Entwicklung eines Beat Saber-Klons in Unity und Levelgenierung durch Audioanalyse

Tobias Jansing

8. September 2019

Projektdokumentation für das Modul *Praktikum Virtual Reality* im Sommersemester 2019

Betreut von
M. Sc. Marcus Riemer, M. Sc. Florian Habib

An der
Fachhochschule Wedel

Inhaltsverzeichnis

1 Einleitung **1**

2 Projektidee **1**

3 Beschreibung des Spiels **1**
3.1 Entwickeltes Endprodukt 1
3.2 Voraussetzungen 1
3.3 Menüführung 1
3.3.1 Hauptmenü 2
3.3.2 Optionsmenü 2
3.3.3 Spielmenü 2
3.3.4 Score-/GameOver-Menü 2
3.4 Im Spiel selbst 2
3.5 Laden von Audiodateien und Mappings 3
3.5.1 BeatSaber Mapping Format 3
3.5.2 Korrekte Songauswahl 3

4 Kompilierung des Projekts **4**
4.1 Verwendete Bibliotheken 4
4.2 Build-Vorgang 5

5 Programmstruktur **5**
5.1 Szenen 5
5.2 Packages 6
5.2.1 Menu 6
5.2.2 Audio 6
5.2.3 Json 7
5.2.4 Game 7
5.2.5 Global 8
5.2.6 Test 8

6 Audioanalyse **8**
6.1 Implementierung 9

Literaturverzeichnis **12**

1 Einleitung

2 Projektidee

Die Grundidee des Projekts bestand darin, einen Klon des erfolgreichen Virtual Reality-Spiels *Beat Saber* zu entwickeln. Dabei handelt es sich um ein audiobasiertes Rhythmusspiel, in dem zum Takt der Musik auf den Spieler zufliegende Blöcke getroffen werden müssen. Eine sehr begrenzte Musikauswahl führte zusätzlich zu der Idee, eine prozedurale Levelgenerierung durch eine Analyse von Audiodaten durchzuführen, um das Spielen beliebiger Lieder zu ermöglichen. Das entsandene Projekt wurde mit Unity entwickelt.

3 Beschreibung des Spiels

3.1 Entwickeltes Endprodukt

Entstanden ist ein Spiel, in dem alle wichtigen Grundfunktionen von Beat Saber implementiert wurden. Der Spieler startet in einem Hauptmenü, in dem die Schwierigkeit und eine Audiodatei ausgewählt werden können. Anschließend wird entweder ein bereits existierendes Level aus dem Cache geladen oder die Audiodatei zwecks Levelgenerierung analysiert. In einer neuen Szene absolviert der Spieler den Beat Saber Core Game Loop mit heranfliegenden Blöcken, die es zu treffen gilt und Hindernissen, denen ausgewichen werden muss. Anschließend wird ein *Score*-Menü angezeigt. Schafft der Spieler es nicht, das Level abzuschließen, wird ein ähnlicher *Game Over*-Screen angezeigt. Anschließend kann im Hauptmenü ein neues Lied ausgewählt werden.

3.2 Voraussetzungen

Voraussetzung für das Spiel ist die Verfügbarkeit eines *Oculus Rift* Virtual Reality Headsets. Möglich ist dabei sowohl die Verwendung der *Oculus Rift* sowie der kürzlich erschienenen *Oculus Rift S*. Die Anwendung wurde vollständig mit einer Oculus Rift S entwickelt. Weiterhin sind beim Testen der Rift im Virtual Reality-Labor der Hochschule Wedel verzeinzelte Probleme aufgetreten. Daher wird ausdrücklich die Verwendung einer Rift S empfohlen.

Für die Verwendung und Kalibrierung des Headsets ist eine Firmware von Oculus notwendig (1). Weitere geläufige VR-Anwendungen wie beispielsweise *SteamVR* werden nicht benötigt.

3.3 Menüführung

Bereits in den Menüs hält der Spieler zwei Laserschwerter in den Händen. Vom rechten Controller aus wird eine Linie auf einen vor dem Spieler befindlichen Canvas gezeichnet, die zum Zielen verwendet wird. Der Controller kann nun

verwendet werden, um mit Menüelementen wie Buttons oder Slidern zu interagieren, die sich an der selben Position wie der Endpunkt der Linie auf dem Canvas. Das Spiel und die Menüführung wurde so entworfen, dass der Spieler das VR-Headset nicht abnehmen muss. Sämtliche Interaktionen sind in VR mit den Controllern möglich.

