Hochschule der populären Künste

# Prozedurales, parametrisiertes Sounddesign in Computerspielen

Programmierung eines prototypischen Klangerzeugers

Benjamin Feder

Studiengang Audiodesign (B.A.)
Matrikelnummer AD160400009
Erstprüfer: Prof. Marco Kuhn
Zweitprüfer: Prof. Tilman Ehrhorn
30. März 2020

# Inhaltsverzeichnis

# 1 Einleitung

Audiodesign übernimmt in vielen modernen Medien eine wichtige Rolle. Gerade bei Filmen geht es beispielsweise um das Gestalten einer Atmosphäre, das Aufbauen von Spannung und das unterschwellige Übermitteln von Informationen über das abgebildete Geschehen (Schweitzer, 2004). Der Betrachter soll voll und ganz in die Ereignisse des Mediums aufgenommen werden.

Auch wenn der zugrundeliegende Prozess des Sounddesigns für lineare und interaktive Medien sehr ähnlich ist, entstehen allerdings vorallem bei letzteren einige neue Herausforderungen. In einem linearen Medium wird dem Betrachter durch Kameraeinstellungen und Schnitt bereits vorgeschrieben, was er sehen kann. Außerdem hat er keinerlei Einfluss auf den Ablauf der dargestellten Ereignisse. Der zeitliche Verlauf der visuellen und akustischen Eindrücke ist bei einem solchen Medium bereits festgelegt. Der Sounddesigner kann hierauf aufbauen und eine klangliche Atmosphäre gestalten. Etwas schwieriger gestaltet sich ein immersives Sounddesign für interaktive Medien, wie beispielsweise Computerspiele.

Heutzutage sind Computerspiele meist so gestaltet, dass der Spieler einen sehr großen Grad an Handlungs- und Bewegungsfreiheit hat. Das hat zur Folge, dass für den Entwickler des Spiels nur schwer vorhersehbar ist, wann der Spieler wie handelt und was in dieser Zeit alles in der Spielwelt geschieht. Um das Sounddesign dementsprechend zu gestalten und außerdem die Präsenz des Spielers nicht zu verringern, ist es also wichtig, auf Ereignisse in der Spielwelt und das Verhalten des Spielers einzugehen, vor allem dann, wenn beim Spieler die Erwartung geweckt wird, das bestimmte Aktionen auch ein akkustisches Ereignis mit sich ziehen.

Physikalisch betrachtet ist Schall nichts weiter als eine mechanische Welle, die durch unser Ohr für uns hörbar gemacht wird. Diese mechanischen Wellen entstehen, indem ein Körper in Schwingung versetzt wird (Meyer & Schmidt, 2005, S. 125-127). Für den Prozess des Sounddesigns bedeuted das also: sind die initialen Auslöser eines Klanges und die Eigenschaften der Klangquelle bekannt, so sollte es möglich sein, diesen Klang zu ermitteln und Änderungen an der Klangquelle mit in Betracht ziehen zu können. Diese Arbeitsweise wird als prozedurale Klangerzeugung beschrieben.

In dieser Arbeit soll zunächst untersucht werden, wie die prozedurale Klangerzeugung in verschiedenen Medien eingesetzt werden kann und welche Unterschiede dies speziell im Bereich von Copmputerspielen im Vergleich zur einfachen Verwendung von fertigen Samples aufweist. Anhand des Klanges von Feuer soll dann ein exemplarischer Prototyp entwickelt werden, der die Praktikabilität und Einsetzbarkeit prozeduralem Sounddesigns genauer betrachtet. Hierzu wird ein Synthesizer entwickelt, der durch Angabe abstrakter Parameter, die das Feuer beschreiben, den Klang dieses Feuers modellieren soll. Mithilfe der Spielengine Unity [1] wird dieser dann in einer virtuellen Umgebung getestet wird. Der

[1] Bei Unity handelt es sich um eine Entwicklungsumgebung für virtuelle Erlebniswelten wie Filme, Simulationen und vorallem auch Spiele.

