Ein innovatives Tool zur Integration von Sound in Computerspiele

An der Hochschule der Medien Stuttgart,
Fakultät Electronic Media
im Bachelorstudiengang **Audiovisuelle Medien**

Bachelorarbeit

Zur Erlangung des akademischen Grades eines
Bachelor of Engineering

Vorgelegt von

Josua Wolf
Matrikel-Nr.: 31298

Erstgutachter: Prof. Dipl.-Ing. Uwe Schulz
Zweitgutachter: Prof. Oliver Curdt

Eingereicht am:

Ein innovatives Tool zur Integration von Sound in Computerspiele

Zusammenfassung / Abstract 3

Eidesstattliche Erklärung 4

Einleitung 5

1 Sound in Spielen: Allgemein

1.1 Wichtigkeit von Sound in Spielen 6

1.2 Besonderheiten: 2D/3D Sound 9

1.3 Interaktive Musik 11

1.4 Procedual Sounddesign 14

2 Sound in Mobile Games: Spezielle Anforderungen

2.1 Unterschied zu PC Spielen 17

2.2 Praxistipps 18

3 Anforderungen an ein Sound Tool

3.1 Benötigte Features Design 20

3.2 Benötigte Features Programmierung 22

3.3 Vergleich: Gegenläufige Interessen? Überlappungen? 23

3.4 Wie könnte ein guter Workflow für beide Seiten aussehen? 24

3.5 Übersicht geplanter Features für das Spiel „Devolution 26

3.6 Unity: Sound in der Engine 29

4 Entwicklung eines Sound Tools auf Basis der Anforderungen

4.1 Ergebnis der Recherche aus den vorangegangenen Artikeln 32

4.2 Anwendungsfälle von Sounddesign in Computerspielen 33

4.3 Planung Programmierung; Design Ansatz 39

4.4 Problemstellungen zur Umsetzung 40

4.5 Umsetzung der Features im Code 41

5 Das neue Tool in der Praxis

5.1 Umstieg von FMod auf mein Tool 55

5.2 Auswertung des programmierten Tools 56

Fazit 58

Literaturverzeichnis 59

Abkürzungsverzeichnis 60

Zusammenfassung

Die vorliegende Arbeit beschäftigt sich mit dem aktuellen Workflow von Sounddesign in Computerspielen, die mit der Unity Engine erstellt werden. Dabei wird zunächst die Bedeutung von Sound in Computerspielen beleuchtet und die damit einhergehende Anforderung ans Sounddesign. Dazu werden einige Anwendungsfälle und Beispiele aufgeführt.

Die meisten Spieleprojekte in Unity werden mit Hilfe eines externen Tools vertont. Das erschwert den Workflow und erfordert eine gute Kommunikation zwischen Programmierern und Sounddesignern. Der Grund für diesen Workflow ist das kaum vorhandene Sound System der Unity Engine. Ein Sounddesigner kann dort nicht mal einfachste Töne ohne Programmieraufwand einbauen.

Ziel dieser Arbeit ist es, ein Tool zu programmieren, mit dem Sounddesigner direkt in der Unity Engine arbeiten können. So wird der Workflow deutlich vereinfacht und der Sounddesigner besser eingebunden.

Abstract

This thesis deals with the current workflow of sound design in computer games created in the Unity Engine. For this purpose, a deeper look into the meaning of sound in computer games and the coherent requirements for sound design is taken. Some examples and use cases on this subject are illustrated.

External tools are used to create sound for most gaming projects in Unity. This makes the workflow much more difficult and requires good communication between programmers and sound designers. The reason for this workflow is a mostly non-present sound system within the Unity Engine. For a sound designer, it’s not possible to add the simplest sound without any programming.

The goal of this thesis is to create a tool with which sound designers can work directly within the Unity Engine. This way the workflow gets simplified and the sound designer is better integrated.

Eidesstattliche Erklärung

Hiermit versichere ich, Josua Wolf, an Eides Statt, dass ich die vorliegende Bachelorarbeit mit dem Titel: „Ein innovatives Tool zur Integration von Sound in Computerspiele“ selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ich habe die Bedeutung der eidesstattlichen Versicherung und die prüfungsrechtlichen Folgen (§26 Abs. 2 Bachelor-SPO (6 Semester), § 24 Abs. 2 Bachelor-SPO (7 Semester), § 23 Abs. 2 Master-SPO (3 Semester) bzw. § 19 Abs. 2 Master-SPO (4 Semester und berufsbegleitend) der HdM) einer unrichtigen oder unvollständigen eidesstattlichen Versicherung zur Kenntnis genommen.

____

Josua Wolf, Matrikelnr.: 31298, Stuttgart, den 24.10.2019

Einleitung

Im Rahmen dieser Arbeit wurde ein Tool entwickelt, mit dem Sounddesigner direkt in Unity arbeiten können. Damit soll ein besserer und direkterer Workflow für Programmierer und Sounddesigner gewährleistet werden. Bis jetzt funktioniert die Soundintegration oftmals über externe Tools (wie zum Beispiel FMod). Das ist allerdings weder für Programmierer noch für Sounddesigner der direkteste Weg, denn so muss jede kleinste Änderung kommuniziert und vom jeweils anderen Gewerk bearbeitet werden. Zudem müssen sich dann sowohl Programmierer als auch Sounddesigner in ein neues Tool einarbeiten, das (im Falle von FMod) aus Sicht der Programmierer überhaupt nicht intuitiv zu integrieren ist. Auch Bugfixing im Sound-Bereich kann so immer nur stattfinden, wenn sowohl Programmierer als auch Sounddesigner anwesend sind. Der Fokus auf dem entwickelten Tool liegt also neben einer besseren Integration auf einer leichteren Kommunikation zwischen den Gewerken.

Vor der Entwicklung des neuen Tools, das einen besseren Workflow gewährleistet, wurden Anforderungsprofiele für Sounddesigner und Programmierer erstellt und ausgewertet. Dabei wurde zunächst darauf eingegangen was Sound in Spielen so besonders macht und wo es zu eventuellen Schwierigkeiten beim Erstellen von Sound für Computerspiele kommt. Danach wurden technische Aspekte behandelt, die vor allem für Programmierer relevant sind. Beispielsweise in Bezug auf Mobile Games und auch in Bezug auf das Sound System der Unity Engine, da das Tool darauf aufsetzt. Schließlich wurden Schnittstellen definiert, mit denen ein einfacher Workflow erstellt werden kann und anhand eines konkreten Projekts erläutert. Das programmierte Tool wurde schließlich mit FMod verglichen, um festzustellen in welchen Bereichen die Arbeit mit dem neuen Tool sinnvoll ist und wo ein bewährter Workflow mit FMod besser funktionieren kann.

Neben der Vereinfachung des Workflows ist auch Ziel des Tools möglichst erweiterbar zu sein. Diese Arbeit dient also auch als Dokumentation, damit interessierte Programmierer sich schneller mit dem Tool zurechtfinden. Ein weiterer wichtiger Punkt ist die Gestaltung des Interfaces. Das Tool soll direkt von Sounddesignern benutzt werden und muss deshalb in der Unity Engine ohne weiteren Programmieraufwand bedienbar sein.

Diese Arbeit ist vor allem für Sounddesigner und Programmierer gleichermaßen relevant. So sollen Sounddesigner ein besseres Verständnis für die Programmierung bekommen und umgekehrt. Das entwickelte Tool, erlaubt eine schnellere und einfachere Integration von Ton in Computerspiele. Zudem steht die Kommunikation der Gewerke im Vordergrund. Mit diesem Tool sollen sowohl Programmierer als auch Sounddesigner in der Lage sein, selbstständig Sound einzubauen.

1 Sound in Spielen

1.1 Wichtigkeit von Sound in Spielen

Wie in jedem audiovisuellen Medium spielt der Ton eine essenzielle Rolle. Mit Ton werden Emotionen transportiert oder die Aufmerksamkeit auf bestimmte Details gelenkt. So wird visuellem Inhalt Bedeutung verliehen. Mehr noch, die Immersion des Konsumenten wird durch den Ton enorm gesteigert.

Das gilt für Computerspiele, genauso wie für andere Medien, wenn nicht sogar noch mehr. Computerspiele sind, um zu funktionieren, darauf angewiesen von den Nutzern auch verstanden zu werden. Von der Onlineplatform gameanalytics daher auch der Tipp: “Everything that’s related to the player’s interaction must be of the highest quality possible. [...] Everything that will help the player understand the situation he's in.” (Nathan Lovato 2017) Um den Spieler durch die vielen visuellen Reize zu locken, muss mit Sound darauf aufmerksam gemacht werden, worauf die Aufmerksamkeit aktuell zu lenken ist. Ein klassisches Beispiel dafür sind Texteinblendungen mit Tutorial-Hinweisen. Um aufzufallen, werden diese Einblendungen meistens mit einem Sound begleitet. Das kann je nach Setting ein Papierrascheln oder ein synthetischer Signalton sein. Der Spieler lernt so automatisch, was bestimmte Geräusche bedeuten und kann selbst entscheiden, wie wichtig diese Sounds für ihn sein können. Das wiederrum ist eine Möglichkeit jedem Spieler ein eigenes Spieltempo zu ermöglichen. Wer selten Computerspiele spielt, möchte sich vielleicht sämtliche Tipps durchlesen, die während des Spiels eingeblendet werden. Jemand, der das Spiel zum zweiten Mal spielt, kennt sich vermutlich schon aus und weiß, Einblendung, die mit diesem Geräusch verbunden sind, sind für ihn irrelevant. Diese Art von Kommunikation zwischen Spieler und Spiel nennt sich Audification. Es geht darum Signale zu schaffen, die dem Spieler als Handlungsaufforderung dienen bestimmte Dinge zu tun, wie zum Beispiel eine Texteinblendung zu oder an einen bestimmten Punkt im Level zu gehen. Dabei gibt in dem Fall das Spiel ein Audiosignal ab, welches dem Spieler signalisiert, dass eine Texteinblendung mit Informationen eingeblendet wird. Nach einer kurzen Prägungsphase werden verwendete Sounds von dem Benutzer wie Signale wahrgenommen und die Reaktion des Users läuft unbewusst ab. (Szczypula und Hofmann 2008, S. 56) Diese Art unbewusster Reaktion erleichtert die Bedienbarkeit eines Computerspiels sehr effektiv, zumal der Mensch diese Art der Signale aus dem echten Leben kennt. Ein Handyklingeln oder das Geräusch, wenn die Mikrowelle fertig ist sind zwei nicht unähnliche Signale, trotzdem kennt wohl jeder den Unterschied. In Computerspielen wird zwischen Auditory Icons und Earcons unterschieden. Auditory Icons ähneln sehr stark dem, was wir aus dem echten Leben kennen. Ein Telefonklingeln ist im Computerspiel eine deutliche Aufforderung zum Telefon zu gehen und den Hörer abzuheben. Diese Art von Signalen muss nicht erst erlernt werden. Abstrakter hingegen sind die

eben erwähnten Texteinblendungen. Es gibt für Texteinblendungen im Sichtfeld keine wirkliche Parallele in der echten Welt. Das Signal, das eine Einblendung also verdeutlicht, wird dementsprechend abstrakt sein. Diese Art von Signal wird als Earcon bezeichnet. Um den Lernprozess trotzdem so einfach wie möglich zu gestalten wird oftmals der abstrakte Klang mit einem vertrauten Geräusch kombiniert, das ins Setting des Spiels passt. Spielt das Spiel im Mittelalter könnte ein Papierrascheln passen, als würde sich der Spielcharakter Notizen machen.

Neben einer einfacheren Bedienbarkeit des Spiels wird durch gutes Sounddesign die Immersion des Spielers stark gesteigert (Nathan Lovato 2017). Man könnte denken, dass Immersion in Computerspielen ein geringeres Problem ist, da tatsächlich mit einer anderen Welt interagiert werden kann, anstatt sie nur zu sehen und zu hören. Gutes Sounddesign ist allerdings unbedingt nötig, um eine Spielwelt auch realistisch und echt erscheinen zu lassen. Das Fehlen von Sound oder schlechtes Sounddesign wird das Spielgefühl brechen (Nathan Lovato 2017). Die Schwierigkeit hierbei besteht darin, für jede Situation Sounds vorbereitet zu haben. Im Vergleich zum Film, muss schließlich auf spontane Aktionen des Spielers reagiert werden können. In der echten Welt existiert aus eigener Erfahrung des Spielers kein Ort, an dem nichts zu hören ist. Der Sound muss also jederzeit realistisch erscheinen und für Sound gilt meistens, wenn er nicht auffällt, ist er gut. Ein abrupter Wechsel zwischen zwei Atmos oder die ständig gleiche Wiederholung von Samples würde die Immersion zum Beispiel sofort brechen. Im Film hingegen ist genau vorherzusehen, wann was passiert. Dort kann dann dementsprechend zu jedem Zeitpunkt eine einzigartige Mischung erstellt werden. Schon die Hintergrundgeräusche eines Levels sind also enorm wichtig, denn darauf baut alles andere auf (Stevens und Raybould 2016, S. 24). Der Hintergrund ist aber natürlich nicht alles. Ein sich ständig Wiederholender Loop mit derselben Geräuschabfolge würde die Immersion sofort brechen. Um der Spielwelt mehr Leben einzuhauchen werden gezielt einzelne Sounds in der Welt verteilt, die nur dann spielen, wenn der Spieler in der Nähe ist.

In Computerspielen hat Sound allerdings oftmals noch mehr Funktionen, da neben Emotionen auch Gameplay über Soundtrigger gesteuert wird. Ein gutes Beispiel sind Shooter, bei denen genau auf Schrittgeräusche anderer Spieler geachtet wird, um deren Position zu bestimmen. Sound wird hier zu einer Möglichkeit für die Entwickler das Spiel fair zu gestalten. Das muss aber nicht nur für Multiplayer-Shooter gelten, auch Singleplayer Spiele bieten diesen Ansatz. Beispielsweise kann ein Boss Gegner kurz vor seiner Spezialattacke ein bestimmtes Geräusch von sich geben. So ist der Spieler vorgewarnt und kann in Deckung gehen. Auch das sind Signale, die ein Spieler lernen muss, um eben zum Beispiel einen Bosskampf zu gewinnen. Manchmal basieren auch ganze Rätsel darauf, genau diese Dinge selbst herauszufinden. In Situationen, wie einem Bosskampf, erwarten erfahrene Spieler oft auch genau

diese Signale, ohne sich dessen überhaupt bewusst zu sein. Letzten Endes reagiert der Sound auf Aktionen des Spielers und der Spieler wiederum auf Signale im Sound (Nathan Lovato 2017).

Für jede Form der interaktiven Anwendung sind außerdem Sounds bei Interaktionen sehr wichtig. Ohne diese Sounds wären Anwender niemals sicher, ob die Interaktion tatsächlich stattgefunden hat. Wird beispielsweise in einem Spiel ein Gegenstand aufgehoben, ist ein Sound zur Bestätigung der Interaktion enorm wichtig. Zudem ist alles, was wir im echten Leben tun, von einem Geräusch begleitet. Eine Aktion, die kein akustisches Signal auslöst, wird als unrealistisch wahrgenommen. So wird die Immersion des Spiels gebrochen. Auch das ist eine Form der Audification. Der Spieler würde sonst jedes Mal ins Inventar schauen müssen, um sich sicher zu sein, dass er den Gegenstand tatsächlich aufgehoben hat. Um die Audification in diesem Zusammenhang möglichst leicht verständlich zu gestalten, wird oft ein Geräusch verwendet, das im direkten Zusammenhang mit dem Gegenstand steht. Beispielsweise wird das Füllen eines Eimers durch ein Wassergeräusch gestützt. Dieses Plätschern dient als Auditory Icon und gibt dem Spieler ein Feedback über seine ausgeführte Aktion (Szczypula und Hofmann 2008, S. 56).

Neben diesen Gameplay relevanten Sounds spielt natürlich auch die Musik in Computerspielen eine wichtige Rolle. Musik ist eine der besten Methoden im Sounddesign, um Emotionen zu vermitteln. Um den Spieler längerfristig an ein Spiel zu binden, hilft eine derartige emotionale Verknüpfung mit Aktionen, die der Spieler erreicht hat (Szczypula und Hofmann 2008, S. 13). Eine triumphierende Musik am Ende eines schweren Jump’n’Run-Levels bestätigt den Spieler in seinem Können und spornt gleichzeitig an, dasselbe Gefühl auch beim nächsten Level nochmal erleben zu wollen. Auf die Spitze getrieben wurde diese Erfahrung bei „Rayman Legends“ (2013). Dort wurde nach je zehn bestanden Level ein komplettes Musiklevel freigeschalten, bei dem jeder Sprung und jedes Stück im Level auf die Musik synchronisiert ist.

Das Gesamtpaket aus Musik, Atmo, sonstigen Geräuschen und gegebenenfalls Dialog muss in einem Computerspiel jederzeit gut ineinandergreifen. Auf der einen Seite soll dem Spieler ein möglichst realistisches Hörerlebnis geboten werden und somit eine deutlich höhere Immersion erzielt werden. Auf der anderen Seite benötigt der Spieler die gesamte Geräuschkulisse, um sich in der virtuellen Welt zurechtzufinden. Unabhängig von den schon aufgeführten Audification Elementen, trägt auch die Musik zur Orientierung bei. Diese kann einzelne Spielabschnitte miteinander verbinden oder trennen (Szczypula und Hofmann 2008, S. 50–51). So ist jedem Spieler bei einem Wechsel der Musik klar, dass er sich jetzt in einem anderen Gebiet befindet oder das beispielsweise Gefahr droht und ein Kampf beginnt. Das Erlebnis Computerspiel ergibt ohne Ton oft wenig Sinn oder macht zumindest weniger Spaß. So wirkt sich Game Sound, der sich situativ anpasst, auch auf die Erregung des Spielers aus. Atmung und Herzschlag beschleunigen oder verlangsamen sich abhängig von der Geräuschkulisse und

steuern auch auf diese Weise den Spieler durch die virtuelle Spielwelt (Raffaseder 2010, S. 247–248). Gerade Horror-Spiele funktionieren nur mit einer guten Soundkulisse. Mal wird der Spieler in Sicherheit gewogen, um dann überrascht zu werden, mal soll er sich einfach nicht sicher fühlen. Auf diese Weise wird der Spieler durch die Spielwelt geleitet. So kann ein einfaches „den Gang entlang gehen“ zu einem schrecklichen Erlebnis werden, ohne dass visuell überhaupt etwas passiert ist (Tobias Heidemann 2012). Eine Studie zu diesem Thema zeigt die Auswirkungen auf Herzschlag und Atmung von Sound in Spielen.

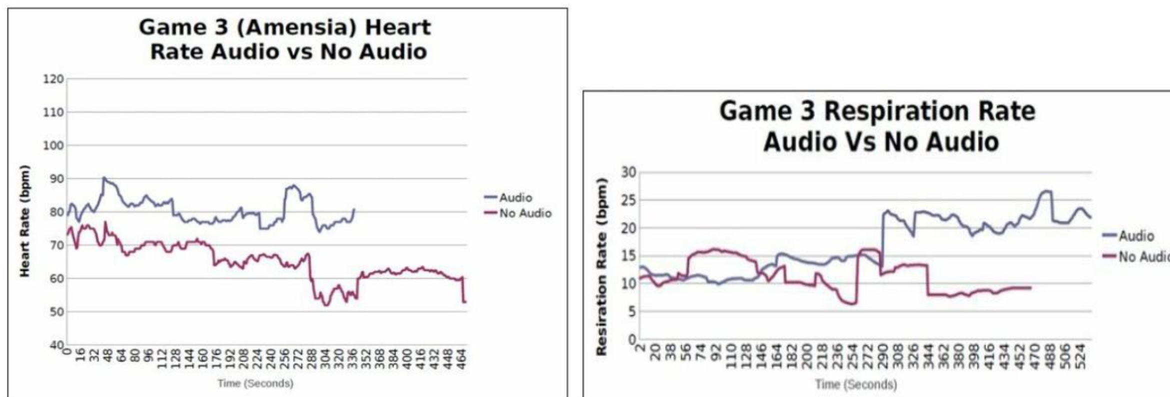

Abbildung 1 und 2: Tobias Heidemann 2012 - Games Studie.jpg (Tobias Heidemann 2012)

1.2 Besonderheiten 2D/3D Sound

Zunächst ist wichtig zu wissen, dass 2D und 3D Sound in Computerspielen nicht der Unterschied zwischen Stereo und Surroundsound sind. Vielmehr bezeichnet 3D Sound all die Sounds, die mit einer räumlichen Koordinate belegt sind und dementsprechend lauter werden, wenn der Spieler darauf zuläuft oder die Position im Stereospektrum ändern, wenn er sich dreht. 3D Sounds können aber (und sind es auch meistens) Monosignale sein. 2D Sound hingegen bezeichnet all die Sounds, die unabhängig von der Position und Rotation des Spielers immer gleich abgespielt werden. Das betrifft normalerweise die Atmo eines Raumes, die ja an jeder Stelle im Raum zunächst gleich sein soll.