3.3.1 Hauptmenü

Im simplen Hauptmenü befinden sich Buttons, um in das Optionsmenü oder das Spielmenü zu wechseln. Weiterhin kann die Anwendung über einen *Quit*-Button geschlossen werden.

3.3.2 Optionsmenü

Das Optionsmenü verfügt über einige visuelle Elemente wie Sliders, mit denen interagiert werden kann. Jedoch haben diese aus Zeitgründen keinerelei Funktionalität, die über den visuellen Aspekt hinausgehen. Das Menü ist für den Rest des Spiels also gänzlich irrelevant.

3.3.3 Spielmenü

Im Spielmenü wird eine Audiodatei ausgewählt, die gespielt werden soll. Ebenso kann eine Schwierigkeit ausgewählt werden. Die Anpassung der Schwierigkeit verändert bestimmte Parameter für die spätere Audioanalyse.

Klickt der Spieler auf den *Load File*-Button, so öffnet sich rechts vom Menücanvas ein einfacher Dateiexplorer, in dem eine Audiodatei ausgewählt werden kann. Mithilfe eines Sliders kann zusätzlich die Schwierigkeit angepasst werden.

3.3.4 Score-/GameOver-Menü

Das Score-Menü und das GameOver-Menü sind bis auf die angezeigte Überschrift identisch. Die vom Nutzer erreichten Score-Werte werden hier angezeigt. Über einen Button kann wieder zurück ins Hauptmenü gesprungen werden.

3.4 Im Spiel selbst

Der Core Game Loop gleicht BeatSaber. Heranfliegende Blöcke müssen im Takt und aus der richtigen Richtung getroffen werden. Zusätzlich fliegen Hindernisse auf den Spieler zu, denen dieser ausweichen muss. Der Spieler hält ein rotes Laserschwert in der linken und ein blaues Laserschwert in der rechten Hand. Ebenso gibt es rote und blaue Blöcke. Die Farben von Schwert und Block müssen übereinstimmen, ansonsten wird der Treffer als Fehler gewertet. Ebenso als Fehler gewertet werden Blöcke, die der Spieler gar nicht trifft, oder jeder Kontakt mit einem Hinderniss. Für die Berechnung der Score-Werte gibt es einen *Multiplier*- und einen *Combo*-Wert. Diese basieren darauf, wie viele Blöcke der Spieler ohne Fehler hinereinander getroffen hat. Jeder Fehler setzt die Werte zurück.

Weiterhin gibt es einen für den Spieler unsichtbaren Wert für die Lebenspunkte des Spielers. Dieser reicht von 100 (Maximum) bis 0 (Game Over) und wird lediglich durch eine rote OnScreen-Overlay Grafik visualisiert, deren Alphawert entsprechend angepasst wird.

3.5 Laden von Audiodateien und Mappings

3.5.1 BeatSaber Mapping Format

Die durch den Beat Saber-Klon generierten Mappings werden im BeatSaber-Format gespeichert. Dabei handelt es sich jeweils um einen Ordner mit einer *Info.dat*-Datei und mindestens einer Datei, die die Mappings für einen bestimmten Schwierigkeitsgrad enthalten, beispielsweise *Easy.dat*. Bei diesen .dat-Dateien handelt es sich eigentlich um JSON-Dateien. Zusätzlich befindet sich dort eine Audiodatei im .ogg-Format, die jedoch die Dateiendung *.egg* zugewiesen bekommt. Der Grund für diese Verschleierung ist aktuell unbekannt, hier könnten höchstens Vermutungen angestellt werden.

3.5.2 Korrekte Songauswahl

Die korrekte Auswahl von Liedern mit dem richtigen Schwierigkeitsgrad erfordert Zusatzwissen. Der standardmäßig ausgewählte Dateipfad für den im Spielmenü zu öffnenden Dateiexplorer ist */Assets/Resources/SongData/BeatMappings*. Dieser Ordner dient als Caching für sämtliche erstellten Mappings, daher sollten auch die Audiodateien dort hinterlegt werden.

