

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

<b>1 Einleitung</b>	<b>1</b>
<b>2 Projektidee</b>	<b>1</b>
<b>3 Beschreibung des Spiels</b>	<b>1</b>
3.1 Entwickeltes Endprodukt . . . . .	1
3.2 Voraussetzungen . . . . .	2
3.3 Menüführung . . . . .	3
3.3.1 Hauptmenü . . . . .	3
3.3.2 Optionsmenü . . . . .	3
3.3.3 Spielmenü . . . . .	3
3.3.4 Score-/GameOver-Menü . . . . .	4
3.4 Core Game Loop . . . . .	5
3.5 Laden von Audiodateien und Mappings . . . . .	5
3.5.1 BeatSaber Mapping Format . . . . .	5
3.5.2 Korrekte Songauswahl . . . . .	5
<b>4 Kompilierung des Projekts</b>	<b>6</b>
4.1 Verwendete Bibliotheken . . . . .	6
4.2 Build-Vorgang . . . . .	7
<b>5 Programmstruktur</b>	<b>8</b>
5.1 Szenen . . . . .	8
5.2 Packages . . . . .	8
5.2.1 Menu . . . . .	9
5.2.2 Audio . . . . .	9
5.2.3 JSON . . . . .	10
5.2.4 Game . . . . .	10
5.2.5 Global . . . . .	12
5.2.6 Test . . . . .	12
<b>6 Audioanalyse</b>	<b>12</b>
6.1 Audiovorverarbeitung . . . . .	13
6.2 Implementierung der Onset Detection . . . . .	16
<b>Literaturverzeichnis</b>	<b>22</b>

# 1 Einleitung

In diesem einleitenden Kapitel wird die grundsätzliche Projektidee umschrieben, um einen groben Überblick über das entstandene Projekt zu geben.

## 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 zufliegenden 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 entstandene Projekt wurde mit Unity entwickelt.

## 3 Beschreibung des Spiels

Im Verlauf dieses Kapitels wird eine ausführliche Beschreibung des Endproduktes geliefert.

### 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.

In Abbildung 1 ist eine frühe Version des Spiels während der Entwicklungsphase zu sehen. Weiterhin kann in Abbildung 2 ein Eindruck über das fertige Spiel gewonnen werden.



Abbildung 1: Frühe Alphaversion des Spiels



Abbildung 2: Finale Version des Spiels mit Effekten etc.

### 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. In Abbildung 3 ist das Hauptmenü zu sehen.



Abbildung 3: Hauptmenü

#### 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.

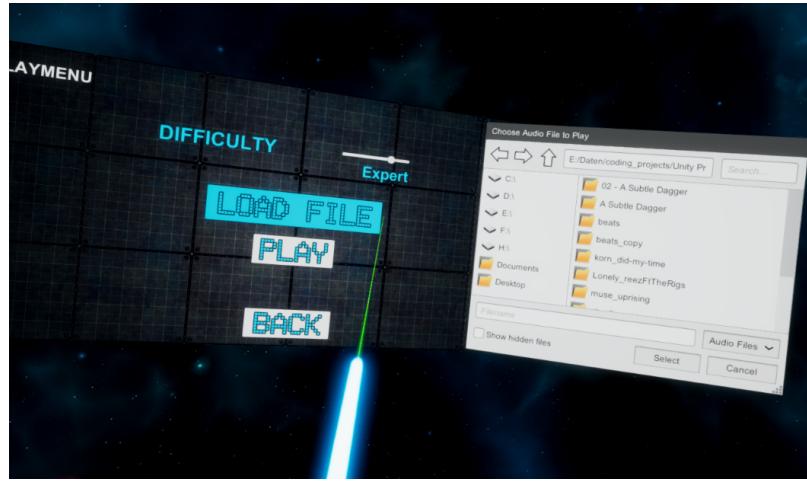


Abbildung 4: Spielmenü

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. In Abbildung 5 ist ein beispielhaftes Bild des Score-Menüs zu sehen.



