



Bringing Your Hands Into Virtual Reality

Schlussbericht

Studiengang: Informatik
Autoren: Simon Meer
Betreuer: Prof. Urs Künzler
Experten: Yves Petitpierre
Datum: 11.06.2015

Versionen

Version	Datum	Status	Bemerkungen
0.1	14.04.2015	Entwurf	Outline erstellt
0.9	08.06.2015	Entwurf	Draft zum Durchlesen erstellt
1.0	11.06.2015	Definitiv	Finale Version erstellt

Inhaltsverzeichnis

1 Einleitung	1
2 Aufgabenstellung und Zielformulierung	2
2.1 Ausgangslage	2
2.2 Ursprüngliche Zielformulierung	3
2.3 Änderungen an der Zielformulierung	3
2.4 Hilfsmittel & Hardware	4
3 Design	5
3.1 Systemübersicht	5
3.2 Systemarchitektur	7
3.3 Systemdesign	10
4 Implementation des Indexers	32
4.1 Aufbau	32
4.2 Datenquellen	33
4.3 Datenstruktur	34
4.4 Herausforderungen	35
5 Implementation von IMVR	36
5.1 Unity 5	36
5.2 Aufbau	37
5.3 Interaktionskonzept	38
5.4 Visual Design	43
5.5 Herausforderungen	52
6 Testergebnisse	61
7 Projektmanagement	62
7.1 Soll-Ist Gegenüberstellung	62
7.2 Zeitplan	65
8 Schlussbetrachtung	66
8.1 Ergebnis	66
8.2 Reflexion	66
8.3 Ausblick	67
Selbständigkeitserklärung	69
Glossar	70
Abbildungsverzeichnis	73

1 Einleitung

Eine der faszinierendsten Entwicklungen unserer Zeit ist zweifelsohne das Aufkommen von Virtual Reality (VR) zu einem erschwinglichen Preis. Mithilfe einer speziellen Brille, einem sogenannten Head-Mounted Display (HMD), taucht der Träger hierbei in eine andere Welt ein, die ihm vorgegaukelt wird. Passend zu dieser Entwicklung finden auch zunehmend neue Peripherie-Geräte ihren Weg in den Markt. Mitunter zu den offensichtlich natürlichsten Methoden der Eingabe gehören ganz klar die eigenen zwei Hände. Es scheint deshalb logisch, diese auch wirklich zu verwenden.

Nun geht es also darum, ein HMD, die Oculus Rift, mit einem Hand-Sensor, der Leap Motion, zu koppeln, und ein voll funktionierendes virtuelles Erlebnis damit zu entwickeln.

In einer Vorarbeit wurde bereits geprüft, wie die zwei Geräte zusammenspielen und ein gewisses *Know-How* wurde erarbeitet. Die *Lessons Learned* waren, dass sich die Leap Motion beispielsweise nicht wirklich eignet, um darauf ein ganzes Spiel aufzubauen und dass eine gute Überlegung zum Design der Interaktion nötig ist.

Mit diesen Erkenntnissen im Schlepptau soll nun eine Musikapplikation entwickelt werden, welche die Hände sinnvoll und effektiv einsetzt.

2 Aufgabenstellung und Zielformulierung

2.1 Ausgangslage

Wie bereits erwähnt, mangelt es in der Medienwelt nicht an Begeisterung für Virtual Reality. In Foren wird diskutiert, an Events wird demonstriert und täglich finden neue Infos ihren Weg in die Öffentlichkeit.

Ein wesentliches Problem ist allerdings die Eingabe. Glücklicherweise gibt es bereits zu einem erschwinglichen Preis eine Möglichkeit, die eigenen zwei Hände zu verwenden und zu erkennen: die Leap Motion.

Sie ist zu einem Preis von 89.99 € erhältlich und ermöglicht eine rein visuelle Handerkennung ohne Handschuhe oder ähnlichen Zusatz.



Abbildung 2.1: Die Leap Motion im Einsatz

Quelle: <https://www.leapmotion.com/>

Im Vorprojekt, das ebenfalls bereits erwähnt wurde, stellte sich aus, dass jedoch Vorsicht beim Einsatz der Leap Motion geboten ist. Die Erkennung ist nicht stabil und gewisse Positionen funktionieren besser als andere. Es stellten sich also ein paar Anforderungen an eine Einbindung:

- Die Applikation darf nicht auf schnelle und genaue Eingabe des Benutzers basieren
- Gesten müssen mit Bedacht des Sichtfeldes konzipiert werden

Besonders der erste Punkt schliesst eine Anwendung der Leap Motion in einem Spiel aus. Bei einem Spiel würden die Hände unausweichlich in den Vordergrund des Spielkonzeptes rücken und eine essenzielle Rolle übernehmen. Da allerdings die Handerkennung nicht zuverlässig ist, geschehen Fehler. Unbeabsichtigte Operationen passieren, der Spieler verliert, er ist frustriert und nach kurzer Zeit gibt er das Spiel auf.

Der gewählte Ansatz ist jedoch ein anderer: Kein Spiel soll erstellt werden, sondern eine Applikation, die durch die Hände gesteuert werden kann und dadurch einen speziellen Charme erhält. Es folgt die Zielformulierung.

2.2 Ursprüngliche Zielformulierung

Es soll eine Applikation namens "Images & Music in Virtual Reality (IMVR)" entwickelt werden, welche Gebrauch von der Oculus Rift macht, um die Bild- und Musiksammlung des Anwenders ansprechend darzustellen, z.B. in Form eines 3D-Karussells. Die zusätzliche "Tiefe", die durch den Einsatz eines stereoskopischen HMD entsteht, soll dem Anwender helfen, sich in seiner Medienbibliothek schneller zurechtzufinden.

Zusätzlich dazu soll die Leap Motion dazu verwendet werden, um vollständige Handfreiheit zu gewähren: Der Anwender soll komplett ohne Maus und Tastatur imstande sein, sich durch seine Bilder zu navigieren.

Kurz zusammengefasst muss die Applikation:

- Die Bilder- und Musikbibliothek des Benutzers in stereoskopischem 3D darstellen.
- Diese freihändig durchsuchbar machen mit Sortier- und evtl. Gruppierfunktion.
- Die Bilder betrachtbar und die Musik abspielbar machen.
- Metainformationen darstellen (z.B. in Form von Diagrammen).

Zusätzlich zur Applikation selbst soll noch ein zusätzliches Tool entwickelt werden, welches im Voraus die Dateien auf dem Host-System indexiert und für die visuelle Applikation bereitstellt.

2.3 Änderungen an der Zielformulierung

Aufgrund der begrenzten Zeit und eines vermehrten Interesse in die Darstellung von Musik, wurde die Zielformulierung während des Projekts leicht abgeändert:

- Die Bilderbibliothek fällt weg
- Die Sortier- und Gruppierfunktion fällt weg
- Es wird mehr Gewicht auf die Darstellungsweise der Musik gelegt

Dazu ist zu sagen, dass auch weiterhin stark auf Bildervisualisierung gesetzt wird. Anstatt den Computer des Anwenders auf Bilder abzuscannen wird diese jedoch vollständig um die lokale Musik aufgebaut. Konkret bedeutet das, dass Artwork von Alben und Fotos von Artisten dargestellt wird.

Die im zweiten Punkt erwähnte Sortier- und Gruppierfunktion könnte in einem weiteren Schritt nachgereicht werden, wurde aber im momentanen Stadium zugunsten von anderen Features vernachlässigt.

2.4 Hilfsmittel & Hardware

Verschiedene Hilfsmittel werden für die Durchführung des Projektes gebraucht. Seitens Software lauten diese:

Tabelle 2.1: Übersicht der eingesetzten Software

Name	Beschreibung
Unity3D	Spielengine und Entwicklungsumgebung
Visual Studio 2012 & 2013	Entwicklungsumgebung von Microsoft
Blender	Open-Source 3D Modelling-Tool
Krita	Open-Source Grafikbearbeitungs-Tool
GIMP	Open-Source Grafikbearbeitungs-Tool
LaTeX	Hilfsmittel für die Erstellung wissenschaftlicher Dokumente
Microsoft Visio	Tool zur Erstellung von Diagrammen

Im Hardware-Departement wurde folgendes eingesetzt:

Tabelle 2.2: Übersicht der eingesetzten Hardware

Name	Beschreibung
Oculus Rift	HMD von Oculus VR
Leap Motion	Hand- und Fingererkennungsgerät von Leap Motion, Inc.

Für eine genaue Erklärung der Oculus Rift und der Leap Motion sei an dieser Stelle auf das Pflichtenheft und die Vorarbeit verwiesen, welche diese im Detail einführt.

3 Design

Bei der Entwicklung wurden diverse Design-Entscheidungen gefällt, welche zum Teil bereits in einem Vorprojekt analysiert wurden. Diese sollen in diesem Kapitel aufgelistet und erläutert werden.

3.1 Systemübersicht

Das Zusammenspiel der Oculus Rift und der Leap Motion ist auf den ersten Blick vielleicht nicht so offensichtlich. Erstere wird aufgesetzt und verdeckt die Sicht, letztere wird auf den Tisch gestellt und erkennt Hände über sich.

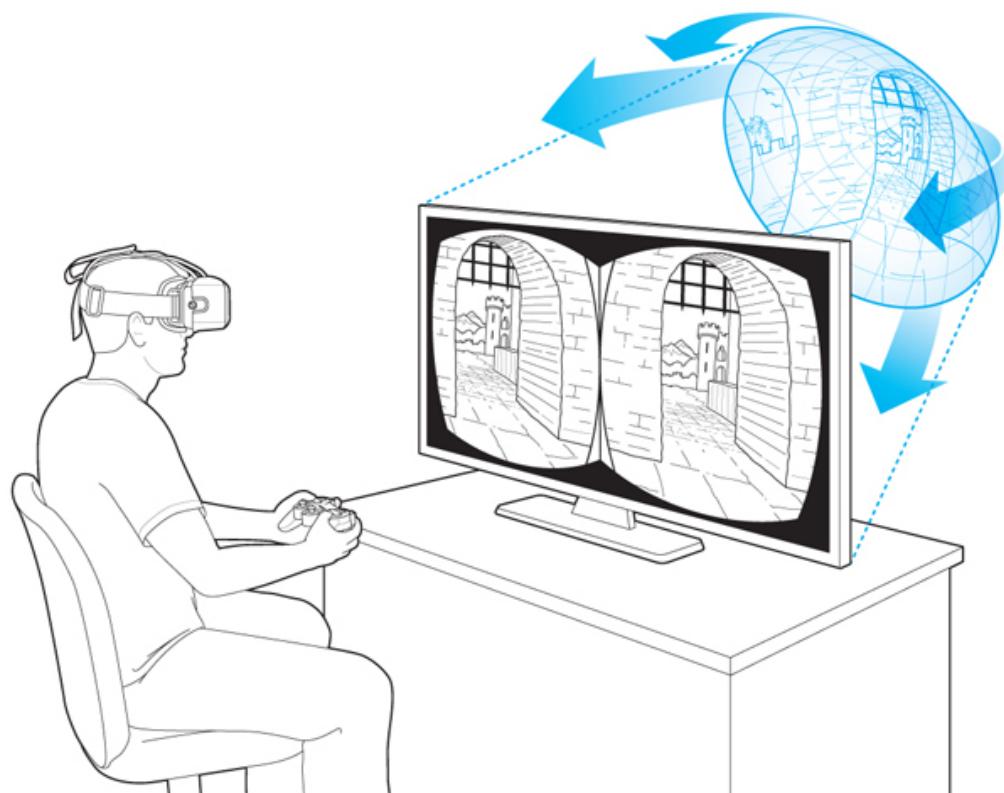


Abbildung 3.1: Funktionsweise der Oculus Rift

Quelle: <http://www.chrisphilpot.com/>

Zusammen mit der Oculus Rift kann die Leap Motion jedoch auch anders angewandt werden. Die Herstellerfirma selbst hat einen Aufsatz entwickelt (Abbildung 3.2), der an das HMD *aufgeklebt* werden kann, und somit den Bereich vor dem Anwender erfasst.



Abbildung 3.2: Aufsatz für die Leap Motion

Quelle: <https://www.leapmotion.com/product/vr>

Der Vorteil bei dieser Verlinkung ist, dass beide Geräte den gleichen Ursprungspunkt und die gleiche Ausrichtung haben. Es ist deshalb ohne Weiteres möglich, die zwei Koordinatensystem aufeinander abzugleichen.

Abbildung 3.3 verdeutlicht, wie die Geräte schliesslich an das System angebunden werden. Ebenfalls erkennbar ist, dass beide per USB mit dem Host-Computer verbunden sind und Daten an entsprechende Services schicken, die auf dem Computer installiert und aktiv sind. Diese Services werden durch die in der Applikation verwendeten Plugins angesteuert, und schliesslich zur Darstellung und Steuern in IMVR verwendet.

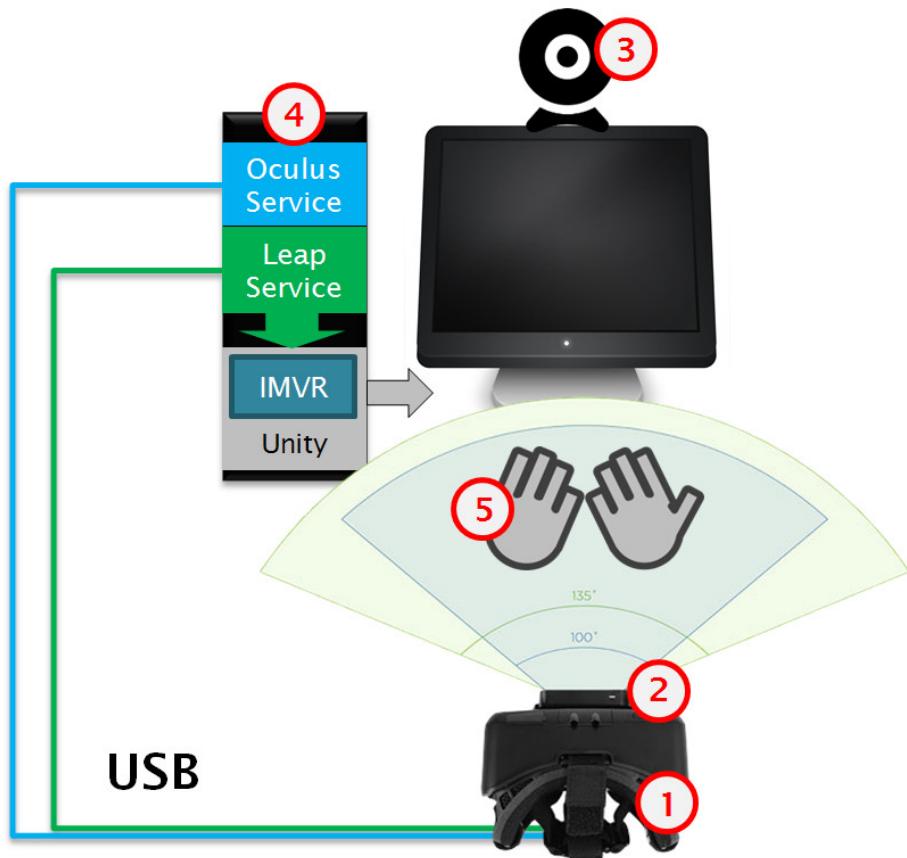


Abbildung 3.3: Eine Übersicht der Technologien und wie sie verbunden sind.

Nr.	Komponente	Beschreibung
1.	Oculus Rift DK2	HMD für den grafischen Output.
2.	Leap Motion	Gerät, welches Hände erkennt und ihre Koordinaten an den Computer sendet.
3.	Oculus Rift Kamera	Kamera, welche seit dem DK2 für das örtliche Tracking zuständig ist.
4.	Computer	Host-System für IMVR.
5.	Benutzer	Benutzer, der die Oculus Rift trägt und mit seinen Händen das Programm steuert.

3.2 Systemarchitektur

Die Komponenten sind erklärt und es sollte nun klar sein, wie die Hardware miteinander verknüpft ist. Das Kernproblem bzw. die Aufgabe dreht sich jedoch um die *Software* dazu. Diese soll ab jetzt in den Vordergrund treten.

3.2.1 Grober Ablauf

Wie in der Aufgabenstellung erwähnt, besteht das Projekt aus zwei Teilen: einem Teil zur Indexierung und einem interaktiven Teil. Diese Gliederung zieht sich durch das ganze Design der Applikation durch. So wird beispielsweise die Indexierung in einem komplett abgespaltenen Projekt durchgeführt, und gilt somit als eigenes Programm.

In Abbildung 3.4 lässt sich diese Unterteilung gut erkennen. Der Anwender startet zwei verschiedene Applikationen, die über eine zentrale Datenbank miteinander kommunizieren. Um jedoch nicht zu weit vorzugreifen, folgt zuerst eine Erklärung der einzelnen Schritte.

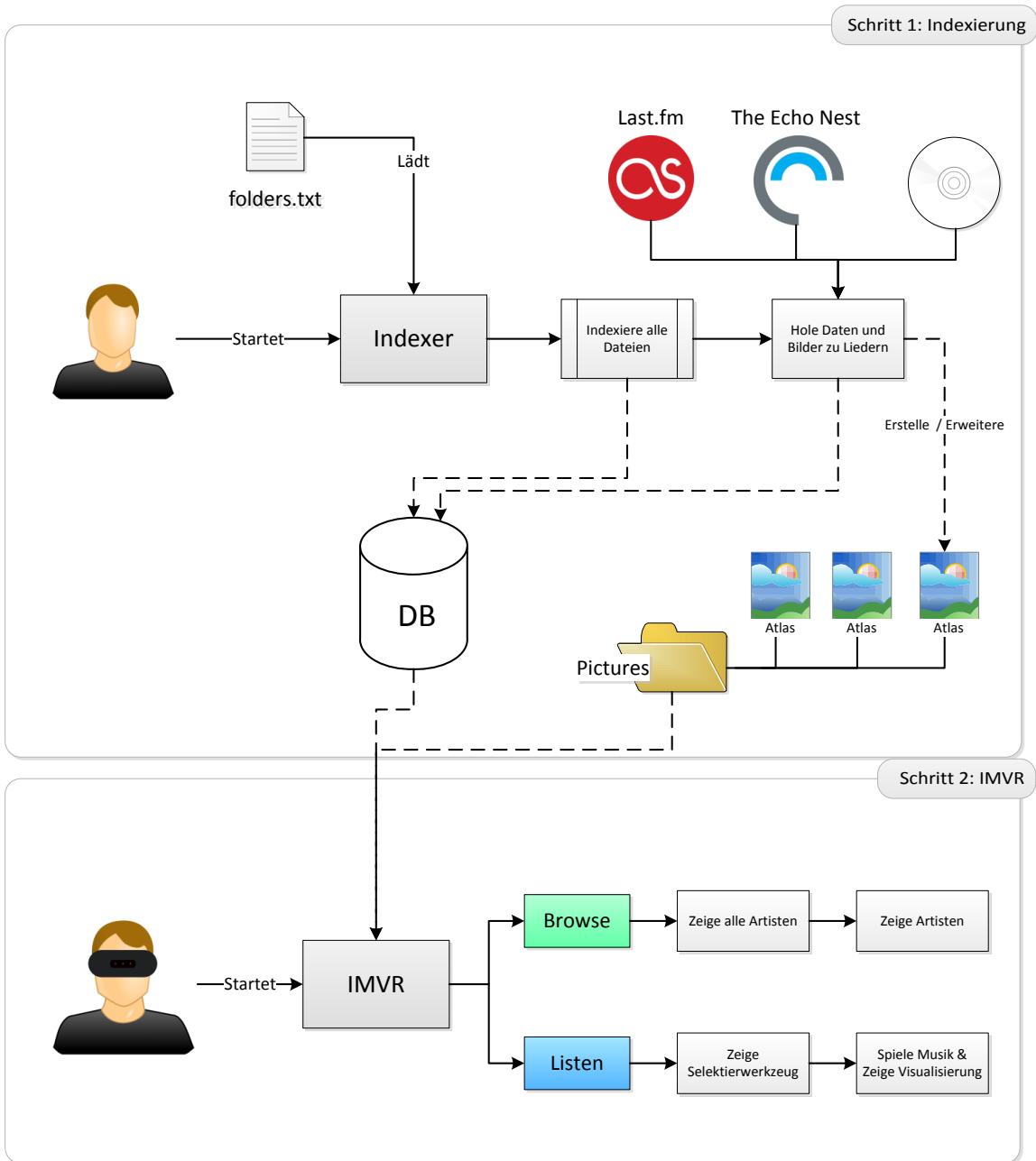


Abbildung 3.4: Grober Überblick des Programmablaufs

Man erkennt wie der Benutzer zuerst ausserhalb der Unity-Applikation den Indexer startet und durchlaufen lässt. Dieser durchläuft alle Ordner, die er in der Datei *folders.txt* findet und schreibt diese in die Datenbank. Im nächsten Schritt werden aus diversen Quellen weitere Metadaten zusammengetragen.

Was ebenfalls auf der Grafik zu sehen ist, sind die Atlassse. Um Ressourcen zu sparen (siehe Kapitel 5.5.1), werden alle Bilder in sogenannten Atlassen, sprich Bildersammlungen, gespeichert.

Die Bilder, von denen hier die Rede ist, sind Fotos der Artisten und das Artwork der indexierten Alben.

Sobald die Datenbank im ersten Schritt erstellt wurde, setzt der Anwender, wie in Kapitel 3.1 beschrieben, seine Oculus Rift mit dem Leap Motion Aufsatz auf und startet IMVR. Er erhält dann die Auswahl, welchen Modus er benutzen will und je nach Wahl entsprechende weitere Optionen.

3.2.2 Ablauf von IMVR

Abbildung 3.5 liefert einen genaueren Einblick in den Programmfluss von IMVR. Im Zentrum steht ganz klar die Wahl des Modus, welche grundsätzlich zu jedem Zeitpunkt der Applikation getätigt werden kann. Auch beendet wird die Applikation bei dieser Verzweigung.

3.3 Systemdesign

Da der Programmfluss nun klar geworden ist, bedarf es einer genaueren Erklärung der einzelnen Komponenten im System. Aufgrund der Brückenfunktion und deshalb der globalen Relevanz, wird zuerst ein Blick auf die Klassenstruktur der Datenbank geworfen. Danach wird der Aufbau des Indexers untersucht, und zuletzt schliesslich die eigentliche Applikation.

Zu beachten: Die meisten Klassen in den folgenden UML-Diagrammen sind Unity-Komponenten und erben somit von der Klasse MonoBehaviour. Um die Diagramme lesbar zu halten, wurde diese Beziehung oft ignoriert. Weiterhin wurden C#-Properties, wo zusätzliche Logik vorhanden ist, als "get_Property" und "set_Property" ausgeschrieben¹.

3.3.1 Commons (Datenbank)

Um Daten zwischen den zwei Programmteilen zu transportieren, wurde noch eine weitere Komponente entwickelt, welche als Datenbank agiert und über eine entsprechende Datenstruktur verfügt. Diese ist in Abbildung 3.6 ersichtlich.

¹Compiler-intern werden Properties als Methoden mit dieser Namensgebung dargestellt.