Im Unterschied zu Filmen, bei denen normalerweise ein zweidimensionales Bild gezeigt wird, kann sich in einem Spiel der Blickwinkel jederzeit ändern. Eine statische Mischung der Atmo ist schlicht nicht möglich. Der Vogel, der gerade noch vor dem Spieler im Baum gezwitschert hat, muss wenn sich der Spieler umdreht hinter ihm sein. Und der Spieler kann sich jederzeit umdrehen. In Computerspielen findet die „Übersetzung“ von den Soundsignalen über die Game Engine statt. Dort wird während der Laufzeit berechnet, wie welcher Ton ausgegeben werden soll. Alle Sounds, die als 3D Sound in der Spielwelt platziert sind, müssen also im Verhältnis zum Spieler gebracht werden. Die virtuelle Spielfigur

dient normalerweise als „Audio Listener". Alle platzierten 3D Sounds in der Nähe des Audio Listeners werden nun je nach ihren Parametern im 3D Raum abgespielt. Wichtig dabei ist, dass der Audio Listener nicht unbedingt auf der Spielfigur des Spielers liegen muss. Da viele Spiele nicht in der First-Person-Perspektive spielen, hat der Spieler oft tatsächlich eine andere virtuelle Position in der Spielwelt (auch referenziert als Kamera), als seine Spielfigur. Oft ist es allerdings leichter den Audio Listener trotzdem auf die Spielfigur zu legen. Sonst müssten die 3D Sounds mit demselben Versatz zu ihren ingame Objekten platziert werden, mit der auch die virtuelle Kamera (also das, was der Spieler tatsächlich sieht) zur Spielfigur positioniert ist.

Anders als im Kino mit Surround-Mischungen, werden die meisten Geräusche nur als Monosignale importiert und dann in die virtuelle 3D-Welt gesetzt. Auf diese Weise wird nicht nur Speicherplatz gespart, ein Stereo oder Surround-Sound macht in Computerspielen einfach nur in wenigen Fällen Sinn. Beim eben genannten Beispiel mit dem Vogelzwitschern soll der Vogel ja immer im Baum sitzen. Wäre das Vogelzwitschern eine Stereomischung, dürfte das Panorama der Mischung nie breiter sein, als der Bildabschnitt, den der Baum einnimmt (oder einnehmen würde, falls gar nicht zum Baum geschaut wird). Und selbst, wenn man diese Bedingung erfüllen kann, würden diese leichten Nuancen in der Richtung, aus der das zwitschern kommt, keinen Einfluss auf die Qualität des Tons haben. Denselben Effekt könnte man auch erzielen, indem sich die Soundquelle in einem bestimmten Radius zufällig bewegt.

Eine Stereomischung ergibt also normalerweise nur für den Ton Sinn, der keinem bestimmten Objekt in der virtuellen Welt zuzuordnen ist, der also egal, wohin der Spieler schaut, immer gleich klingt. Das können durchaus auch Atmos sein, die zum Beispiel den Raumklang wiedergeben. Vor allem sind allerdings die Musik und eventuell Sounds zu UI-Elementen Kandidaten für Stereosound oder gar Surroundmischungen. Es gilt allerdings zu beachten, dass sich Surroundmischungen eher nach etwas anfühlen, das tatsächlich in der Spielwelt ist. Das heißt Musik mit einer Surroundmischung könnte auch dazu führen, den Spieler zu verwirren, da er die Musik tatsächlich an bestimmten Stellen im Level vermutet (Stevens und Raybould 2016, S. 130). Bei sehr intensiven Szenen zum Beispiel in einem Horror-Spiel, kann das aber auch genau der gewünschte Effekt sein.

Je nach Spielsituation sollen bestimmte Emotionen erzeugt werden. Der Spielsound funktioniert auf mehreren Ebenen, um die gewünschte Stimmung hervorzurufen. Normalerweise liegt über jedem Raum oder jedem Gebiet ein Atmo Loop (etwa 2 bis 3 Minuten) als 2D Stereosound, der den Raumklang wiedergibt und die allgemeine Stimmung setzt. In einer ungemütlichen Szene könnte das zum Beispiel ein nervenaufreibendes tiefes Brummen sein. Über die Atmo kommen kleinere Loops von Geräuschen, die bestimmten Gegenständen zugewiesen sind. Eine tickende Uhr, ein tropfender Wasserhahn oder ein Radio. Diese Loops sind 3D Monosignale, die nur zu hören sind, wenn sich der

Spieler in der Nähe aufhält. Mit solchen Loops wird der Szene der nötige Realismus gegeben. Trotzdem kann eine Spielszene so relativ statisch wirken. Es gibt keine wirkliche Varianz und auf Dauer wiederholt sich alles immer wieder. Um der Szene Leben einzuhauchen müssen kurze Loops von zufälligen Ereignissen eingespielt werden. Das können vorbeifahrende Autos sein, ein Vogelzwitschern oder Schritte im Nebenzimmer. Die Quellen des Sounds müssen hier nicht zu sehen sein, es geht nur darum, dem Spieler das Gefühl einer lebendigen Welt zu vermitteln. Auch diese Sounds sind normalerweise 3D Monosignale, die platziert und zu zufälligen Zeitpunkten abgespielt werden. Um den Spieler besser steuern zu können, müssen manche 3D Sounds immer gehört werden, unabhängig von der Position des Spielers. Lediglich die Richtung spielt hier eine Rolle. Sogenannte Player-oriented Sounds (POS) füllen genau diese Rolle aus. Wie bei normalerweise jedem 3D Sound werden auch hier Monosignale verwendet (Stevens und Raybould 2016, S. 26). Zu den verschiedenen Geräuschen kommt oft noch die Musik, die wie die Atmo als 2D Stereosound läuft. Außer in speziellen Situationen soll die Musik normalerweise nicht an einen Punkt in die Spielwelt gesetzt werden können.

Für alle Sounds können je nach Position des Spielers Filter angewendet werden. Geräusche aus dem Nebenraum sollen gedämpft klingen. Allein auf Grund des Realismus ergibt das oft nur für 3D Sounds Sinn. 2D Sounds sind für den Spieler oft eher abgekoppelt vom tatsächlichen visuellen Inhalt und dienen eher der emotionalen Atmosphäre. In besonderen Situationen, können aber selbstverständlich auch 2D Sounds mit Filtern versehen werden.

1.3 Interaktive Musik

Neben Sounddesign spielt natürlich auch die Musik eine wichtige Rolle. Allerdings gilt auch hier zunächst das gleiche Problem, wie bei allen Geräuschen. Es ist normalerweise unmöglich vorherzusagen, wann genau welche Musik spielen soll. Und was bei Musik vielleicht nochmal stärker ins Gewicht fällt, als bei Atmos: Wenn die Musik wechseln soll, klingt ein Fade zwischen zwei Stücken normalerweise sehr billig und offensichtlich. Das geht gegen die Immersion, die eigentlich erzeugt werden soll.

Teil von dem, wie Musik wahrgenommen wird und was für Emotionen sie erzeugen soll, ist das Tempo. Beginnt also eine Kampfsequenz kann es gut sein, dass das für den Kampf komponierte Stück ein höheres Tempo hat, als das Stück, das davor lief. Ein anderes Problem ist, das Musik in Takten und Phrasen funktioniert. Damit Musik verstanden und als das wahrgenommen wird, was sie ausdrücken soll, müssen bestimmte Elemente zu bestimmten Zeitpunkten passieren (Stevens und Raybould 2016, S. 130). Wenn das Stück mitten im Takt oder in einer Phrase gewechselt wird, verliert die Musik jeden

emotionalen Wert. Nun gilt es trotzdem zu jedem Zeitpunkt in der Lage zu sein, zwischen den Musikstücken zu wechseln, ohne den musikalischen Fluss zu brechen.

Es gibt verschiedene Formen, in denen Musik in ein Computerspiel implementiert werden kann. Je nach Art der Implementierung kann die Musik einen anderen emotionalen Wert haben oder flexibler zu wechseln sein. Ein einfacher Ansatz wäre es, die Musik möglichst unklar zu gestalten. Wenn dem Spieler nicht auffällt, dass eine Melodie nicht vollständig gespielt wird, dass eine Akkord Folge nie aufgelöst wird oder dass ein Rhythmus unterbrochen wird, bricht auch die Illusion von einem durchgängigen Musikstück nicht (Stevens und Raybould 2016, S. 175). So kann problemlos jederzeit von einem Stück zum anderen gefadet werden. Ein Stück so zu komponieren, dass es spannend und gleichzeitig völlig undefiniert ist, ist so allerdings kaum möglich. Es muss also irgendwo immer ein Kompromiss gefunden werden zwischen schneller und direkter Reaktion der Musik auf das Gameplay und dem musikalischen Aspekt eines Wechsels.

Eine Möglichkeit ist die sogenannte „Ornamental Form“. Dabei läuft das Musikstück für das jeweilige Level im Loop und an bestimmten gameplayrelevanten Stellen spielen „Stinger“. Das sind kurze musikalische Elemente, die an jeder beliebigen Stelle im Musikstück auftauchen können. Damit wird zum Beispiel das aufheben eines Boosts bei einem Rennspiel betont. Der Vorteil an dieser Form der interaktiven Musik ist die Flexibilität. Stinger können meistens auf jede beliebige Zählzeit eingespielt werden, dementsprechend kann auf die Aktionen des Spielers ohne Verzögerung reagiert werden. Nachteil ist ganz klar eine gewisse Eintönigkeit, die entstehen kann, da für eine gewisse Zeit immer nur dasselbe Stück spielt.

Um diese Eintönigkeit zu vermeiden, kann die „Parallelform“ genutzt werden. Bei dieser Form laufen mehrere Stücke gleichzeitig. Über Parameter, die durch das Gameplay gesteuert werden, können die Stücke jederzeit ein- und ausgefadet werden. Bedingung hierfür ist, dass die verschiedenen Stücke dieselben Akkordfolgen nutzen und sich auch von der Melodie nicht zu sehr unterscheiden. Diese Form wird oft auch für einzelne Stücke genutzt, um zum Beispiel bei Bedarf Trommeln einzuspielen. So kann ein situativer Mix für jede Situation geschaffen werden. Diese Form der interaktiven Musik bietet ein sehr flüssiges Gameplay mit sehr situativer Musik. Der emotionale Kick, der bei einem gut abgepassten Stinger oder bei einem deutlichen Wechsel in der Musik entsteht, ist allerdings nicht vorhanden.

Die „Transitional Form“ oder auch „Horizontales Neu Sequenzieren“ will genau diesen emotionalen Kick hervorrufen. Hier läuft ein Stück im Loop und wird bei bestimmten ingame Events zu einem anderen Stück gefadet. Auf diese Weise kann man den Spieler maximal belohnen und die damit einhergehende ingame Errungenschaft, die erzielt wurde, noch wichtiger erscheinen lassen. Diese Form muss nicht zwangsläufig zwischen zwei komplett verschiedenen Stücken wechseln. Je nach

Bedarf kann auch eine Art Strophe-Chorus-Wechsel stattfinden. Da würde die „Strophe“ im Loop laufen, bis am Ende des Levels der Bosskampf stattfindet, woraufhin die Musik zum „Chorus“ wechselt. Ist der Boss besiegt, fadet die Musik wieder zurück in die Strophe. Das deutliche Problem dieser Form ist das Timing. Ein Wechsel der Stücke oder einzelnen Abschnitte ist immer nur am Ende einer musikalischen Phrase sinnvoll. Die einzelnen Phrasen müssen also möglichst kurzgehalten sein, um schnell auf das Gameplay reagieren zu können. Bei einem schlecht getimten Wechsel läuft man Gefahr, dass der Spieler die Orientierung verliert. Fängt die Bosskampf Musik erst während des Kampfes an, erwartet der Spieler eine noch größere Gefahr und wird sich vermutlich panisch umschauen.

Die vierte Form wird als „Algorithmic Form“ bezeichnet und soll die flexibelste Form sein, ohne den Verlust der emotionalen Wirkung von starken Wechseln in der Musik. Deshalb ist es auch die komplexeste Form der interaktiven Musik. Teil dessen ist, dass bei dieser Form Musik prozedural generiert wird. Auch wenn interaktive Musik auf gewisse Weise immer prozedural ist, basiert diese Form auf sehr viel kleinteiligeren, teils notenbasierten Elementen. Von jedem Element gibt es verschiedene Varianten, die so miteinander verknüpft sind, dass der Computer, der die Musik letzten Endes zusammensetzt, immer weiß welche Elemente hintereinander passen. Es wird dann immer für das jeweils nächste Element entschieden, welches am besten zur Spielsituation passt. Auf diese Weise kann sehr schnell auf Änderungen im Gameplay reagiert werden. Gleichzeitig können genauso effektiv große Änderungen in der Musik eingestreut werden. Ein weiterer Vorteil ist, dass fast nie die immergleiche Musik läuft. Anders als bei der Transitional oder der Ornamental Form, bei denen es vorkommen kann, dass fünf Minuten die gleiche Musik im Loop läuft (Stevens und Raybould 2016, S. 132).

Was bei allen Formen der interaktiven Musik immer zum Problem werden kann sind die Wechsel zwischen den Stücken. Besonders auffällige Wechsel klingen, trotz ihres hohen emotionalen Wertes, nicht gut und können die Immersion des Spielers brechen. Im schlimmsten Fall geht sogar der emotionale Wert des Wechsels verloren (Stevens und Raybould 2016, S. 174). Da es normalerweise schwierig ist immer komplett saubere, perfekt getimte Wechsel hinzubekommen, müssen diese oft kaschiert werden. Da die Musik meistens unter Dialog und Atmo liegt, können diese Elemente dazu verwendet werden, ein Stück zu wechseln. Ist die Aufmerksamkeit des Spielers auf den Dialog gerichtet, wird eher den Wechsel nicht so bewusst wahrnehmen und erlebt dann vielmehr einen Aha-Effekt am Ende des Gesprächs, wenn die Musik wieder etwas mehr in den Vordergrund rückt. Etwas feiner ist der Wechsel allerdings, wenn er durch Atmo-Geräusche kaschiert wird. Da wir diese Geräusche als gegeben hinnehmen, fällt es uns kaum auf, wenn hinter einem Bretter knarren ein Wechsel zu bedrohlicher Musik stattfindet. An speziellen Stellen könnte man sogar so weit gehen, diese Geräusche mit in die Musik einzubinden (Stevens und Raybould 2016, S. 177).

Allgemein sollten Atmos in ihrer Tonalität auf die Musik abgestimmt sein (Nathan Lovato 2017). Dadurch lassen sich Wechsel beispielsweise deutlich leichter maskieren. Aber auch insgesamt für die Stimmung des Spiels ist es wichtig, nicht das Gefühl zu bekommen, ständig einen etwas dissonanten Klang zu haben (außer es soll ein unangenehmes Gefühl erzeugt werden). Meistens reißt solche leichte Dissonanz den Spieler allerdings aus dem Spielgefühl, ohne das er klar einordnen kann, was so unangenehm ist. In vielen Open-World spielen ist es außerdem üblich Musik zwischendurch komplett auszufaden. Wenn der Spieler durch die virtuelle Welt wandert und keine bestimmten vorgefertigten Aufgaben verfolgt wäre das immergleiche Musikstück spätestens nach zehn Minuten nur noch lästig. Ein leichtes ein- und ausfaden im Wechsel mit Wind und Blätterrascheln ist da sehr viel subtiler und immersiver.

Neben dem emotionalen Wert, den die Musik mitbringt, kann Musik auch das Zeitempfinden des Spielers stark beeinflussen. Bei gezieltem Einsatz fühlen sich zwei Stunden Spielzeit wie nichts an (Szczypula und Hofmann 2008, S. 50).

1.4 Procedual Sounddesign

Da, wie in den drei vorigen Unterkapiteln beschrieben, Sound in Computerspielen nicht linear gestaltet werden kann, braucht es einen neuen Ansatz, wie Sounds für Computerspiele konzipiert werden können. Mehrere Faktoren spielen dabei eine Rolle:

- Der Ton darf nicht repetitiv wirken
- Der Ton muss flexibel auf Aktionen des Spielers reagieren
- Der Ton muss die Spielwelt lebendig erscheinen lassen
- Der Ton muss den Spieler durchs Spiel leiten

Natürlich müssen nach wie vor herkömmliche Mittel zur Soundproduktion genutzt werden. Eine Atmo mit all ihren Nuancen lässt sich nach wie vor am besten in einer Digital Audio Workstation (DAW) erstellen, ganz zu schweigen von Musikkompositionen. Nichtsdestotrotz ist die Arbeit nach erstellen der Sounds noch lange nicht getan.

Das Problem an Computerspielen für den Ton ist die Interaktivität. Anders als beim Film kann dieselbe Szene (was den Bildinhalt angeht) mehrmals vorkommen, jedes Mal denselben Ton abzuspielen würde die Illusion einer lebendigen Welt zerstören. Gleichzeitig ist es aber sehr üblich in Spielen, dass der Spieler an einer Stelle mehrmals vorbeikommt, gegebenenfalls sich sogar länger dort aufhält als erwartet. Eine Atmo für diese Stelle muss also auf Dauer interessant und abwechslungsreich genug sein, damit dem Spieler möglichst nicht auffällt, dass er sich in einer virtuellen Welt befindet (Stevens

und Raybould 2016, S. 37). Ein Geräusch, an dem das besonders über den Lauf des Spiels besonders auffällt, sind beispielsweise die Schrittgeräusche des Spielers. Auf verschiedenen Untergründen müssen verschiedene Geräusche entstehen. Wenn jeder Schritt auf dem jeweiligen Untergrund gleich klingt, bricht das direkt aus der vermeintlichen Realität aus, denn im echten Leben, klingt ein Geräusch nie wie ein Anderes (Stevens und Raybould 2016, S. 59). Ziel von prozeduralem Sounddesign soll sein, aus möglichst wenigen unterschiedlichen Sounddateien möglichst viele verschieden Sounds zu generieren. Auf diese Weise sollen Sounds einfacher in großem Umfang erzeugt werden und es kann im Optimalfall besser auf Anforderungen des Spiels reagiert werden (Stevens und Raybould 2016, S. 59). Natürlich könnten auch für jedes Schrittgeräusch auf jedem Untergrund zehn bis zwanzig unterschiedliche Varianten aufgenommen werden, aber das verbraucht nicht nur mehr Speicherplatz, sondern ist auch ein unverhältnismäßig höherer Arbeitsaufwand (Stevens und Raybould 2016, S. 60). Ein Ansatz um aus verhältnismäßig wenigen Aufnahmen von Schrittgeräuschen, viele verschiedene in game Sounds zu erzeugen, ist die Verkettung von Teilen des Sounds aus verschiedenen Aufnahmen. Schrittgeräusche eignen sich dafür sehr gut, da jeder Schritt aus zwei Teilen besteht. Der Verse und den Zehen. Schneidet man die Aufnahmen zwischen den beiden Teilen auseinander und lässt bei jedem Schritt eine zufällige Kombination aus Verse- und Zehgeräusch abspielen, können aus vier verschiedenen Aufnahmen 16 verschiedene Schrittgeräusche entstehen.

Ein einfaches Mittel zur Generierung von unterschiedlichen Sounds ist der Pitch. Muss beispielsweise ein tropfender Wasserhahn vertont werden, kann der Pitch bei jedem Tropfgeräusch leicht verändert werden. In Kombination mit nur zwei bis drei verschiedenen Tropf-Soundsamples entsteht so schon genug Varianz, um dem Spieler den Eindruck von völlig willkürlichem und damit natürlichem Sound zu geben. Gleichzeitig gilt es bei einem tropfenden Wasserhahn auch darauf zu achten, die einzelnen Samples nicht in einem gleichmäßigen Abstand abzuspielen. Wenn derart kurze Samples mit relativ kurzen Pausen wiederholt werden, klingt das auf Dauer wieder sehr artifiziell (Stevens und Raybould 2016, S. 40).