Abbildung 5: Score-Menü

### 3.4 Core Game Loop

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 *Multiplierv*- 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

Folgend wird beschrieben, wie sowohl generierte als auch von externen Quellen bezogene Mappings geladen und gespeichert werden.

#### 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/Beat-Mappings*. Dieser Ordner dient als Caching für sämtliche erstellten Mappings, daher sollten auch die Audiodateien dort hinterlegt werden.

Neben der Audiodatei 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 Schwierigkeitsgrad 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 eine Audioanalyse generiert und im Cache abgelegt.

Es können auch externe, fertige Mappings von Seiten wie [bsaber.com](http://bsaber.com) verwendet 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.

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 weiterer 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 Grading 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 Bibliothek 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

### 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 Helperszene, 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 Helperszenen, die lediglich den Entwicklungsprozess vereinfacht haben.

### 5.2 Packages

In diesem Kapitel wird die Programmstruktur erläutert, indem die erstellten packages einzeln betrachtet werden. In Abbildung ?? ist die im Folgenden beschriebene Programmstruktur zu sehen.

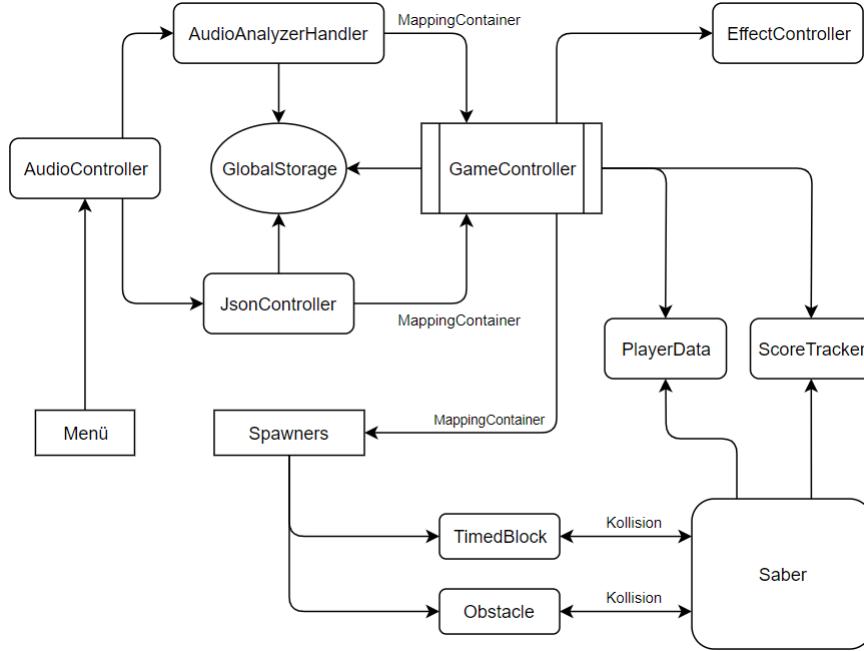


Abbildung 6: Visualisierung der Programmstruktur; abgerundete Bereiche repräsentieren einzelne Klassen, rechteckige beschreiben eine Struktur aus mehreren Klassen

### 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 grundlegende 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 PostBeat-Detector

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 Informationen über Songs, die als eine Art Lesezeichen gespeichert wurden - diese Informationen 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 vewaltende 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` veröffentlichte 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 Kern-element dar. In einer Update-Funktion werden Updates der Sub-Komponenten ausgelöst, die das Spawning 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 NoteConfigs, EventConfigs 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. Ein beispielhafter Effekt kann in Abbildung 7 betrachtet werden.