Neben der Audiodateu wird eine *.info*-Datei sowie eine Datei für den Schwierigkeitsgrad benötigt, beispielsweise *Easy.dat*. Wenn ein Mapping erstellt wird, dann wird im Caching-Ordner aus Subordner erstellt, der diese Daten enthält. Der Name des Ordners entspricht dem Namen der Audiodatei. Audiodateien können theoretisch von jedem beliebigen Ort geöffnet werden. Der Name der Audiodatei ist ausschlaggebend für das Auffinden von Mappings und die Erstellung der Cache-Dateien. Wenn bereits ein Subordner für die Audiodatei existiert wird in diesem Ordner anschließend geprüft, ob ein Mapping für den ausgewählten Schwierigskeitsgrad existiert. Falls dies der Fall ist, wird diese Datei geladen und das Mapping daraus generiert. Daher ist es also sinnvoll, sämtliche Audiodateien direkt in einem Ordner mit identischem Dateinamen im BeatMapping-Order abzulegen. Wenn ein Mapping für den gewählten Schwierigkeitsgrad noch nicht vorhanden ist oder der Suborder ganz fehlt, wird ein neues Mapping durch eien Audioanalyse generiert und im Cache abgelegt.

Es können auch externe, fertige Mappings von Seiten wie *bsaber.com* verwender werden. Die Audioanalyse wird somit gänzlich umgangen. Wenn diese verwendet werden, so ist es wichtig, dass der Ordername auch hier exakt dem Dateinamen entspricht und der Ordner sich im BeatMapping-Ordner befindet. Weiterhin muss eine Schwierigkeit ausgewählt werden, die von diesem externen Mapping angeboten wird - ansonsten wird automatisch die Audioanalyse ein Mapping generieren und speichern, da keine Datei .dat-Datei gefunden wurde.

// TODO bpm und info file weiter erwähnen. erstellen einiger mappings kann info file überschreibne. vielleiht ein kapitel: bekannte probleme? Besonders beim Verwenden externer Mappings können Probleme auftreten

Ein Problem, das während der Entwicklung einige Mal aufgetreten ist, ist das Überschreiben von info.dat-Dateien. Selbst wenn bereits eine info.dat-Datei, sowie ein Mapping für einen bestimmten Schwierigkeitsgrad vorhanden ist, so wird trotzdem ein Mapping generiert, wenn für den ausgewählten Schwierigkeitsgrad kein Mapping existiert. In diesem Fall wird auch die info.dat neu geschrieben. Der BPM-Wert in der Datei wird standardmäßig auf 1 gesetzt, da wir keine Möglichkeit haben, die BPM ohne extrem großen Mehraufwand zu ermitteln. In diesem Fall kann so die info-Datei eines Custom Mappings von bsaber.com überschrieben werden. Die Geschwindigkeit für das Custom Mapping ist dann falsch. Dieses Problem könnte leicht behoben werden, indem geprüft wird, ob bereits eine info.dat-Datei für das bestimmte Mapping existiert.

4 Kompilierung des Projekts

4.1 Verwendete Bibliotheken

Es wurden diverse externe Bibliotheken verwendet. Der Sourcecode für diese Bibliotheken befindet sich direkt im Projekt, es ist also nicht erforderlich, weitere Schritte zu unternehmen. Alle Bibliotheken und Plugins sind im *Plugins*-Ordner zu finden - sämtlicher weitere Code (d.h. alle im Logic-Ordner befindlichen Dateien) wurden selbst implementiert.

Es wurden sowohl Bibliotheken aus dem Unity Asset Store als auch Bibliotheken von externen Quellen wie Github-Repositories verwendet. Folgende Bibliotheken wurden benutzt:

- *Oculus Unity Framework* (2): Zugrunde liegende Bibliothek für sämtliche Basis VR-Features. Bietet vorgefertigte Prefabs für Kameras, Spielersteuerung, VR-Eventsystem etc.
- *Post Processing Stack* (3): Unity-Bibliothek für Postprocessing. Postprocessingeffekte wie Motion Blur, Antialiasing sowie Color Gradic und weitere nachträgliche Farbanpassungen wurden verwendet.
- *MkGlow* (4): Shaderbibliothek für Glow-Shader. Diverse Objekte im Spiel (zum Beispiel die Laserschwerter und die Laserlichteffekte) besitzen diesen Glüheffekt.
- *SimpleFileBrowser* (5): Ein einfacher Dateiexplorer, der auf einem Canvas zu sehen ist.
- *EzySlice* (**?**): Mit EzySlice kann das Durchschneiden von Objekten simuliert werden. Die Bibiliothek bietet mehrere Funktionen an, um ein beliebiges Mesh zu zerteilen. Dabei werden zwei neue GameObjects erstellt, die bei Bedarf erneut zerteilt werden können.