Für prozedurales Sounddesign ist es normalerweise hilfreich die zur Verfügung stehenden Sounddateien nicht als finales Produkt zu betrachten, das an einer bestimmten Stelle im Spiel abgespielt werden soll. Vielmehr soll durch Kombination der einzelnen Dateien für jede Situation der passende Sound erstellt werden können. Eine Explosion ist beispielsweise nicht einfach ein eine Datei mit dem passenden Geräusch, sondern je nach Material, das das explodiert braucht es ein Splittergeräusch, ein Zischen und einen Knall. Diese Kombination an Sounds wird nicht schon in der DAW erstellt, sondern soll während des Spiels je nach Situation gemixt werden.

Grundsätzlich sollten Loops verhindert werden. Die Gefahr das zu wenig Varianz entsteht, ist zu groß. Loops komplett vermeiden lassen sich allerdings eher selten, zumal sie einfacher zu erzeugen sind, als

viele einzelne One-Shots, die durchdacht alle in die virtuelle Welt platziert werden müssen. Gerade im realen Leben stellt sich eine Atmo allerdings eher aus Clustern zusammen. Innerhalb eines Clusters ist relativ viel Aktivität im Ton und zwischendurch entstehen längere Pausen. Daher sollten Loops normalerweise nur mit Pausen (die in der Länger variieren können) zwischen den einzelnen Durchläufen implementiert werden.

Oft ist der Ton entscheidend für den Spieler, um zu verstehen, in welcher Situation er sich befindet. Zum Beispiel muss der Spieler wissen, ob er sich gerade in Gefahr befindet, weil ein schwerer Gegner in der Nähe ist. Diese Situation wiederum muss im Ton richtig abgebildet sein. Dabei spielen zwei Faktoren eine Rolle. Zum einen muss der Spieler anhand des Tons erkennen können, dass er sich in Gefahr befindet. Es muss also ein Signal vorhanden sein, das der Spieler kennt und als Gefahr interpretiert. Dieses Signal muss immer gleich oder sehr ähnlich klingen. Ziel des Sounddesigns ist es hier nicht, möglichst variabel zu sein, sondern hier geht es um den Wiedererkennungswert. Zum anderen sollte dem Spieler klar werden, dass dieses Mal die Gefahr eventuell größer ist. Über prozedurales Sounddesign soll hier also ein bekanntes Geräusch der Situation entsprechend angepasst werden. Es könnte schon reichen, den Pitch ein wenig zu verändern, damit der Ton tiefer klingt. Dem Spieler fällt im Optimalfall nicht auf, warum ihm diese Situation gefährlicher vorkam als andere Situationen, bei denen er in Gefahr war.

Mit Hilfe von derartigen Signalen wird der Sound, der so in der realen Welt nicht existiert, Teil der Spielwelt. Diese sogenannten „Empathic Sounds“ müssen fast nie mit Hilfe von prozeduralem Sounddesign verändert werden. Der Effekt, den eine Veränderung aber hervorruft, hat allerdings oft eine starke Wirkung auf den Spieler. Im Gegensatz dazu stehen die „Ecological Sounds“. Diese repräsentieren alle Geräusche, die in der echten Welt wahrgenommen werden. Bei diesen Geräuschen ist prozedurales Sounddesign von sehr großer Wichtigkeit. Auf diese Weise werden die Geräusche überhaupt erst als real und nicht als artifiziell wahrgenommen (Grimme Game 2018).

2 Sound in Mobile Games

2.1 Der Unterschied zum Computer

In Mobile Games gelten zunächst dieselben Regeln, wie für den PC: Sound ist wichtig! Allerdings ist schon wegen der Menge an Speicher, die auf einem mobilen Endgerät zur Verfügung steht, die Anzahl an verschiedenen Sounds begrenzt. Daher ist es wichtig bei der Entwicklung für Mobile Games, die Größe der verwendeten Sounds immer im Blick zu haben. Und zwar schon bevor diese implementiert werden.

Eine weiter Designentscheidung ist, ob auf 3D Sound verzichtet werden soll. Es ist kaum möglich bei der Soundausgabe des Handys zu erkennen, aus welcher Richtung Sound kommen, da die Lautsprecher des Handys so nah beieinander liegen. Allerdings können natürlich Kopfhörer angeschlossen werden, mit denen eine Richtung wiederum sehr genau bestimmt werden kann. Letzten Endes ist also die Frage, wie wichtig bei einem Mobile Game (wo die meisten Spiele sowieso nicht in einer Third- oder First-Person-Perspektive stattfinden) der 3D Sound ist und ob man zu Gunsten der Performance darauf verzichten kann.

Ob das Spiel auf dem Smartphone mit Kopfhörern gespielt wird hat außerdem auch Auswirkungen auf die Tonqualität an sich. Da Handylautsprecher meistens nur einen begrenzten Frequenzumfang abbilden können, können manche Töne, wie sie vielleicht in einem Computerspiel implementiert werden, gar nicht verwendet werden. Vor allem tiefe Töne klingen entweder sehr unangenehm oder werden gar nicht gehört, wenn sie auf Handylautsprechern wiedergegeben werden. Bei der Auswahl der Töne muss neben der Speichergröße also auch auf das Frequenzspektrum geachtet werden.

Viele Mobile Games werden in der Öffentlichkeit auch oft ganz ohne Ton gespielt. Den Ton grundsätzlich wegzulassen, ist allerdings auch keine Lösung. Es müssen vielmehr mit einer Anwendung drei verschiedene Zielgruppen bedient werden: Kopfhörer, Lautsprecher und kein Ton. Und für die Zielgruppen, die etwas hören, muss der Ton nach wie vor von hoher Qualität sein. Das heißt, er darf nicht repetitiv werden und muss trotz schlechtem Frequenzgang gut klingen.

Wird ein Spiel auf mehreren Plattformen veröffentlicht, muss von vorne herein damit gerechnet werden, dass Ton auf einem Smartphone nicht so klingt, wie am Computer. Allerdings spielt nicht nur die Tonausgabe eine Rolle. Auch die Menge an Ton kann variieren, denn ein Mobile Game kann nur begrenzt viel Speicherplatz belegen. Dabei spielt auch die Art und Weise, wie der Ton abgespielt wird eine wichtige Rolle. Es muss also jederzeit klar sein, welche Töne nur auf dem Computer abgespielt werden sollen und so gegebenenfalls für eine Mobile Version ersetzt werden müssen.

2.2 Praxistipps

In der Praxis gilt für die Entwicklung eines Mobile Games was den Sound betrifft vor allem: So wenig wie möglich Speicher verbrauchen. Denn wer wenig Speicher verbraucht, kann mehr mit den Sounds machen. Das fängt an mit der Speichergröße der zu importierenden Sounds. Ein tiefes grollen benötigt nur tiefe Frequenzen und dementsprechend muss so ein Sound nach der Nyquist-Shannon-Theorie nicht mit 44,1 kHz abgespeichert werden. Wie allerdings schon erwähnt sind tiefe Töne für Mobile Games nicht besonders gut geeignet aufgrund der schlechteren Lautsprecherqualität. Diese Art Speicherplatz zu sparen ist also oft nicht besonders praktikabel. Sie sollte aber auch nicht komplett abgeschrieben werden. Für einige Geräusche, die aus einem tiefen und einem hohen Teil bestehen (Schrittgeräusche, Explosionen, …), kann der Ton in zwei Teilen abgespeichert werden. Ähnlich wie beim Beispiel mit den Schrittgeräuschen (siehe Kapitel 1.4), werden die Teile dann während des Spiels neu zusammengesetzt. So kann der tiefe Teil des Tons platzsparender gespeichert werden. Und nicht nur das: Auf diese Weise lässt sich leichter Varianz in den Ton einbauen.

Eine weiter Möglichkeit Speicherplatz zu sparen, ist ausschließlich Mono-Sounds zu benutzen. Allerdings gilt das weitestgehend auch für PC Spiele, da die Richtung, aus der ein Sound kommen soll, normalerweise nicht über die Mischung, sondern über die tatsächliche Positionierung des Sounds in der 3D-Weilt festgelegt wird. Das gilt natürlich nicht für Musik oder gegebenenfalls UI Soundelemente, bei denen kein Objekt mit dem Ton gekoppelt ist.

Die einfachste Möglichkeit Speicher zu sparen ist die Kompression. Am effektivsten und am weitesten verbreitet ist da das MP3-Format. Für eine geringe Speichermenge, gerade für den Download der Spiele, ist MP3 eigentlich perfekt geeignet. Die Verarbeitung ist allerdings aufwendiger. MP3 und auch andere einige Kompressionsformate benötigen Zeit und Prozessorleistung, um dekodiert zu werden. Beides Ressourcen, die gerade für Mobile Games sehr wertvoll – vielleicht sogar wertvoller als Speicher – sind. Was den Prozessor angeht kommt es selbstverständlich auf das jeweilige Endgerät an, aber normalerweise sollen Spiele aufgrund der größeren Reichweite möglichst für alle Smartphone Besitzer zugänglich gemacht werden und nicht nur für diejenigen, die teurere und leistungsstärkere Handys besitzen. Normalerweise wird ein Ogg-Vorbis- oder ein PCM-Format genutzt. Die Sounds können direkt in der Game Engine konvertiert werden. Allerdings benötigt auch dieses Format Prozessorleistung beim Dekodieren. Bei der Implementierung des Sounds muss also durchaus auch darauf geachtet werden, nicht zu viele verschiedene Sounds auf einmal abzuspielen. Außerdem hängt die Art und Weise, wie die Game Engine den Ton lädt stark mit der Performance zusammen. So können die komprimierten Sounddateien bereits beim Start des Spiels dekodiert und in den Arbeitsspeicher geladen werden oder aber während des Abspielens dekodiert werden. Abhängig von der Größe der Datei und der Häufigkeit, mit der sie abgespielt wird, können beide Varianten von Vorteil sein. Ein

großer Atmo Loop beispielsweise würde sehr viel Arbeitsspeicher kosten, wäre er vom Start des Spiels an die ganze Zeit dekodiert geladen. Bei kurzen Sounds hingegen, die häufig abgespielt werden, nimmt man den Speicherverlust gerne in Kauf. Das ist besser, als die erhöhte Rechenleistung, die bei jedem Abspielen aufs Dekodieren verwendet würde.

Zu der Art und Weise, wann das dekodieren stattfindet, kommt die Wahl der Kompressionsstärke und damit einhergehende Qualitätsverluste. Für *kurze* Sounds, die *oft* abgespielt werden, eignet sich ein PCM oder ADPCM Format, da bei diesen Formaten keine bis wenig Dekodierung notwendig ist. Und nicht viel Speicherplatz mit einer starken Kompression gewonnen wird. *Mittlere* und *lange* Sounds, die *oft* wiedergegeben werden und *kurze* Sounds, die *selten* wiedergegeben werden, sollten mit dem ADPCM Format komprimiert werden. Das ist schneller zu dekodieren als Ogg-Vorbis und kleiner als PCM. Sounds, die *selten* abgespielt werden und *mittlere* oder *längere* Dauer haben, sollte auf das Ogg-Vorbis-Format zurückgegriffen werden. Diese Sounds sind zu lange für das ADPCM-Format. Da sie nur selten abgespielt werden, kann der höhere Rechenaufwand zum dekodieren leichter verkraftet werden (The Knights of Unity 2015).

Wie schon im vorigen Abschnitt angesprochen, verfügen Handylautsprecher nur über ein begrenztes Frequenzspektrum. Es ist deshalb wichtig, Sounds so zu gestalten, dass alles gehört werden kann. Zu tiefe Töne erzeugen ein blechernen, höhen lastigen Sound, wenn sie überhaupt abgespielt werden. Diese Töne klingen sehr unangenehm (Matti Luonua 2016). Im Gegensatz dazu fehlen diese tiefen Töne natürlich, wenn das Spiel über Kopfhörer gespielt wird. Der Mangel an Tiefe limitiert das Sounddesign. Es müssen andere Wege gefunden werden, das Gefühl von Tiefe zu vermitteln. Mit der sogenannten Saturation kann dieses Gefühl erzielt werden. Dabei werden dem Mix angenehm klingende Harmonien hinzugefügt. Auf diese Weise klingt der Ton präsenter und wärmer. Hier muss ein Kompromiss zwischen einem Mix für Kopfhörer und Lautsprecher gefunden werden.

3 Anforderungen an ein Sound Tool

3.1 Anforderungen Design

Eine der größten Herausforderungen als Sounddesigner für Computerspiele ist, von einem linearen zeitlich abhängigen Design, zu einem vollkommen unvorhersehbaren flexiblen Design zu kommen. Die meisten Geräusche spielen zu einem unvorhersehbaren Zeitpunkt in einer unvorhersehbaren Richtung und müssen gegebenenfalls sogar mit bestimmten Filtern abgespielt werden, je nach Ort, an dem sich der Spieler befindet. Mit einer herkömmlichen DAW kann hier also nur wenig erreicht werden. Ein anderes Problem, das damit einhergeht, ist die Wiederholung von immergleichen Sounds. Wie reagiert man, wenn der Spieler länger in einem bestimmten Gebiet bleibt, als erwartet oder wenn sich die Atmo-Tonspur schon mehrmals wiederholt hat? Der Spieler wird zwangsläufig wahrnehmen, dass er sich in einer artifiziellen Umgebung ohne echtes Leben befindet, denn in der echten Welt findet man quasi nie zweimal exakt das gleiche Geräusch, beziehungsweise mehrmals die sich exakt gleich wiederholenden Muster im Ton (Stevens und Raybould 2016, S. 60). Und so wird die Immersion, die beim Spielen erzeugt werden soll, komplett gebrochen. Ein Tool, das Sound für Computerspiele machen soll, muss dementsprechend flexibel sein. Sounds müssen einen anderen Auslöser haben, als ein bestimmter Timecode. Dieser Auslöser findet normalerweise über die Programmierung statt. Um einem Sounddesigner aber nicht den Programmieraufwand aufzulasten, soll es im Optimalfall möglich sein, ohne Programmierung die aktuelle Atmo des Levels zu testen.

Gleichzeitig wird es immer Sounds geben, die nur bei bestimmten Gameplayereignissen abgespielt werden sollen. Diese Sounds erfordern Programmierung in einer Form (Skript- oder Nodebasiert). Für den Sounddesigner muss es hier also eine einfache Möglichkeit geben diese Sounds zu testen, ohne auf die korrekte Programmierung angewiesen zu sein.

Da der Sounddesigner teilweise nicht mit einer herkömmlichen DAW arbeiten wird können, sollte die Arbeitsoberfläche so ausgelegt sein, dass bekannte Elemente aus der DAW (wie ein Mixer mit Kanälen) vorahnden und benutzbar sind. So sollen einfache Änderungen im Klang der einzelnen Sounddateien direkt in der Game Engine gemacht werden, anstatt die Dateien neu aus der DAW exportieren zu müssen.

Dazu ist es für den Sounddesigner auch wichtig, immer im Überblick zu haben, welche Sounds bereits in die Game Engine importiert sind, welche Sounds schon verwendet werden und gegebenenfalls sogar welche Sounds noch fehlen. Mit einer übersichtlichen Soundbibliothek kann effektiv und schneller gearbeitet werden. Zudem können flexibel Änderungen vorgenommen werden, die direkt für jeden nachvollziehbar und sichtbar sind (Nathan Lovato 2017).

Teil für ein erfolgreiches Sounddesign ist das Verständnis darum, was der Spieler in bestimmten Situationen hören sollte. Es ist also wichtig vor allem die Sounds, die zum Spielverständnis beitragen klar zu kommunizieren. „[…] You need to design sound with their interactive use in mind.” (Stevens und Raybould 2016, S. 126). Dafür muss auch für alle, die keine Sounddesigner sind, klar werden, wann bestimmte Sounds spielen müssen. Und auch anders herum muss für den Sounddesigner die Spielmechanik in jeder Situation klar sein. Der Schlüssel ist eine klare Kommunikation zwischen den Gewerken. Dabei helfen sollen Indikatoren, die mit Anmerkungen in die Spielwelt gesetzt werden können.

Für die Funktionalität vieler Sounds ist es wichtig auf bestimmte Parameter (wie die aktuellen Lebenspunkte) zugreifen zu können. Um diese Parameter nutzen zu können braucht es normalerweise auch einen Programmieraufwand. Damit Sounds mit noch nicht funktionierenden Parametern trotzdem schon getestet werden können und dementsprechend effektiver gearbeitet werden kann, sollten Sounddesigner selbst Parameter erstellen können. Diese Parameter dienen als Platzhalter, bis die Programmierer die korrekten Werte übergeben können. In der Funktion als Platzhalter können Sounddesigner selbst beispielhaft Werte eingeben, damit bestimmte Fälle getestet werden können.

Ein weiterer Vorteil davon Parameter selbst zu erstellen ist die Benennung. Da diese nicht unbedingt mit der Benennung der im Skript vorhandenen Parameter übereinstimmen muss, sind Sounddesigner nicht darauf angewiesen bestimmte Konventionen in der Schreibweise einzuhalten.

Viele Sounds sind an bestimmte Objekte in der Spielwelt gebunden. Gerade auch im Falle des User Interfaces, bei dem jeder Klick vertont sein sollte, müssen die erstellten Sounds an bestimmte Objekte gebunden werden. Es kann allerdings oft sehr kompliziert sein, die korrekten Objekte zu finden. Und selbst, wenn die Objekte gefunden sind, kann es sein, dass mit dem hinzufügen des erstellten Sounds noch nicht das gewünschte Ergebnis erzielt wird, da nicht der korrekte Auslöser für den Sound vorhanden ist. Das Erstellen dieser Auslöser ist Aufgabe der Programmierer. Eine klare Kennzeichnung, wo genau Sounds hinzugefügt werden sollen, hilft. Allerdings muss der Programmierer dafür natürlich wissen, dass ein Sound für beispielsweise einen Button im Menü hinzugefügt werden soll. Das wiederrum funktioniert am besten mit der schon erwähnten Liste aller vorhandenen und geplanten Sounds.

Zusammenfassend sind folgende Punkte besonders wichtig für Sounddesigner:

- Der implementierte Sound muss in irgendeiner Form immer getestet werden können
- Bekannte Bedienelemente
- Direkte Bearbeitung für kleinere Änderungen oder Effekte an den Sounds
- Überblick über alle vorhandenen und geplanten Sounds

- Erleichterte Kommunikation mit anderen Gewerken
- Erstellen und benennen von Parametern
- Einfaches Hinzufügen von Sounds zu virtuellen Objekten

3.2 Anforderung Programmierung

Sound in ein Computerspiel einzubauen erfordert meistens mehr Programmieraufwand, als erwartet. Die vielen speziellen Anforderungen oder Anwendungsfälle, wann welcher Sound mit welchem Filter aus welcher Richtung gespielt werden soll, erfordern oft für den jeweiligen Sound spezielle Skripte. Das führt zu einem insgesamt unübersichtlicheren Projekt und einer Menge unnötigem Ballast im Zwischenspeicher. Im einfachsten Fall würde ein Programmierer natürlich in einem Skript am Anfang die Methode „Sound.Start()" aufrufen und die Game Engine erledigt dann den Rest, aber das kann so nicht funktionieren, weil jedes Spiel andere Anforderungen an den Sound hat. Ein weiteres Problem ist die Kommunikation zwischen den Gewerken. Ein Sounddesigner wird selten selbst Skripte schreiben. Es benötigt also immer mindestens zwei Personen, um Ton einzubauen. Und dabei muss dann der Sounddesigner dem Programmierer immer erklären, was er genau will. Oder im schlimmsten Fall: Der Sounddesigner bekommt diktiert, welche Sounds zu liefern sind, ohne überhaupt die Möglichkeit zu haben, diese im Spiel zu sehen, um daran weiter zu arbeiten.

Es muss dem Programmierer möglich sein die Sounds, die vom Sounddesigner für bestimmte Gameplayfälle erstellt wurden, zu erkennen und einfach aus seinen Skripten aufzurufen. Dafür ist eine Liste mit Sounds, die vom Sounddesigner bearbeitet werden, sehr hilfreich, da der Programmierer gegebenenfalls schon im Voraus Code für den jeweiligen Sound vorbereiten kann.

Damit Sound wirklich interaktiv werden kann, muss er mit den Spielmechaniken funktionieren. Dafür wiederum müssen Parameter aus den Skripten an den Ton übergeben werden können. Im einfachsten Fall könnten das die aktuellen Lebenspunkte des Spielers ein. Bei einem niedrigen Wert wird dann die Musik intensiver oder es wird ein Herzklopfen abgespielt. Die Logik zu erstellen, wann genau welcher Ton abgespielt werden soll, ist aber grundsätzlich Aufgabe des Sounddesigners.