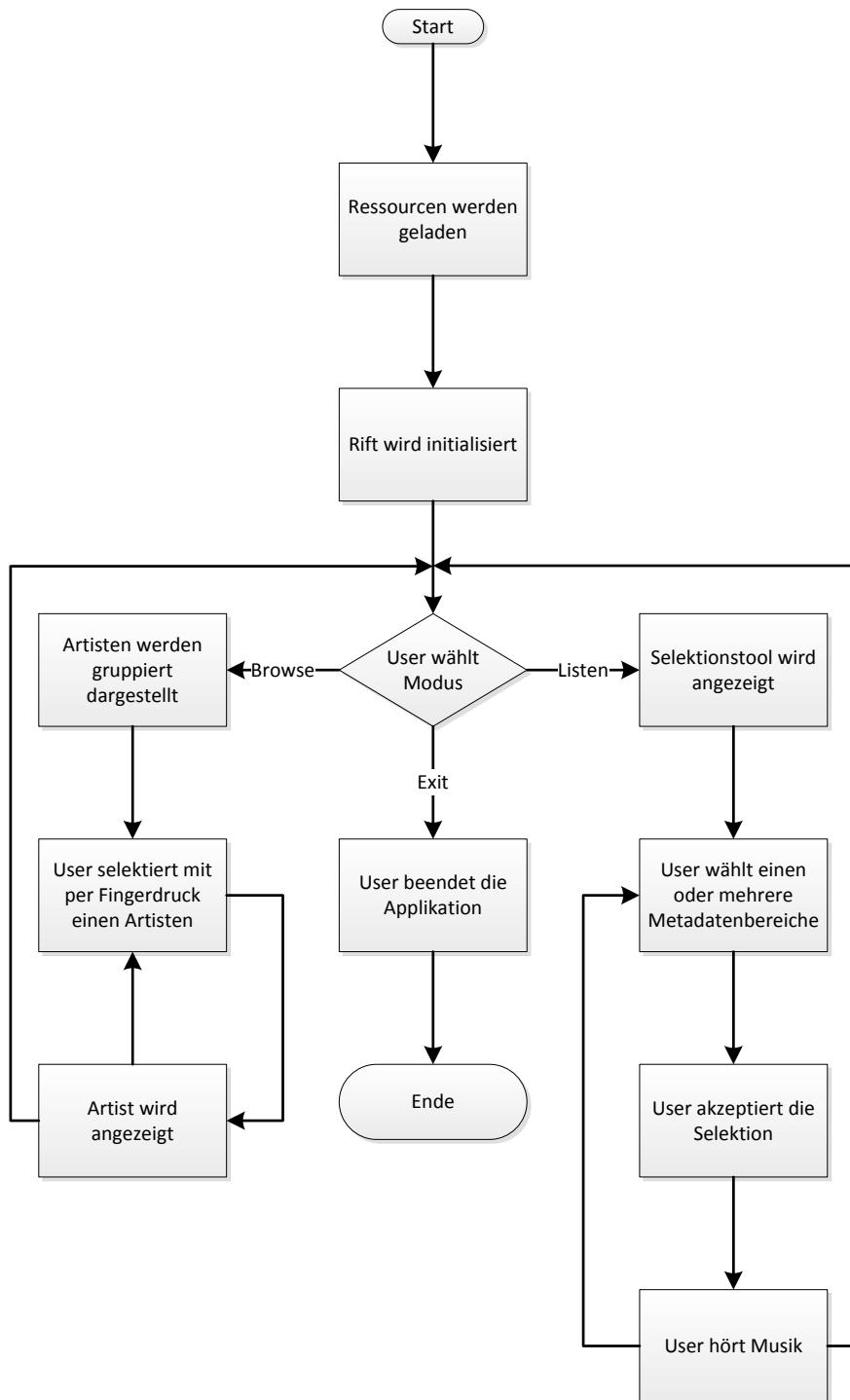


Abbildung 3.5: Linearer Ablauf von IMVR

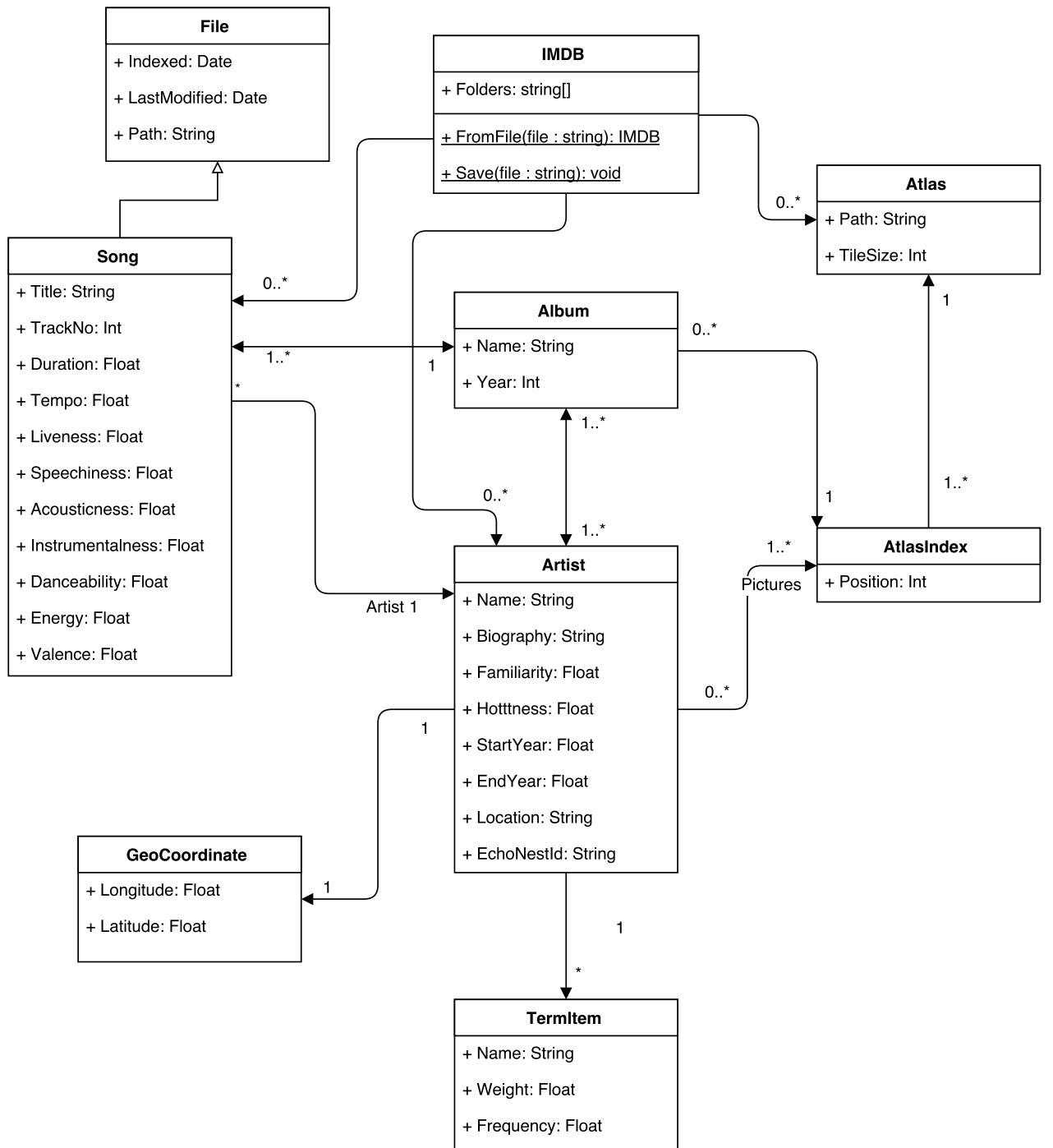


Abbildung 3.6: Klassendiagramm für die Klassen im IMVR.Commons Package

Klasse	Beschreibung
Artist	Klasse für einen einzelnen Artisten mit Biografie und ein paar Kennwerten sowie Koordinaten und Aktivitätsdaten.
Album	Jeder Artist hat ein oder mehrere Alben, an denen er beteiligt ist. Das Album selbst hat nur ein Name und ein Jahr und kann zu mehreren Künstlern gehören.
Song	Repräsentiert einen Song und gehört genau einem Künstler. Jeder Song hat einen Pfad und ist somit von einer reellen Datei gestützt. Ein Song erhält zusätzlich verschiedene Metadaten, die von Services heruntergeladen werden.
TermItem	Ein Begriff (z.B. Genre), der einem Künstler mit einem Gewicht und einer Frequenz zugeordnet wird. Könnte noch weiter normalisiert werden, wurde aber der Einfachheit halber so gelassen.
AtlasIndex	Repräsentiert eine Position bzw. ein Bild in einem Atlas. Mit der Position ist gemeint, an welcher Stelle das Bild im Atlas erscheint.
IMDB	Zentraler Zugangsknoten auf die Daten. Hält Referenzen auf die Artisten, Songs und Atlassse.

Tabelle 3.1: Erklärung der wichtigsten allgemeinen Klassen

Es handelt sich um ein bewusst sehr minimalistisch gehaltenes Schema, um eine Musiksammlung darzustellen. Zu den Problemen, die nicht abgedeckt werden gehören zum Beispiel:

- Jede Ausgabe eines Liedes gilt als ein separates Lied
- Für kombinierte Artisten (z.B. A feat. B) wird jeweils ein neuer Artist erstellt
- Mehrere Aktivitätsperioden eines Künstlers können nicht abgebildet werden

Es gäbe keine Grenzen, wenn man eine perfekte Struktur erreichen wollte, und das liegt ausserhalb des Bereichs dieses Projektes.

Die Metadaten, die in der Klasse Song erkennbar sind, stammen aus der The Echo Nest API und sind wie folgt zu verstehen:

Name	Wertebereich	Bedeutung
Danceability	0..1	Der Wert beschreibt, "wie gut es sich zu dem Song tanzen lässt". Technisch gesehen, wird das Tempo, die rhythmische Stabilität und die allgemeine Regularität in Betracht für den Wert gezogen.
Energy	0..1	Beschreibt, wie energievoll ein Song ist bzw. wie schnell, laut und lärmig er sich anhört.
Speechiness	0..1	Erkennt gesprochene Audiodateien wie z.B. Hörbücher. Dateien mit einem Wert von über 0.66 sind mit grosser Wahrscheinlichkeit vollständig gesprochen, während Dateien mit einem Wert von über 0.33 auch Musik enthalten.
Liveness	0..1	Erkennt, ob Zuhörer in einem Lied hörbar sind. Live-Aufnahmen machen sich mit einem Wert von über 0.8 erkennbar.
Acousticness	0..1	Gibt an, wie hoch die Wahrscheinlichkeit ist, dass ein Song ausschliesslich mit akustischen Mitteln - also ohne Synthesizer, Verstärker und nachbearbeiteten Gesang - aufgenommen wurde.
Valence	0..1	Beschreibt die ungefähre Stimmung eines Liedes. Eine Audiodatei mit einem hohen Wert macht einen fröhlicheren Eindruck, als eine Datei mit einem tiefen Wert.
Tempo	$\mathbb{R}_{\geq 0}$	Beschreibt die BPM eines Liedes.

Tabelle 3.2: Erläuterung der Metadaten

3.3.2 Indexer

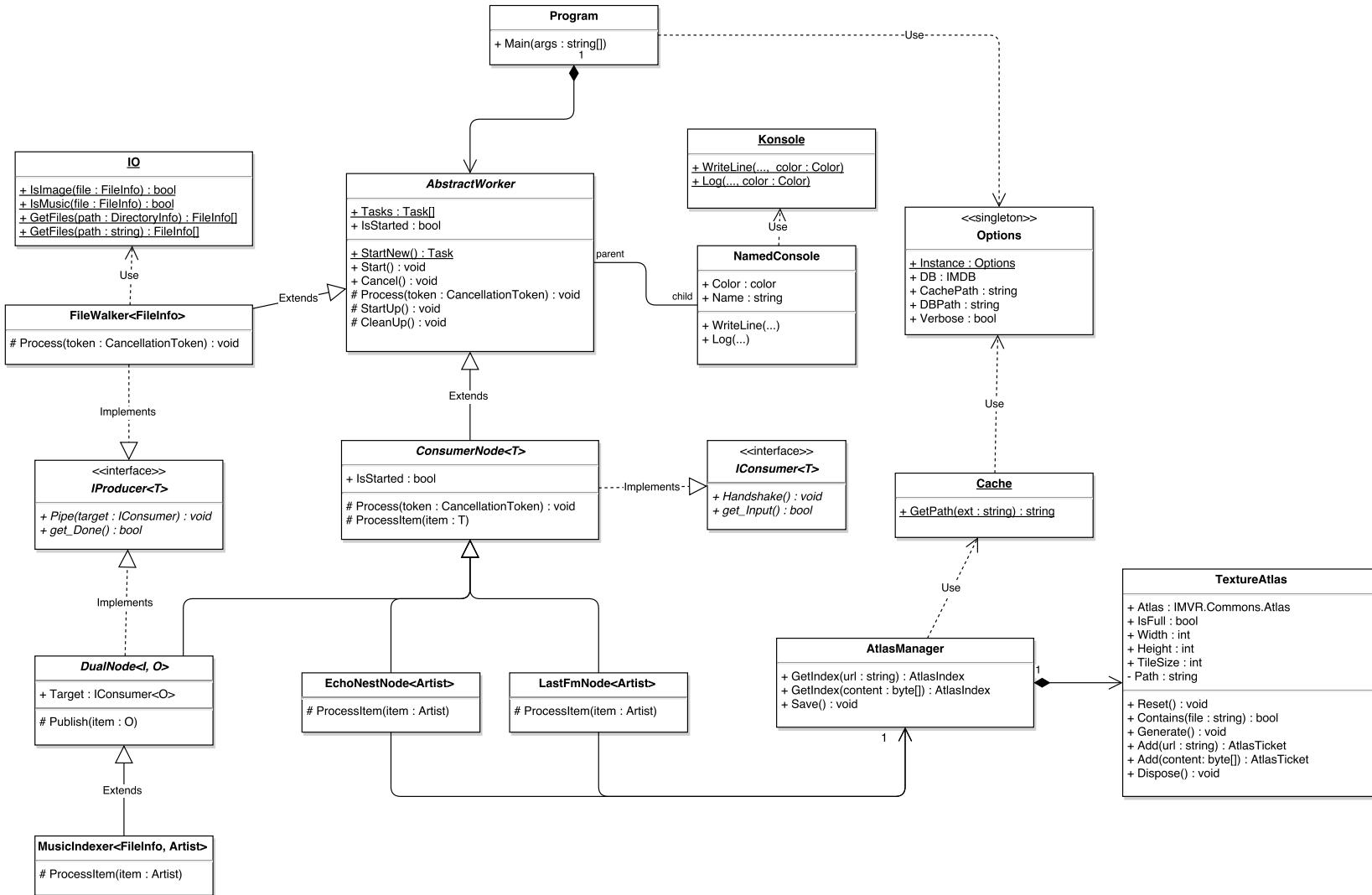


Abbildung 3.7: Klassendiagramm des Indexers

Der Indexer basiert auf *Nodes*. Das Konzept wird dabei mithilfe eines Producer-Consumer Patterns abgebildet.

Alle Nodes im System führen eine Aufgabe auf einem oder mehreren Threads durch. Wenn diese einen Input haben, implementieren sie das IConsumer-Interface oder eine der abstrakten Implementationen davon, und wenn diese einen Output haben, implementieren sie das IProducer-Interface, um die Daten an den nächsten Node weiterzugeben.

Klasse	Beschreibung
Options	Singleton-Klasse, welche mit den Command-Args gebildet wird und die Konfigurationen beinhaltet.
IProducer	Ein Produzent von Daten (→ Output-Knoten)
IConsumer	Ein Konsument von Daten (→ Input-Knoten)
AbstractWorker	Jede Node erbt von dieser Klasse. Sie sorgt dafür, dass die Nodes auf eine einheitliche Weise initialisiert, durchgeführt und aufgeräumt werden können.
ConsumerNode	Hilfsklasse, welche IConsumer implementiert und mithilfe einer Methode ProcessItem den Einsatz erleichtert.
DualNode	Hat die gleiche Funktion wie eine ConsumerNode, aber implementiert zusätzlich IProducer und bietet mit Publish ebenfalls eine Hilfsmethode an.
EchoNestNode	Holt Daten aus der The Echo Nest API und schreibt diese in die Datenbank.
LastFmNode	Holt Daten aus der Last.fm API und schreibt diese in die Datenbank.
MusicIndexer	Konsumiert Files und extrahiert daraus Artisten, Alben und Songs.
FileWalker	Durchläuft die Medienordner und findet Musik und Bilder.

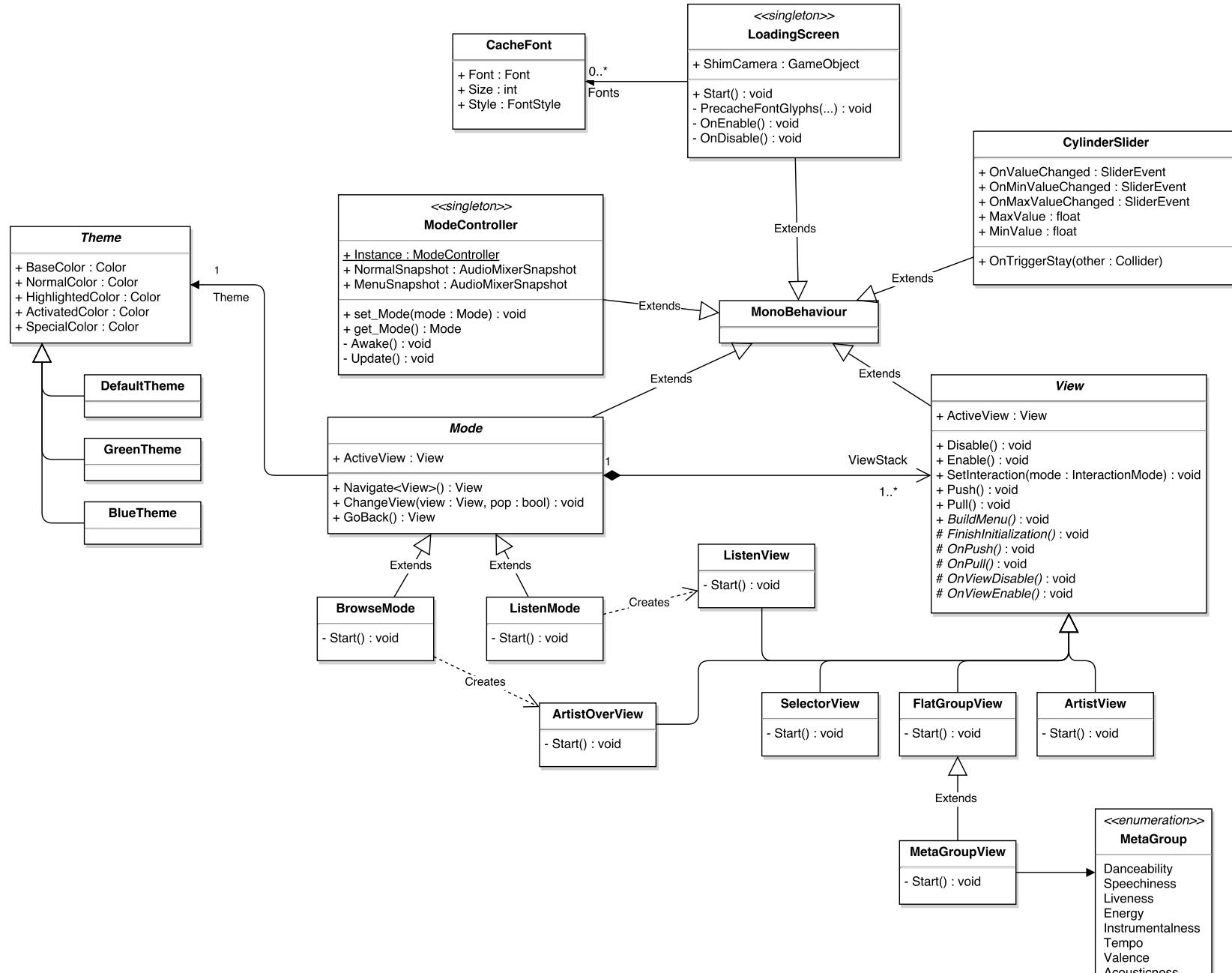
Tabelle 3.3: Erklärung der wichtigsten Klassen des Indexers

3.3.3 IMVR

Nun, da das Design des Indexers festgelegt ist, bedarf auch die Hauptapplikation einer Erklärung. Um das Design besser verständlich zu machen, wurde hierbei das Schema in verschiedene, logische Teilbereiche gegliedert, die auch den Namespaces des Projekts entsprechen.

Es wird empfohlen, diese Schemas zusammen mit den Erklärungen in Kapitel 5 zu verwenden.

3.3.3.1 Interface



Das Interface von IMVR basiert auf sogenannten *Views*. Zu jedem Zeitpunkt der Applikation ist irgendeine View aktiv, die verantwortlich für die momentane Darstellung der Szene ist.

Die Views gehören und werden kontrolliert vom momentan aktiven *Mode*, wovon zwei existieren: der *BrowseMode* und der *ListenMode*. Der aktuelle Modus wird verwaltet durch die Klasse *ModeController*. Das Wechseln des Modus ist Aufgabe des Ring-Panels und wird in einem separaten Kapitel behandelt.

Ein weiteres relevantes Detail ist das *Theme*, welches den Modi zugeordnet wird. Dieses Theme enthält eine Anzahl von Farben, die später von anderen GUI-Elementen verwendet werden und für eine leichtere Unterscheidung des momentanen Modus sorgen sollen.

Klasse	Beschreibung
ModeController	Führt Buch über den momentan aktiven Modus.
Mode	Verfügt über einen eigenen Stack von Views, durch die der Anwender navigiert.
BrowseMode	Modus, in dem der Anwender gezielt durch seine Musiksammlung navigiert, indem ihm eine Übersicht aller Künstler angezeigt wird. Beginnt mit der View <i>ArtistOverView</i> .
ListenMode	Modus, in dem der Anwender nur die Art der Musik angibt, die er hören will. Die Applikation übernimmt den Rest. Beginnt mit der View <i>SelectorView</i> .
Theme	Eine Sammlung von Farben, die einem Modus zugeordnet wird.
View	Eine "Ansicht", die aktiviert und deaktiviert sowie nach vorne und nach hinten geschoben werden kann.
ArtistOverView	Stellt einen gruppierten Zylinder mit den Künstlern in der Musiksammlung dar.
ArtistView	Stellt einen einzelnen Artisten und seine Alben dar.
SelectorView	Stellt ein Auswahlwerkzeug dar, mit dem der Anwender eine Musikart selektieren kann.
ListenView	Verwaltet die visuellen Effekte beim Hören von Musik im Listen-Mode.

Tabelle 3.4: Erklärung der wichtigsten Interface-Klassen

3.3.3.2 Music Arm

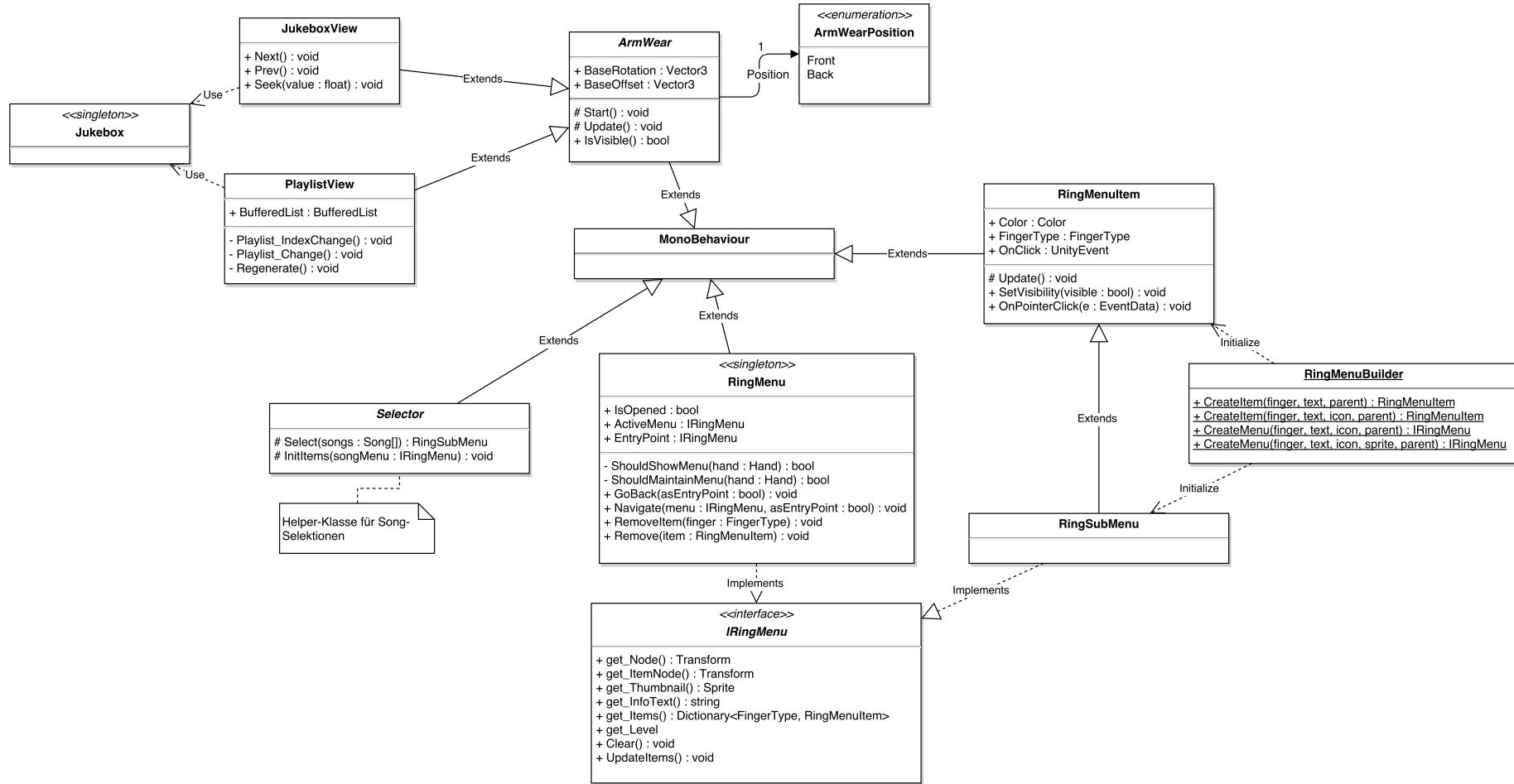


Abbildung 3.9: Klassendiagramm des Music Arms

Das eigentliche Menü, der *Music Arm*, befindet sich im IMVR.Interface.MusicArm Namespace. Er verwaltet die momentan abzuspielende Musik und die selektierten Lieder.