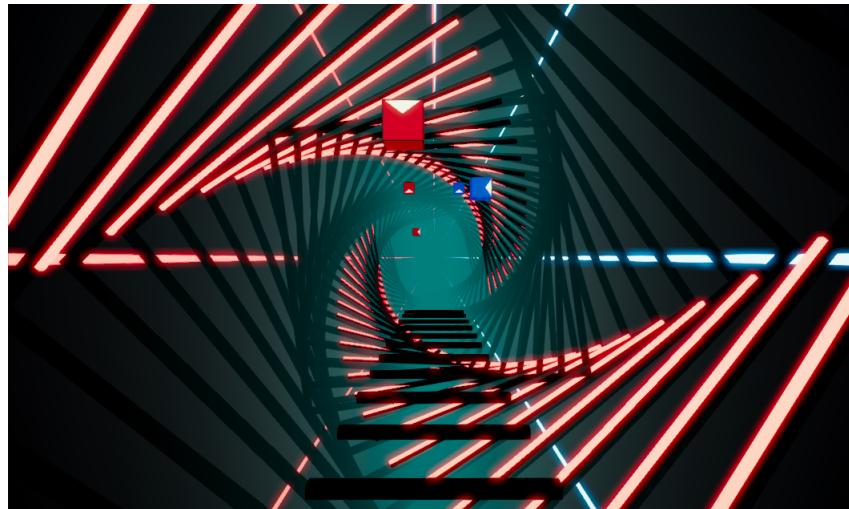


Abbildung 7: Beispielhafter **SpinnerLightEffect**

Zusätzlich ist das Subpackage **Objects** von großer Wichtigkeit. Es enthält Scripts, die das Verhalten sämtlicher Objekte definieren, mit denen der Spieler interagieren kann. Dazu gehören folgende Scripts:

- **Obstacles.cs:** Script für Obstacles. Kümmert sich um Kollision mit Spieler sowie automatische Zerstörung des Objekts.
- **Sabre.cs:** Beschreibt das Verhalten der Laserschwerter inklusive Kolisionsbehandlung mit Blöcken und Zerschneiden der Blöcke in mehrere **SlicedBlocks**.
- **SlicedBlock.cs:** Kümmert sich um automatische Zerstörung der zerschnittenen Blöcken.
- **TimedBlock.cs:** Kümmert sich um automatische Zerstörung der von Notenblöcke.

Besonders interessant ist *Sabre.cs*. Das Script ist für den Core Game Loop des Spiels unerlässlich, da hier das Verhalten der Laserschwerter beschrieben wird. Zunächst wird kontinuierlich geprüft, ob es eine Kollision zwischen Sabre und einem Notenblock gibt. Sobald eine Kollision gefunden wurde, wird diese behandelt. Abhängig vom Winkel des Laserschwerts im Verhältnis zu einem getroffenen Notenblock wird entschieden, ob ein Schlag ein *Hit* oder ein *Miss* gewesen ist. Das Ergebnis wird dem **ScoreTracker** mitgeteilt. Bei einem Treffer wird der Block danach zerschnitten und neue zerschnittene Blöcke mit *EzySlice* generiert.

### 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 die Helferklasse **FastGameAudioLoadingStart**, die sich für die Entwicklung als extrem praktisch erwiesen hat. 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 Energie innerhalb eines Frequenzbands zu betrachten. Dieser Wert wird als *Spectral Flux* beschrieben. Starke, plötzliche Schwankungen weisen darauf hin, dass ein Peak bzw. ein Beat vorhanden ist.

## 6.1 Audiorverarbeitung

Zunächst wird im `AudioController` eine .ogg-Datei über das `WWW`-Modul oder eine .mp3-Datei über den `Mp3-Loader` geladen.

Beide Verfahren zum Laden von Audiodaten geben einen `AudioClip` zurück. Dabei handelt es sich um ein von Unity standardmäßig verwendete Objekt zur Repräsentation von Audiodateien. Der `AudioSampleProvider` stellt anschließend die Audiodaten bereit, die aus dem `AudioClip` mit `AudioClip.GetData` extrahiert werden. Als Resultat erhält man ein float-Array aus Stereosamples, indem abwechselnd Paare aus zugehörigen Samples für den jeweiligen Kanal enthalten sind. Das Array ist also wie folgt aufgebaut:  $[L, R, L, R \dots]$ , wobei jedes  $L$  einem Audiosample für den linken und jedes  $R$  einem Audiosample für den rechten Audiokanal entspricht. Um die Werte dieses Arrays genauer nachzuvollziehen sei Folgendes zu erwägen: In der digitalen Signalverarbeitung wird ein kontinuierliches Audiosignal in ein zeitdiskretes Signal umgewandelt. Ein Audiosample beschreibt dann einen Wert für einen bestimmten Zeitpunkt bzw. für einen kurzen Zeitraum. Ein Sample beschreibt den Signalpegel für diesen Zeitraum. Mit einer typischen Abtastrate von 48 kHz kann dabei eine maximale Frequenz von 24 kHz entsprechend der Nyquist-Frequenz, also der halben Abtastrate, dargestellt werden.