Für einen reibungslosen Workflow ist es allerdings nicht nur wichtig, diese Parameter zu übergeben, sondern auch früh genug zu wissen, welche Parameter vom Sounddesigner benötigt werden. Neben der Liste an geplanten Sounds sollte zu jedem Sound auch eine Liste an benötigten Parametern existieren. Im besten Fall können Parameter dann direkt an diese Liste übergeben werden. Der Platzhalter Parameter wird dann einfach vom korrekten Parameter ersetzt.

Da während der Entwicklung des Spiels die Programmierer meistens den besseren Überblick über den aktuellen Entwicklungsstand haben, als die Sounddesigner, kann es gerade bei Gameplayrelevanten Sounds dazu kommen, dass Programmierer zum Testen Sounds benötigen. Besonders Dialoge und Dialogsysteme werden hauptsächlich von Programmierern entworfen, da diese mit anderen Spielelementen (die bei jedem Spiel anders sein können) harmonieren müssen.

Gameplay relevante Sounds müssen oft auch auf bestimmte Objekte gelegt werden, damit sie zur richtigen Zeit am richtigen Ort sind. Wenn für den konkreten Fall noch keine Töne vorhanden sind, benötigt es auch hier zum Testen Platzhaltertöne. Im Optimalfall soll der Ton dann eigenständig vom Sounddesigner ersetzt werden können. Hierfür benötigt es einen zentralen Punkt im Editor, an dem alle verwendeten Töne aufgelistet sind. Gleichzeitig soll von der Liste auch zu den Objekten gesprungen werden können, damit gegebenenfalls Änderungen am Verhalten des Objektes vorgenommen werden können.

Zusammenfassend sind folgende Punkte für einen Programmierer besonders wichtig:

- Möglichst geringe Anzahl an verschiedenen Skripten
- Ein Überblick über geplante und vorhandene Sounds
- Die Möglichkeit relevante Parameter an das Tool zu übergeben.
- Auf einen Blick erkennen, welche Parameter für den Ton benötigt werden.
- Die Möglichkeit Gameplayrelevante Sounds anzufordern und Platzhaltertöne abzuspielen.
- Sammelstelle für alle Platzhaltertöne für eine leichtere Navigation.

3.3 Vergleich der Anforderungen

Vergleicht man die Anforderungen so fällt schnell auf, dass beide Seiten einen direkteren Workflow benötigen. Sounddesigner möchten gerne selbst Sound einbauen und unmittelbar ein Ergebnis zum Testen haben. Das würde auch das Leben der Programmierer vereinfachen, die dann mit einem deutlich fertigeren Produkt auf von der Soundseite her arbeiten können. Teil dieses besseren Workflows soll eine Liste mit verwendeten und geplanten Sounds sein.

Da Programmierer und Sounddesigner eng zusammenarbeiten müssen, profitieren alle davon, wenn die Kommunikation zwischen den Gewerken möglichst reibungslos verläuft. Kommentare und Anmerkungen zu verwendeten Sounds in der Szene und in der Soundliste können dabei sehr helfen.

Für funktionalen Sound muss aber auch vor allem die Kommunikation zwischen den Skripten der Programmierer und den vorbereiteten Sounds funktionieren. Um das nicht unnötig kompliziert für die Programmierer zu gestalten, sollen relevante Parameter möglichst einfach übergeben werden können.

Dafür benötigen die Programmierer natürlich auch eine Möglichkeit möglichst schnell und einfach alle benötigten Parameter einzusehen. Sounddesigner wiederrum profitieren von einer einfachen Benennung von Parametern und vor allem davon, selbst Parameter erstellen zu können.

Im Gegenzug muss es auch Programmierern möglich sein Testsounds an bestimmten Stellen abzuspielen. Die Logik, wann diese Sounds abgespielt werden sollte mit dem Hinzufügen der echten Sounds nicht verloren gehen. Unabhängig davon soll es möglich sein, Sounds, die auf bestimmten Objekten in der Welt gelegt wurden, leichter zu finden. Im Optimalfall gibt es eine zentrale Stelle, von der aus zu allen Sounds navigiert werden kann.

Beiden Gewerken ist stark damit geholfen, wenn die jeweils andere Seite ein besseres Verständnis für die eigene Arbeit hat. Dadurch muss man an bestimmten Stellen nicht immer auf ein Ergebnis von anderer Stelle warten, sondern kann gegebenenfalls schon weitere Anwendungsfälle vorbereiten.

3.4 Wie könnte ein besserer Workflow aussehen?

Ein wichtiges Stichwort hierfür ist Kommunikation. Solange alle Bescheid wissen, wo welche Sounds einzufügen sind, kann leichter unabhängig voneinander gearbeitet werden. Dadurch wird die Effizienz deutlich gesteigert. Mit wachsender Projektgröße wächst auch die Menge an Sounds, die benötigt werden. Um auch später einen guten Überblick über alle Sounds zu haben, sollten diese von Anfang an gegliedert werden. Grundsätzlich können Sounds in fünf verschiedene Kategorien gegliedert werden. Im sogenannten INFORM-Modell teilen sich Töne in (Stevens und Raybould 2016, S. 310):

Instruction: Töne, die bestimmte Gameplaymechaniken hervorheben, um diese dem Spieler näher zu bringen. (Zum Beispiel Dialoge oder Sounds, die Infoboxen symbolisieren.)

Notification: Töne, die dem Spieler Informationen über den Charakter, andere Objekte oder das Spiel selbst liefern. Auf diese Weise kann der Spieler aufgefordert werden, auf bestimmte Gegebenheiten zu reagieren. (Zum Beispiel Schrittgeräusche anderer Spieler oder Musik.)

Feedback: Töne, die auf Aktionen des Spielers reagieren und somit Belohnung oder Bestrafung sowie Bestätigung oder Ablehnung symbolisieren. (Zum Beispiel Sounds beim Türe öffnen oder Musik.)

Orientation: Töne, die dem Spieler geographisch (oder metaphorisch) beim Orientieren helfen. (Zum Beispiel Atmo oder One-Shots für bestimmte Objekte in der virtuellen Welt.)

Rhythm-action: Töne, die mit dem Input des Spielers synchronisiert sind. (Zum Beispiel Karaoke oder andere Musikspiele)

Mechanic: Töne, die als Spielmechanik genutzt werden (außer Rhythm-aciton-Töne). (Zum Beispiel Rätsel mit Tonsequenzen oder Ablenkungsmanöver.)

Dieses Modell kann von Sounddesignern genutzt werden, um die geplanten Sounds besser zu gliedern. Außerdem kann so leichter abgeschätzt werden, wie viel Varianz für die verschiedenen Anwendungsbereiche vorhanden ist und welche Sounds gegebenenfalls noch benötigt werden. Für Programmierer auf der anderen Seite hilft diese Aufschlüsselung dabei den Programmieraufwand für die verschiedenen Sounds besser abzuschätzen. Instruction, Notification und Feedback-Töne sind meistens mit geringem Aufwand und Orientation-Töne im Optimalfall gar keinen Aufwand verbunden. Rhythem-action-Töne hingegen sind meist eigene Spiele und Mechanic-Töne bedeuten individuellen Aufwand für jeden dieser Töne.

Eine Liste an geplanten Sounds sollte sowohl von Sounddesignern als auch von Programmierern gefüllt werden können. Sounddesigner werden von sich aus eher Instruction, Notification, Feedback und Orientation-Töne erstellen, wohingegen Programmierer vor allem Mechanic-Töne benötigen. Mechanic-Töne wiederrum sind für Sounddesigner nur erkennbar, wenn sie das Spiel selbst spielen würden, dementsprechend müssen Programmierer in der Lage sein Anwendungsfälle, wie diese der Liste hinzuzufügen. Natürlich hilft es jedem Sounddesigner, wenn er so früh wie möglich Prototypen spielen kann. Gerade Gameplay-Sound-Interaktionen können so leichter erkannt werden, aber auch alle anderen Formen von Tönen können so besser konzipiert werden. Den Sounddesigner schon in die Konzeption des Levels und der Charaktere mit einzubeziehen ist für einen effizienten Ablauf unerlässlich (Nathan Lovato 2017).

Nachdem eine Liste an geplanten Sounds existiert und Sounds erstellt wurden, müssen diese implementiert werden. Die größte Herausforderung hier besteht darin einen guten Workflow zwischen Programmierern und Sounddesignern zu finden, was all die Töne betrifft, die der Sounddesigner nicht selbstständig implementieren kann. Töne, die ohne Hilfe der Programmierer eingebaut werden können, sind normalerweise nur Atmos und teilweise Musik, die ohne bestimmten Auslöser von Anfang an spielen. Alle anderen Töne benötigen Programmierunterstützung. Um für beide Seiten ein möglichst gutes Ergebnis zu erzeugen, sollte der Sounddesigner so viel wie möglich vorbereiten. Im Optimalfall muss der Programmierer dann nur noch relevante Parameter übergeben oder die Sounddatei zum richtigen Zeitpunkt mit einem einfachen Aufruf abspielen. Durch eine gute Vorbereitung des Sounddesigners wird das Ergebnis mit höherer Wahrscheinlichkeit dem Entsprechen, was der Sounddesigner geplant hatte.

Die Art und Weise, wie die Zusammenarbeit zwischen den Gewerken an dieser Stelle funktioniert, ist in höchstem Maße vom verwendeten Tool abhängig, mit dem der Sounddesigner arbeitet. Vor allem

im Hinblick auf die Menge an Vorbereitung, die der Sounddesigner treffen kann, um bestimmte Geräusche zu implementieren. Um konkrete Schwierigkeiten aufzuführen soll hier ein beispielhafter Workflow mit FMod aufgezeigt werden. FMod ist ein externes Tool, mit dem interaktiver Sound für Computerspiele erstellt werden kann. FMod funktioniert sowohl mit der Unreal Engine als auch mit Unity. Da FMod kostenfrei zu benutzen ist, kommt es in den meisten Studentenprojekten zum Einsatz. Aber auch große Produktionen (u. a. Rise of the Tomb Raider, Kingdom Come: Deliverance) setzen auf FMod.

3.5 Sound im Spiel „Devolution“

Devolution ist ein Mobile Game, bei dem mehrere Spieler um die Vorherrschaft kämpfen. Besiegte Spieler werden zu Vasallen der Gewinner, die somit eine stärkere Armee zur Verfügung haben, um weitere Gegner zu besiegen. Das Spiel funktioniert als rundenbasiertes Strategiespiel. Die Spieler können auf dem Spielplan, der aus mehreren Hexagonen besteht, Städte gründen und Armeen ausheben. Jedes Hexagon steht für einen bestimmten Landschaftstyp. Je nach Landschaftstyp sind verschiedene Interaktionen möglich. Da es sich um ein Studentenprojekt handelt, ist die Menge an Sound und Anwendungsfälle begrenzt. Trotzdem kann man anhand dieses Projektes leicht einen Überblick über den Workflow mit FMod bekommen.

Die implementierten Sounds sind neben einer Atmo und einer Musik verschiedene User Interface (UI)-Sounds und einige One-Shots bei bestimmten Events.

Die Atmo beschränkt sich auf Blätterrascheln, Vogelgezwitscher und Wind. Der Spieler kann zwar die Distanz zum Spielplan verändern, aber er kann nur begrenzt nahe heran und nur begrenzt weit wegzoomen. Die Atmo soll hier einen möglichst neutralen Hintergrund schaffen. Ein zusätzliches Feature soll der Wind sein. Je weiter der Spieler herauszoom (und dementsprechend höher über der Weltkarte schwebt), desto stärker soll der Wind werden. Je nach Aktionen auf dem Spielplan wird die Atmo untermalt von verschiedenen One-Shots. Zum Beispiel, wenn der Auftrag erteilt wird eine Stadt zu bauen oder wenn ein Marschbefehl erteilt wird.

Zu dem kommen UI Sounds für das Klicken von Buttons im Hauptmenü. Der Play-Knopf klingt dabei etwas anders, als die anderen Buttons. Außerdem gibt es ein Papierrascheln, das Nachrichten und Einblendungen während des Spiels signalisiert.

Unter allem liegt ein Musikloop. Zwischen Hauptmenü und tatsächlichem Spiel wechselt das Stück. Damit wird eine klare Trennlinie geschaffen und Spannung erzeugt. Während des Spiels werden bei bestimmten Aktionen Stinger eingespielt. Bei gewonnenem Kampf, spielt ein positiver Stinger, bei

verlorenem Kampf, ein Negativer. Außerdem spielt ein warnender Stinger, wenn der Spieler angegriffen wird.

FMod eignet sich gerade für längere Atmos und Musik sehr gut, da leicht mehrere Spuren übereinandergelegt werden können. Mit Hilfe von benutzerdefinierten Parametern kann dann beispielsweise die Lautstärke der einzelnen Spuren angesteuert werden. So können Soundevents gebaut werden, die aus verschiedenen Sounddateien bestehen. Mit Hilfe der Parameter können diese Events dann auf das Spielgeschehen reagieren. Für kurze One-Shots bietet FMod keine besonderen Vorteile.

FMod Events können in der FMod Anwendung erstellt und getestet werden. Um die erstellten Events dann aber auch im Spiel benutzen zu können, müssen die einzelnen Events über Skripte aufgerufen und abgespielt werden. Das instanziieren von FMod Events erfordert leider eine etwas umständliche Programmierung und im Falle von 3D-Events muss das erstellte Event zusätzlich noch einem physikalischen Objekt in der virtuellen Welt zugeteilt werden. Diese Zuteilung kann erst stattfinden, wenn das Event in den Speicher geladen wurde. Das bedeutet, dass in irgendeiner Form im Skript abgefragt werden muss, wann das Event geladen ist, um es dann mit einem Objekt zu verknüpfen.

Eine weitere Schwierigkeit ergibt sich in der Benennung der Events. Events werden immer über den Eventpfad „event:/Eventname“ verknüpft. Ist der Eventname nicht eindeutig, ist es dem Programmierer nicht möglich, das Event eigenständig zu implementieren. Oder nur, wenn extern eine Liste existiert, in der festgelegt wird, wann welches Event spielen soll. Dasselbe Problem ergibt sich für Parameter. Diese können zwar, wie die Events auch, vom Programmierer in Unity selbst eingesehen werden, genaue Timings oder allgemeine Infos zu den Parametern müssen allerdings anders geklärt werden.

Die übliche Herangehensweise ist also eine Liste, in der alle geplanten und vorhanden Sounds zu finden sind, in der erklärt wird, wann und wie der Sound zu spielen hat und wie der Sound heißt. So kann der Programmierer zumindest die einzelnen Anwendungsfälle vorbereiten. Der Sounddesigner hat allerdings bis nach diesem Schritt nie die Möglichkeit, die Sounds im Spiel selbst zu hören. Änderungen in FMod lassen sich aber zum Glück leicht machen, ohne dass die Programmierung davon betroffen wäre. Trotzdem hängt eine Menge Arbeitsaufwand am Programmierer. Wo genau ändert sich eine Atmo, auf welchen Objekten liegt ein 3D-Sound? Das sind Fragen, die der Sounddesigner zwar beantworten kann, aber nicht selbst lösen kann. Für solche Fälle müssen normalerweise sowohl ein Programmierer und ein Sounddesigner anwesend sein, um diese Dinge gemeinsam einzubauen.

Ein weiters Hindernis ist Abhängigkeit voneinander. Ohne Programmierung, kann der Sounddesigner schlicht nichts in der Game Engine testen. Der Programmierer kann ohne fertige Events wiederrum

kaum etwas vorbereiten. Es können nur wenige Arbeitsschritte parallel ablaufen. Das kostet unnötig Zeit.

Im Falle von „Devolution“ ist der Programmieraufwand auch auf Grund von wenigen Sounds begrenzt. Konkret müssen folgende Fälle abgedeckt werden:

1. Alle 2D Events. Dazu zählen auch die Stinger bei der Musik. Alles, was hierfür benötigt wird, ist der Eventpfad.
2. Alle 3D Events. Da diese Sounds mit Objekten verknüpft werden müssen, braucht es noch ein konkretes physikalisches Objekt.
3. Eventuelle Parameter der Sounds. Musikwechsel beispielsweise funktionieren meistens über Parameter, da der Wechsel oft zum Ende eines Taktes oder einer Phrase stattfinden soll.

Diese drei Punkte sind im Grunde bei jedem Spiel mit FMod unumgänglich. Die größte Schwierigkeit hier ist die Art und Weise wie und wo die Programmierung tatsächlich umgesetzt wird. Jedes Event braucht einen Auslöser. Für Atmo und Musik ist das ganz simpel der Spielstart, für die Stinger zum Beispiel ist das aber schon nicht mehr so einfach. Da ist eine konkrete Stelle in einem bestimmten Skript relevant. Wenn jetzt aber für jeden Stinger das jeweilige Event dann an der betreffenden Stelle im Skript instanziiert und abgespielt wird, führt das zu schrecklich unübersichtlichen Skripten. Fehler im Ton zu finden (spielt zu früh, spielt gar nicht, …) ist da enorm aufwendig. Der Ton kann ja quasi von überall aufgerufen worden sein. Außerdem kann sich auf diese Weise nicht ein Programmierer speziell um den Ton kümmern, sondern alle Programmierer müssen einen Teil davon einbauen. Natürlich muss am Ende in jedem Fall aus den betreffenden Stellen im Skript ein Aufruf an den Ton kommen, aber der Ton selbst sollte nur von einem bestimmten Skript aus gestartet werden, um eben zu verhindern, dass Ton spielt, der nicht spielen sollte oder umgekehrt. Und selbst wenn Probleme im Ton sind, gibt es nur ein Skript, dass fehlerhaft sein kann, solange der Aufruf funktioniert.

Ein weiterer Grund für ein zentrales Skript ist die Art und Weise, wie die FMod Application Programming Interface (API) funktioniert. FMod Events müssen zunächst als EventInstance mit dem Eventpfad instanziiert werden. Der Eventpfad wird als String übergeben. Sobald ein Pfad für die EventInstance gesetzt ist, kann dieses mit EventInstance.start() gestartet werden. Dank einer überarbeiteten API seit der FMod Version 2.0 müssen Parameter nicht mehr als ParameterInstance instanziiert werden. In älteren Versionen musste eine ParameterInstance mit der EventInstance, aus der der Parameter genutzt werden soll und einem String für den Namen des Parameters erstellt werden. Und auch das setzen und holen von Parameterwerten war unnötig kompliziert. Mit der neuen API können Parameter über EventInstance.setParameterByName(string name, float value) gesetzt werden. Passend dazu gibt es auch eine getParameterByName Methode.

Der Workflow bei „Devolution“ sieht konkret so aus: Nachdem die grundsätzlichen Regeln und Funktionen von Seiten des Gamedesign festgelegt sind und erste Concept Arts vom Grafikdepartement erstellt werden, kann der Sounddesigner erste Entwürfe für Atmo und Musik erstellen. Sobald sich abzeichnet, wie die Spielmechaniken umgesetzt werden, kann von Seiten der Programmierung und des Gamedesign eine Liste erstellt werden, mit allen benötigten Sounds. Diese Liste wird dann vom Sounddesign um alle weiteren Töne ergänzt. Mit der Liste im Blick, kann der für den Ton zuständige Programmierer einen „Sound Manager“ programmieren, der die oben genannten Fälle abdeckt. Alle Events werden über dieses Skript abgespielt. Der Aufruf, den Sound abzuspielen, muss dann nur noch in den anderen Skripten hinzugefügt werden. Ist das FMod Projekt mit den geplanten Sounds bereit, setzen sich der Sounddesigner und der Programmierer zusammen, klären welche Funktion die erstellten Parameter haben und ob bestimmte Events eventuell anders aufgebaut werden müssen, um besser mit der Logik der anderen Skripte zu funktionieren. Danach kann der Sounddesigner weiterhin Änderungen an den Sounddateien vornehmen, ohne dass die Programmierung davon betroffen ist. Neue Sounds oder weiter Parameter erfordern natürlich weiteren Programmieraufwand. Da „Devolution“ anders als beispielsweise bei einem Rollenspiel in nur einer Umgebung (der Landkarte) spielt, ist zunächst kein Wechsel in der Atmo abhängig von der Position des Spielers nötig. In diesem Projekt fällt also ein großer Teil der üblichen Arbeit weg, bei dem innerhalb des Unity Editors bestimmte Bereiche für verschiedene Atmos festgelegt werden müssen.

3.6 Das Unity Sound System

Die Unity Engine bietet (natürlich) auch ein Sound System. Das ist allerdings weder für Programmierer noch für Sounddesigner leicht zugänglich. So gibt es quasi keine Oberfläche, mit der ein Sounddesigner arbeiten könnte, ohne zu Programmieren. Ein Programmierer wiederrum müsste wissen, was der Sounddesigner vorhat, um ihm die benötigten Features an die Hand zu geben. Es ergibt sich also mehr oder weniger dieselbe Pattsituation wie zuvor, mit dem Unterschied, dass der Sounddesigner nicht einmal Soundevents vorbereiten kann.