Beim Design der Klassen war ursprünglich ein viel allgemeinerer Ansatz für den Music Arm geplant. Konkret gesagt, wurden die Klassen so gestaltet, dass das Ring-Menü, welches zum Einsatz kommt, im Editor für *irgendeine* Aufgabe konfiguriert werden kann.

In einem gewissen Stadium des Projekts wurde jedoch der Music Arm konzipiert und das Ring-Menü stark umstrukturiert. Neu erhält jede View beim Laden die Möglichkeit, mithilfe der Helper-Klasse RingMenuBuilder ein eigenes Menü zu gestalten, wobei die Belegung des Zeige-, Mittel- und Ringfingers vorgegeben ist.

Klasse	Beschreibung
RingMenu	Die zentrale Klasse des Music Arms und gleichzeitig auch die Root-Ebene des Menüs. Existiert nur einmal, verwaltet, welche Stufe momentan angezeigt wird, und sorgt für das Anzeigen und Verstecken des Menüs.
RingMenuItem	Ein einzelner Menüeintrag im Menü. Platziert sich in den entsprechenden Finger wenn sichtbar und hat einen Event-Handler für Click-Events (in diesem Fall Auswahl des Menüs).
IRingMenu	Interface für Menüs mit Einträgen.
RingSubMenu	Implementiert IRingMenu und funktioniert wie RingMenu, nur ist es selbst auch ein Eintrag in einem anderen IRingMenu.
RingMenuBuilder	Hilft beim dynamischen Erstellen von neuen Menüeinträgen.
ArmWear	Beschreibt eine Komponente, die am Arm "getragen" werden kann.
JukeboxView	Zeigt das momentane Lied in der Jukebox an und wird auf der Frontseite des Arms getragen.
PlaylistView	Zeigt die momentan Playlist an und befindet sich auf der Rückseite des Arms.
Selector	Hilft Komponenten, die im Ring-Menü eine Songselektion machen wollen, dabei.

Tabelle 3.5: Erklärung der wichtigsten Music-Arm-Klassen

3.3.3.3 Ring Panel

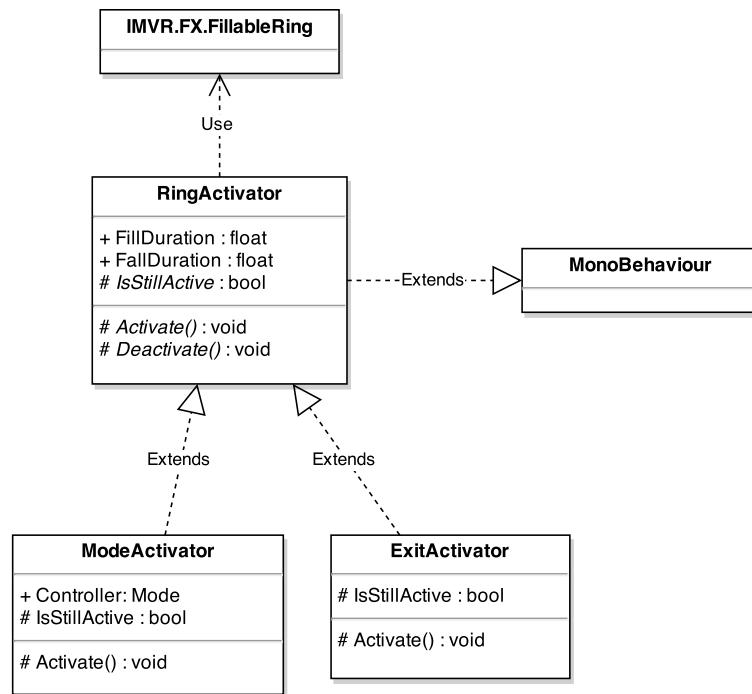


Abbildung 3.10: Klassendiagramm des Ring Panels, womit der Anwender Modi auswählt

Das Ring-Panel befindet sich unter den Füßen des Anwenders und wird mit dem Blick bedient. In diesem Namespace befinden sich nur die Klassen, die für die Logik zuständig sind - der visuelle Teil ist bei den Effekten (IMVR.FX) zu finden.

Klasse	Beschreibung
RingActivator	Abstrakte Klasse, die visuell von einem sich füllenden Ring-Mesh dargestellt wird, und beim anhaltenden Blick aktiviert wird.
ModeActivator	Eine Unterklasse des RingActivators, die nur dazu zuständig ist, einen Modus zu aktivieren (siehe <i>IMVR.Interface</i>).
ExitActivator	Ein weiterer Activator, der beim Aktivieren die Applikation beendet.

Tabelle 3.6: Erklärung der Klassen des Ring-Panels

3.3.3.4 Daten Handling

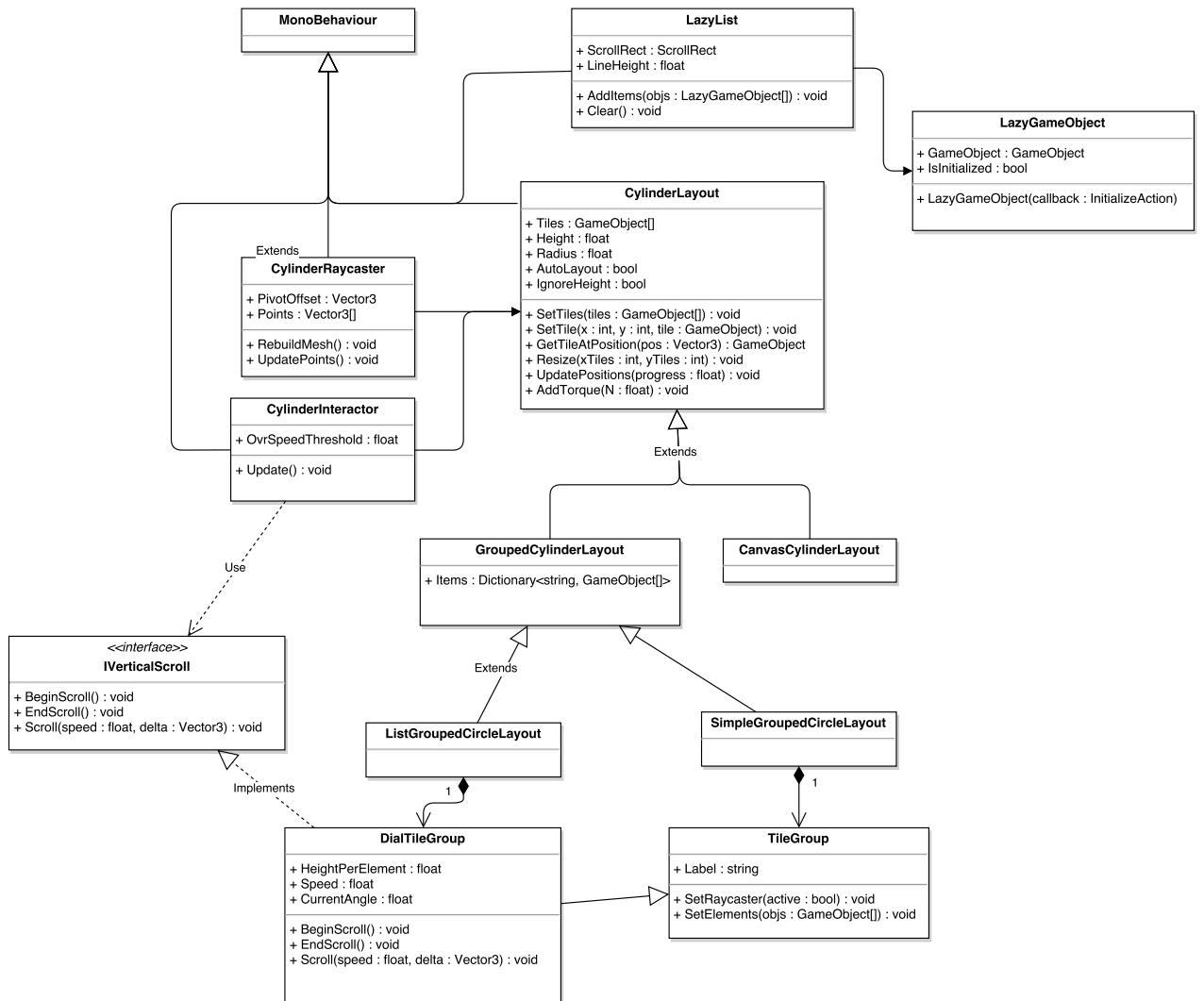


Abbildung 3.11: Klassendiagramm des Teils, der die Daten der Datenbank verwendet

Die Klassen, die sich mit den Daten aus der Datenbank bzw. der Darstellung dieser beschäftigen, befinden sich im Namespace `IMVR.Data`.

Der Namespace ist gegliedert in `Data`, `Data.Music` und `Data.Image`. Ersterer befasst sich mit der generellen Darstellung der Daten, der zweite kümmert sich um das Arbeiten mit Musik, und `Data.Image` schliesslich um Aufgaben mit Bildern.

Ursprünglich war geplant, einen Modus nur für die Bilderbibliothek des Anwenders zu entwickeln, und die dabei verwendeten Klassen in den entsprechenden Namespace zu platzieren. Ein grosser Teil dieser Klassen wurde jedoch zusammen mit dem Modus selbst gestrichen. Übrig geblieben sind die Klassen, die noch Verwendung bei der Musik finden (z.B. für das Darstellen von Artwork).

Eine weitere Klasse, die weiterhin guten Dienst leistet, ist beispielsweise die Klasse `CylinderRaycaster`. Deren Aufgabe war ursprünglich, die hohe Anzahl Bilder handhabbar zu machen, weil einzelnes

Klasse	Beschreibung
CylinderLayout	Komponente für die allgemeine Darstellung von GameObjects in einem Zylinder-Layout. Kann per CylinderInteractor rotiert werden und verfügt über einen optimierten Raycaster.
GroupedCylinderLayout	Ein spezielles CylinderLayout, welches die Elemente gruppiert mit jeweils einem Label darstellt.
SimpleGroupedCylinderLayout	Ein gruppiertes CylinderLayout, welches auf normale GameObjects spezialisiert ist.
ListGroupedCylinderLayout	Ein gruppiertes CylinderLayout, welches auf Listen von Canvas-GameObjects spezialisiert ist.
CylinderInteractor	Kümmert sich um die User-Interaktion mit dem Layout. Hilft beim Rotieren und vertikalen Scrollen.
CylinderRaycaster	Sorgt für ein effizientes Raycasting der Elemente im Layout. Wird wichtig, wenn über 100 interaktive Elemente im Zylinder verteilt sind.
LazyList	Hilft bei der effizienten Darstellung der Playlist (siehe Kapitel 5.5.4).

Tabelle 3.7: Erklärung der wichtigsten Klassen von IMVR.Data

Raycasting zu viel Zeit beanspruchen würde. Da Performance immer ein Problem ist, erweist sich diese weiterhin als nützlich.

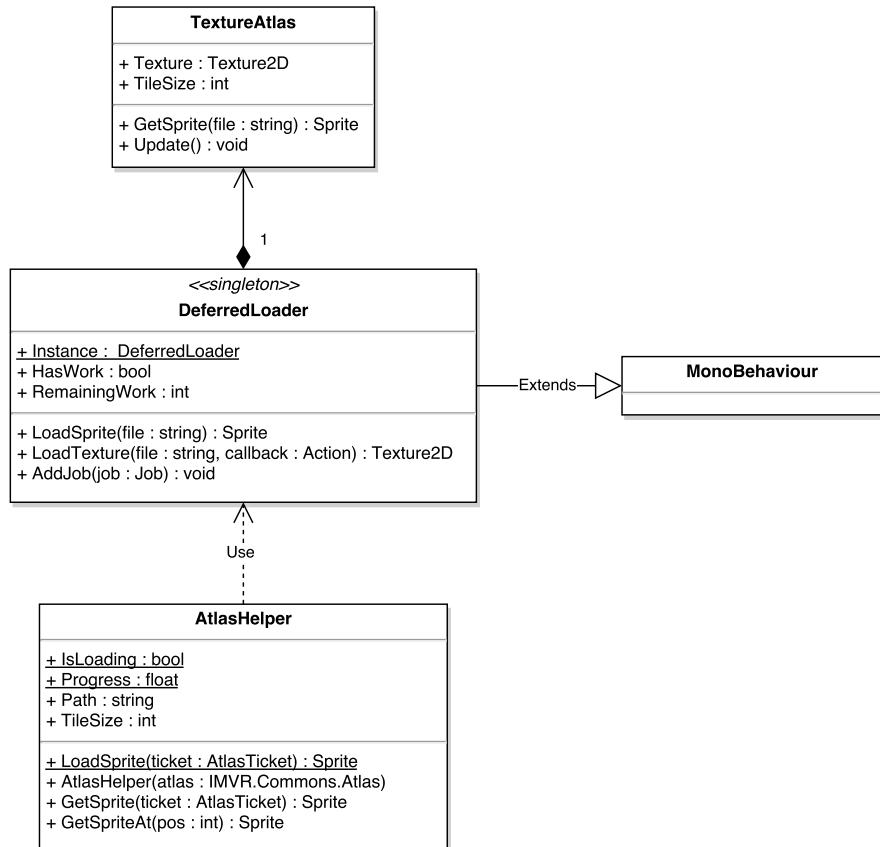


Abbildung 3.12: Klassendiagramm des Bilddaten-Handlings

Der Teil, der für die Bilder zuständig ist, enthält drei Klassen: eine Helper-Klasse, die auf einer hohen Ebene für das einfache Laden von Atlassen aus IMVR.Commons sorgt, eine intelligente Klasse zum Laden von Bildern und eine tiefe Klasse, welche direkt mit den Daten der Atlassse arbeitet.

Klasse	Beschreibung
AtlasHelper	Hilft beim Laden von Atlassen aus der Datenbank, indem er logische Methoden fürs Laden liefert. Gibt zudem Auskunft über die aktiven Ladevorgänge.
DeferredLoader	Lädt und verwaltet Atlassse vom Filesystem, und liefert Texturen und Sprites für den Zugriff darauf.
TextureAtlas	Ein einzelner Texturenatlas auf dem Filesystem mit einer bestimmten Kachelgrösse.

Tabelle 3.8: Erklärung der Klassen von IMVR.Data.Image

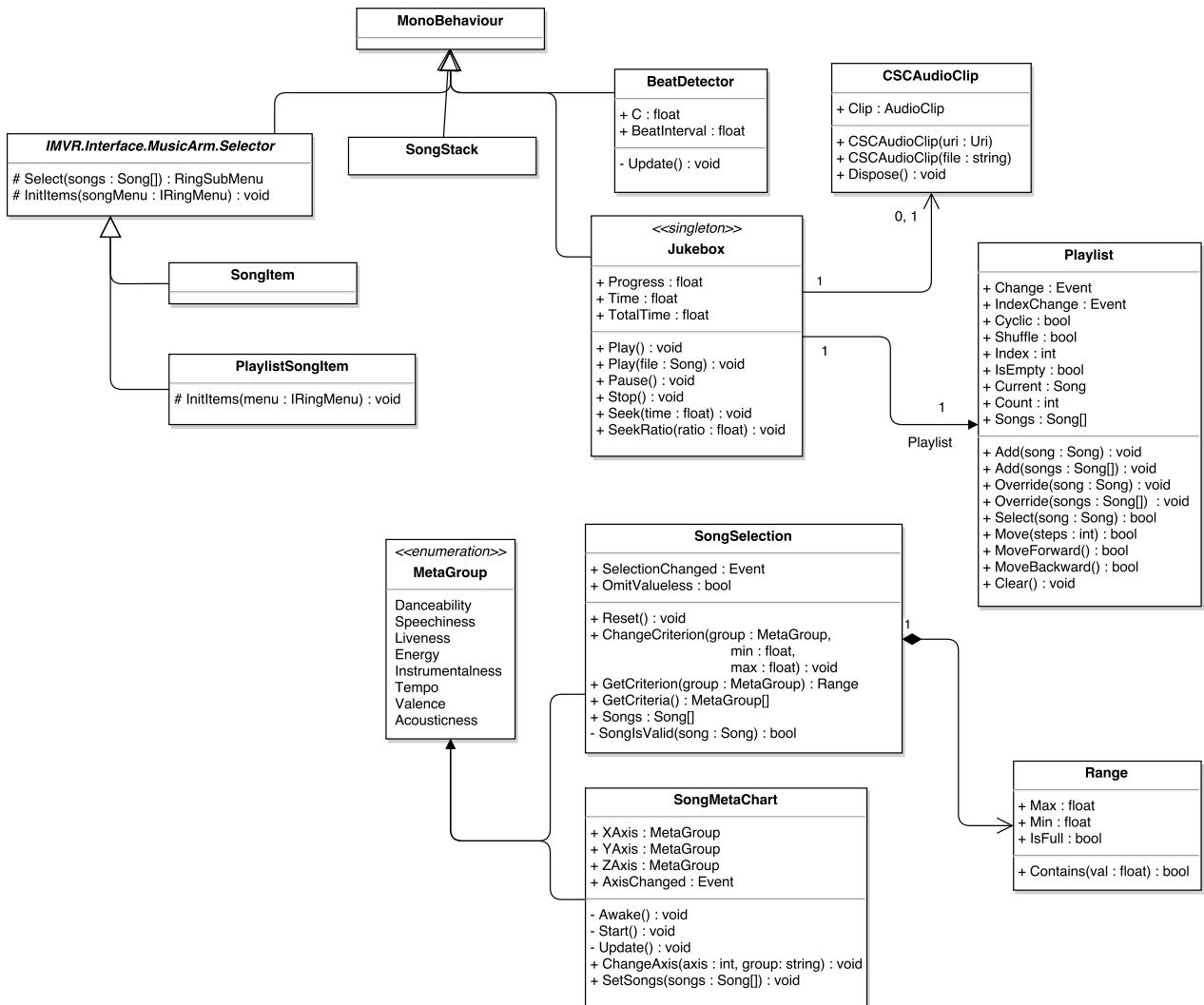


Abbildung 3.13: Klassendiagramm des Musikdaten-Handlings

Im Namespace für die Musik gibt es zwei Arten von Klassen: die Jukebox und die Klassen zur Darstellung von Musikdaten.

Klasse	Beschreibung
Jukebox	Kontrolliert die momentane Wiedergabe und reagiert auf Events der Playlist.
Playlist	Eine Liste von Songs, die für die Wiedergabe vorgesehen sind. Hat eine momentane Selektion, die geändert werden kann. Sender Events beim Ändern der Liste bzw. der momentanen Selektion.
SongItem	Zeigt ein Lied an.
PlaylistSongItem	Zeigt ein Lied an, optimiert für die Anzeige in PlaylistView.
SongMetaChart	Steuert die FX-Klasse PointChart um damit Songmetriken dreidimensional darzustellen (siehe Kapitel 5.4.2)
SongSelection	Enthält eine Ansicht auf die Musikdaten, die durch Auswahlbereiche in verschiedenen MetaGroups eingeschränkt ist. Jede MetaGroup ist ein Kriterium mit einem Wert im Bereich [0..1].

Tabelle 3.9: Erklärung der wichtigsten Klassen von IMVR.Data.Music

3.3.3.5 Effekte

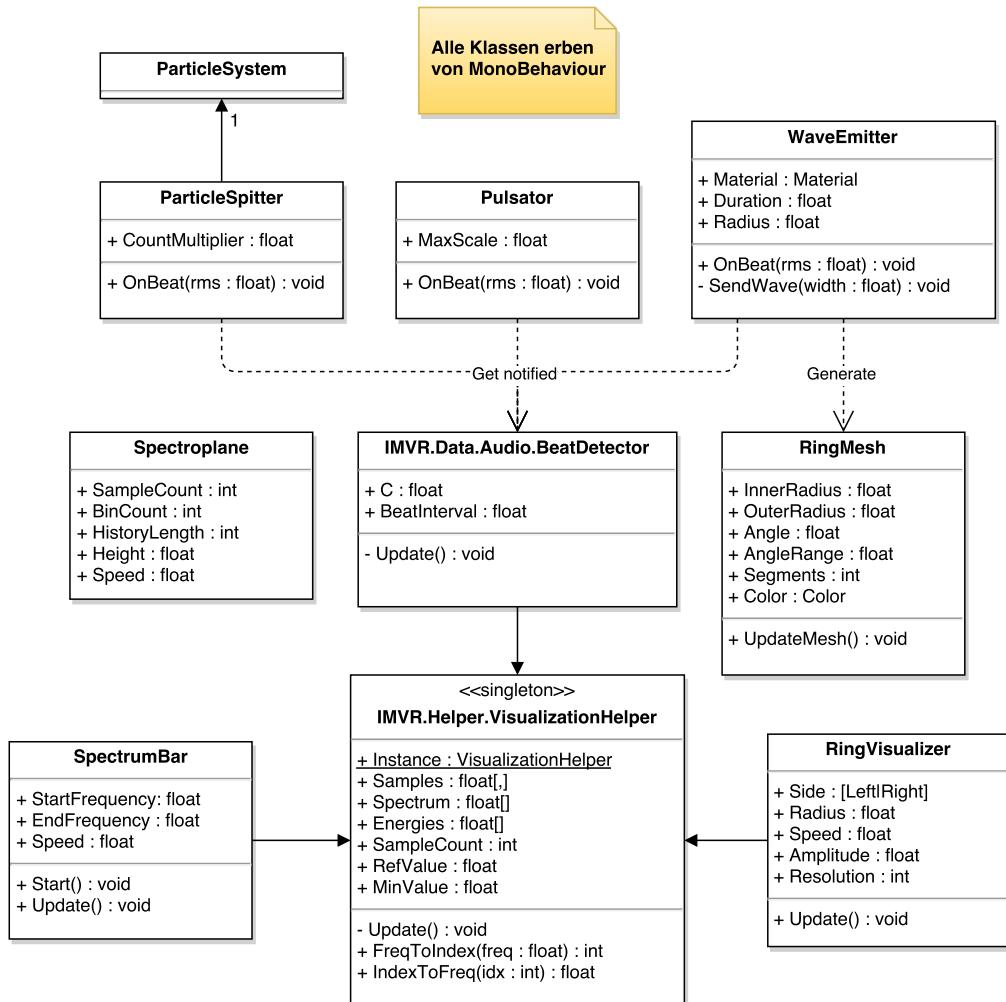


Abbildung 3.14: Klassendiagramm der Effekte zur Musikvisualisierung

Der visuelle Aspekt der Applikation ist im Namespace *IMVR.FX* aufbewahrt. Hier befinden sich die prozeduralen Meshes und die Partikelsysteme. Was man hier nicht finden wird, sind die visuellen Elemente, die direkt für das Interface verwendet werden. Die Klassen in *IMVR.FX* haben eine unterstützende Funktion.

Auch in diesem Namespace gibt es Klassen in leicht unterschiedlichen Anwendungsszenarios. In einem Szenario geht es darum, Musik zu visualisieren wie es z.B. ein Musik-Player tut.

Klasse	Beschreibung
VisualizationHelper	Steht im Mittelpunkt der Musik-Visualisierung. Verwaltet die Sample- und Spektrumdaten, die in Echtzeit aus der Musik gewonnen werden.
SpectrumBar	Ein Mesh, welches je nach Stärke des Signals in einem bestimmten Frequenzbereich, höher steigt bzw. tiefer sinkt.
RingVisualizer	Eine kreisförmige, kontinuierliche Linie, welche die Lautstärke eines Kanals (links oder rechts) darstellt.
BeatDetector	Klasse zur Erkennung von Beats in der wiedergegebenen Musik.
Pulsator	Komponente, welche ein Objekt im Takt der Musik pulsieren lässt.
WaveEmitter	Sendet im Takt der Musik Wellen in entsprechender Stärke aus (siehe Kapitel 5.4.4)
ParticleSpitter	Wirft im Takt der Musik Partikel in die Szene.

Tabelle 3.10: Erklärung der FX-Klassen, die sich mit der Musik-Visualisierung befassen

Neben diesen Visualisierungsklassen gibt es auch Komponenten, die sich mit Effekten befassen, die nicht von der momentanen Musik abhängen.

