1)*intensity;

             int color1ID=int(x);
             int color2ID=color1ID+1;
             float factor=fract(x);
             vec3  c0=colors[color1ID];
             vec3  c1=colors[color2ID];
             float temp=0.0;
             //Set the steps of Interpolations
             if(interpolation>1)
             {
               float step=1.0/float(interpolation);

               while(temp<=factor)
               {
                       temp+=step;
               }
             }else
             {
               temp=factor;
             }
             gl_FragColor=vec4(mix(c0,c1,temp),1.0);

        }
```