Bei einer typischen Abtast- bzw. Samplerate von 48 kHz bei zwei Kanälen ergibt sich pro Sekunde  $ns = 48.000 * 2 = 96.000$  Samples/s. Mit 32 Bit C floats ergibt sich dadurch eine Datenrate von  $d = ns * 32 = 3.072.000Bit = 384.000Byte = 0,366MB$ . Bei einer typischen Lieddauer von 4 Minuten ergibt sich somit eine Datenmenge von etwa 88 Megabyte. Diese relativ hohe zu verarbeitende Datenmenge kann durch die Verwendung von Monodaten halbiert werden. Weiterhin ist die Verwendung von Monodaten vorteilhaft, um nur einen Kanal verarbeiten zu müssen. Das Analysieren von nur einem statt zwei Kanälen führt also sowohl zu einem hohen Performancezuwachs bezüglich der Dauer der Audioanalyse als auch zu einem geringeren zusätzlichen Speicherbedarf. Aus diesem Grund werden (nur für den Zweck der Audioanalyse, nicht aber beim eigentlichen Abspielen) die Stereodataen in Monodaten konvertiert, indem jeweils ein zusammengehöriges  $L-R$ -Paar aus Audiosamples aufaddiert werden. Anschließend wird die Summe der Werte halbiert, um eine Normalisierung vorzunehmen. Die Länge des neuen Arrays entspricht also der halben Länge des Arrays mit Stereodataen.

Tatsächlich arbeitet die Verarbeitung der Audiodaten mit beliebig vielen Kanälen, sodass beispielsweise auch 5.1-Audiodaten eingelesen werden können. Das Array aus rohen Audiodaten würde dann keine  $[L, R, L, R \dots]$ -Paare, sondern folgende Struktur enthalten:  $[FL, FR, C, B, SL, SR, FL, FR, C, B, SL, SR \dots]$  mit  $FL = Front\ Left$ ,  $FR = Front\ Right$ ,  $C = Center$ ,  $SL = Surround\ Left$  und  $SR = Surround\ Right$ . Im vorherigen Abschnitt wurde nur zum leichteren Verständnis beispielhaft davon ausgegangen, dass Stereodaten vorliegen. Bei der Konvertierung in Monodaten werden eigentlich immer so viele Samples eingelesen, wie Kanäle vorhanden sind und anschließend der Mittelwert aus diesen Daten gebildet, um einen einzelnen Monosample daraus zu bilden.

Zum Verarbeiten der Daten wäre auch die Nutzung einer externen Bibliothek möglich gewesen, durch die Simplizität des Vorgangs und durch persönliche Vorlieben erschien es jedoch sinnvoller, das Verfahren selbst zu implementieren.

Unter der Annahme, dass verschiedene Instrumente, deren Beats erkannt werden könnten, sich vorwiegend in unterschiedlichen Audiokanälen aufhalten, könnte dieses Vorgehen verbessert werden. Die Komplexität würde sich dadurch jedoch erhöhen und breit verteilte Instrumente ihre Zusammengehörigkeit verlieren. Für eine einfache Audioanalyse ist der Nutzen einer solchen komplexeren Analyse fraglich.