Ein Vorteil des Unity Sound Systems ist allerdings der Mixer. Ähnlich, wie in einer herkömmlichen DAW können hier verschiedene Soundevents zu einem virtuellen Mischpult hinzugefügt werden. Dort können dann die einzelnen Kanalzüge gepegelt werden, zu Gruppen hinzugefügt werden oder mit Effekten versehen werden. Auf Basis dieser Funktion soll auch das neue Tool funktionieren.

Um einen Sound in Unity abzuspielen wird die sogenannte „Audio Source“ Komponente benötigt. Komponenten können auf jedem beliebigen Objekt beigefügt werden. Beispielsweise das Model einer Türe. Fügt man dem Türobjekt eine Audio Source Komponente bei, und startet diese über ein ebenfalls

hinzugefügtes Skript, wenn sich die Türe öffnet, so spielt der beigefügte Sound an der Stelle der Tür. Grundsätzlich werden 3D Sounds immer in Relation zum „Audio Listener“ abgespielt. Der Audio Listener ist eine Komponente, die üblicherweise auf der virtuellen Kamera, die der Spieler steuert, liegt.

Jede Audio Source benötigt eine Audiodatei, die abgespielt werden soll. Unity unterstützt dafür einige gängige Audioformate (.aif, .wav, .mp3, .ogg, .xm, .mod, .it, .s3m). Die Audiodateien werden in die Unity Engine importiert. In der Engine selbst kann dann – wenn gewünscht – ein Kompressionsmethode und eine Kompressionsrate ausgewählt werden.

Sounds können auf verschiedene Weisen abgespielt werden. Der Unterschied besteht darin, wann der abzuspielende Sound in den Speicher geladen und dekodiert wird. Abhängig von der Größe der Sounds sind unterschiedliche Methoden Laufzeiteffizienter. Die „Decompress On Load“ Option dekodiert die komprimierten Sounds, sobald diese in den Speicher geladen werden. Damit wird ein hoher Rechenaufwand beim Abspielen vermieden. Diese Option sollte aber auf keinen Fall mit großen Dateien verwendet werden. Vorbis-kodierte Dateien benötigen nach dem Laden und dekodieren so etwa zehn Mal so viel Speicher. Für diese Dateien bietet sich eher die „Compressed In Memory“ Option an. Mit dieser Option werden die Dateien komprimiert in den Speicher geladen und während des Abspielens dekodiert. Das erzeugt zwar auch einen erhöhten Rechenaufwand, für große Dateien ist das aber besser, als diese komplett dekodiert in den Speicher zu laden. Die dritte Option nennt sich „Streaming“. Damit wird kein RAM benötigt, da die Dateien direkt von der Festplatte geladen und dekodiert werden. Es wird allerdings Rechenleistung benutzt, selbst wenn keine Sounds abgespielt werden.

Um Audio Sources dem Mixer hinzuzufügen, muss am Mixer ein Kanal erstellt werden. Dieser kann dann wiederrum bei der Audio Source Komponente als Ausgang eingefügt werden. Grundsätzlich muss eine Audio Source, um hörbar zu sein, nicht über den Mixer ausgegeben werden. Der Mixer ist vielmehr ein Tool, das zwischen Audio Source und Audio Listener geschalten ist. So können beispielsweise im Menü Einstellungen getroffen werden, wie laut Hintergrundgeräusche oder wie laut Musik sein sollen. Außerdem kann innerhalb einer Atmo Feinabstimmungen getroffen werden oder es können einzelne Instrumente zum Gesamtmix hinzugefügt werden. Außerdem kann über den Mixer Hall oder ein Equalizer eingefügt werden.

Die Audio Source Komponente bietet neben der Möglichkeit, das Signal über den Mixer zu schicken, weiter Einstellungen. Dazu gehören: Das Stummschalten, das Ignorieren von Mixer Effekten, wiederholtes Abspielen, Pitch, das Starten der Sounddatei, sobald das Spiel gestartet wird, Einstellungen für den 3D Klang und mehr. Alle Einstellungen auf der Audio Source Komponente können

auch über andere Skripte gesteuert werden. Für ein Audio Tool, das direkt in Unity funktioniert, kann diese Funktion dafür genutzt werden, alle Audio Sources an einer Stelle zu bündeln und von dort steuern zu können, anstatt diese immer die einzelnen Objekten in der Szene suchen zu müssen, die eine Audio Source Komponente haben.

Teil des Konzeptes der Unity Engine ist ein modularer Aufbau. Jedes Objekt ist zunächst ein „Game Object“. In einers Szene existieren mehrere Game Objects. Von einem Game Object können auch mehrere Instanzen existieren, die dann unabhängig voneinander mit dem selben Verhalten ausgeführt werden. Durch Komponenten, wie die Audio Source Komponente, wird dem Objekt eine Funktion verliehen. Komponenten können selbst programmiert werden (also nur in Form von angehängten Skripten existieren), wenn für den gewünschten Anwendungsfall noch keine vorgefertigte Komponente existiert. Auf diese Weise kann beispielsweise eine Komponente geschaffen werden, die alle Audio Sources an einer Stelle bündelt und diese verwalten lässt. Komponenten haben normalerweise immer zwei Funktionsfelder. Zum einen müssen sie im Editor verständlich und bedienbar sein, zum anderen erfüllen sie eine Funktion während der Laufzeit. In diesem Fall dient die Komponente allerdings nur der besseren Bedienbarkeit im Editor und erfüllte keinerlei Funktion während der Laufzeit. Diese Art von Skripten nennen sich Editor Skripte. Dabei werden Komponenten unabhängig von ihrer Funktion anders dargestellt, um so eine leichtere Bedienbarkeit des Editors zu ermöglichen. Gleichzeitig können mit Editor Skripten schon bestimmte Verknüpfungen im Hintergrund automatisch gesetzt werden. Die Kombination von einer hoffentlich leichteren Bedienbarkeit und der Automatisierung von kleineren Arbeitsschritten, die im Workflow sonst wenig Sinn ergeben, macht Editor Skripte zum perfekten Mittel, um eigene Tools zu programmieren.

4 Entwicklung eines Sound Tools auf Basis der Anforderungen

4.1 Ergebnis der Recherche

Ein wichtiges Ziel des neuen Tools soll die einfache Kommunikation und damit eine einfachere Implementierung sein. Teil dessen ist, dass der Sounddesigner direkt in Unity arbeiten soll. Das aktuelle Soundsystem in der Unity Engine bietet allerdings für einen Sounddesigner kaum Anhaltspunkte. Das neue Tool soll das verbessern und dem Sounddesigner so die Möglichkeit geben mit dem Soundsystem zu arbeiten. Dafür muss vor allem die Benutzeroberfläche des Unity Editors besser an die Anforderungen eines Sounddesigners angepasst werden. So sollen auch einige Unity spezifische Schritte, die im Workflow des Sounddesigners keinen Sinn ergeben automatisiert werden.

Sound in Computerspielen kann verschiedene Funktionen erfüllen. Anhand des INFORM-Modells können Sounds gegliedert werden. Eine Gliederung der geplanten und vorhanden Sounds hilft Programmierern und Sounddesignern dabei, leichter den Überblick zu behalten. Umso leichter diese Gliederung zugänglich für alle ist und in den ganz normalen Workflow mit eingebunden werden kann, umso besser.

Je nach Anwendungsfall des Sounds, muss der Sounddesigner in der Lage sein, verschiedene Samples zu kombinieren. Und zwar nicht in seiner DAW, sondern im Tool, mit dem der Sound final eingebaut wird. Mit diesem prozeduralen Ansatz wird Speicherplatz gespart, während gleichzeitig eine größere Varianz im Ton entsteht, was dem Spieler einen realistischeren Eindruck gibt. Um effektiv Samples zu kombinieren, muss das Soundtool bestimmte Werkzeuge bieten. So sollen Samples beispielsweise in einem bestimmten Umfang zufällig gepitcht werden oder die Reihenfolge der Samples soll sich ändern. Viele von diesen spontanen Änderungen sind auch vom aktuellen Spielgeschehen beeinflusst.

Für die Erstellung von interaktivem Sound spielen also vor allem auch Parameter eine wichtige Rolle. Diese sollen zur Laufzeit die abgespielten Töne ans Spielgeschehen anpassen und dem Spieler so beispielsweise Feedback geben. Die Parameterlogik muss sowohl in den Skripten als auch im Soundtool erstellt werden. Dabei muss für den Programmierer klar sein, was der Parameter genau tut und wie er zu bedienen ist, um die Logik dementsprechend zu implementieren. Falls das nicht auf einfachem Wege möglich ist, muss der Programmierer Vorschläge geben können, wie eventuell dasselbe Ergebnis auf andere Weise erzielt werden kann.

Neben der Logik der Parameter muss auch die Kommunikation zwischen Soundtool und Skript möglichst leicht funktionieren. Das hilft nicht nur bei Parametern, sondern auch für ganz allgemeine Steuerung der Soundevents, wie das starten oder stoppen.

4.2 Anwendungsfälle von Sounddesign in Computerspielen

Wie in Kapitel 1 schon erwähnt spielt Sound in Computerspielen eine enorm wichtige Rolle. Im Vergleich zu anderen Medien kann (und wird) Sound hier allerdings sehr viel vielfältiger eingesetzt. In Computerspiele gibt es mehr als nur Musik, Atmo und Sprache. Jede Atmo muss zu jedem Musikstück passen, die Musik muss sich an den Spielinhalt anpassen und so weiter. Um ein Sound Tool zu entwickeln, mit dem Sounddesigner effektiv Sound für Computerspiele erstellen können, müssen zumindest die herkömmlichsten Anwendungsfälle beleuchtet und abgedeckt werden.

Einige von den Anwendungsfällen sind schon beispielhaft erwähnt worden. Auch der Workflow beim Spiel „Devolution“ hat einige dieser Anwendungsfälle beinhaltet. Für das neue Tool sollen aber so viele Anwendungsfälle wie möglich abgedeckt werden.

Die folgenden beschriebenen Features sind in keiner bestimmten Reihenfolge.

Soundliste: Mit einer Soundliste (vergl. FMod-Soundliste) soll stets im Überblick behalten werden, welche Sounds importiert und mit welchen Sounds dementsprechend gearbeitet werden kann. Um diese Liste übersichtlicher gestalten zu können sollen die importierten Sounds farblich kodiert werden können. Diese farbliche Kodierung kann auch dabei helfen Sounds nach bestimmten Gruppen zu gliedern. Beispielsweise, wenn mit dem INFORM-Modell gearbeitet wird. Außerdem soll die Soundliste nicht nur einen Überblick über alle Sounds bieten, sondern auch Parameter zu den jeweiligen Sounds anzeigen. Außerdem soll bei Sounds, die einem bestimmten Objekt zugeordnet sind, direkt zu diesem Objekt in der Szene gesprungen werden. Die Soundliste wird somit zum zentralen Teil des neuen Tools, da hauptsächlich damit die Kommunikation zwischen den Gewerken vereinfacht werden soll. Das erfordert auch die Möglichkeit neue Sounds in der Liste anzulegen, die noch nicht existieren. Später können diese dann einfach vom Sounddesigner ergänzt werden, während die Programmierer schon die Logik für diesen Sound erstellen konnten. Damit all diese Eigenschaften erfüllt werden können, muss in Unity ein Editor Skript erstellt werden. Die Funktionalität des Tools ist allerdings nicht davon abhängig. Für die Funktionalität ist es vielmehr entscheidend, wie importierte Sounds weiterverarbeitet werden können. Die Soundliste existiert in Unity also auch nur als Editor Fenster und hat keine Effekte oder Auswirkungen auf das Verhalten des Tons.

Sounds: Importierte Tondateien sollen Scriptable Objects zugewiesen werden. In Unity gibt es neben Objekten, die nur in der jeweiligen Spielszene existieren (was auch Objekte, wie eine unsichtbare Tonquelle sein kann), auch sogenannte Scriptable Objects, die unabhängig von der geladenen Szene aktiv sind. Diese können beispielsweise dafür genutzt werden, Variablen über eine Szene hinweg zu speichern. So können außerdem am effektivsten bestimmte Game States gespeichert werden. Scriptable Objects speichern außerdem alles, was während des Testens im Editor geändert wurde und

setzen sich nicht, wie normale Objekte wieder auf ihre Ausgangsposition zurück, wenn der Playmodus im Editor beendet wird. Der Vorteil von Scriptable Objects im Zusammenhang mit Sounddateien, sind grundsätzliche Funktionen, die direkt mit der Sounddatei in Verbindung stehen. So können hier Informationen über den Sound gespeichert werden. Beispielsweise, ob der Sound irgendwo verwendet wird. Gleichzeitig können zusätzliche Variablen erstellt werden, die dann über dasselbe Scriptable Object abrufbar sind. So zum Beispiel eine String-Variable, die als Kommentarzeile zum betreffenden Sound genutzt werden kann. Der Nachteil eines zusätzlichen Objektes ist ein leicht erhöhter Speicherplatzbedarf und ein minimal größerer Arbeitsaufwand, da nach dem Importieren der Datei, noch ein zusätzliches Objekt erstellt und benannt werden muss. Der erhöhte Speicherbedarf wird dadurch gerechtfertigt, dass die im Scriptable Object gesetzten Werte ansonsten anderswo erstellt und gesetzt werden müssen. Das macht diese Werte weniger leicht zugänglich und zuordnungsbar. Der zusätzliche Arbeitsschritt kann in einer späteren Version des neuen Tools automatisiert werden, ist aber für die Funktionalität zunächst nicht relevant. Dieser Arbeitsschritt kann aus mehreren Gründen durchaus allerdings auch sinnvoll sein. Wie in Kapitel 3.6 beschrieben, sollten Audiodateien je nach Größe und Häufigkeit auf verschiedene Weisen abgespielt werden. Mit einem zusätzlichen Arbeitsschritt nach dem Importieren, kann die richtige Einstellung beim Verknüpfen mit dem Scriptable Object getroffen werden. Passiert das automatisch, läuft man Gefahr, das zu vergessen. Gleichzeitig sollten Programmierer ein Sound Objekt erstellen können, ohne eine Sounddatei setzen zu müssen. Die von den Programmierern erstellten Objekte sind dann in der Soundliste als „noch nicht belegt“ markiert und können dann als angeforderte Sounds fungieren. Zum Testen können Programmierer genauso händisch einen anderen Testsound setzen.

Sound Events: Mit jedem Sound aus der Soundliste kann ein Event erstellt werden. Das Event beinhaltet mehrere Parameter, mit denen festgelegt werden soll, wann das Event anfängt und wie lange es läuft (Loop oder One-Shot). Außerdem können hier Effekte, wie Lautstärke, Modulation, Fade-In/Fade-Out und mehr hinzugefügt werden. Ein Event kann auch aus mehreren Sounds bestehen, die entweder nacheinander abgespielt werden oder von denen jeweils zufällig nur ein Sound wiedergegeben wird. Events können Gegenstände aus der Spielwelt zugeordnet werden, müssen aber in jedem Fall auf ein Objekt in der Szene gelegt werden, wie beispielsweise eine Audio Source. So kann zum Beispiel ein Sound-Event für alle Bäume erstellt werden und es muss nicht für jeden Baum dasselbe Event immer wieder kopiert werden. Ein Sound Event ist im Prinzip dafür da, einen bestimmten Sound entweder zu wiederholen oder nur einmal abzuspielen. Im Sinne des prozeduralen Sounddesigns können aber verschiedene Variationen desselben Sounds erzeugt werden. Im INFORM-Modell können Sound Events für alle Kategorien verwendet werden. Das Sound Event funktioniert in Kombination mit der Unity eigenen Audio Source Komponente. Zum einen wird diese benötigt, um Sounds in Unity abzuspielen, zum anderen kann über diese auch der Zugang zum Mixer geschaffen

werden. Mit dem Hinzufügen des Sound Event Skripts zu einem Objekt, wird automatisch auch eine Audio Source Komponente hinzugefügt. Aus den gegebenenfalls mehreren Sounds im Event, setzt das Sound Event Skript dann immer den aktuell abzuspielenden Sound mit den jeweiligen Einstellungen in die Audio Source Komponente ein und spielt diesen ab. Die Schwierigkeit für die Programmierung des Tools liegt bei diesem Ansatz daran, dass zwei Sounds aus demselben Sound Event nicht gleichzeitig abgespielt werden können, da eine Audio Source immer nur eine Datei auf einmal abspielen kann. Für diesen Fall müssen dann gegebenenfalls mehr Audio Source Komponenten auf demselben Objekt erzeugt werden.

Atmos: Atmos sollen vor allem dafür da sein, große Gebiete in der Spielwelt mit Hintergrundgeräuschen abzudecken. Atmos sind eine besondere Form der Sound Events. Anders als ein Sound Event besteht diese vielmehr aus einer Vielzahl von verschiedenen Geräuschen, die zu einem langen Loop immer wieder spielen. Einer Atmo können mehrere Events zugewiesen werden, die dann je nach Einstellung und Positionierung des Spielers zur Gesamtatmosphäre beitragen. Der Unterschied zwischen Events und Atmos besteht darin, dass Events normalerweise einem bestimmten Objekt zugeordnet sind (Ein Baum mit Vogelzwitschern, eine Tür, …) wohingegen Atmos einem ganzen Raum ein „Grundrauschen“ geben sollen. Auf diese Weise soll außerdem schneller sichtbar sein, wo bestimmte Events stattfinden. Atmos sind im INFORM-Modell meistens das O, in manchen Fällen auch das N. Es geht dabei um die Orientierung des Spielers in der Spielwelt. Dazu gehört auch ein grundsätzliches Gefühl für den Raum zu bekommen und die Stimmung des Spielers zu beeinflussen. Gleichzeitig kann die Atmo dem Spieler Hinweise darauf geben, wo bestimmte Objekte zum Interagieren zu finden sind. Beispielsweise sucht der Spieler in einem Survival-Spiel Wasser und kann deswegen auf Wasserplätschern achten. Um eine Atmo zu implementieren muss im besten Fall keine zusätzliche Programmierung stattfinden. Die Atmo mit all den Sound Events wird gestartet, wenn der Spieler einen bestimmten Bereich betritt. Es ist Aufgabe des Sounddesigners die verschiedenen Events in der Szene zu platzieren und diese entweder nur zu zufälligen Zeitpunkten abzuspielen oder diese sich ständig wiederholen zu lassen. In Unity funktionieren diese Trigger über sogenannte Collider. Das sind Elemente, die normalerweise dazu genutzt werden, die Kollision von physikalischen Objekten zu berechnen, man kann allerdings die Kollision ausschalten und die Collider ein Signal senden lassen, sobald der Spieler durch einen dieser Collider läuft.

Musik: Die Musik ist ähnlich wie die Atmo nicht an ein bestimmtes Objekt gebunden, sondern eher an einen bestimmten Bereich im Spiel. Im Gegensatz zur Atmo, die sowohl 2D als auch 3D Elemente hat, ist die Musik in den allermeisten Fällen in 2D. Es muss außerdem möglich sein, die Musik zu dämpfen, das Stück zu wechseln und einzelne Spuren hinzuzufügen beziehungsweise auszublenden. Die Schwierigkeit bei diesem Feature besteht darin, dass Musik viel mehr als andere Events/Atmos von

einem zeitlichen Aspekt abhängig sind. Ein Stück soll nur selten hart zu einem anderen Stück wechseln. Die Triggerpunkte, wann die Musik wechselt sind also anders zu bedienen. Außerdem wird Musik nicht ausschließlich abhängig von der Spielerposition abgespielt, sondern kann auch aus anderen Gründen wechseln. In Kapitel 1.3 wurden die verschiedenen Arten der interaktiven Musik beleuchtet. Die Algorithmic Form wird beim neuen Tool zunächst außer Acht gelassen, da diese sehr individuelle Programmierung benötigt und in den meisten Fällen (noch) nicht genutzt wird. Es bleiben also noch drei andere Formen der interaktiven Musik. Betrachtet man diese im Hinblick auf den Wechsel zwischen den verschiedenen Musikstücken, so lässt sich erkennen, dass die Ornamental Form und die Transitional Form die Musik immer nur zu einem bestimmten Zeitpunkt wechseln, während in der Parallelform alle Musikstücke gleichzeitig abspielt und bei Bedarf gefadet werden. Das neue Tool muss also neben dem Wechsel zum Ende von Musikstücken auch das parallele Abspielen von mehreren Tonspuren gewährleisten. Gleichzeitig sollen Wechsel zwischen Musikstücken niemals direkt wieder rückgängig gemacht werden. Wechselt ein Spieler zum Beispiel von einem Gebirgspass in eine Höhle, soll sich die Musik zwar ändern, aber wenn der Spieler wieder direkt umdreht, hätte sie sich eigentlich doch nicht ändern sollen. Für diesen Fall soll ein Toleranzwert in Sekunden angegeben werden können. Innerhalb dieser Zeit wechselt die Musik noch nicht, geht der Spieler zurück, setzt sich der Wert zurück und die Musik hat sich nicht geändert, geht der Spieler weiter, ändert sich die Musik ganz normal. Dieser Wert hat vor allem für die Parallelform eine Auswirkung, da in den anderen Fällen sowieso bis zum Ende einer musikalischen Phrase mit dem Wechsel gewartet wird. Für die Parallelform ist es außerdem sehr wichtig, die Kontrolle über mehrere Spuren zu haben. Dafür eignet sich der Unity Audio Mixer sehr gut. Jede Audio Source kann den Output über den Mixer ausgeben, mit dem verschiedene Snapshots für bestimmte Audiomixe eingestellt werden können. Für jede Spur muss allerdings auch eine eigene Audio Source zum Mixer verlinkt sein. Diese Verlinkung kann automatisch passieren. Allerdings kann mit einer händischen Verlinkung die Benennung der einzelnen Kanalzüge und das eventuelle Gruppieren von solchen besser organisiert werden. Um Musik zu implementieren braucht es also auf jeden Fall ein anderes Skript, als für Atmos und Events. Einstellungen, wie Pitch und Lautstärke können über den Mixer geregelt werden. Das Musikskript muss vielmehr „nur" zum richtigen Zeitpunkt zwischen verschiedenen Mixer Snapshots wechseln.