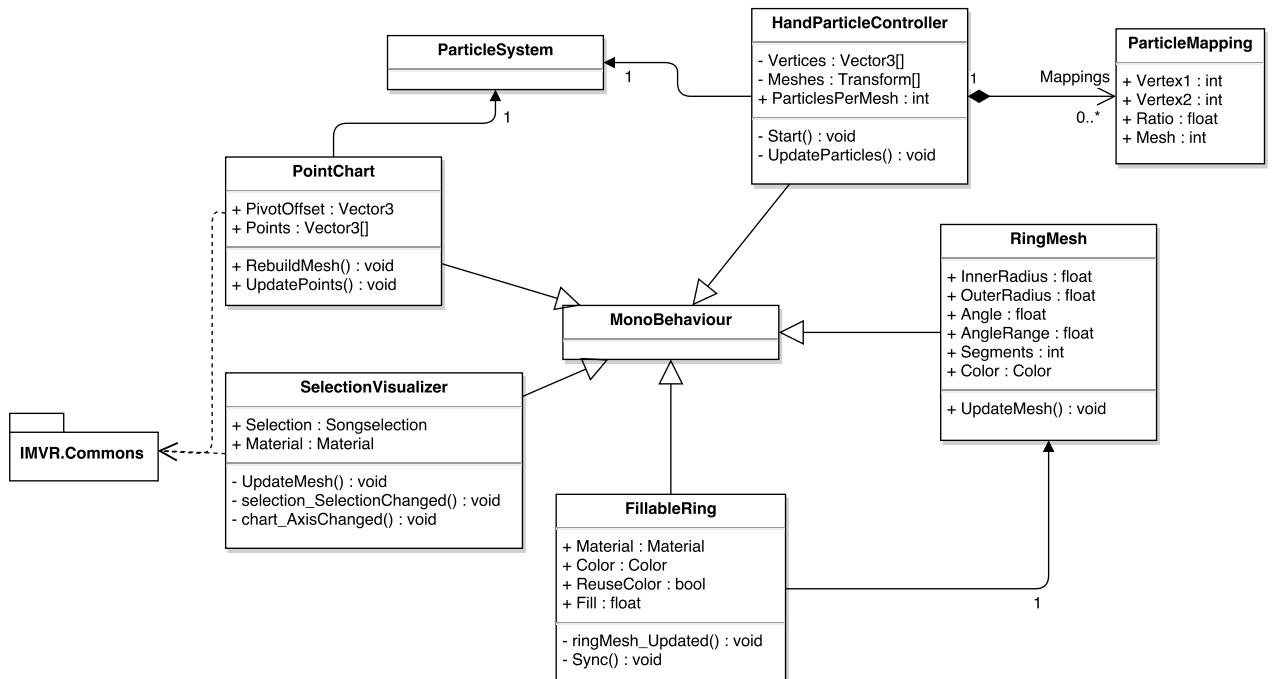


Abbildung 3.15: Klassendiagramm der allgemeinen Effekte

Klasse	Beschreibung
RingMesh	Ein zweidimensionales Mesh, welches einen beliebigen Kreisring darstellen kann (siehe Kapitel 5.4.4).
FillableRing	Erweitert ein Ring-Mesh mit einer Füllung, die mit der Property Fill auf einen Wert im Bereich [0..1] eingestellt werden kann.
PointChart	Stellt ein dreidimensionales Point-Chart dar, welches auf Partikel aufbaut (siehe Kapitel 5.4.2).
HandParticleController	Kontrolliert die Positionen und Bewegungen der Partikel einer Handseite (siehe Kapitel 5.4.1).
ParticleMapping	Mappt einen Partikel auf eine Kombination aus zwei Vertices.

Tabelle 3.11: Erklärung der allgemeinen FX-Klassen

3.3.3.6 Gesten

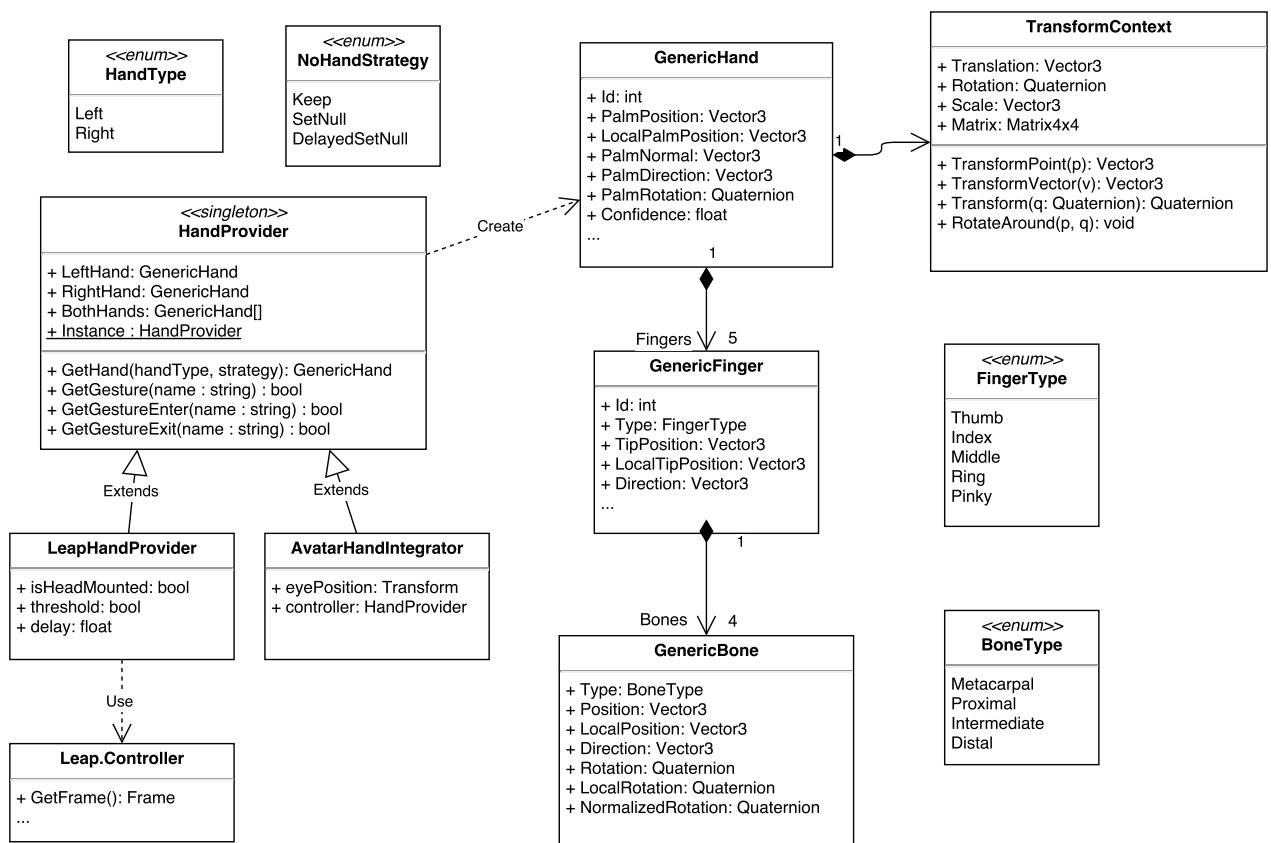


Abbildung 3.16: Klassendiagramm des Abstraktionslayers für die Leap Motion

Im Vorprojekt wurde ein Abstraktionslayer für die Hände und Gesten der Leap Motion entwickelt. Dabei wird die in diesem Fall sinnvolle Beschränkung eingeführt, dass nur *eine* linke und *eine* rechte

Hand zu jedem gegebenen Zeitpunkt in der Applikation existieren kann.

Ausserdem wird ein einfacher Zugriff auf die Handdaten per HandProvider möglich, welcher auch ein eigenes System zur Gestenerkennung umfasst. Für eine genauere Detaillierung der Funktionsweise dieses Abstraktionslayers sei auf den Schlussbericht der Vorarbeit verwiesen.

4 Implementation des Indexers

Ein Teil der Arbeit war es, einen Indexer zu implementieren, der die Dateien des Benutzers durchläuft und - wie es der Name vermuten lässt - Audiodateien indexiert. In diesem Kapitel soll auf den Aufbau und die Implementationsdetails eingegangen werden.

4.1 Aufbau

Der Indexer sowie die Datenstruktur und alle anderen Tools, die eine ergänzende Funktion zur Hauptapplikation haben, wurden in einer separaten Visual Studio Solution zusammengefasst. In dieser befinden sich vier Projekte:

Tabelle 4.1: Die Projekte in Auxiliary Tools

Name	Beschreibung
IMVR.Commons	DLL-Projekt, welches die Klassen enthält, die zwischen Applikation und Indexer geteilt werden.
IMVR.Commons.Tests	Testprojekt mit Unit-Tests um die Integrität der Daten sicherzustellen.
IMVR.Indexer	Konsolenprojekt, welches die Musikdateien auf dem Host-System indexiert.
IMVR.SpeechServer	Konsolenprojekt, welches in der Vorarbeit verwendet wurde und in dieser Arbeit keine Verwendung fand.

Der Indexer (IMVR.Indexer) ist als parallelisiertes, knotenbasiertes System konzipiert worden. Die Parallelität wurde deshalb gewählt, weil es beim Einholen der verschiedenen Datenquellen teils zu Wartezeiten kommt, die gut anders genutzt werden können. Dieser Faktor kam besonders ins Spiel als noch zusätzlich zu den Musikdaten auch Bilderdaten gesammelt worden sind.

Ein weiterer Grund für die Parallelisierung ist, dass als mögliches Feature eine real-time Indexierung vorgesehen war, die es letztendlich allerdings nicht in das fertige Programm schaffte. Weitere Informationen zu diesem Thema werden in Abschnitt 4.4 gegeben.

4.2 Datenquellen

Da IMVR zum Ziel hat, die Musik des Benutzers in verschiedenen Formen und Farben darzustellen, werden Daten von diversen Quellen benötigt. Wir leben in einer wundervollen Zeit in Sachen Datenvielfalt: Es gibt viele Online-Services, welche zu einem Grossteil gratis sind, von denen man diverse Daten erhalten kann.

Es folgt eine kurze Zusammenstellung von untersuchten Datenquellen.

Tabelle 4.2: Eine Übersicht von verfügbaren Online-Datenquellen.

Name	Daten	API-Limite
Last.fm	Bilder, Meta-Daten	5 Request / Sekunde
The Echo Nest	Bilder, Meta-Daten, Analyse-Daten	120 Requests / Minute
Spotify	Bilder, Meta-Daten, Playlisten, Streams	?
Amazon	Bilder, Beschreibungen	1 Request / Sekunde
Gracenote	Bilder, Fingerprinting, Meta-Daten	~1000 Requests / Tag
MusicBrainz	Bilder, Meta-Daten	~1 Request / Sekunde

Im Falle von IMVR kommen die Daten grundsätzlich von drei verschiedenen Quellen:

- ID3 Tags der Musik-Dateien
- The Echo Nest
- Last.fm

Diese wurden gewählt aufgrund der durchsichtigen API-Limite und den verfügbaren Daten. The Echo Nest aggregiert zudem die Daten von anderen Seiten wie Spotify und MusicBrainz, und enthält wertvolle Analyse-Daten, die eine zentrale Position in IMVR haben.

In einem ersten Schritt werden grundlegende Daten wie der Titel des Liedes, der Name des Artisten, usw. aus der Datei selbst entnommen. Wenn möglich wird auch gleich geprüft, ob ein Album-Cover hinterlegt ist.

In einem zweiten Schritt wird über die .NET Bibliothek *echonest-sharp* [1], welche im Rahmen des Projektes geforkt und erweitert wurde, zur API von The Echo Nest verbunden und diverse Meta-Daten zur Musik heruntergeladen.

Was bei der Originalimplementation leider fehlt, sind neuere Features wie Genres und Ortsdaten, sowie ein intelligenter Bremsalgorithmus, der dafür sorgt, dass die Datenlimite eingehalten wird. Deshalb wurde ein Fork erstellt, der diese Daten und Funktionalitäten ergänzt¹.

¹https://github.com/EusthEnoptEron/echonest-sharp/tree/additional_apis

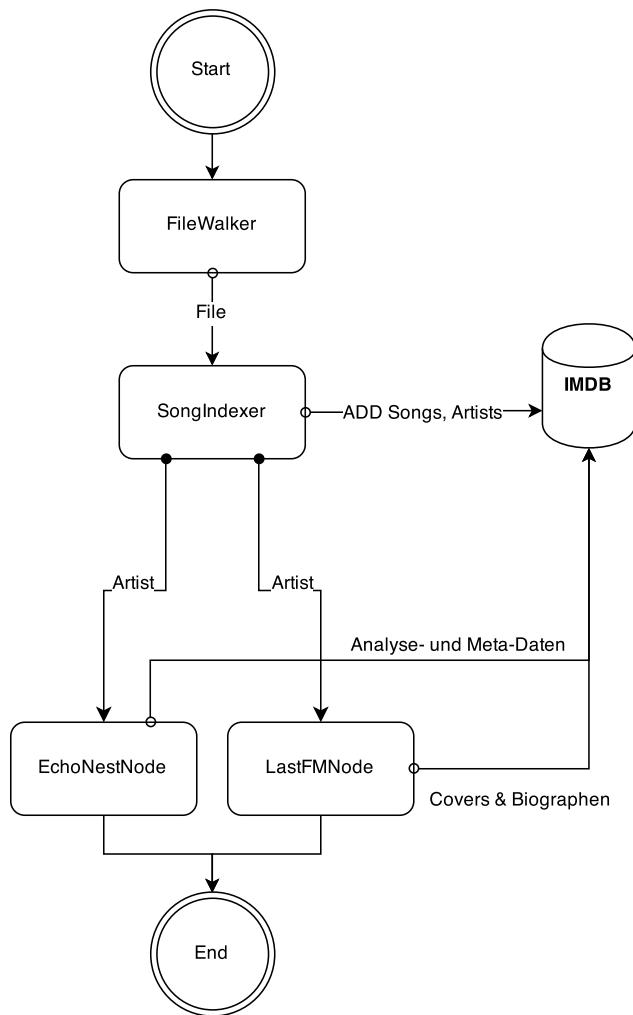


Abbildung 4.1: Der Datenfluss, den die Files beim Indexieren nehmen.

4.3 Datenstruktur

Für die Abspeicherung der Daten wurde zuerst ein Ansatz gewählt, der auf einer SQLite Datenbank basierte. Diese wurde in einem Prototyp auch erfolgreich implementiert. Es stellte sich jedoch heraus, dass diese Abhängigkeit das Programm unnötig verkomplizieren würde und eine simple In-Memory Datenstruktur völlig ausreicht.

Das finale Produkt verwendet also eine simple, objektorientiert Datenstruktur, die serialisiert und so zwischen den zwei Projekten (Indexer und IMVR) geteilt werden kann. Das Schema ist in Abbildung 3.6 zu sehen. Damit beide Projekte Zugriff auf die genutzten Klassen haben, wurde das Schema in eine separate DLL ausgelagert, die ebenfalls als Abhängigkeit in beiden Projekten referenziert wird.

Anzumerken ist, dass mehrere Einstiegspunkte auf die Daten existieren und so eine gewisse Redundanz geschaffen wird. Diese Redundanz ist hilfreich, weil in IMVR mehrere Modi eben diese Einstiegspunkte benötigen. Aufgrund der gewählten Serialisierungsmethode entsteht jedoch kein grosser Speicher-Overhead.

Die Serialisierung wird durch sogenannte Protocol Buffers [16] realisiert, wofür eine Open-Source

Bibliothek namens *protobuf-net*² existiert. Protocol Buffers ist ein binäres Datenformat, welches von Google Inc. entwickelt wurde und bekannt ist für seine Kompaktheit und Simplizität. Ge wählt wurde es, weil die herkömmliche Serialisierung mit .NETs BinaryFormatter zum Teil zu Problemen mit Unity führen kann.

4.4 Herausforderungen

Während der Entwicklung des Indexers kristallisierten sich diverse Schwierigkeiten heraus, welche grundsätzlich in drei Kategorien einordnen lassen: organisatorische und datentechnische.

Die Organisation, oder Planung, war deshalb problematisch, weil einer der wichtigsten Faktoren für den Aufbau des Indexers die Vielfalt der Datentypen war. Da anfangs geplant war, Musik *und* Bilder zu indexieren und darzustellen, machte eine parallele Verarbeitung der Daten Sinn und war auch notwendig.

Es wurden parallel mehrere Bilder auf mehreren CPU-Kernen analysiert, und gleichzeitig wurde auch die Musikdatenbank erweitert. Mit dem Wegfall des Bilder-Parts wird die parallele Verarbeitung also um einiges unwichtiger. Dies gilt besonders, weil bei den Metadatenquellen für die Musikindexierung jeweils nur ein Thread Sinn macht, da die APIs mit Limiten ausgestattet sind.

Datentechnisch stellte es sich als problematisch heraus, geeignete Datenquellen zu wählen, welche die notwendigen Daten simpel und mit guten API-Limiten liefern konnten. Von Anfang an war klar, dass der Service von The Echo Nest verwendet würde, doch dieser alleine ist nicht ausreichend. Bevor schliesslich Last.fm als Quelle für Album-Covers gewählt wurde, wurden besonders zwei APIs untersucht: Amazon und Gracenote.

Im Falle von Amazon führten zwei Probleme zum Ausschluss: das Fehlen einer guten C#-Bibliothek und die schwerfällige Registrierung. Momentan sieht die Situation so aus, dass recht viel manuell gemacht werden muss, bzw. eine umfangreiche Service-Description importiert werden muss. [13] In Sachen Registrierung wird erwartet, dass man eine Webseite registriert, Kontaktdaten angibt und dann geprüft wird.

Bei Gracenote beläuft sich das Problem hauptsächlich auf die Daten-Limite. Diese ist nirgends öffentlich ersichtlich, und sobald diese überschritten wird, sind keine Requests mehr möglich für den ganzen Tag. In einer Applikation wie IMVR, wo in einem Schritt alles indexiert werden soll, ist dies ein Killer-Kriterium.

²<https://code.google.com/p/protobuf-net/>

5 Implementation von IMVR

Dieses Kapitel dokumentiert die eigentliche Applikation *IMVR*. Vieles hat sich im Laufe des Projekts verändert und entwickelt, und einige Probleme traten zum Vorschein, die an dieser Stelle genauer erläutert werden sollen.

5.1 Unity 5

Die Implementation von IMVR lässt sich leider nicht ohne einen gewissen Hintergrund in Unity 5 erklären.

5.1.1 Grundkonzepte

Unity ist eine Entwicklungsumgebung und eine Spiel-Engine, die momentan aufgrund ihrer Bedienungsfreundlichkeit und einer frei erhältlichen Version sehr beliebt in der Szene der Indie-Developer ist. Gleichzeitig dient sie auch als gutes Prototyping-Tool, um schnell Ideen umzusetzen.

Um Unity in groben Zügen zu erklären, sollen zwei Ansichtspunkte beschrieben werden: der Szenenaufbau und die Ressourcenverwaltung.

Ein Projekt in Unity ist in sogenannte *Scenes* (Szenen) gegliedert, welche aus Objekten (*GameObject*) bestehen – das Grundprinzip der Scene-Graphs wird also angewandt. Speziell ist, dass die Interaktion und Spiellogik grundsätzlich nur innerhalb von *Components* geschieht. Jedes Script und jede *Eigenschaft* wird als Component einem GameObject zugeordnet. So hat zum Beispiel ein Licht ein *Light*-Component oder die Kamera ein *Camera*-Component. Selbst die Position jedes GameObjects ist nur ein Wert im *Transform*-Component.

Ein anderer Aspekt von Unity ist die Ressourcenverwaltung. Im Grunde genommen ist es dem Programmierer überlassen, wie er seine Ressourcen verwaltet. Das einzige, was beachtet werden muss, ist, dass alle Ressourcen im *Assets*-Folder abgelegt werden müssen. Üblicherweise wird dann ein Ordner für jede Art von Asset erstellt, z.B. für Materialien, Texturen, Meshes, Scripts, etc.

Ein weiterer wichtiger Begriff sind die *Prefabs*. Damit sind Vorlagen gemeint, die man erstellt, indem man fertige Objekte aus der aktuellen Szene in das Asset-Folder zieht, und danach wiederverwerten kann.

5.1.2 Unity im Kontext von IMVR

Was sofort auffällt bei der Betrachtung der Ressourcenverwaltung, ist, dass diese starke Kopplung von Assets zu Projekten sich mit der grundlegenden Aufgabe dieses Projektes beisst. Unity sieht vor, dass der Programmierer während der Entwicklung alle seine Assets in den dafür vorgesehenen Ordner platziert, das Projekt am Schluss kompiliert, und dann höchstens im Nachhinein neue Assets als *Asset Bundles* an seine Anwender verteilt. In diesem Projekt ist es jedoch zwingend nötig, dynamisch Bilder und Musik anhand der Dateien auf dem Anwender-PC zu laden. Auf die Folgen und Lösungen zu diesem Problem wird in den Kapiteln 5.5.1 und 5.5.2 näher eingegangen.

5.1.3 UI

Ein Feature, von dem die Applikation starken Gebrauch macht, ist das UI-System von Unity. Vor Version 4.6 des Editors wurde das GUI grösstenteils im Programmcode selbst erstellt mithilfe des alten System *nGUI*.

Das neue System ist analog zu nGUI unter dem Namen *uGUI* bekannt. Neuerdings werden GUI-Elemente direkt im Editor platziert und können von einer umfangreichen Layout-Engine profitieren. Ausserdem verfügen UI-Komponenten über ihre eigene Behaviour-Klasse *UIBehaviour*.

Der entscheidende Vorteil davon im Kontext dieses Projektes ist die Möglichkeit, diese UI-Elemente dreidimensional darzustellen. Neben einem Overlay- und einem perspektivischen Kamera-Modus verfügt ein Canvas, das Root-Element eines UIs, über einen World-Modus, wo das Canvas direkt in der Welt positioniert wird.

Eine Implikation, die dieses System in sich birgt, ist, dass es nicht mehr so einfach wie zu Zeiten von nGUI ist, ein GUI im Code zu erstellen. Würde man alle Elemente dynamisch generieren, würde dies zu unleserlichem und schlecht handhabbaren Code führen.

Um dieses Problem zu umgehen, bieten sich Prefabs an. Der Prozess der Erstellung eines einzelnen UIs sieht folgendermassen aus:

1. Designen des UIs im Editor
2. Speichern als Prefab
3. Im Code: Laden und per `SetTransform()` an ein anderes Element anhängen
4. Iterieren bis fertiggestellt

5.2 Aufbau

In gewöhnlichen Projekten wird häufig eine "Szene" pro Spielbildschirm verwendet. Da es in IMVR nur eine wirkliche Szene gibt bzw. ein laufender Übergang gemacht wird, benötigt es nur eine, welche sich um die gesamte Darstellung kümmert.

Die Struktur sieht so aus, dass an oberster Stufe vier Knotenpunkte existieren:

1. einer für das Event-System von Unity

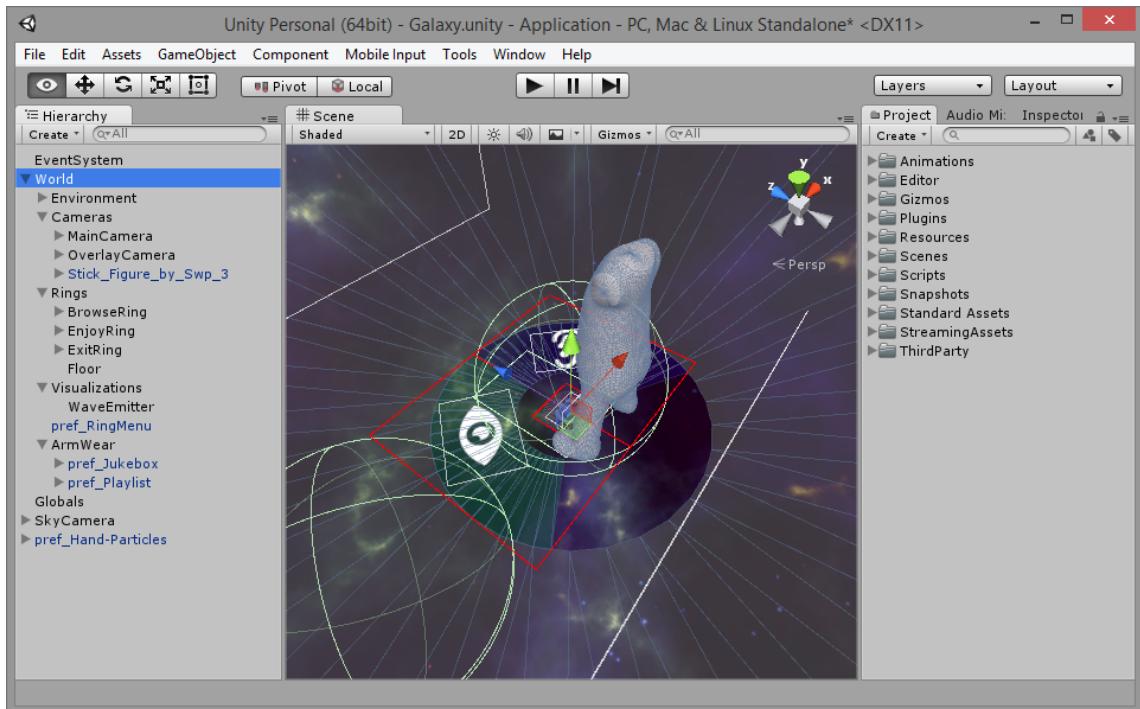


Abbildung 5.1: Aufbau der Szene

2. einer für die gesamte Welt (näher erläutert in Kapitel 5.5.4)
3. einer für die korrekte Positionierung der Kamera (ebenfalls erläutert in Kapitel 5.5.4)
4. einer für die Handhabung der Partikel der Hände

Die Welt selbst besteht aus mehreren Kameras, Visualisierungen, und den Elementen, die für den Music Arm benötigt werden. Zusätzliche GameObjects werden jeweils generiert, wenn neue Views geladen werden.