Anschließend wird über die Monodaten iteriert und jeweils eine *Fast Fourier Transformation* (FFT) (14) ausgeführt. Dieses Extrahieren der Spektraldaten wird im `SpectrumProvider` vorgenommen. Eine FFT wurde dabei aus Performanzgründen einer normalen Fouriertransformation vorgezogen, da sich die Laufzeit durch einen *Divide-and-Conquer*-Ansatz im Gegensatz zu einer DFT (*Discrete Fourier Transformation*) von  $O(n^2)$  auf  $O(n * \log(n))$  verringert (14). Hierfür wird die Bibliothek `DspLib` (10) verwendet. Es 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 (15), 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 Erwähnung 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  $f_b$  von:

$$f_b = \frac{22050}{1024} = 42,982\text{Hz} \quad (1)$$

In der `TrackConfig` werden `AnalyzerBandConfigs` definiert, die die Verteilung und die Anzahl der Bänder bestimmen, die in der Audioanalyse verwendet werden. Die `AnalyzerBandConfig` bestimmt, welche der Bins für die Audioanalyse verwendet sollen. Dadurch wird also das Frequenzband definiert. Zusätzlich werden einige Parameter gesetzt, die für die Onset Detection nötig sind, wie zum Beispiel ein Thresholdwert. Dieser Wert wird durch den ausgewählten Schwierigkeitsgrad beeinflusst und bestimmt effektiv, wie viele Peaks erkannt und somit die Anzahl der generierten vom Spieler zu treffenden Blöcke. Insgesamt werden zwei Frequenzbänder verwendet. Diese wurden so gewählt, dass niedrigfrequente Kickdrum- und höherfrequente 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. Die genaue Festlegung dieser statischen Frequenzbänder wurde durch das Austesten unterschiedlicher Unter- und Obergrenzen gewählt. Die getroffene Auswahl erzielte dabei das beste Ergebnis und repräsentiert in etwa einer typischen Verteilung von Snare- und Kickdrum. Es können allerdings keine Daten über die Güte dieser Verteilung genannt werden, da das *Austesten* dieser Grenzen anhand von subjektiven Merkmalen geschehen ist. Es wurde sozusagen nach Gefühleentschieden, für welche Frequenzbänder Beats am besten erkannt wurden. Dafür wurde die Testszene *OnsetTest* verwendet, in der erkannte Beats und berechnete Fluxwerte für eine Audiodatei visualisiert wurden. Hier besteht Verbesserungspotential durch das Auffinden einer objektiven Metrik für die Bewertung der Güte der Frequenzbänder.

Pro Bin ergeben sich durch die definierten Werte folgende Minimalfrequenz  $f_1$  und  $f_2$ :

$$f_{1min} = f_b * 0 = 0 \quad (2)$$

$$f_{1max} = f_b * 6 = 258\text{Hz} \quad (3)$$

$$f_{2min} = f_b * 30 = 1289\text{Hz} \quad (4)$$

$$f_{2max} = f_b * 400 = 17193\text{Hz} \quad (5)$$

Die Größe der Bins, also auch die Höhe des über die Bins akkumulierten Fluxwertes, unterscheidet sich hier maßgeblich, da das Spektrum des ersten Frequenzbands deutlich mehr Frequenzen beinhaltet als das zweite. Eine Normalisierung muss hier jedoch nicht zwangsweise vorgenommen werden, da die Verteilung der Energie über das Frequenzspektrum ohnehin nicht linear verläuft. Ferner gilt es, relative Unterschiede zwischen Fluxwerten zu finden (Erklärung dazu folgt im Folgenden), sodass eine Normalisierung nicht nötig ist. Indirekt wurde eine Normalisierung im Sinne eines Ausgleichs der Anzahl der detektierten Peaks zwischen den beiden Frequenzbändern durch die Auswahl der Thresholdlevel durchgeführt.

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 `AudioAnalyzerHandler` übergeben. Dieser startet die eigentliche Audioanalyse sowie die Post-Audioanalyse.