UI Sounds: UI Sounds sind im Grunde nichts anderes als Events für 2D-Objekte. Der räumliche Aspekt fällt weg. Ein weiterer Unterschied ist der potenzielle Trigger für den Sound. Beispielsweise sollen Sounds gespielt werden können, wenn die Maus über die Buttons im Menü bewegt wird. In Unity gibt es eine Button Komponente, die normalerweise für die Darstellung und Funktionalität des Buttons genutzt wird. Wird das UI Sound Event auf dasselbe Objekt, wie die betreffende Button Komponente gelegt (oder der Button in anderer Weise mit dem Sound Event verknüpft), kann abgefragt werden, wann sich die Maus über dem Button befindet und wann der Button geklickt wird. UI Sounds müssen

allerdings nicht nur für Buttons verwendet werden können, sondern jede Einblendung benötigt auch einen Sound. Für das neue Tool wäre es von Vorteil, wenn das normale Sound Event Skript auch UI Sounds abdecken kann, zumal sich diese in vielen Fällen Überlagern. Beziehungsweise normale Sound Events dienen gerade in Tutorials auch als Auslöser für UI Einblendungen. Im Gegensatz zu normalen Sound Events müssen UI Sounds allerdings selten über den Mixer ausgegeben werden.

Filter: Bestimmte Räume verändern den Klang von Events. Das Rauschen eines Flusses in einer Höhle klingt anders als, wenn der Fluss unter freiem Himmel fließt. Um nicht für jeden Raum neue Sounds zu erstellen und zu implementieren, können Filter auf Räume gelegt werden. Der Filter betrifft dann alle Atmos und Events, die abgespielt werden, während sich der Spieler innerhalb dieses Raumes befindet. Filter für bestimmte Regionen werden auch über den Mixer angelegt. Das erfordert, dass alle Sound Events auch über den Mixer ausgegeben werden, ansonsten haben Equalizer oder Reverb Effekte keinerlei Auswirkung auf die jeweiligen Events.

Fall-Off-Distance: Jedem Sound-Event ist ein innerer und ein äußerer Radius zugeordnet. Innerhalb des inneren Radius wird das Event mit voller Lautstärke abgespielt, alles innerhalb des äußeren Radius wird kontinuierlich leiser, bis schließlich außerhalb des äußeren Radius nichts mehr zu hören ist. Diese Radien werden im Editor angezeigt und können dort einfach von Hand vergrößert/verkleinert werden. Wie sich der Ton mit zunehmender Entfernung verhalten soll kann an der Audio Source Komponente eingestellt werden. Für die Parameter „Volume“, „Spatial“, „Spread“ und „Reverb“ kann mit Kurven das Verhalten über die Entfernung eingestellt werden. Für die Lautstärke (Volume) ergibt sich meistens eine lineare oder logarithmische Kurve, die gegen Null geht, damit bei zunehmender Entfernung der Ton ausgeblendet wird. Man könnte aber je nach Fall das Geräusch auch immer lauter werden lassen. Mit steigendem räumlichem Wert (Spatial), hat die 3D Sound Engine stärkeren Einfluss auf die Positionierung des Tons. Ein Spatial Wert von 0 ermöglicht keine räumliche Zuordnung. Gerade bei markanten Tönen, wie einem Wasserfall, soll keine Räumliche Zuordnung mehr möglich sein, wenn man direkt danebensteht. Allerdings ist jeder Ton, der auch nur teilweise 2D Anteile hat (also alle Werte unter 1) überall in der Szene zu hören. Mit dem Spread Wert wird festgelegt, an welcher Stelle der Ton im 3D Panorama abgespielt wird. Mit einem Wert von 0 ist der Ton direkt an der Stelle, an der er auch in der Welt platziert ist. 360 invertiert die Ausgabe. Bei 180 ist der Ton maximal verteilt auf das gesamte Panorama und entspricht damit eher einem 2D Monosignal. Mit dem Reverb Regler kann festgelegt werden, wie stark der Ton auf umliegende Reverb Zones ausgegeben werden soll. Reverb Zones verzerren den Ton, wenn sich der Audio Listener in einer solchen Zone befindet. Damit können beispielsweise einfach Innenräume mit dem Ton von außen beschallt werden, ohne dass im Mixer Änderungen vorgenommen werden müssen.

Trigger Zonen: Für Sound Events, Musik und Atmos können Trigger Zones dabei helfen, wann Töne abgespielt oder angehalten werden sollen. Gerade Atmos funktionieren ausschließlich nach diesem Prinzip. Betritt der Spieler einen bestimmten Bereich, fängt eine bestimmte Atmo an zu spielen, während eine andere Atmo ausgeblendet wird. Trigger Zonen sind aber auch für den Mixer wichtig. Am Beispiel der Reverb Zones aus dem vorigen Abschnitt lässt sich ein möglicher Anwendungsfall ableiten. Betritt ein Spieler eine bestimmte Trigger Zone, wird im Mixer ein bestimmter Snapshot aufgerufen, der einen bestimmten Raumklang besser wiedergeben kann. Allerdings muss sich das nicht auf Effekte beschränken. Mit Trigger Zonen für den Mixer können verschiedene Kanäle gefadet werden und so beispielsweise eine intensivere Musik starten. Trigger Zonen bieten außerdem den Vorteil, dass Sounddesigner eigenständig Sounds triggern können. Der Programmieraufwand ist für Atmos deshalb normalerweise gleich Null.

Kennzeichnung im Editor: Komplett ohne Wissen über die Unity Engine und übers Programmieren wird es natürlich nicht möglich sein alle möglichen Sounds einzubauen. Deshalb soll es die Möglichkeit geben im Unity Editor Kommentare zu bestimmten Objekten/Soundquellen zu schreiben. Das kann neben der Kommunikation zwischen den Gewerken auch zur Übersicht genutzt werden. Vorrangig sind dabei natürlich Kommentare in der Soundliste zu den verschiedenen Sounds und Parametern hilfreich. Teilweise müssen aber auch bestimmte Objekte, die in der Szene vorhanden sind, kommentiert werden. Unity bietet dafür nativ die Möglichkeit jedes Objekt mit einer farbigen Kennzeichnung zu versehen, auf der der Name des Objektes steht. Diese Kennzeichnung ist dann im Editor Fenster zu sehen. Allerdings können so keine Kommentare zum Objekt gemacht werden. Eine Möglichkeit so trotzdem Kommentare zu erstellen, ist ein Objekt zu erstellen, das in der Hierarchie unter dem zu kommentierenden Objekt liegt und dieses dann im Wortlaut des Kommentars zu benennen.

Sound zu Animationen: Sound Events müssen auch über Animationen aufgerufen werden können. Jedes Soundevent muss also so konzipiert sein, dass vom Unity-Animator ein Methoden-Aufruf zum Sound Event stattfinden kann. Im Unity-Animator kann ein Methodenname als String angegeben werden. Diese Methode wird dann auf einem Skript, das auf demselben Objekt, wie die Animation liegt, aufgerufen.

Einbindung ins Skript: Natürlich gibt es immer wieder Fälle, bei denen es leichter ist, bestimmte Funktionsweisen über ein Skript zu lösen. Sämtliche Effekte und Einstellungen auf einem Sound Event müssen also auch von einem anderen Skript aus bearbeitet werden können. Das Sound Event Skript (sowie Atmo und Musik Skripte), das mit dem neuen Tool benutzt werden soll, muss also dementsprechend vorbereitet sein.

4.3 Planung der Programmierung: Designansatz

Bei der Programmierung des neuen Tools ging es vor allem darum ein möglichst modulares System zu erschaffen. Auf diese Weise soll zum einen eine einfache Integration in jede mögliche Art von Spiel gewährleistet werden. Zum anderen soll das Tool auch möglichst erweiterbar bleiben. Wie im vorigen Unterkapitel beleuchtet, gibt es eine Menge Anwendungsfälle, die teilweise sehr speziell werden können. Das neue Tool soll es Programmierern also erleichtern diese speziellen Anwendungsfälle mit einzubinden. Gleichzeitig soll es Sounddesignern möglich sein (einfache) Sounds so einzubauen, dass der Programmierer nichts dafür tun muss, damit der Sound zur richtigen Zeit spielt. Das wiederrum erleichtert dem Sounddesigner das Testen und Verbessern seiner Sounds.

Das Tool beruht hauptsächlich auf der Soundliste. Diese soll die erste Anlaufstelle für Programmierer und Sounddesigner werden, wenn sich ein Überblick über den aktuellen Stand des Tons gemacht werden soll. Damit der Ton aber tatsächlich funktioniert, ist die Liste überflüssig. Es geht da „nur" um bessere Kommunikation.

Für die Funktionalität viel wichtiger sind die verschiedenen Skripte, mit denen tatsächlich Ton eingebaut wird. Diese Skripte müssen vor allem für den Sounddesigner intuitiv bedienbar sein. Für die drei Anwendungsfälle Atmo, Musik und Sound Event gibt es jeweils ein Skript. Diese Anwendungsfälle unterscheiden sich so grundlegend in ihrer Funktion und eventuellem zusätzlichen Programmieraufwand, dass es für den Sounddesigner und Programmierer gleichermaßen leichter ist, wenn es je ein eigenes Skript dafür gibt. Diese Skripte sollen aber trotzdem miteinander kommunizieren können. Alle beruhen schließlich auf einem Scriptable Object, dem Sound Object. Nach dem Importieren der Sounddateien, werden diese zu ihrem Sound Object hinzugefügt. Dort können Parameter und Kommentare hinzugefügt werden. Programmierer können auf diese Weise Sound Objects für angeforderte Sounds erstellen. Parameter wiederrum beschreiben Eigenschaften des bestimmten Sounds. Grundsätzlich können Parameter für Lautstärke und Pitch gesetzt werden. Außerdem einige spezifische Einstellungen zum Sound, beispielsweise wie stark ein zufälliger Pitch vom normalen Pitch abweichen soll. Werden keine Parameter gesetzt, werden die Sounds immer mit den Standardeinstellungen abgespielt.

Sounds können nun zu Sound Events, Atmos oder zur Musik hinzugefügt werden. Diese drei Skripte verarbeiten den Sound auf verschiedene Weise. Das Sound Event Skript bietet die Möglichkeit, Sounds direkt hintereinander abzuspielen oder aber einen zufälligen Sound abzuspielen. Zusätzlich kann die Lautstärke oder der Pitch der abgespielten Sounds zufällig gewählt werden. Sound Events haben eine Play-Funktion. Wird diese von einer Trigger Zone oder aus einem anderen Skript aufgerufen, spielt das Sound Event, bis es vorbei ist oder wieder gestoppt wird. Natürlich kann ein Soundevent auch in

ständiger Wiederholung abgespielt werden. Damit ist das Sound Event das Skript, mit dem grundsätzlich alle Sounds abgespielt werden können. Für eine Atmo braucht es aber zwangsläufig eine Trigger Zone, die festlegt, wo die Atmo zu hören ist. Außerdem finden sich in einer Atmo neben 2D Soundloops, auch Sound Events, die zu zufälligen Zeiten spielen sollen. Zu einer Atmo können also Neben den Sounds, die immer zu hören sein sollen, auch Sound Events hinzugefügt werden. Das Atmo Skript startet alle Sounds, sobald der Spieler in die jeweilige Trigger Zone tritt. Die Sound Events werden je nach Einstellung nicht gehört, da der Spieler zu weit weg steht. Ist ein Sound Event zu ende, startet das Atmo Skript das Event innerhalb eines zufälligen Zeitraums wieder.

Den größten Unterschied macht das Musik Skript. Mit dem neuen Tool sollen die drei herkömmlichen Formen der interaktiven Musik unterstützt werden können. Grundsätzlich ist jedes Musik Skript als ein Stück zu interpretieren. Innerhalb des Stücks können verschiedene Sounds hinzu- oder weggefadet werden. Oder gelegentliche Stinger abgespielt werden. Über Trigger Zonen und Input aus anderen Skripten, sollen so die einzelnen Spuren des Stückes je nach Spielsituationen angepasst werden können. Das funktioniert über Mixer Snapshots. Jede Spur eines Stückes muss mit dem Mixer verknüpft werden. Im Mixer können dann verschiedene Zustände des Stücks eingestellt und als Snapshot gespeichert werden. Das Tool kann dann einfach zwischen diesen Snapshots faden.

4.4 Problemstellungen zur Umsetzung

Dadurch, dass das Tool ohne Programmierkenntnisse bedient werden soll, muss die Benutzeroberfläche dementsprechend designt sein. Vor allem in Hinblick auf intuitive Bedienbarkeit. Gleichzeitig muss das Tool auch Programmierern erlauben, an möglichst vielen Stellen eigene Skripte einzubauen. Ohne Wissen über die Unity Engine wird das neue Tool allerdings nicht bedienbar sein. Vor allem das Verständnis dafür, wo die einzelnen Elemente zu finden sind (wie zum Beispiel der Mixer) muss vorhanden sein.

Für die Programmierung wird fast ausschließlich die Unity Sound API (Unity Technologies 2019) verwendet. Diese ist allerdings an einigen Stellen etwas umständlich und nicht besonders ausgereift. Das ist einer der Gründe, weshalb bei vielen Produktionen auf andere Tools zurückgegriffen wird. Für das neue Tool bedeutet das, dass diese Schwachstellen überbrückt werden müssen. So gibt es beispielsweise keine Möglichkeit den aktiven Snapshot eines Mixers abzufragen. Und auch der Wechsel zu einem anderen Snapshot funktioniert über eine Methode auf dem Snapshot selbst und nicht über den Mixer. Um zu einem bestimmten Snapshot zu wechseln, muss zunächst über den Mixer mit einem String danach gesucht werden und diesen dann (wenn der Snapshot gefunden wurde) aufrufen.

Der Ansatz, das Tool möglichst modular und flexibel zu gestalten erfordert die Kommunikation zwischen Scriptable Objects und normalen Game Objects. Da in einem Scriptable Object allerdings keine Instanzen von Game Objects serialisiert werden können, muss für die Kommunikation oft doch ein Skript zwischengeschalten werden. Besonders auffällig ist das bei den sogenannten Switches. Ein Switch ist ein Scriptable Object, in dem verschiedene Zielzustände gespeichert werden können. Soll beispielsweise der Mixer auf einen anderen Snapshot wechseln, kann ein Switch erstellt werden, in dem der Snapshot und das Musikskript referenziert werden. Derselbe Switch kann auch für einen Wechsel in der Atmo zuständig sein. So kann an einer Stelle im Level gebündelt alles verarbeitet werden, was verarbeitet werden soll. Der Switch selbst ist flexibel für Atmos, Sound Events und Musik einsetzbar. Wie aber schon angemerkt, kann der Switch keine Instanzen dieser Skripte referenzieren. Wird ein Switch getriggert, muss also auch eine Referenz zur jeweiligen Instanz mitgegeben werden.

Unabhängig vom technischen Hintergrund der Unity Engine, werden für jede Art von Sound (Atmo, Sound Event, Musik) bestimmte Voraussetzungen benötigt. Atmo und Sound Events sind sich sehr ähnlich, zumal zu einer Atmo verschiedene Sound Events gehören. Die Musik unterscheidet sich davon grundlegend, da die einzelnen Spuren zeitlich aufeinander abgestimmt sein müssen. Die Unity Engine bietet für den Sound allerdings kaum Unterstützung für das exakt getimte abspielen von Sounds. Mit der PlayScheduled()-Funktion können zwar Sounds zu exakten Momenten abgespielt werden (Unity Technologies 2019), allerdings ist diese Funktion nur dafür gut, Sounds nahtlos hintereinander abzuspielen. Soll ein musikalischer Einwurf beispielsweise an bestimmten Stellen im Musikstück auftauchen, kann diese exakte Stelle in Unity nicht einfach ermittelt werden und dann an diese Stelle geschoben werden. Dafür mangelt es an einer grafischen Oberfläche zum Arrangieren von mehreren Sounds. Mit dem neuen Tool soll das zumindest stückweise behoben werden.

4.5 Umsetzung der Features im Code

Das Tool besteht aus mehreren Komponenten, die alle auf dem Sound Objekt beruhen. Ein Sound Objekt ist im Grunde nichts anderes, als der importierte Sound mit Einstellungsmöglichkeiten. Die Sound Komponente muss dementsprechend auch mit allen möglichen Verarbeitungswegen funktionieren. Das Sound Objekt ist außerdem ein Scriptable Object. Dadurch ist es als Asset zu verstehen und nicht als Game Object. Das heißt es muss kein Objekt innerhalb einer Szene existieren, damit der Sound funktioniert. Außerdem können so Sounds über Szenen hinweg mehrfach verwendet werden, ohne dass jedes Mal die Einstellungen neu getroffen werden müssen. Da für jeden Sound, der gleichzeitig spielt je eine Audio Source in der Szene existieren muss, kann es vorkommen, dass aus Performance Gründen ein Sound im Laufe des Spiels auf verschiedenen Audio Sources abgespielt wird.

Einstellungen auf dem Sound werden dann einfach auf die Audio Source übertragen. Das bedeutet, dass wenn der Sound abgespielt wird, eine Audio Source übergeben werden muss, auf der der Sound abgespielt werden soll. Diese Audio Source wird dann den Einstellungen entsprechend angepasst.

```
public void Play(AudioSource source)
{
    source.volume = _volume;
    source.loop = _loop;
    if (_3Dsound)
    {
        source.minDistance = _minDistance;
        source.maxDistance = _maxDistance;
        switch (_volumeRoloff)
        {
            case VolumeRoloff.Logarithmic:
                source.rolloffMode = AudioRolloffMode.Logarithmic;
                break;
            case VolumeRoloff.Linear:
                source.rolloffMode = AudioRolloffMode.Linear;
                break;
            case VolumeRoloff.Custom:
                source.rolloffMode = AudioRolloffMode.Custom;
                source.SetCustomCurve(AudioSourceCurveType.CustomRolloff,
                _customVolumeRoloffCurve);
                break;
        }
        source.SetCustomCurve(AudioSourceCurveType.SpatialBlend,
        _customSpatialRoloffCurve);
        source.SetCustomCurve(AudioSourceCurveType.Spread, _customSpreadRoloffCurve);
        source.SetCustomCurve(AudioSourceCurveType.ReverbZoneMix,
        _customReverbRoloffCurve);
    }
    if (_randomizePitch)
        source.pitch = Random.Range(Mathf.Clamp(_randomPitchMin, -3f, 3f),
        Mathf.Clamp(_randomPitchMax, -3f, 3f));
    else
        source.pitch = _pitch;
    source.clip = _audioClip;
    if (_output != null)
        source.outputAudioMixerGroup = _output;
    else
        source.outputAudioMixerGroup = null;
    source.Play();
}
```

Wie hier zu sehen ist können verschiedene Eigenschaften eingestellt werden. Zunächst einfach die Lautstärke. Diese Einstellung entspricht auf dem Mischpult eher dem Gain. Falls die Audiodatei im Vergleich zu anderen insgesamt zu laut ist, kann mit dieser Einstellung reguliert werden, ansonsten gibt es über den Mixer leichter die Möglichkeit je nach Fall die Lautstärke zu ändern.