5.3 Interaktionskonzept

Unüberraschenderweise findet die Interaktion des Users mit IMVR fast ausschliesslich mit seinen Händen statt. In einer frühen Phase des Projektes war noch geplant, eventuell die Spracheingabe modal zu den Händen zu gebrauchen, jedoch reichte dafür die Zeit nicht mehr. Ein Artefakt dieses Vorhabens ist das *SpeechServer*-Projekt, welches sich immer noch unter den *AuxiliaryTools* befindet.

5.3.1 Fussplatten

Beim Erstellen einer visuellen Applikation stellt sich die Frage, wie man am besten die Struktur verdeutlichen kann. Eine Technik, die dafür gewählt wurde, ist der Einsatz von "Fussplatten".

Hierbei befinden sich unter den Füßen des Anwenders ringförmige Platten, welche die zwei Modi der Applikation repräsentieren. Eine dritte, zentriert abgehobene Platte dient zur Beendigung der Applikation.

Bei diesen Platten handelt es sich um das einzige Interaktionsmittel, welches bewusst keine Eingabe durch die Hände erfordert. In diesem Fall wird die Oculus Rift selbst als Eingabegerät verwendet, und zwar durch den Blickwinkel.

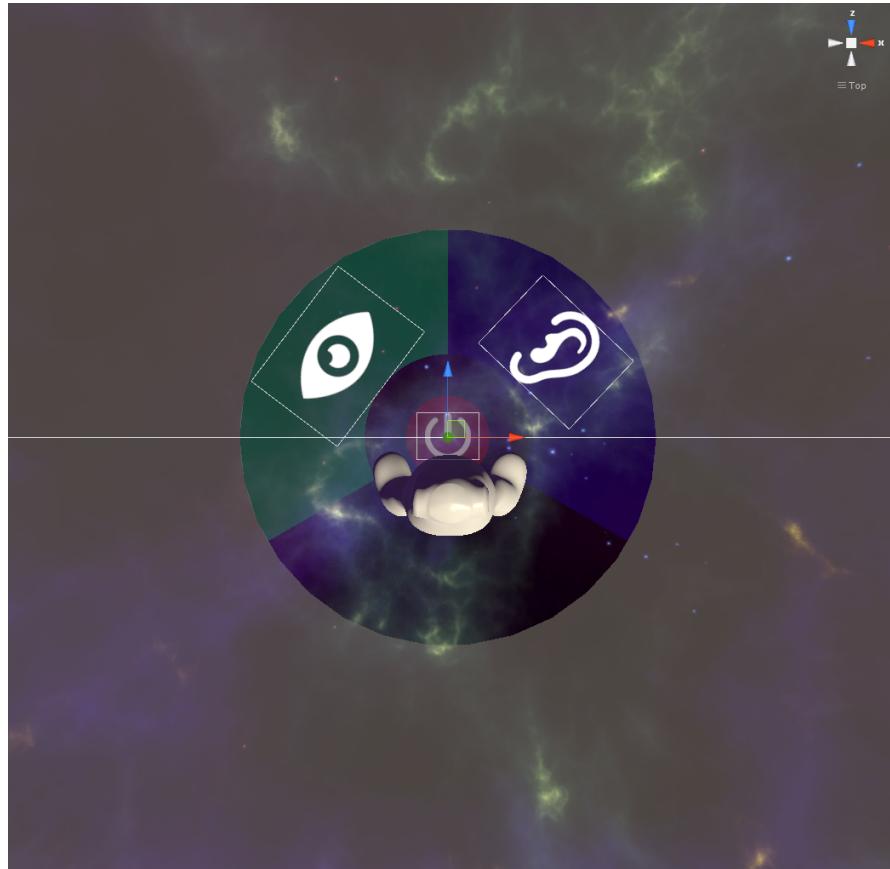


Abbildung 5.2: Vogelperspektive auf die Szene

Die Idee ist, dass der Benutzer feststellen will, "wo er steht", herunterschaut, und durch gehaltenen Blickkontakt mit den Fussplatten diese aktivieren kann. Entsprechend den *Best Practices* wird dabei ein Indikator gefüllt, der anzeigt, wie lange der Blick noch gehalten werden muss.

Diese Art von Eingabe lässt sich oft bei bereits erschienen Demos für die Oculus Rift beobachten. Das Prinzip ist sehr leicht zu implementieren und daher auch verlockend, jedoch muss mit Vorsicht vorgegangen werden: Für den Benutzer ist es auf Dauer unangenehm, wenn von ihm ständige Kopfbewegungen gefordert werden. In IMVR wurde jedoch bewusst Gebrauch von dieser Methode gemacht, weil der Anwender nur selten nach unten schauen wird, und es relativ intuitiv ist.

Implementiert wurden diese Fussplatten mit einem selbst erstellten Ring-Mesh (siehe Kapitel 5.4.4), welches zur Laufzeit generiert werden kann und dynamisch genug ist, um animiert zu werden. Bei der Benutzung werden insgesamt 4 Klassen angewendet: zwei zur Darstellung und zwei für die Logik.

In Abbildung 5.3 sind diese nummeriert abgebildet. Die Rolle der Klassen sieht dabei folgendermassen aus:

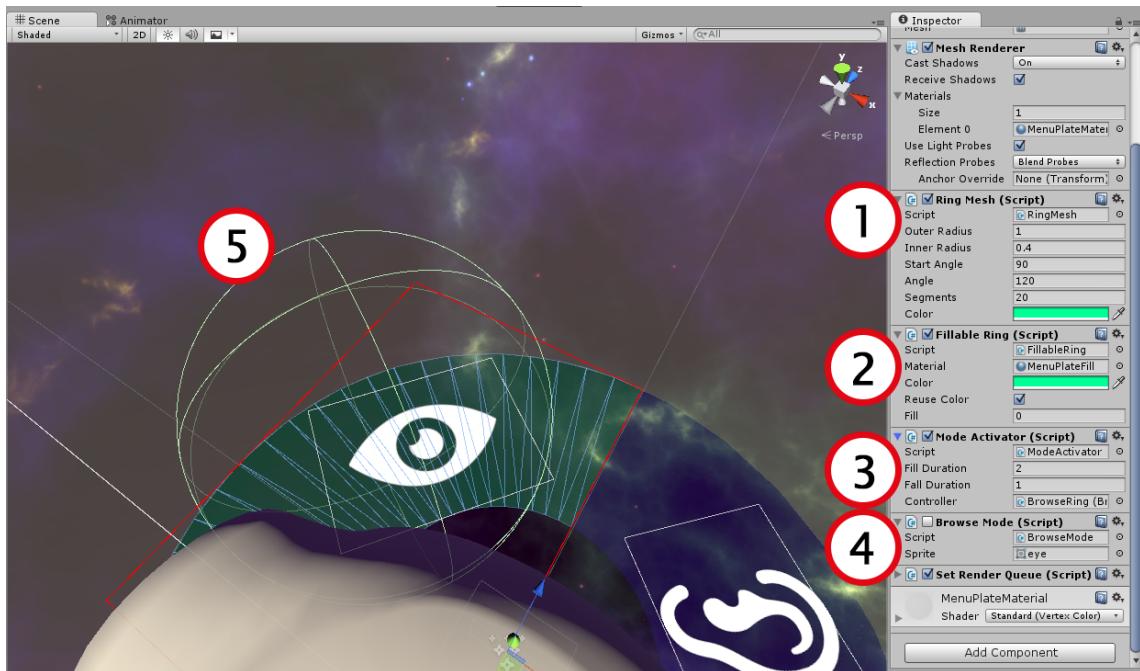


Abbildung 5.3: Aufbau einer Fussplatte

#	Komponente	Rolle
1	Ring Mesh	Sorgt für die Darstellung eines Ring-Meshes, indem ein Mesh anhand der angegebenen Parameter generiert wird.
2	Fillable Ring	Erweitert die Ring Mesh Komponente mit einem zweiten Kreissektor, der die gleichen Werte wie das Ring Mesh hat, bis auf den äusseren Radius. Dieser wird anhand des Fill-Wertes zwischen 0 und 1 verschoben.
3	Mode Activator	Reagiert auf Blicke und steuert den Fill-Wert des Fillable Rings.
4	Mode	Stellt den Modus dar, der vom Mode Activator aktiviert werden soll, sobald der Blick lange genug gehalten wurde.

Tabelle 5.1: Beschreibung des Aufbaus der Fussplatten

Um den Blick des Anwenders zu erkennen, wurde bei den Fussplatten jeweils an einer geeigneten Position ein Collider gesetzt (Nummer 5 auf Abbildung 5.3), der auf einer für die Fussplatten dedizierten Ebene liegt und per Raycasting geprüft wird.

5.3.2 Ringmenü

Bei der Entwicklung von VR-Applikationen stösst man zwingenderweise auf Situationen, in denen herkömmliche Konzepte nicht mehr verwendet werden können. Die Platzierung und der Aufbau des Menüs ist so ein Punkt.

Es ist nicht leicht ein Menü korrekt zu platzieren. Eine statische Platzierung als Overlay hält das Interface zwar im sichtbaren Bereich, kann sich jedoch als störend herausstellen. Lässt man es verzögert mitschweben, gerät das Menü sofort ausser Kontrolle, und stellt man es irgendwo in die Szene und belässt es dabei, verliert man es sofort aus dem Blick.

Im Falle von IMVR bieten sich jedoch die Hände als gut verwendbarer Ankerpunkt für das Menü an. Es gibt ein freies Projekt auf GitHub [6], welches die Finger der Hand für die Platzierung der Buttons in einer ringartigen Struktur verwendet. Leider befand sich das Projekt in einem zu instabilen Stadium für diese Arbeit, aber es lieferte die Idee für eine eigene, ähnliche Implementierung.

Für IMVR wurde ebenfalls ein Ringmenü entwickelt, doch dieses verfügt über keine Schaltflächen im herkömmlichen Sinn. Die Finger werden auch mit Funktionen versehen, aber zum Betätigen benutzt der Anwender nicht seine andere Hand, sondern hebt einen Finger. Wenn ein Finger lange genug gehoben wird, wird die zugewiesene Aktion ausgeführt.

5.3.3 Music Arm

Da es in dieser Arbeit voll und ganz um das Eingreifen ins Geschehen mit den eigenen Händen geht, wäre es bedauerlich, wenn es nicht möglich wäre, die Musik mit den Händen zu steuern. Ein weiteres Konzept, welches genau dieses Problem löst, ist der sogenannte "Music Arm".

Beim Music Arm handelt es sich um das Interface, welches an Stelle des menschlichen Arms angezeigt wird. Dieser digitale Arm bietet die Möglichkeit, die Playlist zu verwalten, sowie die momentan abgespielte Musik anzuschauen bzw. vor- und zurückzuspielen.

Hierbei wird eine Technik verwendet, welche einen Unterschied zwischen der Vorder- und der Rückseite macht. Schaut der Benutzer auf die "Uhr", also hält seinen Handrücken vor sich, dann erscheint ein Interface, in welchem er alle wichtigen Informationen über das momentane Lied erhält (Name, Artist, Cover, Länge und Fortschritt).

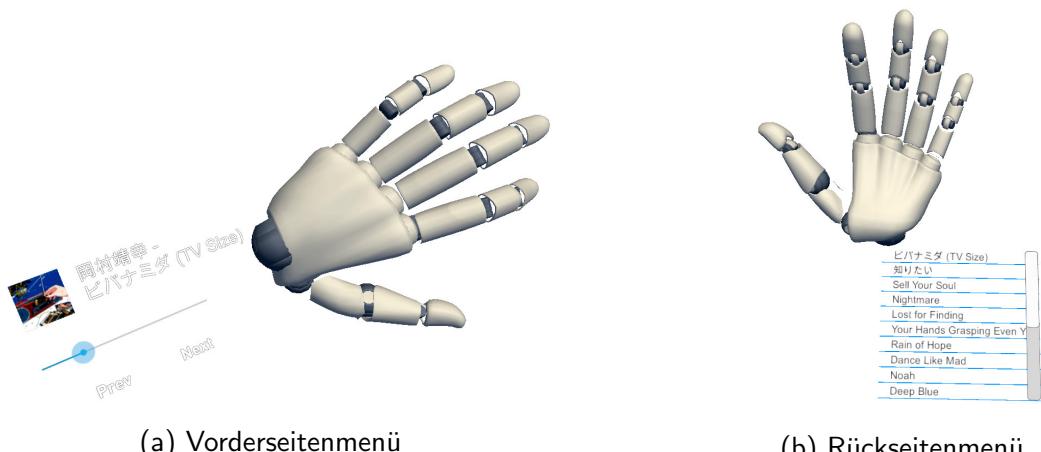


Abbildung 5.4: Darstellung des Music Arms

Dreht der Anwendet jedoch seinen Arm und schaut auf die Rückseite, wird er eine Liste erhalten, die er scrollen und selektieren kann. Bei der Selektion wird das Element auf das Ringmenü "gesendet", wo der Anwender dann die Möglichkeit hat, das Lied abzuspielen, oder die Selektion aufzuheben. Würde man sofort bei der Selektion ein Lied abspielen, ergäbe sich das Problem, dass durch

die mangelhafte Genauigkeit der Leap Motion und der Bedienung allgemein viele Fehlselectionen passieren würden und somit diverse Lieder zufällig abgespielt würden. Abbildung 5.4 zeigt, wie dieses Interface aussieht.

Der Entscheid, welche Seite angezeigt wird, geschieht über ein simples Skalarprodukt. Der Forward-Vektor der Kamera wird verglichen mit der Normale des Arms, welche in die gleiche Richtung zeigt, wie der Handrücken. Wenn der Wert tiefer als -0.5 ist, die Vektoren also entgegengerichtet sind, wird die Unterseite erkannt, und wenn der Wert höher als 0.5 ist, die Oberseite. Der Entscheid könnte auch bei 0 gefällt werden, aber dann würden auch Grenzfälle angezeigt werden, in denen besser gar kein Menü angezeigt wird.

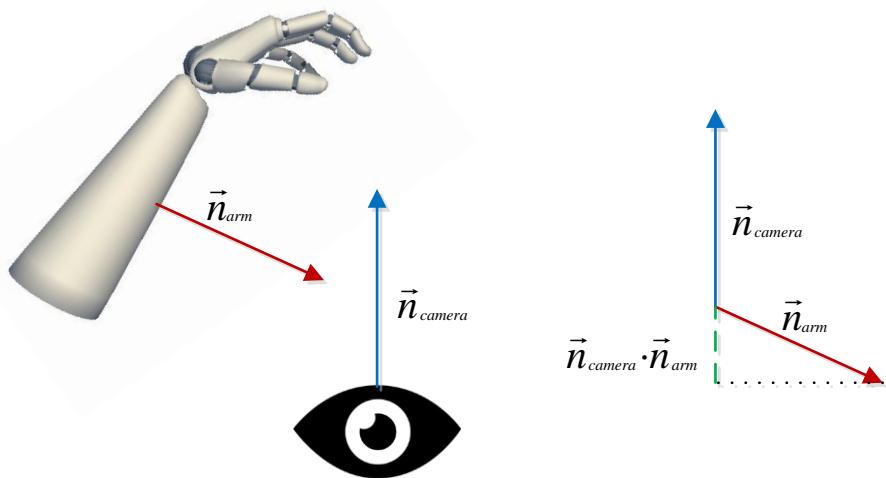


Abbildung 5.5: Bestimmung der momentanen Armseite

5.3.4 Slider Balken

Im Hörteil der Applikation, dem blauen Teil, sollte es dem Anwender ermöglicht werden, die Features der gewünschten Musik anzugeben, und diese dann anzuhören. Wie bereits erwähnt, bilden diese Werte einen Bereich zwischen 0 und 1 ab (mit Ausnahme des Tempos). Will der Anwender eine energievolle Musik hören, muss es ihm deshalb möglich sein, neben dem Maximalwert auch einen Minimalwert anzugeben. Es liegt also ein typischer Use-Case für einen Range-Slider vor.

Da sich kein solcher im Werkzeugkasten von Unity befindet, musste ein eigener erstellt werden. Um einen anderen Approach zu testen, als mit der starken Verwendung von Unitys neuem UI-System in den anderen Teilen der Applikation gewählt wurde, wurde in diesem Fall bewusst ein Slider mithilfe eines Zylinder-Meshes erstellt.

Beim implementierten Slider hat der Anwender die Möglichkeit, direkt mit seinen zwei Händen einen Auswahlbereich einzustellen, indem er diese parallel in den Zylinder hält und entsprechend

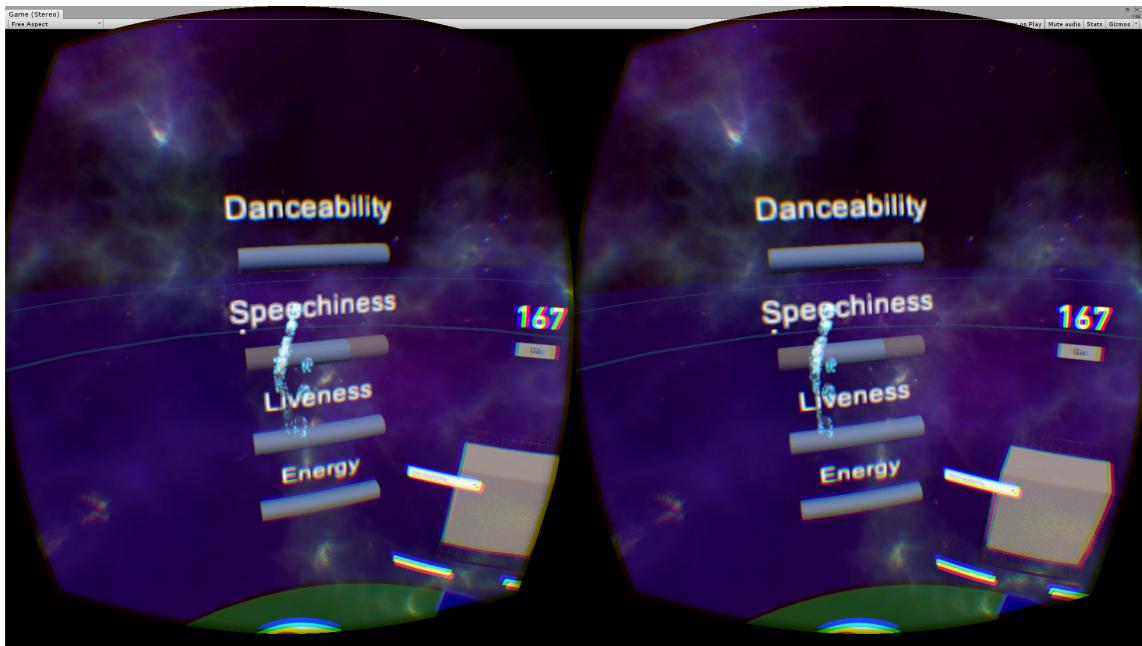


Abbildung 5.6: In-Game Screenshot zur Bedienung der Slider-Balken

bewegt. Anders als die relativ abstrakte Implementation der anderen Interaktionselemente, basiert dieser Slider auf das Kollisionsmodell von Unity und die physikalische Hände, die mit dem Leap Motion Plugin mitgeliefert werden.

Das System funktioniert so, dass zu Beginn eine Achse \hat{v}_{dir} festgelegt wird, in die sich der Zylinder füllt (z.B. von links nach rechts). Bei einer Kollision der Hand mit dem Zylinder, wird nun zuerst geprüft, ob die Hand gültig ist. Sofern das der Fall ist, wird die Position der Handfläche \vec{P}_{hand} auf die Achse \hat{v}_{dir} projiziert und auf die Länge des Zylinders skaliert.

Der Wert, der dadurch entsteht, wird dann je nach geprüfter Handseite als neues Minimum bzw. Maximum verwendet. Allerdings geschieht davor noch eine lineare Interpolation vom vorigen Wert für einen weichen Übergang und ein Snapping zum Vereinfachen der Auswahl des Maximums bzw. Minimums.

Damit andere Teile der Applikation auf diese Wertänderungen reagieren können, wird beim Ändern der Extrema jeweils die Events `onMinValueChanged` bzw. `onMaxValueChanged` sowie `onValueChanged` ausgelöst. Durch Abfangen dieser Events wird z.B. der Zähler im `Selector` bei Änderungen der Selektion aktualisiert.

5.3.5 Zylinderinteraktion

Eine zentrale Funktion von IMVR ist die Darstellung von Artisten in einer Art Karussell bzw. einem Zylinder. Die Interaktion geschieht dabei vollständig mit den Händen.

Um dieses Feature zu implementieren, fiel der Entscheid ebenfalls auf das UI-System von Unity, da dieses die nötigen Elemente (Text, Bilder, Widgets) liefert. Um die Interaktion zu ermöglichen, wurde ein selbst geschriebenes Input-Modul verwendet, welches die Selektierung, Aktivierung und Deaktivierung von UI-Elementen übernimmt. Wie Input-Module funktionieren, wird in der Vorarbeit

genauer erläutert. Wichtig zu wissen ist, dass es diese auch für die Maus- und Tastatursteuerung sowie für Touchpads gibt. Input-Module eignen sich also für diese Funktionalität.

Ein Problem, das diese Module in diesem Kontext mit sich bringen, ist ihre enge Kopplung mit 2D-Interfaces. Die Event-Behälter, die intern verwendet werden, benutzen fast ausschliesslich 2D-Vektoren, um Klickpositionen und ähnliches festzuhalten. Ebenfalls problematisch ist, dass, anders als bei der Maus oder beim Touchpad, die Entscheidung über einen Klick im 3D-Raum gemacht werden muss. Wird der Finger zwischen zwei Schaltflächen gehalten, muss trotzdem die richtige aktiviert werden, auch wenn diese z.B. verdeckt ist. Ausserdem erfordert der Raycaster, der bei der UI verwendet wird, ebenfalls 2D-Koordinaten.

Diese Problematik wurde gelöst, indem die Position des Fingers in 2D-Koordinaten umgewandelt wird. Der Prozess läuft folgendermassen ab:

1. Der Punkt P_{3D} , wo sich die Fingerspitze befindet, wird durch die Event-Kamera (linkes Auge) in einen 2D Punkt P_{2D} umgewandelt.
2. Ein Strahl wird durch P_{2D} in die Szene geworfen und gibt eine Liste von Schnittpunkten S zurück.
3. Die Schnittpunkte S werden nach dem Element durchsucht, welches die niedrigste, rechtwinklige Distanz d von P_{3D} aufweist und wo gilt $d < T$, wobei T ein Schwellwert ist.
4. Das gefundene Element wird in die Input-Pipeline gesendet und weiterverarbeitet.

Es geschehen noch weitere Optimierungen, um z.B. den Fall abzudecken, wenn der Finger bei der Betätigung einer Schaltfläche nicht mehr genau darüber liegt.

5.4 Visual Design

Viele Elemente in IMVR haben mit dem Visuellen zu tun. Sie sollen die Musik unterstützend begleiten, Metadaten darstellen und natürlich gefallen.

5.4.1 Darstellung der Hände

Das erste Thema, das beim Visuellen einfällt, sind die Hände. Bei der Gestaltung der Hände waren grundsätzlich folgende Voraussetzungen zu erfüllen:

1. handförmig
2. abstrakt
3. momentaner Modus ist erkennbar

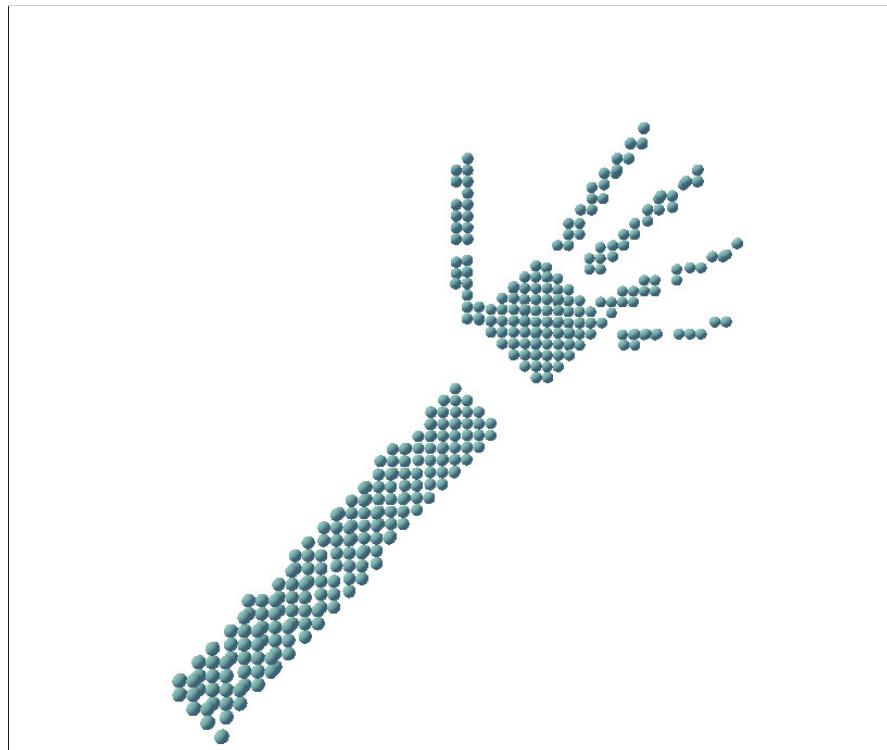


Abbildung 5.7: Die ursprüngliche Visualisierung der Hand

Der erste Punkt versteht sich von selbst. Der zweite Punkt, die Abstraktheit, kommt daher, weil es am besten zur Applikation passt. Eine echt-aussehende, materielle Hand würde fehl am Platz wirken, wenn der Rest der Szene fast ausschliesslich aus Geometrie besteht und keinen Zusammenhang mit der Realität hat.