Fokus soll hierbei auf der Entwicklung eines Arbeitsflusses liegen, der in möglichst vielen Bereichen eine Anwendung finden kann. Für diesen Arbeitsfluss sollen Schritte wie das Analysieren der verschiedenen Klangquellen, eine sinnvolle abstrakte Parametrisierung und eine vereinfachte und trotzdem flexible Integration in verschiedene Anwendungen in Betracht gezogen werden.

# Teil I
# Prozedurales Sounddesign

## 2 Voruntersuchung

### 2.1 Was heißt prozedural?

Das Wort ‘prozedural’ (‘procedural’) kommt von ‘Prozedur’ (‘procedure’) und bedeuted “a series of steps followed in a regular definete order“ (meriam-webster, 2019). Eine weitere Defintion lautet: “den äußeren Ablauf einer Sache betreffend"(Duden, 2019). Eine Prozedur ist also kurz zusammengefasst, eine Reihe an Anweisungen, die in einer bestimmten Reihenfolge durchgeführt werden sollen. Das Ergebnis dieser Prozedur muss nicht von vornherein bekannt sein, sondern ist nur zu sehen, wenn alle Schritte der Prozedur befolgt wurden. Eine prozedurale Generation beschreibt also ‘[...] a method of creating data algorithmically as opposed to manually [...]’ („Procedural generation“, 2019). ‘Prozedural generierter Sound’ bedeuted demnach also, dass nicht zu Begin ein vorbereitetes Sample geladen wird, welches zum rechten Zeitpunkt abgespielt werden soll, sondern vielmehr, dass zuvor Regeln definiert werden, nach denen der Sound generiert wird und sich entwickeln kann. Gerade für computergebundene Anwendungen bietet sich hierdurch die Möglichkeit, Einwirkungen von außen mit in die Klangerzeugung einfließen zu lassen.

### 2.2 Anwendungsbereiche

Ein sehr großer Anwendungsbereich für prozedurales Sounddesign sind interaktive Klanginstallationen. Ohne derartige prozesse wäre die Essenz solcher Installationen, die darin besteht, Besucher am Gesamtbild dieser teilhaben und den Verlauf der Präsentation beeinflussen zu lassen, nur sehr schwer umzusetzen. Ein Beispiel einer solchen Installation wurde vom Team ’lev’ gestaltet. Das Team programmierte das sogenannte loop-Ensemble. Hierbei handelt es sich um eine Zusammenstellung von Softwareinstrumenten, die speziell dafür gestaltet wurden, die Grundlagen der elektronischen Musik zu lehren. Das Team hat die Instrumente des loop-Ensemble so präpariert, dass diese zunächst “[...] vorgestaltete Loops stur und statisch vor sich hin spielen [...]” (lev Team, 2017). Mihilfe von verschiedenen Controllern konnten die Besucher nun die Generierung dieser Loops beeinflussen und sie selbst gestalten. Parametrisierung spielte hierbei eine entscheidende Rolle. Um den Besuchern ein gewisses Erfolgserlebnis zu geben, war es wichtig, dass ihre Interaktion gut klang, ohne dass dafür musikalisches Verständis erforderlich war. Die Eingaben, welche über die Controller getätigt wurden, mussten also so interpretiert werden, dass alles in einem musikalischen Einklang stand.

Klanginstallationen, in denen der Besucher die zu hörende Musik beeinflusst sind eine der häufigsten Formen solcher Installationen. Weitere Beispiele hierfür

sind das Sibelius-Monument in Helsinki (Kühne, 2015) und "5,3 Kilo Meter pro Stunde" in Bonn (Stache, 2012). In all diesen Fällen spielt der Besucher eine essenzielle Rolle.

In der Klanginstallation "Bug" von Mark Bain in Berlin hingegen werden verschiedenen Geodaten und seismischen Sensoren verwendet, um alle noch so winzigen Ereignisse in und um einem Gebäude, an dem sie angebraucht sind, aufzunehmen. Diese Werte werden dann zur Steuerung eines Synthesizers verwendet. Das gesamte Haus und die Umgebung dienen als 'Eingabegerät' mit welchem die Prozedur des Sounddesigns besteuert wird. Auf diese Weise entsteht eine "dauerhafte Komposition" (Bain, 2010).