## 6.2 Implementierung der Onset Detection

Die generierten Spektraldaten werden nun verwendet, um aus den Audiodaten prozedural durch Onset Detection ein Level zu generieren. Das weiterhin im Paper beschriebene Verfahren wird im `PeakDetector` implementiert - die eigentliche Onset Detection findet also dort statt. Spectral Flux SF wird von Dixon wie folgt definiert:

$$SF(n) = \sum_{k=-\frac{N}{2}}^{\frac{N}{2}-1} H(|X(n, k)| - |X(n - 1, k)|)$$

Abbildung 8: Spectral Flux

Es wird über alle Samples im Fenster von  $-N/2$  bis  $N/2 - 1$  iteriert und anschließend das Ergebnis der Rectifier-Funktion  $H$  mit dem aktuellen und dem vorherigen Sample addiert.

$$H(x) = \frac{x+|x|}{2}$$

Abbildung 9: Rectifier Funktion

$X(n, k)$  repräsentiert den Wert des  $k$ -ten Frequency Bins im  $n$ -ten Sample. Für alle Samples wird nun der *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 Wert des zuletzt betrachteten Spektrums wird mit dem aktuellen Spektralwert verglichen. Alle positiven Unterschiede, also jede Vergrößerung des Spektralwerts, werden nun über alle zugehörigen Frequency Bins kumuliert, um den Spectral Flux-Wert zu bilden. Der Fluxwert vergrößert sich also nur, wenn der Signalpegel im Vergleich zum vorherigen gestiegen ist.

Anschließend wird über alle berechneten Fluxwerte in den beiden Bändern iteriert. Dabei wird ein Threshold generiert, der auf den Fluxwerten basiert, einen Zeitpunkt umgeben (also die Frames vor und nach dem aktuellen Wert). Dadurch kann entschieden werden, ob eine Veränderung der Aktivität des Audiospektrums signifikant genug gewesen war, um einen Beat darzustellen. Ist der aktuelle Fluxwert höher als der Durschnitt aller Fluxwerte, die im Bereich des Fensters liegen, multipliziert mit dem 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 Variable sorgt dafür, dass während einer bestimmten Zeit nach der Erzeugung einer `NoteConfig` keine weiteren Notes erzeugt werden können.

Weiterhin werden hier `ObstacleConfigs` und `EventConfigs` erzeugt. Dabei handelt es sich um Konfigurationsobjekte, die später für das Spawning 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.

`PostOnsetAudioAnalyzer` Anschließend wird vom `AudioAnalyzerHandler` eine nachfolgende Post-Audioanalyse durch den `PostOnsetAudioAnalyzer` 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. Anschließend werden Blöcke beliebigen Typs die zu nah beieinander liegen so angepasst, dass ihre Positionierung sowie die Schnittrichtung spielbar bleibt. Erneut sorgt ein regelbasiertes Verfahren dafür, dass nur spielbare Kombinationen zulässig sind. Genauer heißt das unter anderem, dass die Richtungen

und Positionen der Blöcke überprüft werden. Nach festgelegten Regeln in Form von **if-else**-Konstrukten werden unzulässige Kombinationen von Blöcken, basierend auf ihren Schnittrichtungen und Positionen, herausgefiltert. Wenn beispielsweise ein roter und ein blauer Block auf der selben Höhe direkt nebeneinander liegen, so sollten sie keine horizontale Schnittrichtung haben, da das spielerisch nicht umzusetzen ist. Bei der Auswahl der Positionen der Blöcke und Doppelblöcke gibt es zusätzlich gewisse Zufallschancen für die Positionierung von Blöcken, um Variation zu erzeugen. Diese unterliegen selbstverständlich ebenfalls den festgelegten Kriterien und Regeln. Stets wird ebenfalls darauf geachtet, dass die bewegten Noten, die sich zeitlich mit Obstacles überschneiden, außerhalb dieser positioniert werden.

Ohne eine Nachbearbeitung ist das Resultat der Onset Detection nur bedingt spielbar. Eine Feinjustierung der Parameter der Audioanalyse allein reicht mit der entwickelten Implementierung nicht aus, um sauber beliebige Beats für unterschiedliche Songs und Genres zu erkennen. Stattdessen hat das erzeugte Ergebnisse stets zu einer der folgenden Situationen geführt:

- *False Positives*: Es werden zu viele Beats detektiert, wodurch ebenfalls Nicht-Beats fälschlicherweise erkannt werden
- *False Negatives*: Es werden keine/wenig falsche Beats erkannt, jedoch werden nicht alle Beats erkannt.
- *Unspielbare Blockkombinationen*: Die Kombination der Blöcke in ihren Positionen weist zu viel Varianz auf, sodass unspielbare Kombinationen von Blöcken entstehen.
- *Blockdichte*: Es werden generell zu viele Blöcke erkannt, um ein tatsächlich spielbares Erlebnis zu gewährleisten.a
- *Überschneidungen*: Es gibt Überschneidungen zwischen Blöcken und Hindernissen.