Neben dem erwähnten Argument, hat eine abstrakte Hand auch den Vorteil, dass die Darstellung der momentan Musik angepasst werden kann, z.B. durch Farbe oder Bewegung.

Beim dritten Punkt ist mit *Modus* der aktuelle Applikationsmodus gemeint, also der Browse- bzw. Listen-Modus. Diese werden unter Anderem durch die Farbe unterschieden, und diese Farbe zeigt sich auch bei der Visualisierung der Hände.

Für die erste Art der Visualisierung (Abbildung 5.7) wurde eine leicht abgeänderte Form einer Standardhand im Leap Motion Package gewählt. Dabei handelte es sich um eine auf Voxel basierende Darstellung. Für jeden Knochen in der Hand wurde ein Voxel-Sheet erstellt, welches dann den Bewegungen der Hand gefolgt ist. In der Implementation in IMVR wurden lediglich die Voxels mit Kugeln ersetzt, um ein bisschen Individualität zu schaffen.

Diese Darstellungsform wurde jedoch im Laufe des Projekts zugunsten einer partikelbasierten Implementation verworfen. In der ersten partikelbasierten Form, wurde versucht eine gerigchte Hand mit Partikeln zu umrahmen statt das Mesh selbst zu rendern. Das war jedoch mit ein paar grossen Nachteilen verbunden:

- (a) Die Bone-Weights werden auf der GPU angewendet, deshalb muss das Mesh in jedem Frame auf der CPU nachgebildet werden.
- (b) Durch die hohe Zahl der Vertices werden über 10 000 Partikel pro Hand erstellt, was sich sehr schlecht auf die Performance auswirkt.

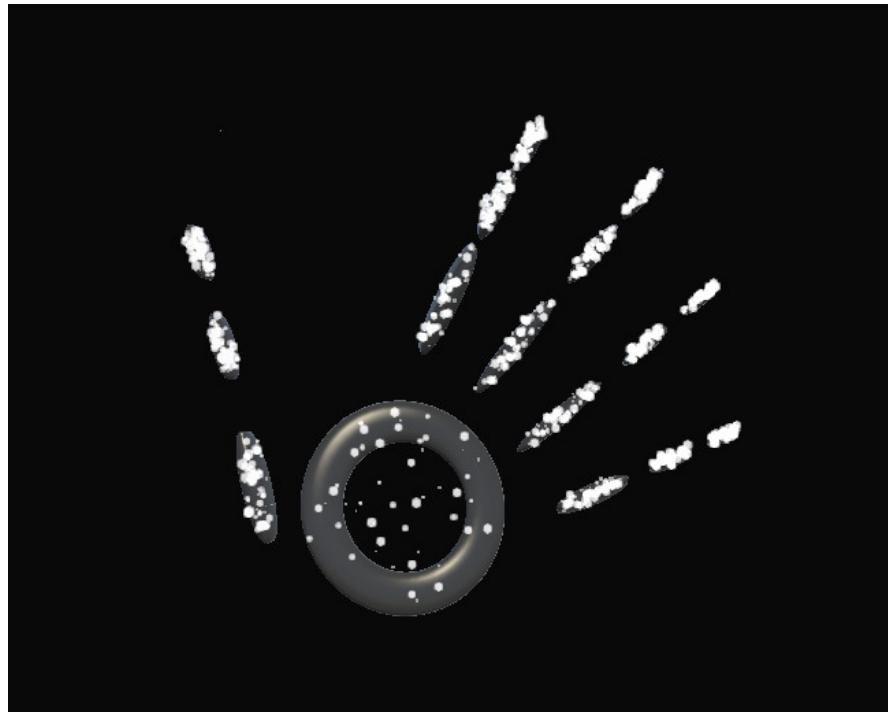


Abbildung 5.8: Die finale Visualisierung der Hand mit Partikeln

- (c) Da das Finden von Punkten innerhalb eines komplizierten Meshes nicht trivial ist, müssen alle Punkte auf der Hülle platziert werden.

Aufgrund dieser Nachteile wurde die Implementation schliesslich noch einen Schritt weiter abgeändert. In der finalen Visualisierung (Abbildung 5.8) wird eine abstrakte Hand, welche aus mehreren Meshes besteht, mit Partikeln *gefüllt*. Damit erhält der Anwender ein intuitives Feedback, wenn die Hand erkannt wird, anstatt dass diese aus dem Nichts erscheint. Damit er trotzdem nicht auf den Effekt warten muss, erhält die Hand einen halb-durchsichtigen Rahmen.

Ein Grossteil des Codes konnte von der vorhergehenden Implementation mit dem geriggten Modell übernommen werden. Der Vorteil hier ist, dass die Hand auf *Transforms* aufbaut: Die Submeshes, ein Mesh pro Knochen, werden durch ihre *Transforms* korrekt platziert, damit eine Hand gebildet wird. Dadurch entsteht zwar nicht ein fliessender Übergang wie bei einer geriggten Hand, aber man kann mit primitiven Elementen arbeiten.

In diesem Fall wurde ein Handmodell gewählt, welches grösstenteils aus einfachen, abgerundeten Quadern besteht. Bei der Erstellung der Hand, werden alle Meshes abgelaufen und für jedes Mesh eine gewisse Anzahl Partikel erstellt. Damit diese *innerhalb* der Hand erscheinen, und nicht auf der Hülle wie bei der vorherigen Implementation, wird jede Partikelposition aus zwei zufälligen, verschiedenen Vertices gebildet mit einer ebenfalls zufälligen Gewichtung. Weil die Formen alle konvex sind, ist garantiert, dass die gebildeten Punkten alle innerhalb der Meshes liegen.

Um schliesslich einen schönen Effekt zu erzielen, wurden noch ein paar zusätzliche Details eingebaut:

- Die Partikel sind animiert
- Die Partikel erhalten die Farbe des aktuellen Modus'

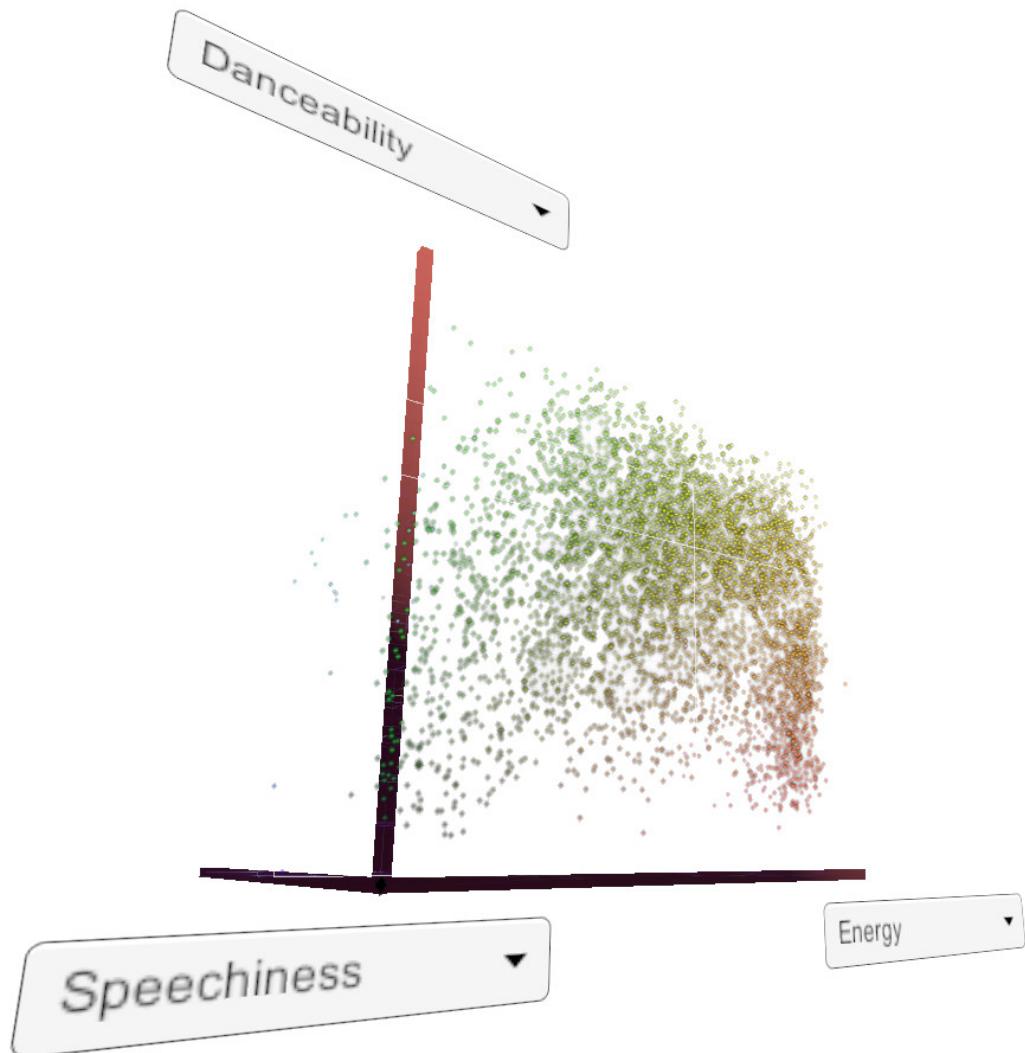


Abbildung 5.9: Visualisierung der Musiksammlung innerhalb eines Point Charts

- Die Partikel werden beim Verlieren der Handerkennung zerstreut

Alles in allem wurde damit eine ansprechende und intuitive Darstellung der Hände erreicht.

5.4.2 Point Chart

Wenn man von Datenvisualisierung redet, ist es ganz klar, dass auch Diagramme dargestellt werden wollen. Umso besser, wenn dies stereoskopisch geschehen kann: Dadurch lässt sich die dritte Dimension erheblich besser wahrnehmen.

Im Rahmen dieses Projekts wurde ein Point-Chart entwickelt, der dazu dient, die Musik des Anwenders in ein dreidimensionales System einzurichten und auf einen Blick darzustellen. Die Dimensionen sind 1:1 die Features, die während der Indexierung von The Echo Nest heruntergeladen wurden, also Merkmale wie *Danceability*, *Energy* und *Tempo* (siehe Kapitel 3.3.1).

Wie auf dem UML (Abbildung 3.15) ersichtlich, wurde die Implementation des Point Charts in drei verschiedenen Klassen durchgeführt.

PointChart ist eine relativ generisch gehaltene Implementation eines Point Charts, welche sich nur um die Darstellung kümmert.

SongMetaChart ist eine Komponente, die auf der ersten basiert und diese steuert, und die Metadaten-Logik ins Spiel bringt.

SelectionVisualizer ist eine weitere Komponente, die ihrerseits auf SongMetaChart aufbaut und die momentane Selektion (nur im Listen-Modus) visualisiert.

Die zentrale Klasse der Funktionalität, PointChart, besteht visuell ebenfalls aus drei Teilen. Es werden drei Elemente bei Änderungen prozedural erstellt:

- Ein Koordinatensystem, das aus drei Balken besteht, welche mit *Triangles* generiert werden
- Ein dreidimensionales Gitternetz, welches mit Linien erstellt wird
- Einer Partikewolke aus Partikeln

Die zwei Meshes werden grundsätzlich nur einmal generiert, können aber mithilfe der Methode RebuildMesh() jederzeit neu erzeugt werden. Die Linien des Gitters bestehen aus einem durchsichtigen Material-Shader, welcher ihnen eine passend subtile Sichtbarkeit gibt.

Das Partikelsystem, welches für die Punkte des Diagramms zuständig ist, sollte immer bei Änderungen der Punktewerte mithilfe der Methode UpdatePoints() erneuert werden.

Ursprünglich war geplant, diese Punkte durch ein *Point*-Rendering zu implementieren, doch diese Art des Renderings ist im Falle von Unity relativ limitiert. Will man lediglich Punkte als einzelne Pixel darstellen, stellt das kein Problem dar. Problematisch wird es, sobald die Grösse der Punkte geändert werden soll, denn dann ist man auf einen OpenGL-Build angewiesen [7].

Die Implementation per Partikelsystem funktioniert gut. Auf was geachtet werden muss, ist die Lebenszeit der einzelnen Partikel. Da sie in diesem Fall nicht kurzlebig sind sondern persistent, wird ihre Lebenszeit bei der Emission der Partikel auf einen Wert gesetzt, der mit sehr hoher Wahrscheinlichkeit nie erreicht werden wird.

```
1 public void UpdatePoints()
2 {
3     float scale = transform.lossyScale.x;
4
5     var particles = new ParticleSystem.Particle[points.Length];
6     particleSystem.Emit(points.Length);
7     particleSystem.GetParticles(particles);
8     for (int i = 0; i < points.Length; i++)
9     {
10         particles[i].size = Mathf.Lerp(0.3f, 0.05f, points.Length / 5000) *
11             scale;
12         particles[i].lifetime = particles[i].startLifetime = 100000f;
13         particles[i].velocity = Vector3.zero;
14         particles[i].position = (points[i] + pivotOffset) * scale;
15         particles[i].color = new Color(points[i].x, points[i].y, points[i].z);
16         particles[i].angularVelocity = Random.RandomRange(-45f, 45f);
17     }
18     particleSystem.SetParticles(particles, particles.Length);
19 }
```

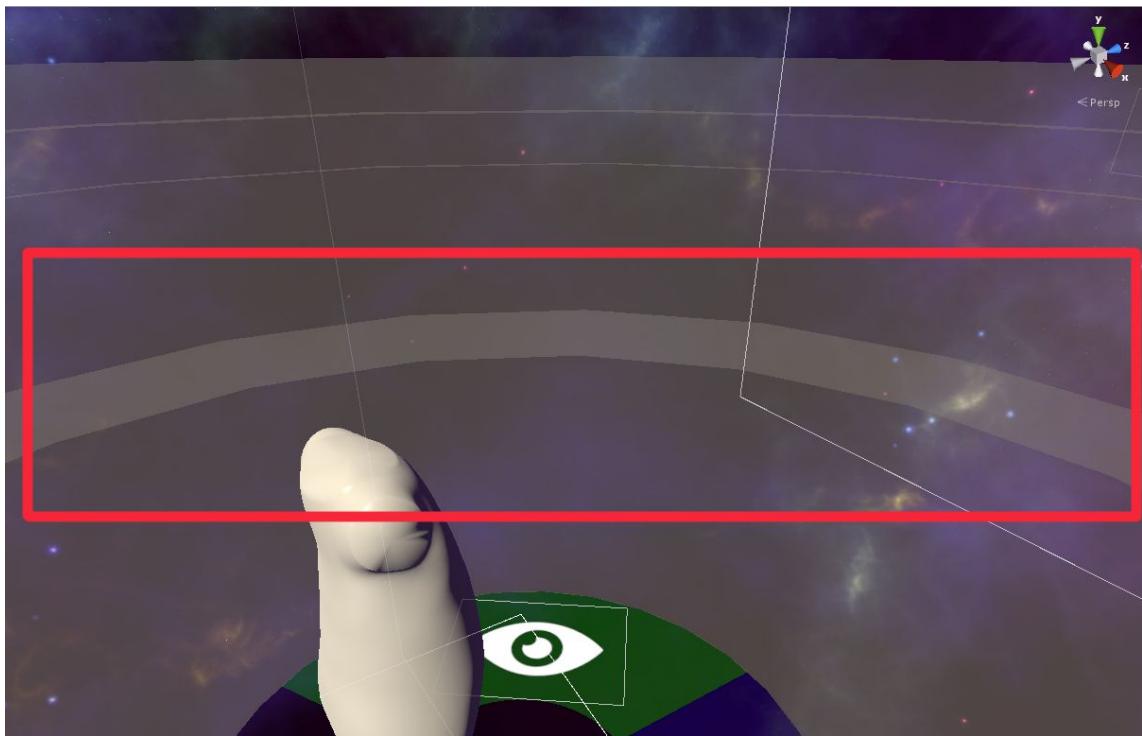


Abbildung 5.10: Beats werden mithilfe von Wellen dargestellt (rot umrahmt)

Listing 5.1: Erstellung der Punkte in einem PointChart

5.4.3 Farben und Symbolik

Um die einzelnen Modi mit Farben abzugrenzen, wurden *Themes* eingeführt, die jeweils eine Kollektion von Farben beinhalten. Beim Aktivieren eines Modus wird dieses Theme ebenfalls aktiviert - zugreifbar per `Theme.Current` - und auf die UI-Elemente angewendet.

Als farbliche Vorlage wurde das UI von Android [3] gewählt. Farben und UI-Ideen wurden teilweise aus den Spezifikation des Material-Designs entnommen.

Damit alle Elemente auf einen Themenwechsel reagieren können, verfügt die Klasse `Theme` über ein `Change-Event`.

5.4.4 Beat Detector und Wellen

Damit die Szene, in der sich der Anwender wiederfindet, nicht zu steril und still ist, wurde ein sehr rudimentärer Beat Detector implementiert, der bei jedem Beat eine Welle mit entsprechender Stärke aussendet.

Für die Programmierung der Beat-Detection wurde ein Artikel von GameDev.com zurate gezogen [10]. Ganz grob gesagt, wird einfach nach Peaks im Energieverlauf der Musik gesucht. Die

benötigten Werte werden alle durch die selbsterstellte Helper-Klasse `VisualizationHelper` bereitgestellt. Als kleine Veränderung zum ursprünglichen Algorithmus wurde die Suche nach Beats auf das Frequenzspektrum zwischen 50Hz und 200Hz beschränkt.

Anstatt einer dynamischen Beat-Detection hätte auch ein Pre-Processing durchgeführt bzw. die Werte von The Echo Nest verwendet werden können. Allerdings ist eine so genaue Beat Detection in diesem Fall nicht notwendig, und für ein exaktes Timing wäre eine stetige Synchronisation zwischen Daten und Musik notwendig.

Die Ringe, die ausgesendet werden, werden alle prozedural generiert und passen sich der *Theme*-Farbe an. Das System basiert auf drei Klassen:

- Der `BeatDetector` versendet bei Beats eine `OnBeat(strength)` Nachricht an Komponenten im gleichen `GameObject`.
- Der `WaveEmitter` fängt die Nachricht ab und erstellt ein neues `GameObject` mit einer `RingMesh`-Komponente.
- `RingMesh` sorgt dann schliesslich dafür, dass der Kreisring bzw. das Kreissegment korrekt dargestellt wird.

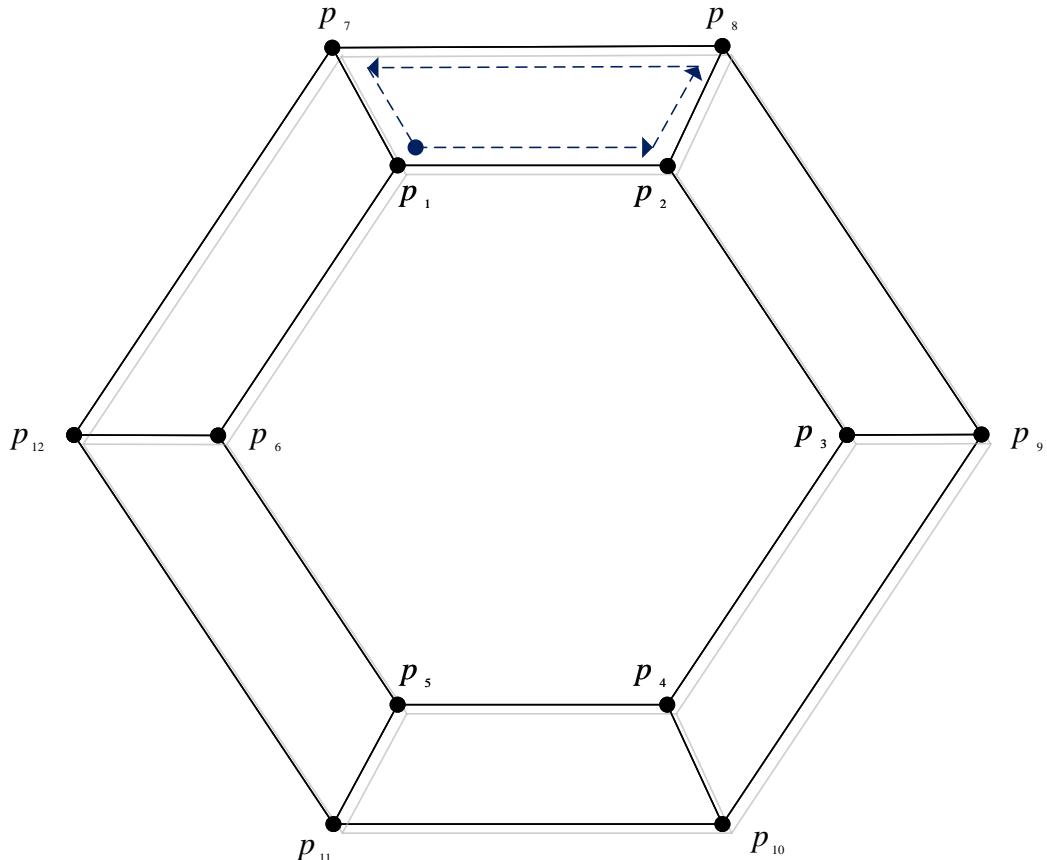


Abbildung 5.11: Die Ring-Meshes werden mithilfe von *Quads* generiert

Die Erstellung des Kreisrings ist sehr simpel gehalten. Die Komponente erlaubt das Konfigurieren von Innenradius, Aussenradius, Startwinkel, Winkelbereich und Anzahl Segmenten. Diese Eigenschaften werden überwacht, und bei jeder Änderung wird beim nächsten Update der Engine eine Neugenerierung des Meshes ausgelöst.

Bei der Generierung wird ein Array der Länge $n_{\text{Segmente}} * 2$ erstellt, welches die Vertices enthält. Die Punkte des Aussen- und Innenkreis werden dann gleichzeitig in dieses Array gefüllt, wobei der Aussenkreis einen Offset von n_{Segmente} erhält (siehe Abbildung 5.11). Schliesslich werden dann die Quads mit jeweils vier Indexen gebildet.

5.4.5 Visualisierung der Musik

Damit das Eintauchen in die Musik nicht zu langweilig und unspektakulär wird, wurden ein paar Klassen für die Musikvisualisierung entwickelt.

In einem ersten Schritt wurde mithilfe von NAudio [4] versucht, selbst Daten über den Frequenzbereich der momentanen Musik per FFT zu sammeln. Dies funktionierte grundsätzlich, brachte aber den Nachteil mit sich, dass ausschliesslich NAudio für die Wiedergabe genutzt werden konnte.

Es stellte sich dann heraus, dass Unity mit GetOutputData und GetSpectrumData selbst über eine Analysefunktion für Musik verfügt¹, und das sogar mit verschiedenen Analysefenstern.

Um die gewonnenen Daten zentral zu verwalten, wurde nun eine Klasse namens VisualizationHelper erstellt. Diese hilft ausserdem dabei, die momentane Lautstärke der Stereo-Kanäle zu bestimmen, indem sie ein RMS-Wert über alle Samples erstellt [8].

Es gibt drei verschiedene Arten, wie die Musik momentan visualisiert wird:

1. die Beats
2. die Lautstärke der Stereokanäle (Waveform)
3. die Lautstärke von Frequenzbereichen

Wie die Detektion und Visualisierung von Beats durchgeführt wird, wurde bereits in Kapitel 5.4.4 beschrieben.

Für die Lautstärke der Stereokanäle wird mit dem Linienrenderer von Unity dynamisch eine Welle generiert, die entlangläuft. Die Klasse RingVisualizer steuert hierbei den Renderer. Die Implementation ist äussert simpel:

```
1 volumes.AddFirst(helper.energies[(int)side]);
2 volumes.RemoveLast();
3
4 Vector3 position = Vector3.zero;
5
6 int i = 0;
7 foreach (float vol in volumes)
8 {
9     float rate = i / (resolution - 1f);
10    position.x = Mathf.Cos(rate * TWOPI) * radius;
```

¹Dokumentation: <http://docs.unity3d.com/ScriptReference/ AudioSource.GetSpectrumData.html>

```

11     position.y = vol * amplitude;
12     position.z = Mathf.Sin(rate * TWOPi) * radius;
13
14     //points[i] = position;
15     points[i] = Vector3.Lerp(points[i], position, Time.deltaTime * speed);
16     lineRenderer.SetPosition(i, points[i]);
17
18     i++;
19 }
20
21 lineRenderer.SetPosition(0, points[resolution - 1]);

```

Listing 5.2: Aktualisierung der Punkte im RingVisualizer

Jedem Punkt im Line-Renderer wird ein sich verschiebender Volume-Wert zugewiesen, der vom VisualizationHelper bereitgestellt wird. Hierbei gibt dieser Volume-Wert die Position auf der Y-Achse vor (die Höhe), während die Position auf der XZ-Ebene per Sinus und Kosinus bestimmt wird.