Für jeden Sound kann außerdem eingestellt werden, ob dieser im Loop laufen soll oder nicht. Der Sounddesigner sollte das für jeden Sound wissen können. Manche Skripte beeinflussen allerdings das Loop-Verhalten, sodass ein Sound, der auf Loop gestellt wurde, sich am Ende gar nicht direkt wiederholt. Diese Funktion klingt zunächst nicht besonders intuitiv, sollte sich aber von selbst erklären, wenn später die betreffenden Stellen im Skript aufgezeigt werden.

Soll der Sound als 3D Sound fungieren, gibt es einige Einstellungen, die exklusiv dafür benötigt werden. Konkret das Verhalten des Sounds mit zunehmender Entfernung vom Spieler. Für die Lautstärke, den Reverb, den Spread und die räumliche Zuordnung können eigene Kurven gesetzt werden. Die Lautstärke bietet dabei auch zwei vorgefertigte Kurven. Eine mit linearem und eine mit logarithmischem Fall-Off. Für die andern drei Werte gilt, wird nichts eingestellt, werden die Standardwerte verwendet.

Für den Pitch des Sounds kann entweder bei jedem abspielen ein zufälliger Pitch gewählt oder der Standardwert benutzt werden. Soll der Pitch zufällig bestimmt werden, wird ein zufälliger Wert zwischen einem vom Nutzer gesetzten Minimum und Maximum gewählt. Dieser Wert fällt allerdings niemals unter -3 oder übersteigt 3. Diese Werte sind die Unity internen Grenzwerte für den Pitch.

Ist dem Sound ein Mixer Ausgang zugeordnet, wird dieser beim Abspielen mit der jeweiligen Audio Source verknüpft. Es müssen keine Ausgänge auf dem Mixer zugewiesen werden, allerdings ist es für die allermeisten Sounds sinnvoll. Beim Musikskript ist es sogar dringend notwendig, den Mixer zu nutzen und damit sollten Sounds für die Musik mit einem Mixerausgang versehen werden.

Nachdem alle Einstellungen auf die Audio Source übertragen wurden, wird der Sound abgespielt. Für Sounds, die zu einem bestimmten Zeitpunkt abgespielt werden sollen, gibt es eine überladene Play-Methode, bei der ein konkreter Zeitpunkt als Double-Wert übergeben werden muss. Mit PlayScheduled wird der Sound dann zu exakt diesem Zeitpunkt wiedergegeben.

```
public void Play(AudioSource source, double time)
{
    […]
    source.PlayScheduled(time);
}
```

Ist der Sound angelegt, kann er auf verschiedene Weisen weiterverarbeitet werden. In den drei Skripten Atmo, Sound Event und Music werden die eingefügten Sounds für den jeweiligen Zweck benutzt. Ein Sound Event beschreibt hierbei ein aus mehreren Sounds zusammengesetztes Event, bei dem zufällige Sounds hintereinander abgespielt werden. Dabei kann entweder aus nur einer Auswahl an vielen Sounds immer einer ausgewählt werden oder aus mehreren Auswahlen hintereinander. Diese Auswahl an Sounds nennt sich Sound Group und ist ein essenzieller Teil des interaktiven Sounddesigns. Auf diese Weise können zum Beispiel Schrittgeräusche prozedural generiert werden, wie in Kapitel 1.4 erwähnt. Die Sound Group dient also dazu mehrere Sounds zu bündeln und beim Abspielen, einen dieser Sounds auszuwählen. Die Sound Group kann allerdings auch als Pause genutzt werden. So kann in einem Sound Event auch eine organische Lücke zwischen verschiedenen Sounds entstehen.

```
public void Play(AudioSource source)
{
    _r = Random.Range(0, _sounds.Length);
    if (_sounds.Length != 0 && _sounds[_r] != null)
    {
        _sounds[_r].Play(source);
    }
    else
    {
        if (!_useSetTime)
        {
             _silenceTime = Random.Range(Mathf.Abs(_minSilenceTime),
             Mathf.Abs(_maxSilenceTime));
        }
    }
}
```

Wird auf einer Sound Group die Play Methode aufgerufen, muss auch hier die entsprechende Audio Source übergeben werden. Für einen getimten Aufruf, steht auch hier eine überladene Play Methode mit Zeitpunkt zur Verfügung. Zunächst wird ein zufälliger Integer-Wert bestimmt. Dieser liegt zwischen 0 und der Anzahl der Sounds, die der Sound Group angehören. Die Random.Range-Methode kann bei der Übergabe von Integer-Werten alle Zahlen inklusive des Minimums und exklusive des Maximums zurückgeben. Deshalb muss von _sounds.Length nicht 1 abgezogen werden. Ist an der Stelle im Array der verknüpften Sounds eine Lücke oder ist der Array komplett leer, wird nichts abgespielt, sondern der Sound stattdessen als Pause interpretiert. Die Pause kann entweder jedes Mal dieselbe Länge haben oder zwischen einem angegebenen Minimum und Maximum variieren. Da die Pausenlänge mit Float-Werten angegeben wird, funktioniert die Random.Range-Methode hier anders. Es können alle Zahlen inklusive des Minimums und des Maximums zurückgegeben werden.

```
public Sound CurrentSound
{
    get
    {
        if (_sounds.Length != 0 && _sounds[_r] != null)
            return _sounds[_r];
        else
            return null;
    }
}
public float CurrentSoundLength
{
    get
    {
        if (_sounds.Length != 0 && _sounds[_r] != null)
            return _sounds[_r].AudioClip.length;
        else
            return _silenceTime;
    }
}
```

Von einer Sound Group kann jederzeit der aktuell beziehungsweise zuletzt abgespielte Sound, sowie dessen Länge erfragt werden. Wird der aktuelle Sound als Pause interpretiert, gibt die Anfrage nach der Sound Länge die Länge der Pause zurück.

Die erstellten Sound Groups können dann in einem Sound Event verwendet und miteinander verknüpft werden.

```
public void Play()
{
    if (_useSingleSound)
    {
        if (!_playing)
        {
            _eventSound.Play(_source1);
            if(_loop)
                _source1.loop = true;
        }
    }
    else
    {
        if (!_playing)
        {
            _source2.loop = false;
            _referenceTime = AudioSettings.dspTime;
            _eventSounds[_currentEventSound].Play(_source1, _referenceTime);
            _sourceUsed = true;
            _nextEventSound = _currentEventSound + 1;
            if (_nextEventSound >= _eventSounds.Length && _loop)
            {
                _nextEventSound = 0;
                ScheduleNextSound();
            }
            else if(_nextEventSound >= _eventSounds.Length && !_loop)
            {
                _source1.loop = false;
                _source2.loop = false;
            }
            else
            {
                ScheduleNextSound();
            }
        }
    }
    if(_eventSound != null && !_useSingleSound)
    {
        _backgroundLoop = true;
        _source3 = gameObject.AddComponent<AudioSource>();
        _source3.loop = true;
        _eventSound.Play(_source3);
    }
    _playing = true;
}
```

Wird ein Sound Event abgespielt, wird entweder ein einfacher Sound gestartet oder aus der ersten Sound Group ein Sound ausgewählt, der abgespielt werden soll. Der einfache Sound spielt entweder einmal oder wird ständig wiederholt. Diese Funktion dient vor allem für Signale an Spieler, die jedes

Mal gleich klingen sollen und die zu einer bestimmten Aktion im Spiel abgespielt werden sollen. Auch Töne für UI-Elemente können damit bedient werden. Wird das Sound Event eher für Atmo und im prozeduralen Sinn verwendet, werden Sound Groups verwendet. Wenn das Sound Event wiederholt werden soll, wird aus jeder Sound Group immer ein Sound (oder eine Pause) ausgewählt und nahtlos hintereinander abgespielt. Ist aus der letzten Sound Group ein Ton ausgewählt, wird danach wieder ein Ton aus der ersten Sound Group abgespielt. Natürlich kann auch nur eine einzelne Sound Group verwendet werden. Da wird dann ständig ein Ton aus derselben Gruppe gewählt und direkt hintereinander abgespielt. Soll das Sound Event nicht wiederholt werden, endet das Event sobald aus der letzten Sound Group ein Element abgespielt wurde.

Um Sounds nahtlos aneinander zu knüpfen, muss die PlayScheduled-Funktion genutzt werden. Diese erfordert eine Zeitangabe als Double-Wert. Um diesen konkreten Zeitpunkt festzustellen, wird die DSP Zeit genutzt. Damit wird angegeben, wie viele Audio Samples unabhängig von der Framerate verarbeitet werden. Dadurch ist die DSP Zeit sehr akkurat. Somit ist die PlayScheduled-Funktion auch sehr viel akkurater als die vergleichbare PlayDelayed-Funktion, bei der nur ein Float-Wert basierend auf der Spielzeit übergeben wird. Die Spielzeit berechnet sich anhand der Frames und ändert sich innerhalb eines Frames nicht.

```
private void ScheduleNextSound()
{
    if (_eventSounds[_currentEventSound].CurrentSound != null)
        _timeTillNextClip =
              (double)_eventSounds[_currentEventSound].CurrentSound.AudioClip.samples /
              _eventSounds[_currentEventSound].CurrentSound.AudioClip.frequency;
    else
        _timeTillNextClip = _eventSounds[_currentEventSound].CurrentSoundLength;
    _referenceTime += _timeTillNextClip;
    if (_sourceUsed)
    {
        _eventSounds[_nextEventSound].Play(_source2, _referenceTime);
        if(_eventSounds[_nextEventSound].CurrentSound != null)
            _sourceUsed = false;
    }
    else
    {
        _eventSounds[_nextEventSound].Play(_source1, _referenceTime);
        if (_eventSounds[_nextEventSound].CurrentSound != null)
            _sourceUsed = true;
    }
    StartCoroutine(WaitAClip());
}
```

Um den nächsten Sound zum exakt richtigen Zeitpunkt abspielen zu können, muss die Länge des aktuellen Sounds genauso exakt berechnet werden. Deshalb wird, wenn der nächste Sound keine Pause ist, die Anzahl der Samples durch die Frequenz geteilt (John Leonard French 2018). Im Gegensatz zu AudioClip.Length, was einen Float-Wert zurückgibt, wird so auch wie bei der DSP Zeit auf Basis der Samples gerechnet und ein viel genaueres Ergebnis erziehlt. Ist der aktuelle Sound eine Pause, muss

die Länge der Pause nur abgefragt werden. Die Länge des aktuellen Sounds (oder der aktuellen Pause), wird auf die Startzeit des aktuellen Sounds addiert. So wird der Zeitpunkt für den Start des nächsten Sounds errechnet.

Da auf einer Audio Source immer nur ein Sound abgespielt werden kann, muss beim Aufruf des nächsten Sounds auch eine andere Audio Source verwendet werden. In einem Soundevent spielt allerdings maximal ein Sound auf einmal, das heißt, es kann ständig zwischen zwei Audio Sources hin- und hergewechselt werden. Der Wechsel darf bei einer Pause allerdings nicht stattfinden, da diese keine Audio Source benötigt und ansonsten der Sound auf der anderen Audio Source unterbrochen wird, sobald der nächste Sound auf dieselbe Audio Source ausgegeben werden soll. Nachdem der nächste Sound eingeplant ist, muss für die länge des aktuellen Sounds gewartet werden, bevor auf dieselbe Audio Source der übernächste Sound geplant werden kann.

```
private IEnumerator WaitAClip()
{
    yield return new
        WaitForSeconds(_eventSounds[_currentEventSound].CurrentSoundLength + .1f);
    _currentEventSound = _nextEventSound;
    if (++_nextEventSound >= _eventSounds.Length && _loop)
    {
        _nextEventSound = 0;
        ScheduleNextSound();
    }
    else if(_nextEventSound >= _eventSounds.Length && !_loop)
    {
        _source1.loop = false;
        _source2.loop = false;
    }
    else
    {
        ScheduleNextSound();
    }
}
```

Mit Hilfe der IEnumerator-Klasse kann das Sound Event für eine bestimmte Zeit pausiert werden. Bevor der übernächste Sound eingeplant wird, muss allerdings wie in der Play-Funktion überprüft werden, ob das Sound Event wiederholt werden soll, falls aus allen Sound Groups ein Sound abgespielt wurde.

Ein Sound Event kann über andere Skripte gestartet werden oder über Trigger, die der Sounddesigner in der Szene platzieren kann. Sobald der Trigger betreten wird, startet das Sound Event und sobald der Trigger wieder verlassen wird, stoppt das Event. Um zu sicher zu gehen, dass das Sound Event nur vom Spieler ausgelöst werden kann, wird der Tag des Objektes, das den Trigger berührt, mit einem gesetzten Tag verglichen. Entsprechen sich die Tags, wird das Sound Event ausgelöst. Es besteht allerdings auch die Möglichkeit die Trigger-Kollision zu ignorieren.

```
private void OnTriggerEnter(Collider other)
    {
        if(other.tag == _playerTag && _useCollision)
            Play();
    }
```

Um eine Atmo für die Szene zu bauen werden für gewöhnlich mehrere Sound Events benötigt. Alle Gegenstände in der Spielwelt, die Geräusche von sich geben haben zumindest ein Sound Event. Schon allein wegen der Performance ergibt es aber keinen Sinn immer alle Sound Events laufen zu lassen, wenn der Spieler nicht in der Nähe ist und diese sowieso nicht hören kann. Mit dem Atmo Skript können sämtliche zur Atmo gehörende Sound Events gestartet werden. Das Atmo Skript funktioniert ausschließlich über Trigger. So können große Bereiche des Levels mit einer Atmo abgedeckt werden, ohne dass zusätzlicher Programmieraufwand notwendig ist.

```
public void Start2DLoop(bool fade)
{
    _atmoPlaying = true;
    if (!_single2Dsound && _randomizeAtmoLoop)
        _r = Random.Range(0, _2Dsounds.Length);
    _2Dsource.clip = _2Dsounds[_r].AudioClip;
    if (fade)
    {
        _2Dsource.volume = 0;
        _2Dsource.Play();
        StartCoroutine(Fade2DSound(1));
    }
    else
    {
        _2Dsource.Play();
    }
    if (!_single2Dsound)
        StartCoroutine(WaitForNext2DSound());
}
```

Zusätzlich zur Kontrolle über beliebig viele Sound Events kann das Atmo Skript einen oder mehrere Sounds als 2D Sounds loopen lassen. Diese dienen dazu die allgemeine Stimmung in der aktuellen Szene zu setzen. Unabhängig von der Position des Spielers werden diese immer zu hören sein. Die 2D Sounds können entweder immer zufällig ausgewählt werden oder in einer bestimmten Reihenfolge hintereinander wiedergegeben werden, um dem Sounddesigner möglichst viele Anwendungsfälle für dieses Skript zu geben. Zudem kann zwischen den einzelnen 2D Sounds eine Blende eingebaut werden. Ist ein 2D Sound am Ende, blendet er aus, während der nächste Sound einblendet.

```
private IEnumerator Fade2DSound(float to)
{
    float startValue = _2Dsource.volume;
    if(to > _2Dsource.volume)
    {
        while(_2Dsource.volume < to)
        {
            yield return null;
            _2Dsource.volume += Time.deltaTime * Mathf.Abs(to - startValue);
        }
        _2Dsource.volume = to;
    }
    […]
}
```

Auch hierfür wird die IEnumerator-Klasse benutzt. Mit dem „yield return null“-Befehl wird einen Frame gewartet, bevor das Skript weiter ausgeführt wird. So wird Frame für Frame die Lautstärke der Audio Source stückweise erhöht. „Time.deltaTime“ gibt die Zeit in Sekunden seit dem letzten Frame an. Damit kann effektiv die Länge eines Frames gemessen werden. Da die Lautstärke einer Audio Source nur zwischen 0 und 1 festgelegt werden kann, würde das Einblenden des Sounds immer so viele Sekunden benötigen, wie zwischen dem Start- und dem Endwert liegt. Damit jede Einblendung immer eine Sekunde lang ist, wird Time.deltaTime mit der Differenz multipliziert. Ist diese kleiner als 1, wird dementsprechend immer nur ein Bruchteil der Zeit seit dem letzten Frame zur Lautstärke hinzugefügt. Unabhängig von der Framerate ist der Sound so immer nach einer Sekunde bei der Ziellautstärke. Da die Lautstärke des Sounds auf diese Weise theoretisch im letzten Durchlauf der While-Schliefe leicht über den Zielwert gesetzt werden kann, wird er final nochmals auf die korrekte Lautstärke korrigiert. Diese Korrektur findet allerdings innerhalb eines Frames statt, womit sie für den Menschen quasi nicht wahrnehmbar ist.

Als drittes Skript zu Atmo und Sound Event gibt es das Musik Skript. Interaktive Musik funktioniert anders als Sound Events, da Phrasen aufeinander abgestimmt sein müssen und nicht jederzeit zwischen Stücken gewechselt werden kann. Um harte Wechsel zu vermeiden, wird oft auf die Parallelorm zurückgegriffen. Die Idee hinter dem Musik Skript ist ein Musikstück in seine einzelnen Spuren aufzuteilen, um diese dann bei Gelegenheit ein- und auszublenden.

```
public void Play()
{
    _currentSnapshot = _defaultSnapshot;
    _defaultSnapshot.TransitionTo(0);
    for (int i = 0; i < _tracks.Length; i++)
    {
        if (_playCounter == 0)
        {
            _audioSources.Add(gameObject.AddComponent<AudioSource>());
            _audioSources.Last().loop = true;
            _audioSources.Last().spatialBlend = 0;
            _tracks[i].Play(_audioSources.Last());
        }
        else
        {
            _tracks[i].Play(_audioSources.ElementAt(i));
        }
    }
    _playCounter++;
}
```

Das Ein- und Ausblenden soll über Snapshots im Mixer funktionieren. Wichtig dafür ist, dass alle Sounds, die zum Skript hinzugefügt wurden, einen Ausgang auf dem Mixer haben. Grundsätzlich werden alle Spuren (im Skript als „tracks“ bezeichnet) von Anfang an abgespielt und geloopt. Welche Tracks wie laut gehört werden, hängt von den Einstellungen im Mixer ab. Beim erstmaligen Start des Musik Skriptes werden für alle Sounds jeweils eine Audio Source erstellt. Da immer nur ein Sound pro Audio Source abgespielt werden kann und im Voraus nicht klar ist, wann welche Sounds spielen, ist es notwendig eine Audio Source pro Spur zu erstellen. Alle Audio Sources werden außerdem auf einen Spatial Blend von 0 gesetzt, damit die Musik unabhängig von der Position des Spielers zu hören ist.

```
public void SwitchSnapshot(int[] trackRestarts, AudioMixerSnapshot snapshot)
{
    if(_currentSnapshot != snapshot)
    {
        for (int i = 0; i < trackRestarts.Length; i++)
        {
            Restart(trackRestarts[i]);
        }
        _currentSnapshot = snapshot;
        snapshot.TransitionTo(_snapshotTransitionTime);
    }
}
```

Das Musik Skript funktioniert hauptsächlich durch den Wechsel von Snapshots. Snapshots können vom Sounddesigner sehr leicht angelegt und bedient werden. Die Musik kann so sogar schon bevor tatsächliche Dateien zum Abspielen existieren, vorbereitet werden. Wird der Snapshot eines Musikstücks gewechselt können außerdem ausgewählte Spuren neu gestartet werden. Das ist vor allem für Stinger und bei Übergängen hilfreich. Auf diese Weise können auch nicht loopende Elemente in die Musik eingebaut werden, die bei Gelegenheit abgespielt werden.