- *Spherical Fog* (7): Dabei handelt es sich um ein simples Shader-Script
- *NLayer* (8): Audio-Decoder, mit dem zur Laufzeit MP3s eingelesen und zu WAVs konvertiert werden können, die von Unity unterstützt werden. Teil der NAudio-Bibliothek.
- *Json.NET* (9): Hilfsbibliothek für das Parsing von Json-Dateien.
- *DspLib* (10): DSP-Bibliothek speziell für Fourier-Transformationen. Wird genutzt, um Spektrumsdaten aus Audiodaten zu extrahieren.
- *TextMesh Pro* (11): Textbibliothek von Unity Technologies zur erweiterten Darstellung von Text. Wird beispielsweise genutzt, um Text außerhalb eines Canvas als Mesh mit beliebiger Position innerhalb des Raumes darzustellen.

4.2 Build-Vorgang

Für die Anfertigung des Projekts wurde die *Unity 2019.1.9f1* in der Pro-Version verwendet. Da keine Pro-Features verwendet wurden ist diese vollständig abwärtskompatibel mit der *2019.1.9f1* Personal-Version. Für das Deployment ist zu beachten, dass unbekannt ist, ob das Projekt mit einer neuen Version als *2019.1.9f1* kompatibel ist. Darüber hinaus sollte es ausreichen, das Projekt im Master-Branch zu klonen, in Unity zu importieren und zu bauen.

5 Programmstruktur

3) Übersicht über die Struktur deines Programms. In Unity kann man das häufig gut an den Szenen aufhängen. Ich brauche nicht unbedingt ein formales UML-Diagramm, aber ich möchte wissen wie der Code strukturiert ist.

5.1 Szenen

Für das Projekt wurden mehrere Szenen angefertigt:

- *MainMenu*: Enthält Menüs mit insgesamt drei Untermenüs - Das Hauptmenü, das Optionenmenü und das Spielmenü. Je nach Auswahl des jeweiligen Untermenüs werden die anderen Untermenüs deaktiviert und ausgeblendet.
- *ScoreMenu*: Enthält den Scorebildschirm.
- *Game*: Die eigentliche Spielszene.

- *FastGame*: Hierbei handelt es sich um eine Helferszene, die sich in der Entwicklung als sehr praktisch erwiesen hat. Ohne jegliche nötige Menüinteraktionen wird hier sofort eine vorgegebene Audiodatei geladen und analysiert, um Änderungen im Spiel direkt sichtbar zu machen und die Entwicklungszeit zu beschleunigen.
- *OnsetTest*: Eine Testszene für die Audioanalyse. Diese wird nicht mehr gebraucht und wurde während der Entwicklung verwendet, um die Audioanalyse zu visualisieren. Siehe *PlotDemoAudioAnalyzerController.cs* und *SpectrumPlotter.cs*.
- *SabreTest*: Eine Testszene, in der viele Blöcke zufällig instantiiert werden. Ziel der Szene war es, die Funktionen der Laserschwerter zu testen.
- *JsonTest*: In dieser Testszene wurde das entwickelte Json-Parsing für die .dat-Mappings getestet, indem direkt ein Mapping eingelesen wurde, was das Debuggen erleichtert hat.

Einige weitere Szenen wurden zwischenzeitlich angefertigt und anschließend verworfen, da sie nicht mehr benötigt wurden oder sich als unnötig erwiesen haben.

Von den genannten Szenen werden nur die ersten drei Szenen *Game*, *MainMenu* und *ScoreMenu* tatsächlich verwendet. Bei den weiteren Szenen handelt es sich um Helferszenen, die lediglich den Entwicklungsprozess vereinfacht haben.

5.2 Packages

Im folgenden wird die Programmstruktur erläutert, indem die erstellten packages einzeln betrachtet werden.

5.2.1 Menu

Das Menu-Package enthält sämtlichen Code, der für Menüelemente verantwortlich ist. Das PlayerMenu, das OptionsMenu sowie das MainMenu haben hier ihre eigenen Handlerklassen, die beispielsweise Buttoncallbacks beinhalten und Interaktionen mit den Menüs ermöglichen. Sobald der Spieler im PlayMenu einen Song ausgesucht hat und den *Play*-Button drückt, wird der Ladebildschirm angezeigt und der *AudioController* instantiiert.