Etwas interessanter ist jedoch die Berechnung der Lautstärke in einem bestimmten Frequenzbereich.

```

1 // Determine the start and end index in the spectrum array
2 int m_startIndex = VisualizationHelper.Instance.FreqToIndex(startFrequency);
3 int m_endIndex = VisualizationHelper.Instance.FreqToIndex(endFrequency);
4
5 // Calculate root mean square over the frequency range
6 float rms = 0;
7 for(int i = m_startIndex; i < m_endIndex; i++) {
8     rms += Mathf.Pow(VisualizationHelper.Instance.spectrum[m_startIndex], 2);
9 }
10 rms = Mathf.Sqrt( rms / (m_endIndex - m_startIndex) );

```

Listing 5.3: Berechnung des RMS-Wertes über einen Frequenzbereich

Diese Berechnung wird in der Klasse SpectrumBar durchgeführt. Wie auf dem UML (Abbildung 3.15) zu sehen ist, verfügt jede Instanz über eine Start- und eine Endfrequenz, der den Frequenzbereich festlegt. Dieser Bereich sollte mit höheren Frequenzen unbedingt logarithmisch anwachsen, da sonst die Kurve ungleichmäßig verteilt ist.

Zuerst wird im Code dieser Frequenzbereich in Indexe für das Spectrum-Array umgewandelt. Die Grösse dieses Arrays, und somit die Abstände der Frequenzen, ist abhängig von der Anzahl Samples, die für den FFT eingesetzt werden.

Die Werte, die zwischen den zwei Indexen liegen, werden dann quadratisch akkumuliert. Der Durchschnitt des Resultats wird dann schliesslich für die Höhe eines Quaders verwendet.

5.4.6 Avatar

Auch der Anwender selbst muss irgendwie dargestellt werden. Ein Avatar gilt generell als ein Hilfsmittel, um die Immersivität zu steigern, kann jedoch auch verschlechternd wirken [9].



Abbildung 5.12: Der verwendete Avatar mit Kopf und Armen

Quelle: <http://www.turbosquid.com/FullPreview/Index.cfm/ID/833108>

Im Falle von IMVR wurde jedoch die Entscheidung getroffen, einen simplen Avatar einzubauen. Aufgrund der Art der Applikation, fiel hierbei die Wahl auf eine abstrakte Figur, die frei verfügbar und vollständig geriggt ist. Dadurch bleibt jederzeit die Möglichkeit offen, per IK die Figur zu steuern.

Da die Hände und der Kopf nicht benötigt werden, wurden diese kurzerhand mithilfe von 3DS Max 2015 demontiert. Die vorherig erwähnte Kontrolle über IK wurde nicht implementiert, könnte aber in einem nächsten Schritt eingebaut werden.

5.5 Herausforderungen

Während der Entwicklung kristallisierten sich zahlreiche Probleme, die gelöst werden mussten. Teilweise reichte die Zeit allerdings nicht ganz, um die Fehler zu beheben.

5.5.1 Ressourcen-Management

Ein zentrales Problem bei der Erstellung von IMVR waren die Ressourcen. Besonders im Stadium, wo noch eine enorme Anzahl Bilder dargestellt werden sollten, war es wichtig, die Bilderressourcen effektiv zu verwalten.

Das Problem bei Bildtexturen ist, dass jede verwendete Textur auf die GPU gesendet werden muss und einen Draw-Call auslöst. Werden also 1000 Bilder in der Szene angezeigt, geschehen mindestens 1000 Draw-Calls, was die Performance der Applikation stark herunterzieht.

Als Lösung für dieses Problem wurde die Darstellung der Bilder in IMVR eng um den Einsatz von Atlassen konzipiert.

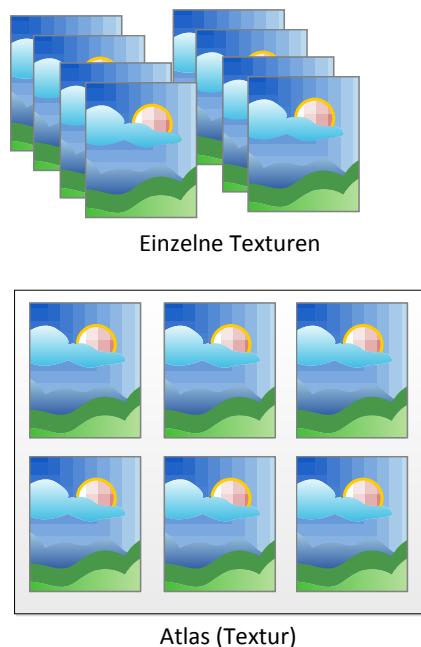


Abbildung 5.13: Illustration der Funktionsweise eines Atlases

Bei einem Atlas werden mehrere Texturen in eine Textur zusammengefasst und die Shader greifen dann jeweils nur auf einen Bereich der Atlas-Textur zu. In Unity ist das relativ leicht durch die Sprite-Klasse anzuwenden:

```
1 Sprite.Create(texture, rect, new Vector2(0.5f, 0.5f), tileSize, 0,  
    SpriteMeshType.FullRect);
```

Der Parameter `texture` referenziert hierbei die Textur, und `rect` beschreibt den benötigten Ausschnitt in Pixel.

5.5.2 Abspielen externer Musik

Ein Problem von Unity ist wie gesagt die enge Anbindung eines Projektes an sein Assets-Ordner. Eine Auswirkung dieses Design-Entscheids ist, dass Unity zur Laufzeit nur unkomprimierte WAVE-Streams und Ogg Vorbis encodierte Streams lesen kann. Diese Limitation wurde anfangs mithilfe

der Bibliothek NAudio umgangen, aber da der Einsatz von NAudio, wie in Kapitel 5.4.5 erläutert, die Benutzung von GetOutputData und GetSpectrumData verhindert, wurde dieser Ansatz verworfen.

Stattdessen wurde auf eine andere C#-Bibliothek ausgewichen: CSCore **??**. Im Gegenzug zu einer schlechteren Portabilität als NAudio (läuft nur auf Windows) bietet CSCore Klassen zur einfachen Konversion von codierten Audio-Streams zu simplen PCM-Streams an.

In Unity dreht sich die Audiowiedergabe um die Klassen AudioListener, AudioSource und AudioClip. Die Rolle dieser Klasse ist grösstenteils selbsterklärend: Ein AudioListener (die Kamera) hört eine Anzahl von AudioSource, welche jeweils einen AudioClip abspielen.

Im Falle von CSCore war es nötig, den AudioClip zu überschreiben. Da die Klasse jedoch sealed und somit nicht erweiterbar ist, wurde kurzerhand ein Wrapper darum erstellt. Die Wrapper-Klasse, CSCAudioClip, bietet eine Property *Clip* an, welche einen gültigen AudioClip zurückliefert.

Der Clip selbst wird direkt im Konstruktor der Klasse erstellt:

```
1 // Init source stream (Codec -> PCM)
2 var m_source = CodecFactory.Instance.GetCodec(filename);
3
4 // Determine the bit depth and choose an appropriate decoder
5 // (PCM (byte) -> Samples (float))
6 switch (m_source.WaveFormat.BitsPerSample)
7 {
8     case 8:
9         m_decoder = new Pcm8BitToSample(m_source);
10        break;
11    case 16:
12        m_decoder = new Pcm16BitToSample(m_source);
13        break;
14    case 24:
15        m_decoder = new Pcm24BitToSample(m_source);
16        break;
17    default:
18        Debug.LogError("No converter found!");
19        return;
20    }
21
22 // Create a clip with the decoder stream as the input
23 Clip = AudioClip.Create(m_name,
24     (int)(m_decoder.Length / m_decoder.WaveFormat.Channels),
25     m_decoder.WaveFormat.Channels,
26     m_decoder.WaveFormat.SampleRate,
27     true,
28     OnReadAudio,
29     OnSetPosition);
```

Listing 5.4: Initialisierung eines Audio-Clips per CSCore

Zuerst wird ein Codec-Stream erstellt, der die Datei in einen PCM-Stream umwandelt. Da Unity die Byte-Werte dieses Streams nicht interpretieren kann, müssen diese noch in Floats umgewandelt

werden. Dies geschieht über einen WaveToSampleBase-Stream. Die Unterscheidung der Bit-Tiefe wird gemacht, damit der Decoder weiß, wie viele Bits zur Bildung der Floats benutzt werden müssen.

Nach diesem Prozedere kann der Clip wie ein herkömmlicher Audio-Clip einer AudioSource zugeordnet und abgespielt werden. Ein Nachteil dieser Implementation ist, dass der Clip nach Benutzung per Dispose() vom Memory gelöscht werden muss.

5.5.3 Canvas-Elemente

Zur Darstellung der visuellen Elemente von IMVR wurde sehr stark auf das UI-System von Unity gesetzt, da dieses ideal für umfangreiche Interfaces geeignet ist und verschiedene Widgets bereitstellt. Zu Beginn waren jedoch ein paar interne Implementationsdetails nicht bekannt, welche schliesslich zu Problemen bei der Anwendung führte.

Canvas-Elemente, welche jeweils an oberster Stelle eines uGUIs stehen, werden als ganze Meshes betrachtet, und seriell gerendert. Überschneidungen mit gewöhnlichen Meshes werden zwar korrekt gehandhabt, schneiden sich aber zwei Canvas-Elemente führt dies zu unerwünschten Ergebnissen (Abbildung 5.14).

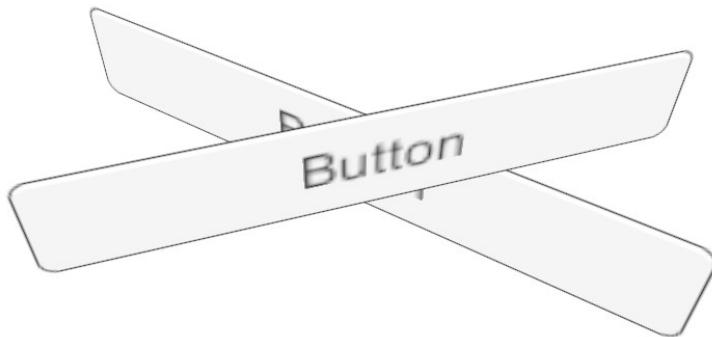


Abbildung 5.14: Überschneidung zweier Canvas-Elemente

Das Problem ist allerdings nicht gravierend und kann verkraftet werden, denn Canvas-Elemente haben eine grundlegende Z-Testing Funktionalität und werden die meiste Zeit korrekt gerendert. Ein anderes, ähnliches Problem stellte sich als viel verheerender heraus: die Verwendung von komplexen Canvas.

Mit "komplexen Canvas" sind Canvas-Elemente gemeint, die tief reichend sind und deren Unter-Elemente eine Positionierung auf der Z-Achse ("in die Tiefe") besitzen. Ein Canvas ist entweder hinter einem anderen Canvas oder davor. Wenn jedoch Unterobjekte mit unterschiedlichen Tiefen existieren, kommt es zu Fällen, wo manche Objekte hinter anderen Canvas liegen und andere weiter vorne. Abbildung 5.15 stellt dieses Verhalten bildlich dar.

Ursprünglich wurden versucht, bei der Erstellung von UI-Elementen möglichst wenige Canvas zu benutzen, um einen Overhead zu vermeiden. Dieses Vorgehen stellte sich jedoch aufgrund des

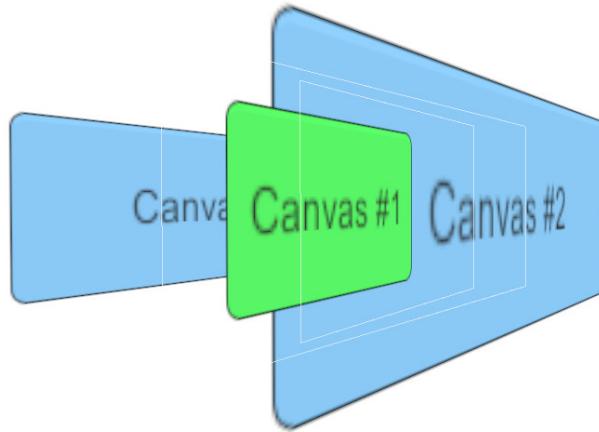


Abbildung 5.15: Probleme mit dem Rendering von komplexen Canvas

oben erwähnten Problemes als Fehler heraus, denn die Rendering-Reihenfolge war dadurch beinahe willkürlich.

Als Lösung wurden deshalb alle Unterobjekte, welche eine andere Position auf der Z-Achse hatten, in eigene Canvas-Elemente ausgelagert.

Eine Ausnahme dazu stellt das "Dial"-Menü in der Künstlerübersicht dar. Dieses hätte stark überarbeitet werden müssen, was zeitlich nicht mehr reichte. Außerdem wäre dadurch die Anzahl der benötigten Draw-Calls wieder drastisch gestiegen, da Unity Canvas-Elemente nicht *batcht* (sprich, die Draw-Calls nicht zusammennimmt).

5.5.4 Performance

Ein mühsames Problem, das wohl in jedem visuell fordernden Projekt früher oder später auftritt, war die Performance der Applikation. Ein Ziel war es, eine konstante Framerate von 60Fps zu erreichen. Leider konnte dies nicht ganz erreicht werden, aber es wurden diverse Techniken angewendet, um die Performance zu verbessern.

Was bereits früh im Projekt auffiel, war das blockierende Laden von Bildern auf die Grafikkarte. Lädt man eine Textur auf die Grafikkarte - was bei der dynamischen Erstellung von Texturen basierend auf Bilder auf dem System häufig passiert - wird das Programm für diese Zeit eingefroren. Bei kleineren Grafiken (128x128) fällt dies nicht sehr ins Gewicht, aber sobald die Texturen grösser werden (1024x1024) wird der Lag schnell bemerkbar.

Um dieses Problem zu beheben gab es 3 Möglichkeiten:

1. Benutzen der Low-level Native Plugin API von Unity
2. Verwenden von kleinen Texturen
3. Pre-Loaden der Texturen

Die erste Möglichkeit, die auf einer tiefschichtigen API von Unity basiert [14], hätte eventuell zu einer Lösung geführt, doch die Unsicherheit einer Lösung und die ungenügende Dokumentation des Features hat schliesslich zu einem Gegenentscheid geführt.

Das Benutzen von kleinen Texturen bringt wiederum andere Probleme mit sich. Zum einen ist es dann nicht mehr möglich qualitativ hochwertige Bilder anzuzeigen, andererseits müsste auf Atlasses verzichten werden, was zu *vielen* Draw-Calls führen würde. Performance-technisch wäre dies also keine Lösung.

Schliesslich fiel der Entscheid auf die letzte Variante - das Pre-Loading. Dafür wurde ein Ladebildschirm implementiert, der mithilfe von `AtlasHelper.Progress` (siehe Abbildung 3.11), prüft, wie viel geladen wurde, und wann er sich wieder verstecken kann.

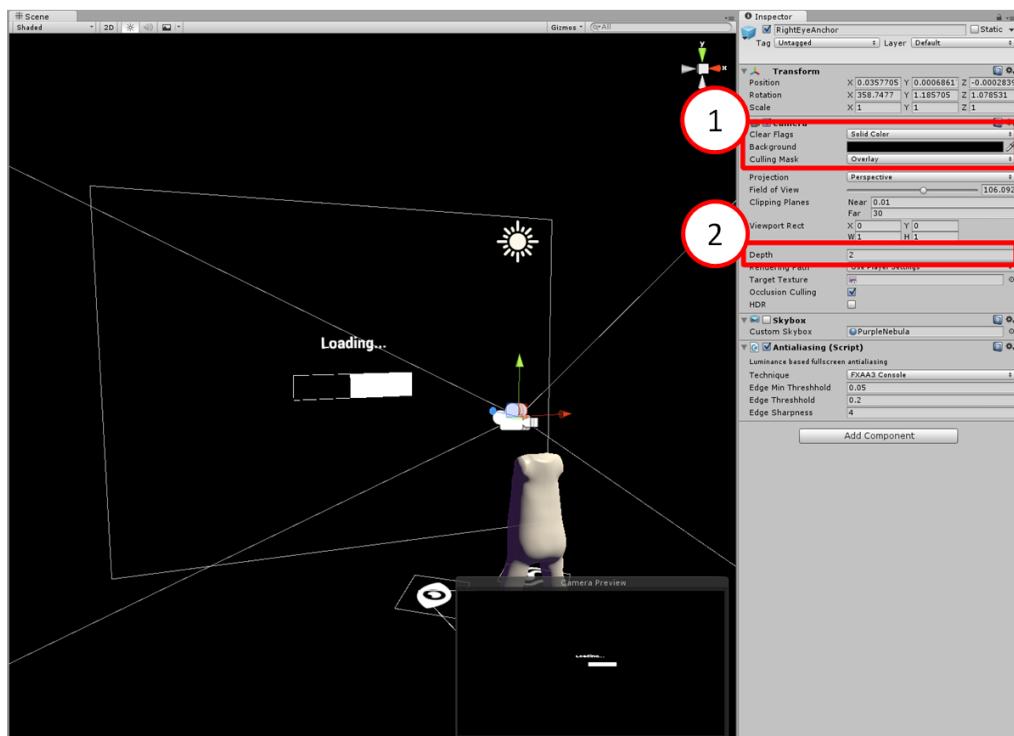


Abbildung 5.16: Aufbau des Ladebildschirms

Implementiert ist das mit einem zweiten Set von Kameras, die *über* die vorhandene Szene rendern, und jederzeit bei aufwendigen Operationen aktiviert werden können. Um das Umzusetzen, benötigt es ein paar spezielle Einstellungen in der Kamera (vgl. mit Abbildung 5.16):

1. Das vorherig gerenderte muss mit der Farbe Schwarz überschrieben werden und die Culling-Maske so eingestellt werden, dass nur die *Overlay*-Ebene gerendert wird.
2. Die Tiefe der Kameras muss höher eingestellt werden, als die eigentlichen Kameras. Das ist notwendig, damit der Ladebildschirm *nach* dem Rest gerendert wird.

Der Ladebalken selbst wird mithilfe der Fill-Property von Bildelementen gefüllt. Das Feature ist tief im UI-System von Unity verbaut und eignet sich perfekt für diese Anwendung. Damit der Ladebildschirm an korrekter Stelle gerendert wird, wurde das Objekt der Kamera angehängt und bewegt sich daher mit.

Im Verlauf der Entwicklung stellte sich ebenfalls heraus, dass die Generierung der *Font*-Texturen relativ zeitaufwendig ist. Deshalb werden im Loader ebenfalls Fonts vorinitialisiert.

Was sich allerdings mit Abstand als am Performance-lastigsten herausstellte, war das generieren von Canvas-Meshes. Dies ist eine grundlegende Limitierung von Canvas-Elementen und wird eventuell in zukünftigen Versionen verbessert. Das Problem ist, dass Canvas-Elemente zu einzelnen Meshes zusammengefasst werden und dieser Vorgang sehr aufwendig ist. Das geschieht bei jeder noch so kleiner Änderung in der Canvas-Hierarchie - selbst bei einer Änderung der Position [12]. Wenn der Anwender also die Zylinderansicht rotierte, führte dies zu einem enormen Performance-Loch.

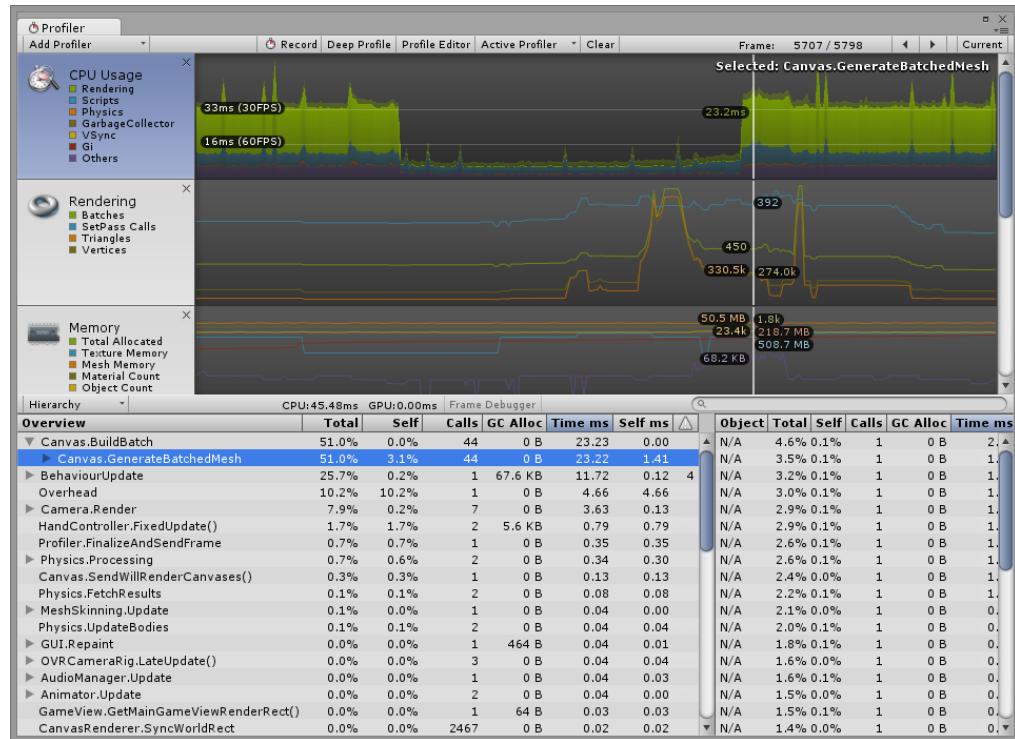


Abbildung 5.17: Performance-Einbussen bei BuildBatch()

Als Workaround um dieses Problem wurde beim Rotieren des Zylinders nicht der Zylinder rotiert, sondern die Welt. Das hat zur Folge, dass der Zylinder selbst unberührt bleibt und kein BuildMesh()-Call ausgelöst wird. Leider hat diese Lösung sehr grosse Einschränkungen: Physik wird nicht mehr funktionieren, weil sich die Welt immer wieder bewegt, auch äussere Zylinderschichten müssen sich mitdrehen, da sonst das gleiche Problem auftritt und alle Elemente müssen umständlich in einem Transform, der alle Welt-Objekte beinhaltet, organisiert werden.

Ein weiteres Problem bei diesem Workaround ist, dass sich die Skybox nicht mitdreht, wenn die Welt rotiert wird. Das hat umgekehrt zur Folge, dass es für den Anwender so aussieht, als würde er den Himmel rotieren, wenn er den Zylinder rotiert. Als Lösung dafür wurde eine weitere Kamera ausserhalb der Welt platziert, die nur für die Aufnahme des Hintergrunds zuständig ist. Die Kamera bleibt, wo sie ist, wenn die Welt gedreht wird, und entsprechend bleibt auch der Hintergrund still.

Neben der Performance-Einbussen existierender Elemente, traten auch Probleme beim *Initialisieren* von Objekten auf. Die Schwierigkeiten wurden sichtbar bei der Implementierung des Interfaces für die Playlist.

Beim *Listen*-Modus kommt es oft vor, dass eine sehr grosse Anzahl Lieder einem Kriterium gerecht werden. Da jede Zeile in einer Liste in uGUI ein eigenes GameObject ist, hat dies zur Folge, dass entsprechend viele GameObjects *on-the-fly* generiert werden müssen. Selektiert der Anwender also 1000 Songs, werden 1000 GameObjects erstellt, was zu einem enormen Einfall der Framerate führt.

Eine Lösung dafür bietet eine Komponente namens *PooledList* [18], welche immer nur eine kleine Anzahl Objekte erstellt und lediglich den Inhalt mit den aktuell sichtbaren austauscht.

Leider ist das Projekt nicht sehr gut dokumentiert und baut auf einem GridLayout auf. In IMVR wird jedoch ein VerticalLayoutGroup verwendet, welches nicht mit dem Plugin kompatibel ist. Ein Versuch, die Playlist in ein Grid-Layout zu konvertieren schlug fehl.

Als Lösung wurde schliesslich selbst eine sehr simple, ähnliche Komponente entwickelt: die LazyList. Diese akzeptiert eine Liste von LazyGameObjects - eine selbst erstellte Klasse, die auf Anfrage mithilfe eines Callbacks, der im Konstruktor übergeben wird, jeweils ein GameObject erstellt - und nutzt diese, um das vertikale Layout zu füllen. Dabei werden zwei Tricks angewendet:

1. Nur die aktuell sichtbaren Elemente werden angezeigt. Der Rest wird in einem deaktivierten GameObject zwischengelagert und vollständig ignoriert.
2. Alle Elemente werden nur auf Anfrage initialisiert. Das heisst, dass anfangs kein einziges Objekt existiert und diese Schritt für Schritt beim Scrollen geladen werden.