Nicht nur das Musik Skript, sondern auch Atmos und Sound Events können verschiedene Snapshots haben, solange die Sounds mit dem Mixer verknüpft sind. Diese können beispielsweise dafür benutzt werden Sounds zu dämpfen, wenn der Spieler ein Gebäude betritt. Um einfach Trigger für diese und andere Fälle erstellen zu können, existiert neben den Sounds und Sound Groups ein drittes Scriptable Object mit dem Namen Switch. Switches können dafür benutzt werden, Zielzustände zu speichern. Auf diese Weise kann vom Sounddesigner genau festgelegt werden, was bei bestimmten Aktionen passieren soll. Von Programmierseite muss beim jeweiligen Aufruf dann nur der jeweilige Switch referenziert werden. So kann ein Switch auch mehrmals benutzt werden. Zum Beispiel wenn der Spieler auf verschiedene Weisen ins Hauptmenü gelangen kann. Der Sounddesigner muss nur einmal festlegen, was passieren soll, wenn ins Hauptmenü gewechselt wird und der Programmierer kann den Switch dann bei jeder Gelegenheit aufrufen. Natürlich können im Switch nicht alle möglichen Anwendungsfälle festgelegt werden, da jedes Spiel spezielle Anforderungen hat. Der Switch dient hauptsächlich zum Wechseln von Snapshots und dem Starten und Stoppen von Sound Events oder Musikstücken. Mit diesen grundlegenden Funktionen kann der Switch sehr vielfältig eingesetzt werden.

```
public void Trigger(Music m, SoundEvent s, Atmo a)
{
    if(m != null)
    {
        if (_tracksToRestart.Length == 0)
        {
            m.SwitchSnapshot(_musicSnapshot);
        }
        else
        {
            m.SwitchSnapshot(_tracksToRestart, _musicSnapshot);
        }
    }
    if(s != null)
    {
        if (_restartSoundEvent)
        {
            s.Stop();
            s.Play();
        }
        if (_eventSnapshot != null)
        {
            _eventSnapshot.TransitionTo(_transitionTimeEvent);
        }
    }
    if(a != null)
    {
        if (_2DVolume >= 0)
            a.Volume2D = _2DVolume;
        if (_snapshotName != "")
            a.Switch3DSnapshots(_snapshotName, _transitionTimeAtmo);
    }
}
```

Da ein Scriptable Object keine Instanzen von Game Objects referenzieren kann, muss das betroffene Skript übergeben werden, sobald der Switch aufgerufen wird. Die Trigger Methode ist dementsprechend für jede Kombination aus Musik, Sound Event und Atmo überladen, falls nur der Snapshot der Musik gewechselt werden soll.

Um die Musik zu starten, kann auf den Start-Snapshot des Stücks gewechselt und sämtliche Tracks neu gestartet werden. Zum Beenden, kann auf den End-Snapshot gewechselt werden. Gleichzeitig kann der Switch auch für jeden andern Snapshot-Wechsel benutzt werden.

Um das Sound Event zu starten, kann es einfach neu gestartet werden. Auf diese Weise wird auch gewährleistet, dass ein Sound Event, das aktuell läuft, aber gegebenenfalls nicht hörbar ist, wieder von Beginn an spielt. Gleichzeitig kann auch ein Snapshot geändert werden. Das wird bei Sound Events vor allem wichtig sein, wenn Effekte, wie ein Equalizer oder Reverb hinzugefügt oder entfernt werden sollen.

Die Atmo soll über einen Switch nicht gestoppt oder gestartet werden, da das über den Trigger auf dem Atmos Skript selbst geschieht. Allerdings kann die Lautstärke des 2D Sound Loops geändert, also auch auf 0 gesetzt werden. Wird der Wert gesetzt blendet die Lautstärke über die Dauer von einer Sekunde auf den Zielwert. Der Zielwert kann nicht unter 0 sinken und nicht über 1 steigen.

```
public float Volume2D
{
    get { return _2Dsource.volume; }
    set { StartCoroutine(Fade2DSound(Mathf.Clamp01(value))); }
}
```

Neben der 2D Lautstärke können auch Snapshots auf sämtlichen Sound Events, die zur Atmo gehören geändert werden. Solange das Sound Event einen Snapshot mit dem Angegebenen Namen im Mixer hat, wird zu diesem Snapshot geblendet. Diese Funktion ist sehr hilfreich, wenn der Spieler ein Gebäude betritt und alle Sound Events um das Gebäude herum gedämpft werden sollen. Mit dieser Funktion müssen nicht Switches für alle Sound Events einzelnen aufgesetzt werden.

Neben den Switches gibt es das TriggerSwitch-Skript. Das kann dafür verwendet werden eigene Trigger in der Szene zu platzieren, die dann den verlinkten Switch aufrufen.

```
private void OnTriggerEnter(Collider other)
{
    if(other.tag == _playerTag && _enterSwitch != null)
    {
        _exiting = false;
        if(!(_singleUse && _delayUsed))
        {
            if (!_enterSwitch.TriggerDelay && _enterSwitch.DelayTime <= 0)
                _enterSwitch.Trigger(_enterMusic, _enterEvent, _enterAtmo);
            if (_enterSwitch.DelayTime > 0)
                StartCoroutine(WaitDelay(_enterSwitch));
        }
    }
}

private void OnTriggerExit(Collider other)
{
    if(other.tag == _playerTag && !(_singleUse && _delayUsed))
    {
        _exiting = true;
        if(_enterSwitch != null && _enterSwitch.TriggerDelay)
        {
            if (_enterSwitch.DelayTime <= 0)
                _enterSwitch.Trigger(_enterMusic, _enterEvent, _enterAtmo);
        }
        if(_exitSwitch != null)
        {
            if (_exitSwitch.DelayTime <= 0)
                _exitSwitch.Trigger(_exitMusic, _exitEvent, _exitAtmo);
            else
                StartCoroutine(WaitDelay(_exitSwitch));
        }
        if (!_delayUsed && _singleUse)
            this.enable = false;
    }
}
```

Der Trigger soll genau wie die Switches möglichst vielseitig eingesetzt werden können. So können sowohl Switches aufgerufen werden, wenn der Spieler den Trigger betritt als auch wenn der Spieler den Trigger verlässt. Die beiden Switches können völlig unabhängig voneinander agieren. Wichtig ist, dass zum Switch die richtigen Skripte verlinkt werden. Benötigt der Switch, der beim betreten des Triggers aufgerufen werden soll, beispielsweise nur ein Musik Skript, reicht es dieses zu verlinken. Eine Atmo und ein Sound Event müssen dann nicht verlinkt werden.

Manchmal ist es wichtig, dass Switches nicht direkt aufgerufen werden, sondern erst verzögert. So kann der zweite Switch genau mit dem ersten Switch synchronisiert werden. Dafür muss ein sehr schmaler Trigger platziert werden, bei dem beide Switches quasi gleichzeitig aufgerufen werden. Über den Delay auf dem zweiten Switch, kann dann genau bestimmt werden, wie viel später dieser aufgerufen werden soll.

Dadurch, dass der Trigger auch beliebig größer skaliert werden kann, kann die Zeit, in der sich der Spieler innerhalb des Triggers aufhält, auch als Übergangsphase für ein Musikstück genutzt werden. So leitet der erste Switch das Ende eines Stückes ein, der zweite Switch startet dann das nächste Stück.

So lange sich der Spieler im Trigger aufhält läuft gegebenenfalls noch das Ende des ersten Stückes. Diese Funktion soll genauso wie der zeitliche Delay kreativen Spielraum bieten.

Da Trigger manchmal nur einmal verwendet werden sollen, um zu verhindern, dass Spieler auf dem Rückweg dieselben Effekte nochmals aufrufen, kann das Skript automatisch deaktiviert werden, sobald es das erste Mal benutzt wurde.

```
private IEnumerator WaitDelay(Switch s)
{
    if (_exiting && _singleUse)
        _delayUsed = true;
    yield return new WaitForSeconds(s.DelayTime);
    s.Trigger(_enterMusic, _enterEvent, _enterAtmo);
    if (_singleUse)
    {
        this.enable = false;
        _delayUsed = false;
    }
}
```

Dabei muss auch beachtet werden, dass der zweite Switch gegebenenfalls erst zeitverzögert nach dem Verlassen des Triggers abgespielt wird. Falls das der Fall ist und der Trigger nur einmal benutzt werden soll, wird für die Zeit bis der verzögerte Switch aufgerufen wird, jegliche Interaktion mit dem Trigger ignoriert. Soll der Trigger nur einmal benutzt werden, wird das Skript danach deaktiviert. Es kann dann nur von anderen Skripten wieder aktiviert werden. Das ist allerdings ein Anwendungsfall, der vom jeweiligen Spiel abhängt.

5 Das neue Tool in der Praxis

5.1 Umstieg von FMod auf das neue Tool

Ein Umstieg von einer Fremdsoftware auf ein internes Tool wird niemals leicht von statten gehen, solange nicht bei der Programmierung von Anfang an darauf geachtet wurde Soundaufrufe möglichst neutral zu halten. Das wiederrum ist allerdings umso schwieriger, je größer ein Projekt ist. Die Frage, ob sich der Umstieg lohnt, um danach einen schlankeren Workflow zu haben, ist in der Praxis schwer zu beantworten. Das neue Tool ist da keine Ausnahme. Was allerdings bei einem internen Tool von Vorteil ist: Man kann zweigleisig fahren. Der Umstieg muss nicht von einem zum nächsten Moment stattfinden, sondern kann schrittweise geschehen. Sämtliche Skriptaufrufe an die Fremdsoftware behindern das neue Tool in keiner Weise und auch umgekehrt behindert das neue Tool nicht die Fremdsoftware. So können neue Sounds mit dem neuen Tool eingebaut werden und gleichzeitig stückweise die alten Aufrufe überarbeitet und ebenfalls für das neue Tool eingepflegt werden.

Der Unterschied zwischen FMod und dem neuen Tool ist vor allem der Unterschied zwischen Musik, Atmo und Sound Event. In FMod gibt es keinerlei Unterscheidung, für was das Event benutzt wird. Wird es aus dem Skript abgespielt hält es sich an die in FMod erstellten Parameter und Logiken, bis es entweder zu Ende ist oder gestoppt wird. Mit dem neuen Tool muss beim Erstellen des Sound Events oder der Atmo oder der Musik klar sein, wie die Sounds verarbeitet werden sollen. Grundsätzlich ist das System natürlich flexibel. Skripte können durchaus zweckentfremdet werden (was eine leichtere und kreativere Nutzung des Tools unterstützt). Besonders gut können die Skripte allerdings das, nach dem sie benannt sind. Beim Umstieg muss also immer klar sein, welche FMod Events in welche Skripte übertragen werden. In den meisten Fällen werden das Sound Events sein, da das die häufigsten Soundquellen in einem Spiel sind.

Eine weitere Hürde ist das Umdenken vom relativ linearen Design von Events in FMod zu einem mehr aktionsbasierten Design im neuen Tool. In FMod werden alle Events auf einem, für Sounddesigner bekannten, Zeitstrahl erstellt. Dabei kommt es vor, dass verschiedene Zustände des Events hintereinander gereiht sind. Durch FMod Logik, wann welcher Bereich im Loop laufen soll, sind die Zustände voneinander getrennt. Um von Zustand zu Zustand zu springen, müssen Parameter erstellt werden, die wiederrum vom Programmierer in die passenden Skripte eingearbeitet werden müssen. Das neue Tool erlaubt es mehrere Zustände im Mixer zu erstellen und mit eigenen Übergängen zwischen diesen zu wechseln. Die Übergänge können dabei auch leicht selbst in der Szene platziert werden. Für das Event selbst wird mehr Wert auf prozedurales Design der Sounds gelegt. So ist es möglich Sounds in Sound Gruppen zusammenzufassen, aus denen jeweils ein Sound abgespielt wird.

Dadurch, dass es dem Sounddesigner mit dem neuen Tool möglich ist, den Sound im Spiel direkt zu testen und teilweise direkt funktional einzubauen, wird der Workflow vereinfacht. Damit soll es dem Sounddesigner möglich sein, Sounds besser aufeinander abzustimmen. Für den Umstieg bedeutet das aber auch, dass einige Sounds nochmal zusätzlichen Arbeitsaufwand benötigen. So müssen alle Atmos mit Triggern versehen werden und das Fall-Off-Verhalten von 3D Sounds neu gesetzt werden.

Gerade für die Musik bietet FMod oft eine leichtere Implementierung, da diese viel mehr zum linearen FMod Design passt. Ohne Möglichkeit die einzelnen Spuren in Unity auf einem Zeitstrahl anzuordnen, sind komplexe musikalische Themen mit dem neuen Tool nur mit zusätzlichem Arbeitsaufwand in einer DAW umzusetzen. Für solche Fälle kann sich durchaus eine Kombination aus dem neuen Tool und FMod lohnen. Steigt man von FMod auf das neue Tool um, kann so von den Vorteilen beider Tools profitiert werden.

5.2 Auswertung des programmierten Tools

Ein großer Vorteil des neuen Tools ist die einfache Einbindung in die Game Engine. Damit werden der Workflow und die Kommunikation erleichtert. Beides erklärte Ziele des neuen Tools. Zudem ist das Tool einfach erweiterbar. Je nach Spiel und Anforderung können eigene Skripte ergänzt werden. Diese Skripte für besonderes Verhalten im Ton müssten auch bei der Benutzung anderer Sound Software geschrieben werden. Der Vorteil bei einem Game Engine internen Tool ist allerdings, dass diese speziellen Skripte mit weitergegeben werden können. Das neue Tool kann also von jedem Nutzer beliebig erweitert werden.

Ein weiterer wichtiger Punkt für das neue Tool ist die Möglichkeit teilweise Atmos, Sound Events und Musik, ohne jeglichen Programmieraufwand einzubauen. Das grundlegende Sounddesign eines Levels kann also komplett eigenständig und ohne Unterstützung der Programmierer stattfinden.

Allerdings wird nicht nur eine eigenständige Arbeitsweise unterstützt, sondern auch das für Computerspiele so wichtige prozedurale Sounddesign. Jedes Sound Event ist darauf ausgelegt möglichst vielseitig klingen zu können. Der Aufbau und die Bedienung des Tools weisen den Benutzer auf mögliche Variationen im Ton hin. Für die auditive Gestaltung einer lebendige Spielwelt müssen so nicht für jeden Sound neue Parameter erstellt werden, die den Klang in einem vorgegebenen Rahmen anpassen. Die Parameter sind vielmehr schon direkt implementiert und stehen bei Bedarf zur Verfügung.

Im Gegensatz dazu bietet die Unity Engine nur eine sehr beschränkte Möglichkeit die Sound Dateien wie in einer DAW üblich auf einem Zeitstrahl darzustellen. Es ist dementsprechend schwierig zwei

Sounds während der Laufzeit exakt aufeinander abzustimmen. Vor allem für Musik ist das sehr umständlich. Die einzige Möglichkeit zwei Sounds mit einem bestimmten Versatz abzuspielen ist eine exakte Sekundenangabe. Diese muss allerdings erst ausgerechnet werden. Diese Problematik muss nicht zwangsläufig auftreten. Gerade bei Musik, die in der Parallelform abgespielt wird, muss das Timing schon in der DAW konzipiert werden. Grundsätzlich ist es allerdings leichter Musik in FMod zu implementieren.

Ein weiteres Hindernis für das neue Tool ist die Arbeitsoberfläche. Ohne Kenntnisse in der Unity Engine, ist es nicht zu bedienen. Es können natürlich Editor Skripte erstellt werden, die die Bedienung gegebenenfalls erleichtern. Die grundlegende Funktionsweise der Unity Engine oder die des neuen Tools ändern sich dadurch aber nicht. Für einen Sounddesigner, der das erste Mal Sound in Computerspiele einbaut, wäre FMod aber auch nicht ohne Vorwissen zu bedienen.

Fazit

Mit der Konzeption und Programmierung eines Soundtools für die Unity Engine, wurde ein Problem angegangen, dass bei jedem Spieleprojekt auftaucht. Früher oder später muss Ton ins Computerspiel eingebaut werden und die Unity Engine bietet keine einfache Möglichkeit das zu tun. Und auch wenn auf externe Tools zurückgegriffen wird, muss trotzdem ein Programmieraufwand betrieben werden. Verbunden ist das dann mit einem umständlichen Workflow, bei dem der Sounddesigner keine Möglichkeit hat, den erstellten Sound tatsächlich im Spiel zu testen.

Je nach Spiel und den damit verbunden Anforderungen an den Ton, wird so jedes Mal ein kleines individuelles Sound Tool programmiert. Auch in Zukunft wird sich das nicht ändern. Unity plant aktuell nicht mit großen Änderungen am aktuellen System. Lediglich das Aufnehmen von abgespieltem Sound aus der Unity Engine in eine Datei und Snapshot-Funktionen für die Unity-Timeline (mit der Trailer und Zwischensequenzen erstellt werden können) sind Features die aktuell in Entwicklung sind (Unity 2019).

Neben fehlenden Features, ist auch die API der Unity Engine im Sound Bereich nicht immer einfach zu handhaben. Für die Programmierung des neuen Tools mussten einige grundlegende Funktionen erstellt werden. Außerdem lassen sich einige Einstellungen nur sehr kompliziert und damit rechenaufwendiger per Skript setzen. Beispielsweise können zwei Sounds nicht nativ direkt hintereinander abgespielt werden. Stattdessen muss eine zweite Audio Source exakt dann gestartet werden, wenn die erste zu Ende ist. Das wiederrum erfordert die Rechnung, wann exakt die erste Audio Source zu Ende ist, denn auch das ist nicht nativ abfragbar.

Mit dem neuen Tool soll es leichter sein, schnell Sound hinzuzufügen. Gerade auch Programmierer, können sich so relativ einfach eine Atmo zusammenbasteln. Eine Veröffentlichung des Tools auf GitHub, Source Forge oder im Unity Asset Store wäre nach dieser Arbeit der nächste Schritt in der Weiterentwicklung des Tools.

Literaturverzeichnis

Grimme Game (2018): Eine Reise durch die Welt der Ludomusicology - Grimme Game. Online verfügbar unter https://www.grimme-game.de/2018/07/09/ludo-2018-eine-reise-durch-die-welt-der-ludomusicology/#more-1227, zuletzt geprüft am 21.08.2019.

John Leonard French (2018): 10 Unity Audio Tips (That You Won't Find in the Tutorials). Online verfügbar unter https://johnleonardfrench.com/articles/10-unity-audio-tips-that-you-wont-find-in-the-tutorials/, zuletzt geprüft am 27.09.2019.

Matti Luonua (2016): Making Good-sounding Audio for Mobile Games - Unity Connect. Hg. v. Unity Connect. Online verfügbar unter https://connect.unity.com/p/articles-making-good-sounding-audio-for-mobile-games, zuletzt geprüft am 08.09.2019.

Nathan Lovato (2017): 9 Sound Design Tips to Improve your Game's Audio - GameAnalytics. https://www.facebook.com/gameanalytics/. Online verfügbar unter https://gameanalytics.com/blog/9-sound-design-tips-to-improve-your-games-audio.html, zuletzt geprüft am 21.08.2019.

Raffaseder, Hannes (2010): Audiodesign. Kommunikationskette, Schall, Klangsynthese, Effektbearbeitung, Akustische Gestaltung. 2. Aufl. s.l.: Carl Hanser Fachbuchverlag. Online verfügbar unter http://www.hanser-elibrary.com/action/showBook?doi=10.3139/9783446423251.

Stevens, Richard; Raybould, Dave (2016): Game audio implementation. A practical guide using the unreal engine. Boca Raton: CRC Press Taylor & Francis Group CRC Press is an imprint of the Taylor & Francis Group an informa Business. Online verfügbar unter http://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&AN=1261657.

Szczypula, Oliver; Hofmann, Jan (2008): Game Sound. Audiotechnische Aspekte in Computerspielen. Zugl.: Stuttgart, Hochschule, Dipl.-Arb., 2006. Saarbrücken: VDM Verl. Dr. Müller.

The Knights of Unity (2015): Wrong Import Settings are Killing Your Unity Game (part 2). https://www.facebook.com/TheKnightsofUnity/. Online verfügbar unter https://blog.theknightsofunity.com/wrong-import-settings-killing-unity-game-part-2/, zuletzt geprüft am 18.09.2019.

Tobias Heidemann (2012): Games Studie: Wie sehr wirst du von In-Game Sound beeinflusst? https://www.facebook.com/GIGA.GAMES.DE. Online verfügbar unter https://www.giga.de/spiele/amnesia-the-dark-descent/specials/games-studie-wie-sehr-wirst-du-von-in-game-sound-beeinflusst/, zuletzt geprüft am 21.08.2019.

Unity (2019): Leitplan - Unity. Online verfügbar unter https://unity3d.com/de/unity/roadmap, zuletzt aktualisiert am 21.10.2019, zuletzt geprüft am 21.10.2019.

Unity Technologies (2019): Unity - Scripting API. Online verfügbar unter https://docs.unity3d.com/ScriptReference/, zuletzt aktualisiert am 11.10.2019, zuletzt geprüft am 22.10.2019.

Abkürzungsverzeichnis

ADPCM	Adaptive Differential Pulse Code Modulation
API	Application Programming Interface
DAW	Digital Audio Workstation
kHz	Kilohertz
MP3	eigentlich MPEG-1 Layer III oder MPEG-2 Layer III
MPEG	Moving Pictures Experts Group
PCM	Pulse Code Modulation
POS	Player-oriented sounds
UI	User Interface