5.2.2 Audio

Dieses Package ist für sämtliche audiobasierten Operationen zuständig. Die Hauptklasse ist der *AudioController*. Dieser lädt die ausgewählte Audiodateien und speichert diese als AudioClip. Unterstützt werden entweder MP3- oder OGG-Dateien. Anschließend prüft der AudioController, ob bereits ein BeatMapping für das ausgewählte Lied im BeatMapping-Cache vorhanden ist. Weiterhin wird eine *TrackConfig* erzeugt, die grundlegene Informationen zu dem verwendeten Song enthält, beispielsweise Konfigurationsinformationen für die

spätere Audioanalyse sowie dem Songnamen oder der Samplerate. Falls ein solches Mapping existiert, kümmert sich der *JsonController* aus dem Json-Package um das Laden der Cachedatei. Falls noch kein Mapping existiert, so wird das Sub-Package *AudioAnalysis* für die Audioanalyse verwendet. Der *AudioAnalyzerController* wird instantiiert und startet die Audioanalyse. Diese unterteilt sich in zwei Vorgänge: Zunächst - BeatDetector - Dann PostBeatDetector

Sowohl die Audioanalyse als auch das Laden eines Mappings aus dem Cache erzeugt einen *BeatMappingContainer*. Dieser enthält weitere Configs für später im Spiel zu verwendende Notes (fliegende Blöcke), Obstacles (Hindernisse) und Events (Lichteffekte). Weiterhin gibt es eine Config für Bookmarkinformationen, diese werden im Spiel jedoch nicht verwendet.

Dieser BeatMappingContainer wird global gespeichert, damit von anderen Szenen darauf zugegriffen werden kann. Der Audiocontroller startet nach der Audioanalyse bzw. dem Cache-Loading das Laden der Game-Szene.

5.2.3 Json

Dieses Package ist dafür zuständig, Json-Dateien zu schreiben und zu parsen. Als vrewaltende Entität agiert dabei der *JsonController*. Insgesamt drei verschiedene Arten von Dateien können von ihm behandelt werden; Infodateien Highscoredateien und Mappingdateien. Diese Dateien orientieren sich an der Json-Struktur, die sowohl von Beat Saber als auch von bsaber.com für die BeatMappings bzw. Info-Dateien gewählt wurden.

Um eine Json-Datei zu schreiben, muss zunächst ein Json-String gebaut werden. Darum kümmern sich der *JsonStringBuilder*, der drei Methoden für die jeweilige Art von Datei enthält. Anschließend wird diese Datei mit Hilfe des *JsonIOHandlers* weggeschrieben. Dieser enthält Funktionen zum Schreiben und Lesen von Json-Dateien.

Soll eine Datei gelesen werden, so muss diese gesparsed werden. Das passiert ebenso innerhalb des JsonIOHandlers. Als Rückgabewert geben die drei Lesemethoden entweder eine Liste von *HighscoreData*, einen *MappingInfo*-Container oder einen *MappingContainer* mit den Event-, Obstacle- und Notedaten zurück. Diese werden im Audiocontroller über die durch den JsonController veröffentliche API eingelesen und anschließend global gespeichert.

5.2.4 Game

Dieses Package enthält die meisten in der Spielszene verwendeten Scripts, die für das eigentliche Spielgeschehen verantwortlich sind. *Game.cs* stellt das Kernelement dar. In einer Update-Funktion werden Updates der Sub-Komponenten ausgelöst, die das Spawnen von Effekten, Noten und Hindernissen kontrollieren. Diese Spawning-Komponenten befinden sich im *Spawning*-Subpackage. Zunächst wird hier in Abhängigkeit der BPS (*Beats per Second*) berechnet, zu welchem Zeitpunkt der Audioclip abgespielt werden muss. Weiterhin wird die aktuelle, BPS-abhängige Zeit des nächsten Updates der Spawncontroller berechnet. Diese Zeit steht in Abhängigkeit von der Reisezeit, die die Blöcke und Obstacles vom

Spawnpunkt bis zum Spieler zurücklegen müssen. In der Update-Funktion der jeweiligen Spawncontroller wird dann geprüft, ob ein neues Objekt zum jetzigen Zeitpunkt gespawned werden sollte, indem die Zeit mit den Zeiten der Note-Event- oder ObstacleConfigs verglichen wird.

Das Subpackage *Effects* beinhaltet Controllerklassen, die für das Auslösen und Steuern von Effekten zuständig sind. Der *MainEffectController* beinhaltet alle Controllerobjekte für die einzelnen Effekttypen: Nebel (*FogController*), Laser (*LaserController*) und Lichter (*Spinnerlightcontroller*). Für die Nebeneffekte werden lediglich einzelne eingefärbte Neben-GameObjects für kurze Zeit aktiviert und wieder deaktiviert, um einen aufleuchtenden Nebel-Lichteffekt zu kreieren. In der Gameszene sind weiterhin einige Laser angebracht. Dabei handelt es sich um simple, längliche Cube-Objekte mit einem Material, das Licht emittiert und einen Gloweffekt simuliert. Diese werden durch den LaserController langsam oder abrupt bewegt und leuchten auf. Der komplizierteste und teuerste Effektcontroller ist der SpinnerLightController. In der Szene sind einige rotierende, rechteckige und verschachtelte Objekte installiert. Diese verfügen über ähnliches Lichtmaterial wie die Laser, das standardmäßig deaktiviert ist. Der SpinnerLightController kümmert sich darum, die Lichter in diese Spinnerobjekten in einer bestimmten Reihenfolge zu aktivieren, um so blinkende Effekte oder Transitionseffekte zu realisieren. Dem SpinnerLightController kann ein beliebiger Effekt zugewiesen werden, der von ihm ausgeführt wird. Die Effekte werden im Namespace *SpinnerLightEffects* definiert.