Mit diesen zwei Verbesserungen ist es nun möglich, sofort eine beliebig grosse Liste zu erstellen. Der Nachteil ist, dass irgendwann der Speicher ausgehen wird, da momentan nicht mehr angezeigte Objekte nicht aufgeräumt werden. Ein ähnlicher Ansatz wie bei der *PooledList* wäre ideal.

6 Testergebnisse

7 Projektmanagement

Dieses Kapitel soll dienen, um einen organisatorischen Rückblick auf das Projekt zu machen und die Ist-Situation mit der Soll-Situation zu vergleichen.

7.1 Soll-Ist Gegenüberstellung

In der folgenden Tabelle wird eine Gegenüberstellung des Soll-Zustands zum Ist-Zustand gemacht. Die Elemente, die sich mit der Bilderbibliothek befassten, wurden aufgrund der Änderung in der Aufgabenstellung durchstrichen und entziehen sich einer Bewertung.

Nr.	Soll	Ist
1. Elementares		
	✓ 1.1 Die Applikation nutzt die Oculus Rift im direkten Modus für die Ausgabe.	Die Applikation nutzt die Oculus Rift im direkten Modus für die Ausgabe.
	✓ 1.2 Die Applikation ist vollständig mit den Händen bedienbar.	Die Applikation ist vollständig mit den Händen bedienbar, verwendet jedoch den Blick also weitere Eingabemöglichkeit.
	✓ 1.3 Die Applikation erlaubt den Zugriff auf alle gefundenen, validen Dateien.	Die Applikation erlaubt den Zugriff auf alle gefundenen, validen Dateien.
	✓ 1.4 Bilder können angeschaut werden, Musik kann angehört werden.	Musik kann angehört werden und relevante Bilder können angeschaut werden.
	✓ 1.5 Die Applikation kann jederzeit beendet werden.	Die Applikation kann per Blick auf die entsprechende Fußplatte jederzeit beendet werden.
*	1.6 Die Applikation erreicht eine stetige Framerate von mindestens 60fps auf dem Referenzsystem ¹ .	Es gibt noch Einbrüche in der Framerate bei Szenenwechsel.
*	1.7 Die Applikation kann min. 100'000 Dateien verarbeiten.	Das Anzeigen in der Übersicht wird aufgrund von Vertices-Beschränkungen problematisch. Hinzufügen von 100 000 Elementen zur Playlist ist jedoch möglich.

¹Intel(R) Xeon CPU E5-1650 v3 @ 2.50Ghz mit 32GB RAM und einer NVIDIA GeForce GTX 980 Grafikkarte.

Nr.	Soll	Ist
✓	1.8 Hände werden abstrakt dargestellt.	Hände werden abstrakt mithilfe von Partikeln dargestellt.
✓	1.9 Die Hintergrundmusik kann von der Musikbibliothek ausgewählt werden.	Die Hintergrundmusik kann von der Musikbibliothek ausgewählt werden.

2. Übersicht (Bilder)

3. Detailansicht (Bilder)

4. Übersicht (Musik)

✓	4.1 Unterstützt mit ID3-Tags versehene MP3.	Unterstützt mit ID3-Tags versehene MP3.
✓	4.2 Dateien werden alphabetisch gruppiert nach Artisten dargestellt.	Dateien werden im Browse-Modus alphabetisch gruppiert nach Artisten dargestellt.
✗	4.3 Dateien können nach Album, Genre und Jahr gruppiert werden	Eine Gruppierfunktion wird momentan nicht unterstützt.
*	4.4 Alben können anhand ihres Covers wie Bilder dargestellt werden.	Alben werden in der Detailansicht mit einem Cover dargestellt.
✓	4.5 Unterstützt weitere Musikformate (Ogg Vorbis, FLAC, M4A, APE, TAK, etc.)	Alle von CSCore [11] unterstützten Formate sind abspielbar ² .
*	4.6 Dateien können ungruppiert dargestellt werden.	Im Listen-Modus werden Songs unabhängig des Künstlers oder des Albums betrachtet.

5. Detailansicht (Musik)

✓	5.1 Album mit einer Liste von Liedern wird dargestellt.	Album mit einer Liste von Liedern wird dargestellt.
✓	5.2 Eine Datei kann zur Wiedergabe ausgewählt werden.	Eine Datei kann zur Wiedergabe ausgewählt werden.
✓	5.3 Die Wiedergabe wird visuell untermauert (Spektrogramm).	Es gibt diverse visuelle Effekte zur Untermauerung der Musik.
*	5.4 Informationen zum Artisten werden dargestellt (Ort, Beschreibung, Gründungsjahr)	Die Daten sind vorhanden, aber es werden noch nicht alle dargestellt.

6. Indexierung

✓	6.1 Der Benutzer kann einen oder mehrere Ordner angeben, die nach Bilder und Musik gescannt werden.	Der Benutzer kann einen oder mehrere Ordner angeben, die nach Bilder und Musik gescannt werden.
---	---	---

²MP3, WAVE, FLAC, AAC, AC-3, WMA, Ogg Vorbis

Nr.	Soll	Ist
✓	6.2 Die gefundenen Dateien werden mit diversen Kennwerten indexiert.	Die gefundenen Dateien werden mit diversen Kennwerten indexiert.
✓	6.3 Dateien, die nicht mehr in einem der Medienordner liegen, werden aus dem Index gelöscht.	Dateien, die nicht mehr in einem der Medienordner liegen, werden aus dem Index gelöscht.
✗	6.4 Die indexierten Ordner können innerhalb der Applikation angegeben werden	Die Indexierung ist vollständig getrennt von IMVR.
✗	6.5 Die Indexierung kann innerhalb der Applikation durchgeführt werden.	Die Indexierung wird vorher und für sich durchgeführt.

7. Sonstige Funktionen

*	7.1 Es können Filter über die Musik gelegt werden (Low-Pass, High-Pass, Compressor, etc.)	Momentan werden keine Filter unterstützt, die Musik ist jedoch in einem Mixer eingebunden und für Filtereffekte vorbereitet.
	7.2 Bildergruppen können als "Diashow" angesehen werden.	
✗	7.3 Es ist eine Netzwerkfunktion vorhanden, die es erlaubt, die Fotosammlung zu zweit anzuschauen.	Es werden keine Netzwerkfunktionen angeboten.
*	7.4 Gesamtstatistiken können angezeigt werden (Bildalter / Anzahl, Auflösung / Anzahl, Auflösung / Anzahl, Kartenchart mit Anzahl Fotos)	Es sind ein paar Statistiken zu Metadaten im System vorhanden.
✗	7.5 Online-Services können als Datenquelle angegeben werden (Flickr, Spotify, etc.)	Es werden keine dynamischen Online-Services unterstützt.

Tabelle 7.1: Eine Gegenüberstellung der Soll-Kriterien mit dem Ist-Zustand

7.2 Zeitplan

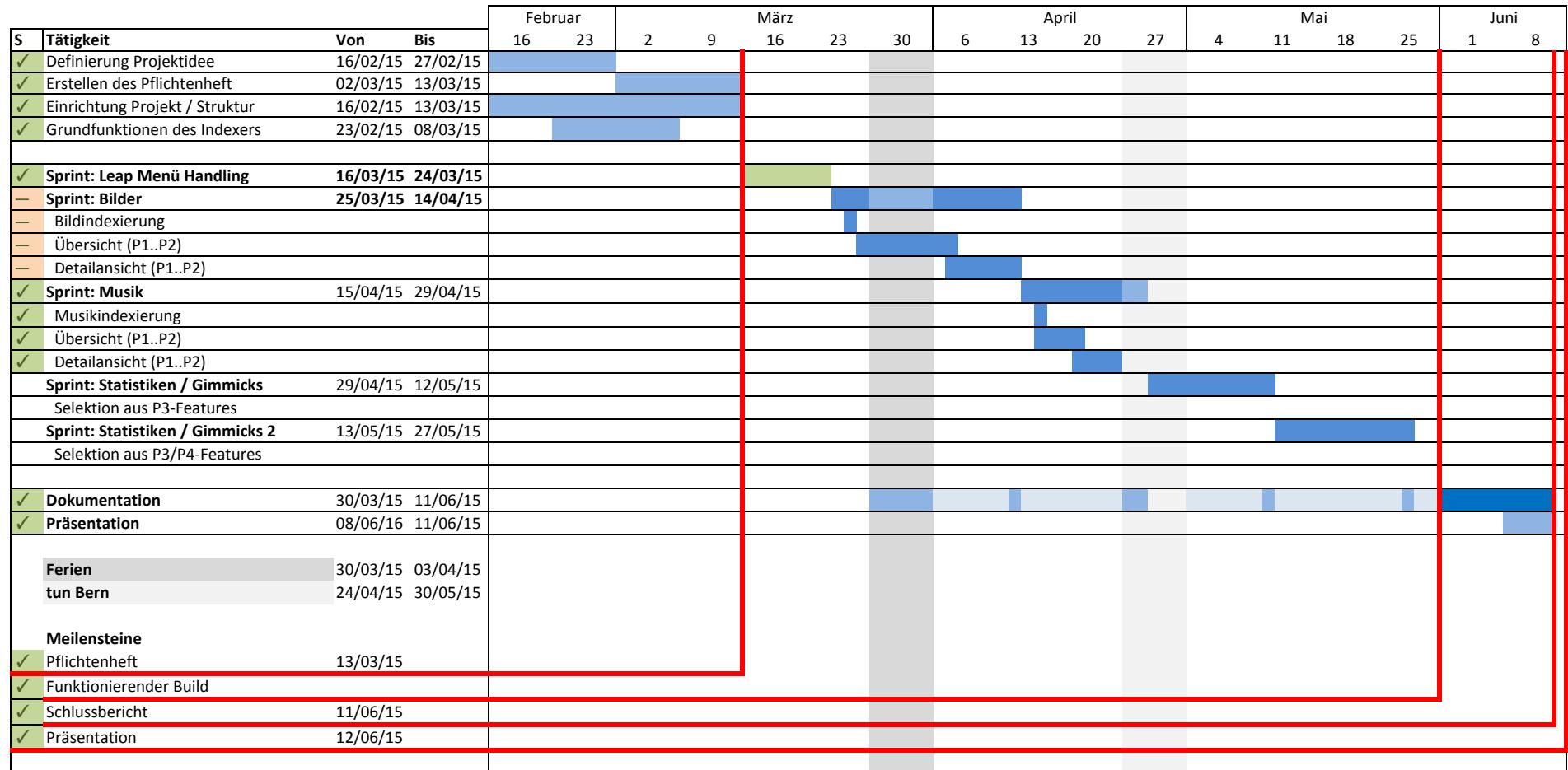


Abbildung 7.1: Der Zeitplan mit Sprints und Prioritäten.

8 Schlussbetrachtung

Zum Schluss soll kurz ein Blick auf die Gegenwart, die Vergangenheit und die Zukunft geworfen werden. War das Projekt erfolgreich? Wo lagen die *Pitfalls*? Wie geht es weiter? Diese Fragen sollen an dieser Stelle beantwortet werden.

8.1 Ergebnis

Wenn man vom gestrichenen Bilderbereich absieht, konnten rückblickend alle geplanten Haupt-features implementiert werden. Leider mussten einige Kompromisse gemacht werden und zum Teil war der Fortschritt nicht so schnell, wie geplant.

Es gibt mehrere Informationen, die aus den API Schnittstellen heruntergeladen wurden, jedoch letztendlich keine Verwendung fanden. Darunter gehören beispielsweise die Biografien und beschreibenden Begriffe für Künstler, oder auch die zusätzlichen Fotos, die von diesen gesammelt werden.

Die Performanceansprüche wurde trotz Versuchen, die Leistung zu verbessern, nicht erreicht. Dies könnte jedoch mit den gesammelten Erfahrungen verbessert werden.

Auch beim Indexierungstool sind noch Mängel vorhanden. Die Datenbank wird bei jedem Durchgang komplett neu erstellt, statt mit Daten ergänzt, da bei der Datenstruktur noch die Mittel fehlen, um die Datenintegrität sicherzustellen bzw. zu erleichtern.

Alles in allem kann jedoch ein guter Fazit gezogen werden. Die Indexierung funktioniert und gibt sinnvolle Log-Meldungen und die Applikation verfügt über die Bereiche, die geplant waren. Im Detail fehlen zwar noch ein paar Funktionen in den einzelnen Bereichen, diese sind jedoch grundsätzlich implementiert und vollständig navigierbar.

8.2 Reflexion

Grundsätzlich ist das Ergebnis zufriedenstellend ausgefallen. Es gibt allerdings zahlreiche Punkte, welche besser hätten gelöst werden können.

Aufgrund der experimentellen Natur des Projektes war die Planung teilweise nicht so ausgearbeitet und zuverlässig, wie sie es in einem herkömmlichen Projekt wäre. Durch das Verändern von Funktionalität, dem wiederholten Refactoring, und der geänderten Zielsetzung entstand zum Teil zusätzliche Komplexität und zu einem gewissen Masse auch "Code-Smell" [17].

Eine Lektion, die ebenfalls gelernt wurde, ist, dass Computergrafik unberechenbar sein kann und viele Tücken bereithält. Mehrmals vergingen Stunden, bis die Ursache eines Bugs oder einer fehlerhaften Funktion herausgefunden werden konnten, gefolgt von zahlreichen Stunden, die zur Behebung aufgewendet werden mussten.

Im Nachhinein kann auch gesagt werden, dass Unity nicht die ideale Wahl für die Umsetzung der Projektidee war. Die Umgebung ist zu sehr um Assets und die Idee von geschlossenen Spielen aufgebaut, und eignet sich nicht sehr gut für dynamische Inhalte. Eventuell hätte die Unreal Engine 4 [2] in dieser Hinsicht bessere Resultate geliefert. Es muss allerdings auch gesagt werden, dass eine Einarbeit in die Unreal Engine in Hinsicht auf das Vorprojekt keinen Sinn gemacht hätte und somit keine grosse Wahl bestand.

Nichtsdestotrotz bzw. eben aufgrund der zahlreichen Hürden erlaubte die Entwicklung von IMVR einen Einblick in viele Features und Eigenheiten von Unity und fiel somit sehr informativ und lehrreich aus. Meshes wurden generiert, ein eigenes Input Modul wurde geschrieben, das UI wurde erforscht und die Grenzen und Asset-Gebundenheit wurde erkannt.

Eine nicht ganz so offensichtliche Kompetenz, die ebenfalls im Verlaufe des Projektes angeeignet und verbessert werden konnte, ist der Umgang von grossen Projekten mit Git bzw. GitHub. Die Ordnerstruktur war zwar von Beginn an geplant, wurde jedoch während des Projekts erweitert und optimiert. Die Verwendung eines Version Control System (VCS) erwies sich allgemein als nützlich, wenn kurz etwas getestet werden sollte oder bei Vergleichen mit älteren Revisionen, wenn ein Feature nicht mehr funktionierte.

Ich kann also abschliessend sagen, dass ich mir dank des Projektes eine solide Basis an Fachwissen in Unity und Computergrafik aneignen konnte, und das Projekt somit als Erfolg betrachte.

8.3 Ausblick

Die Grundfunktionalität von IMVR ist zwar implementiert, es bleibt jedoch noch viel Raum für Verbesserungen und Erweiterungen. Es gibt mehrere Richtungen, in die sich das Tool weiterentwickeln könnte.

Eine Möglichkeit wäre die simple Erweiterung der Funktionalität in die Breite. Damit ist gemeint, dass weitere Ansichten auf die Daten implementiert werden könnten. Momentan ist es nur möglich, alle Künstler auf dem lokalen System in einer Zylinderstruktur darzustellen, aber nichts spräche dagegen, dasselbe für die Alben oder Lieder zu machen. Ebenfalls möglich wären Sortier- und Gruppierungsfunktionen, wie anfangs geplant war.

Anders gesehen, könnte man die Applikation auch noch weiter in die Tiefe entwickeln. Seit Frühjahr 2015 bietet Oculus Rift ein Audio SDK an, welches 3D-Sound mit Stereokopfhörer simulieren kann. Erste eigene Tests führten zum Schluss, dass die Implementierung viel umfangreicher und robuster ist, als Unitys eingebauter 3D-Sound. Es gäbe sicher ein paar interessante Einsatzgebiete für dieses Feature.

Weiterhin könnte man auch noch weitere Metadaten hinzuziehen oder die vorhandenen besser auswerten. Verschiedene Web-Services bieten verschiedene Arten von Daten an, die miteinander kombiniert werden könnten, um neue Schlüsse zu ziehen. Während des Projektes kam beispielsweise

die Idee auf, Selektionen von Metadatenbereichen als selbst definierbare "Presets" abzuspeichern, damit eine leichtere und aussagekräftigere Suche möglich wäre.

Auch der Weg zur vollumfänglichen Mediacenter-Lösung stände offen. Würde man die Bilderbibliothek wieder hinzunehmen, wäre bereits der erste Schritt getan. Die Einbindung von Videodaten würde aufgrund der Limitierungen von Unity jedoch problematisch werden. Jedoch wäre auch dieses Problem keineswegs undurchführbar - zumal es im Asset Store bereits passende Plugins gibt.

Schliesslich gäbe es die Möglichkeit, andere Geräte für die Handerkennung oder für die stereoskopische Visualisierung anzuwenden. Von Leap Motion ist z.B. ein neuer Sensor unter dem Namen *Dragonfly* in Arbeit [5], der in Headsets eingebaut werden soll, und bessere Auflösung sowie ein grösseres Sichtfeld verspricht. Ein weiteres vielversprechendes Projekt ist Nimble Sense [15], dessen Team nun mit Oculus VR zusammenarbeitet.

Der Einbau dieser oder weiterer Geräte würde zu einer zuverlässigeren Erkennung der Hände führen und somit das Erlebnis angenehmer gestalten. Auch durch den Einsatz von HMDs mit höherer Auflösung könnte die Applikation einfach und nachhaltig verbessert werden.

Selbständigkeitserklärung

Ich/wir bestätige/n, dass ich/wir die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der im Literaturverzeichnis angegebenen Quellen und Hilfsmittel angefertigt habe/n. Sämtliche Textstellen, die nicht von mir/uns stammen, sind als Zitate gekennzeichnet und mit dem genauen Hinweis auf ihre Herkunft versehen.

Ort, Datum: Biel, 11.06.2015

Name Vorname: Meer Simon

Unterschrift:

Literaturverzeichnis

- [1] T. Auensen, "echonest-sharp." [Online]. Available: <https://github.com/torshy/echonest-sharp>
- [2] I. Epic Games, "What is Unreal Engine 4." [Online]. Available: <https://www.unrealengine.com/what-is-unreal-engine-4>
- [3] Google, "Android wear materials." [Online]. Available: <https://developer.android.com/design/downloads/index.html>
- [4] M. Heath, "NAudio." [Online]. Available: <https://naudio.codeplex.com/>
- [5] D. Holz, "Leap Motion Sets a Course for VR." [Online]. Available: <http://blog.leapmotion.com/leap-motion-sets-a-course-for-vr/>
- [6] Z. Kinstner, "Hover VR interface kit." [Online]. Available: <https://github.com/aestheticinteractive/Hover-VR-Interface-Kit>
- [7] Lovelock, "How do I use PSIZE in a Unity 4.5.4 shader?" [Online]. Available: <http://answers.unity3d.com/questions/807548/how-do-i-use-psize-in-a-unity-454-shader.html>
- [8] A. Naletto, "GetOutputData and GetSpectrumData, what represent the values returned?" [Online]. Available: <http://answers.unity3d.com/questions/157940/getoutputdata-and-getspectrumdata-they-represent-t.html#answer-158800>
- [9] L. Oculus VR, "Best practices guide," 2015. [Online]. Available: http://static.oculus.com/sdk-downloads/documents/Oculus_Best_Practices_Guide.pdf
- [10] F. Patin, "Beat detection algorithms," 2003. [Online]. Available: http://www.gamedev.net/page/resources/_/technical/math-and-physics/beat-detection-algorithms-r1952
- [11] F. R., "CSCore - .NET audio library." [Online]. Available: <https://github.com/filoe/cscore>
- [12] Revolter, "Why does Canvas.BuildBatch happen at random and how can I avoid it during runtime?" [Online]. Available: <http://forum.unity3d.com/threads/why-does-canvas-buildbatch-happen-at-random-and-how-can-i-avoid-it-during-runtime.277966/>
- [13] O. Trutner, "Signing Amazon product advertising API requests ? the missing C# WCF sample," 2009. [Online]. Available: <https://flyingpies.wordpress.com/2009/08/01/17/>
- [14] Unity, "Low-level Native Plugin Interface." [Online]. Available: <http://docs.unity3d.com/Manual/NativePluginInterface.html>
- [15] N. VR, "Nimble Sense: Bring Your Hands into Virtual Reality & Beyond." [Online]. Available: <https://www.kickstarter.com/projects/nimblevr/nimble-sense-bring-your-hands-into-virtual-reality>

- [16] Wikipedia, "Protocol buffers," 2015, [Online; accessed 24-May-2015]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Protocol_Buffers&oldid=662187371
- [17] ——, "Smell (Programmierung)," 2015, [Online; Stand 1. Juni 2015]. [Online]. Available: [http://de.wikipedia.org/w/index.php?title=Smell_\(Programmierung\)&oldid=137769083](http://de.wikipedia.org/w/index.php?title=Smell_(Programmierung)&oldid=137769083)
- [18] WillzZz, "PooledList." [Online]. Available: <https://github.com/WillzZz/pooled-list>

Abbildungsverzeichnis

2.1	Die Leap Motion im Einsatz	2
3.1	Funktionsweise der Oculus Rift	5
3.2	Aufsatz für die Leap Motion	6
3.3	Eine Übersicht der Technologien und wie sie verbunden sind.	7
3.4	Grober Überblick des Programmablaufs	9
3.5	Linearer Ablauf von IMVR	11
3.6	Klassendiagramm für die Klassen im IMVR.Commons Package	12
3.7	Klassendiagramm des Indexers	15
3.8	Klassendiagramm des Interfaces der Applikation (View-Behavior)	18
3.9	Klassendiagramm des Music Arms	20
3.10	Klassendiagramm des Ring Panels, womit der Anwender Modi auswählt	22
3.11	Klassendiagramm des Teils, der die Daten der Datenbank verwendet	23
3.12	Klassendiagramm des Bilddaten-Handlings	25
3.13	Klassendiagramm des Musikdaten-Handlings	26
3.14	Klassendiagramm der Effekte zur Musikvisualisierung	28
3.15	Klassendiagramm der allgemeinen Effekte	29
3.16	Klassendiagramm des Abstraktionslayers für die Leap Motion	30
4.1	Der Datenfluss, den die Files beim Indexieren nehmen.	34
5.1	Aufbau der Szene	38
5.2	Vogelperspektive auf die Szene	39
5.3	Aufbau einer Fussplatte	40
5.4	Darstellung des Music Arms	41
5.5	Bestimmung der momentanen Armseite	42
5.6	In-Game Screenshot zur Bedienung der Slider-Balken	43
5.7	Die ursprüngliche Visualisierung der Hand	44
5.8	Die finale Visualisierung der Hand mit Partikeln	45
5.9	Visualisierung der Musiksammlung innerhalb eines Point Charts	47
5.10	Beats werden mithilfe von Wellen dargestellt (rot umrahmt)	49
5.11	Die Ring-Meshes werden mithilfe von <i>Quads</i> generiert	50
5.12	Der verwendete Avatar mit Kopf und Armen	53
5.13	Illustration der Funktionsweise eines Atlases	54
5.14	Überschneidung zweier Canvas-Elemente	56
5.15	Probleme mit dem Rendering von komplexen Canvas	57
5.16	Aufbau des Ladebildschirms	58
5.17	Performance-Einbussen bei BuildBatch()	59
7.1	Der Zeitplan mit Sprints und Prioritäten.	65

Tabellenverzeichnis

2.1	Übersicht der eingesetzten Software	4
2.2	Übersicht der eingesetzten Hardware	4
3.1	Erklärung der wichtigsten allgemeinen Klassen	13
3.2	Erläuterung der Metadaten	14
3.3	Erklärung der wichtigsten Klassen des Indexers	16
3.4	Erklärung der wichtigsten Interface-Klassen	19
3.5	Erklärung der wichtigsten Music-Arm-Klassen	21
3.6	Erklärung der Klassen des Ring-Panels	22
3.7	Erklärung der wichtigsten Klassen von IMVR.Data	24
3.8	Erklärung der Klassen von IMVR.Data.Image	25
3.9	Erklärung der wichtigsten Klassen von IMVR.Data.Music	27
3.10	Erklärung der FX-Klassen, die sich mit der Musik-Visualisierung befassen	29
3.11	Erklärung der allgemeinen FX-Klassen	30
4.1	Die Projekte in Auxiliary Tools	32
4.2	Eine Übersicht von verfügbaren Online-Datenquellen.	33
5.1	Beschreibung des Aufbaus der Fussplatten	40
7.1	Eine Gegenüberstellung der Soll-Kriterien mit dem Ist-Zustand	64