Aus diesem Grund ist die Verwendung von Optimierungen wie unter anderem dem `NoteBlockCounter` und dem `PostOnsetAudioAnalyzer` notwendig. Ohne diese Nachbearbeitung ist das Spiel nur bedingt spielbar, da dann die oben genannten Situationen auftreten.

In den folgenden Grafiken sind typische Verteilungsmuster von Blöcken zu erkennen. Dabei handelt es sich um ungefilterte, teilweise nachbearbeitete und vollständig nachbearbeitete Szenarien. Zur besseren Übersicht wurde die Unity-Szenenansicht verwendet sowie Effekte deaktiviert.

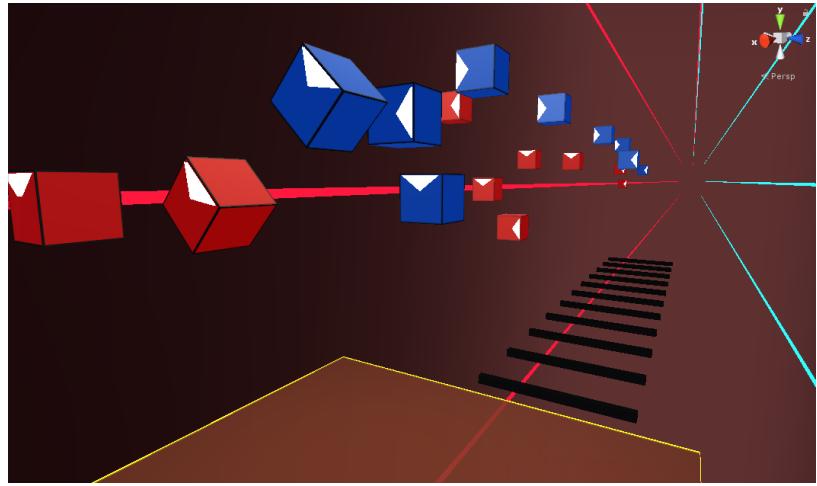


Abbildung 10: Typische Verteilung ohne jegliche Optimierung

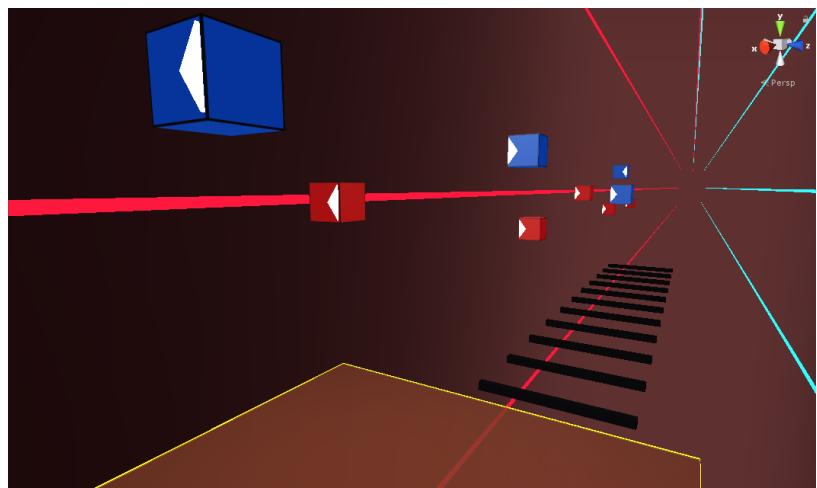


Abbildung 11: Typische Verteilung mit `BlockCounter`

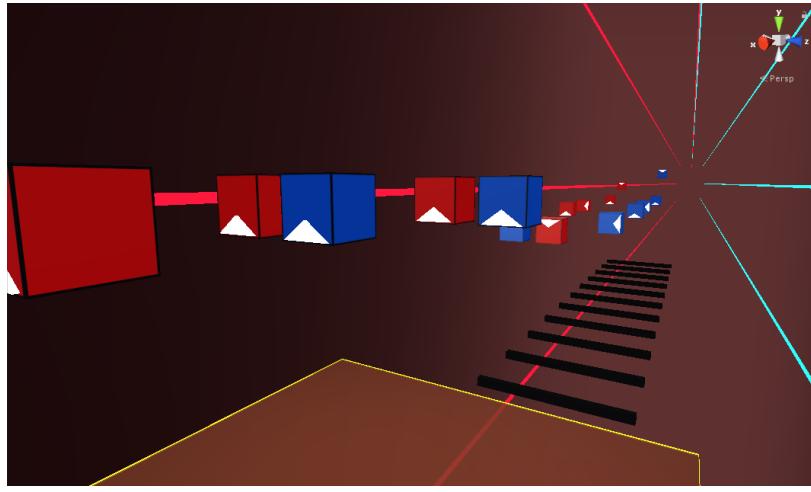


Abbildung 12: Typische Verteilung mit PostOnsetAudioAnalyzer

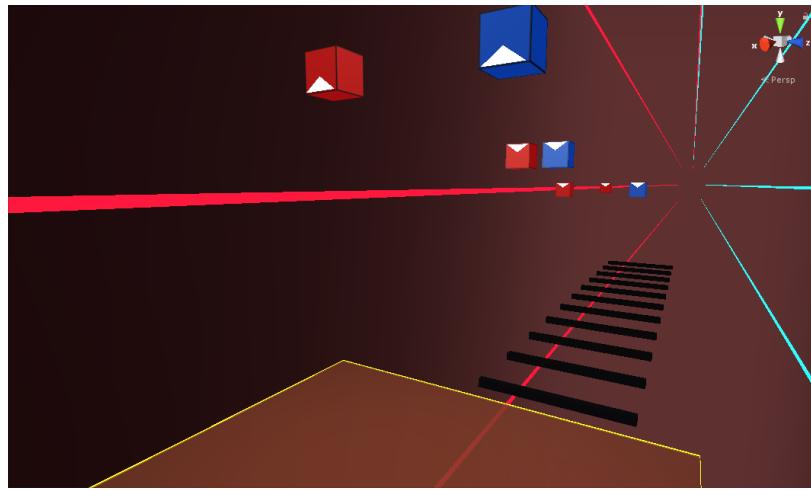


Abbildung 13: Typische Verteilung mit BlockCounter und textttPostOnsetAudioAnalyzer