5.2.5 Global

Hier befinden sich globale Komponenten, deren Aufgabe es beispielsweise ist, Entwicklereinstellungen zu setzen, oder Daten zu speichern. Die meisten dieser Objekte sind Singletons, die über Szenen hinaus existieren können, indem *DontDestroyOnLoad* verwendet wird.

5.2.6 Test

Hier befinden sich einige Testklassen, die für Tests- und Entwicklungszenen verwendet wurden. Einige der Klassen sind durch zwischenzeitlich vorgenommene Refactorings nicht mehr funktionsfähig. Weiterhin befindet sich hier der *FastGameAudioLoadingStart*. Diese Helferklasse hat sich für die Entwicklung als extrem praktisch gewesen. Hier werden einige Entwicklereinstellungen gesetzt, anschließend wird ohne Umwege sofort das Spiel bzw. die Audioanalyse mit einem vordefinierten Lied gestartet. Das Menü wird dadurch übersprungen, wodurch viel Entwicklungszeit beim Testen gespart werden konnte.

6 Audioanalyse

Es findet eine Audioanalyse statt, um Beatmappings zu erzeugen, es wird also eine prozedurale Levelgenerierung durchgeführt. Verfahren für die Erkennung

von Beats in Audiodaten werden als *Onset Detection* bezeichnet. Das im Rahmen dieses Projekts entwickelte Analyseverfahren basiert auf einem im Jahre 2006 von Simon Dixon beschriebenen Ansatz, *Onset Detection Revisited* (12). Als weitere Quelle wurde ein Onset Detection-Tutorial von Mario Zecher (13) hinzugezogen, der die im Paper beschriebenen Ansätze verständlich zusammenfasst.

Grundsätzlich basiert dieses Verfahren darauf, mittels einer Fouriertransformation innerhalb eines Fensters die Entwicklung der spektralen Ennergie innerhalb eines Frequenzbands zu betrachten. Starke, plötzliche Schwankungen weisen darauf hin, dass ein Peak bzw. ein Beat vorhanden ist.

6.1 Implementierung

Zunächst wird im *AudioController* eine .ogg-Datei über das *WWW*-Modul oder eine .mp3-Datei über den *Mp3-Loader* geladen. Der *AudioSampleProvider* stellt anschließend die rohen Audiodaten bereit und konvertiert diese in Monodaten, da Monodaten deutlich einfacher zu handhaben sind. Unter der Annahme, dass verschiedene Instrumente, deren Beats erkannt werden könnten, sich vorwiegend in unterschiedlichen Audiokanälen aufhälten, könnte dieses Vorgehen verbessert werden. Für eine einfache Audioanalyse ist der Nutzen einer solchen komplizierten Analyse jedoch fraglich.

Anschließend wird über die Monodaten iteriert und jeweils eine *Fast Fourier Transformation* (FFT) ausgeführt. Eine FFT wurde dabei aus Performanzgründen einer normalen Fouriertransformation vorgezogen. Hierfür wird die Bibiliothek *DspLib* (10) verwendet. Dabei werden jeweils 1024 Audiosamples eingelesen. Da 1024 Audiosamples bei einer Samplerate von 44.100 Hz etwa 23 ms entsprechen, würde diese Menge von Samples bei einer Live-Onset Detection eine zu große Verzögerung erzeugen - für eine vorverarbeitete Analyse entsteht hier allerdings kein Problem. Als Resultat jeder FFT wird aus diesen Daten eine Menge von Frequenzbins generiert, die der halben Anzahl der Samples + 1 entspricht. Aus 1024 Inputsamples ergeben sich also 513 Frequency Bins. Die Maximalfrequenz richtet sich dabei nach der Nyquist-Frequenz (14), die der halben Frequenz der Samplerate entspricht. Im Regelfall liegt diese bei 44.100 Hz, kann aber oft auch 48.000 Hz betragen. Ein Resampling der Audiodaten, für eine Angleichung auf eine stets gleiche Samplerate wurde in Erwähung gezogen, hat sich jedoch als zu aufwändig herausgestellt. Tatsächlich wird dieser Unterschied Abweichungen in den Resultaten der Onset Detection hervorrufen, diese sind jedoch zu vernachlässigen. Bei einer Samplerate von 44.100 Hz werden durch die FFT also Frequenzdaten mit einem Frequenzspektrum von 0 bis 22050 Hz generiert. Diese werden gleichmäßig auf die Bins aufgeteilt, welche von der Anzahl der eingelesenen Samples abhängig ist. Die Wahl der Größe der eingelesenen Samples ist also ausschlaggebend für die Granularität der Frequency Bins. Pro Bin ergibt sich hier eine Frequenz *fb* von:

$$f_b = \frac{22050}{1024} = 42,982Hz \tag{1}$$

In der *TrackConfig* werden *AnalyzerBandConfig*s definiert, die die Verteilung und die Anzahl der Bänder bestimmen, die in der Audioanalyse verwendet werden. AnalyzerBandConfig bestimmt, welche der Bins für die Audioanalyse in dem jeweiligen Band verwendet sollen. Zusätzlich werden einige Parameter, wie Thresholdwerte, gesetzt. Diese Thresholdwerte werden durch den ausgewählten Schwierigkeitsgrad beeinflusst und bestimmen effektiv, wie viele Peaks erkannt und somimt wie viele vom Spieler zu treffende Blöcke generiert werden. Insgesamt werden zwei Frequenzbänder verwendet. Diese wurden so gewählt, dass Kickdrum- und Snaredrum-Beats möglichst gut erkannt werden können, da sie dem Frequenzbereich dieser Instrumente in etwa entsprechen. Für das Kick-Frequenzband werden die Bins 0 bis 6 verwendet. Für das Snare-Frequenzband werden die Bins 30 bis 400 verwendet. Pro Bin ergeben sich durch die definierten Werte folgende Minimalfrequenz *f1* und *f2*:

$$f1_{min} = f_b * 0 = 0 \tag{2}$$

$$f1_{max} = f_b * 6 = 258Hz \tag{3}$$

$$f2_{min} = f_b * 30 = 1289Hz \tag{4}$$

$$f2_{max} = f_b * 400 = 17193Hz \tag{5}$$

Anschließend werden die komplexen Frequenzdaten in ein nutzbares Format (double[]) konvertiert. Diese Daten und die erzeugte TrackConfig, welche die Konfigurationsdaten für die Analyse enthält, werden dem *AudioAnalyzer-Handler* übergeben. Dieser startet die eigentliche Audioanalyse sowie die Post-Audioanalyse.

PeakDetector Im *PeakDetector* wird das weitere im Paper beschriebene Verfahren angewendet.

Zunächst wird für alle Spektren ein sogenannter *Spectral Flux*-Wert berechnet:

```
for (int i = firstBin; i <= lastBin; i++)
{
    flux += Math.Max(0f, _currentBandSpectrums[_band][i] -
        _previousBandSpectrums[_band][i]);
}
return flux;
```

Zur Berechnung des Spectral Flux, der initial 0 ist, macht nun folgendes für alle Bins, die für das aktuelle Frequenzband definiert wurden: Der letzte Spektralwert wird mit dem aktuellen Spektralwert verglichen. Dem Fluxwert werden nun positive Veränderungen aufaddiert. Der Fluxwert vergrößert sich also nur, wenn der Signalpegel im Vergleich zum vorherigen gestiegen ist.

Anschließend wird über alle Fluxwerte in den beiden Bändern iteriert, wobei rückblickend jeweils die letzten n Fluxwerte betrachtet werden. Ist der aktuelle Fluxwert höher als der Durschnitt aller Fluxwerte, die im Bereich des Fensters liegen, multipliziert mit einem in der Config definierten Thresholdfaktor, so liegt ein Kandidat für einen Peak vor. Es wird anschließend geprüft, ob der Kandidat

ein Peak ist, indem der vorherige und der nachfolgende Fluxwert betrachtet werden. Sind beide Fluxwerte geringer als der Wert des Kandidaten, so liegt ein Peak vor.

```
private bool _isPeak()
{
    return currentPrunedFlux > previousPrunedFlux &&
        currentPrunedFlux > nextPrunedFlux;
}
```