In Grafik 10 ist deutlich zu erkennen, dass die Anzahl der Blöcke zu hoch ist. Die zu hohe Blockdichte wird durch die Verwendung des BeatBlockCounter eingeschränkt, wie in Grafik 11 zu erkennen. Weiterhin sind die Blöcke willkürlich verteilt - eine dahingehende Anpassung durch den PostOnsetAudioAnalyzer ist in Grafik 12 zu sehen. In Abbildung 13 wird anschließend verdeutlicht, dass die Kombination beider Optimierungen zu einem deutlich verbesserten und nun spielbaren Ergebnis hinsichtlich der Verteilung, Anzahl und Positionierung der Blöcke betrifft.

## Literaturverzeichnis

- [1] Oculus Firmware  
[https://www.oculus.com/setup/?locale=de\\_DE](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/DArayan/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=18&paged=3>  
Mario Zechner, Badlogic Games
- [14] Fast Fourier Transformation  
[https://en.wikipedia.org/wiki/Fast\\_Fourier\\_transform](https://en.wikipedia.org/wiki/Fast_Fourier_transform)  
Wikipedia, Wikimedia Foundation, Inc.
- [15] Erfassen von Analogsignalen: Bandbreite, Mess-, Prüf und Regelungstechnik Nyquist-Abtasttheorem und Alias-Effekt  
[https://www.etz.de/files/01\\_14.blumenstein.pdf](https://www.etz.de/files/01_14.blumenstein.pdf)  
Vanessa Blumenstein, National Instruments Germany GmbH, München