Für jeden erzeugten Peak wird eine Konfigurationsdatei *NoteConfig* für einen Block bzw. eine Note generiert. Das Band 0 erzeugt rote, mit dem linken Saber zu treffende Noten. Das Band 1 erzeugt blaue, mit dem rechten Saber zu treffende Noten. Die Position der Blöcke wird zufällig gewählt. Es wurden zusätzlich einige Regeln definiert, die gewährleisten, dass möglichst nur Kombinationen von Noten erzeugt werden, die auch spielbar sind. Beispielsweise können Noten, die nebeneinander liegen und nicht die selbe Farbe haben, nicht in die selbe Richtung zeigen, da ein Treffen mit beiden Sabern dann nicht möglich wäre. Es werden weitere Optimierungen, wie beispielsweise der *NoteBlockCounter* vorgenommen. Diese Varaible sorgt dafür, dass während einer bestimmten Zeit nach der Erzeugung einer NoteConfig keine weiteren Notes erzeugt werden können.

Weiterhin werden hier *ObstacleConfig*s und *EventConfigs* erzeugt. Dabei handelt es sich um Konfigurationsobjekte, die später für das Spawnen von Hindernissen sowie das Triggern von Lichteffekten verantwortlich sind. Die Erzeugung dieser Objekte ist zufällig und basiert lediglich auf einer bestimmten definierten Wahrscheinlichkeit. Für die Hindernisse ist es außerdem wichtig, dass Notes, die sich im selben Zeitraum wie ein Hinderniss bzw. Obstacle befinden, sich nicht mit diesem überschneiden. Die Noten werden dann dementsprechend anders positioniert.

PostAudioAnalyzer Anschließend wird vom *AudioAnalyzerHandler* eine nachfolgende Post-Audioanalyse gestartet. Hier wird mehrmals über alle erzeugten Blöcke iteriert. Zunächst werden von BeatSaber bekannte *Doppelblöcke* erzeugt. Noten des selben Typs, die sehr nah beieinander liegen, werden dabei zu Doppelblöcken zusammengefasst. Erneut sorgt ein regelbasiertes Verfahren dafür, dass nur spielbare Kombinationen zulässig sind. Anschließend werden Blöcke beliebigen Typs die zu nah beieinander liegen so angepasst, dass ihre Positionierung sowie die Schnittrichtung spielbar bleibt. Einige weitere kleine Optimierung werden vorgenommen, wobei stets darauf geachtet, dass die bewegten Noten, die sich zeitlich mit Obstacles überschneiden, außerhalb dieser positioniert werden.

Literaturverzeichnis

[1] Oculus Firmware
https://www.oculus.com/setup/?locale=de_DE
Oculus VR

[2] Oculus Unity Framework
https://developer.oculus.com/downloads/package/oculus-sample-framework-for-unity-5-project/
Oculus VR

[3] Post Processing Stack
https://assetstore.unity.com/packages/essentials/post-processing-stack-83912
Unity Technologies

[4] MKGlowFree
https://assetstore.unity.com/packages/vfx/shaders/fullscreen-camera-effects/mk-glow-free-28044
Michael Kremmel

[5] SimpleFileBrowser
https://github.com/yasirkula/UnitySimpleFileBrowser
Süleyman Yasir Kula

[6] EzySlice
https://github.com/DavidArayan/ezy-slice
David Arayan

[7] SphericalFog
https://forum.unity.com/threads/spherical-fog-shader-shared-project.269771/
Rune Skovbo Johansen

[8] NLayer
https://github.com/naudio/NLayer
Mark Heath, Andrew Ward

[9] Json.NET
https://www.newtonsoft.com/json
James Newton-King, Newtonsoft

[10] DspLib
https://www.codeproject.com/Articles/1107480/DSPLib-FFT-DFT-Fourier-Transform-Library-for-NET-6
Steve Hageman

[11] TextMesh Pro
https://assetstore.unity.com/packages/essentials/beta-projects/textmesh-pro-84126
Unity Technologies

[12] Onset Detection Revisited
https://www.eecs.qmul.ac.uk/ simond/pub/2006/dafx.pdf
Simon Dixon, simon.dixon@ofai.at
Austrian Research Institute for Artificial Intelligence

[13] Onset Detection Tutorial
https://www.badlogicgames.com/wordpress/?cat=18paged=3
Mario Zechner, Badlogic Games

[14] Erfassen von Analogsignalen: Bandbreite, Mess-, Prüf und Regelungstechnik Nyquist-Abtasttheorem und Alias-Effekt
https://www.etz.de/files/01_14_blumenstein.pdf
Vanessa Blumenstein, National Instruments Germany GmbH, München