Prozedural generierter Sound bildet in diesen Fällen also die wichtigste Grundlage. Es gibt bestimmte Regeln, die, in diesen Fällen von Computern, interpretiert werden müssen, welche die Aktionen der Besucher und anderen äußeren Einflüssen mit einbeziehen, aus welchen erst dann das vollständige Klangerlebnis entsteht.

Ein weiterer Anwendungsbereich, der möglicherweise nicht gleich auffällt, ist Sprachsynthese. Vor allem bei Text-zu-Sprache-Synthese kann nicht von vorn herein bestimmt werden, was gesagt werden soll und in welchem Kontext oder Satzbau und daraus folgend in welcher Betonung das Gesprochene wiedergegeben werden muss. Inhalt und Kontext entstehen erst mit der Eingabe des Benutzers. Gerade in modernen Systemen wie Siri und Google Now ist oft nicht einmal der auszugebende Text bekannt, sondern nur der Inhalt, welcher wiedergegeben werden soll (Kontext-zu-Sprache-Synthese) (Huang & Deng, 2010). Es bietet sich an, Sprache für derartige Zwecke also auch prozedural zu generieren.

Zusammenfassend lässt sich sagen, prozedurale Sounds sind am sinnvollsten in Bereichen einzusetzen, in denen zu Begin noch nicht feststeht, wie der auszugebende Sound tatsächlich klingen soll. Computerspiele bilden eine ideale Platform für die Anwendung solcher Prozeduren. Spieler werden zur Ineraktion aufgefordert, durch welche der Klang beeinflusst werden könnte. Außerdem können die Prozeduren bereits innerhalb der Präsentationsplatform (der Computer) durchgeführt werden.

## 2.3 Prozedural generierter Sound in Computerspielen

### 2.3.1 Eine kurze Geschichte

Es ist keine neue Idee, Sounds für Spiele generativ zu erzeugen. Bereits die ersten Arcadeautomaten und später auch Heimkonsolen ab 1970 haben darauf zurückgegriffen, Sounds während des Spielens zu generieren. In diesen Fällen handelte es sich allerdings nicht zwangsweise um eine ästhetische Entscheidung. Sie lag viel mehr in den nur gering zur Verfügung stehenden Resourcen begründet. Der sowohl resourcen- als auch arbeitstechnische Aufwand, Samples in zufriedenstellender Qualität aufzunehmen und wiederzugeben stand einfach nicht im Verhältnis zum daraus resultierendem Ergebnis. Die Sounds zum richtigen

Zeitpunkt zu generieren war wesentlich effizienter. Allerdings waren die Prozessoren der Computer in ihrer Kapazität beschränkt. Deswegen wurden dedizierte Prozessoren für die Erzeugung des Klanges entwickelt und in den Endgeräten verbaut. Zu diesen 'Soundchips' gehören unter anderen der SN76477, der in sehr vielen Arcademaschinen verbaut wurde und der SID6581 der im Commodore 64 zu finden ist. Beide Chips erfreuen sich auch heute noch gerade bei Musikern, die gerne selbst Synthesizer bauen, großer Beliebtheit. Die meisten Soundchips bestehen aus wenigstens einem Stimmoszillator, einem Noisegenerator, mehreren Eingängen, diese beeinflussen können und einem Ausgang für das synthetisierte Signal. Häufig zu finden sind noch weitere Stimmoszillatoren, Niedrig-Frequenz Oszillator, sowie Amplituden- und Frequenzmodulatoren und Zufallszahlgeneratoren (Texas Instruments, 1978, 1981a). Diese Auswahl stellte den Sounddesignern dieser Zeit eine Pallette zur Verfügung, mit der es galt, die jeweiligen Spiele zu vertonen. Hauptsächlich wurden die Soundchips zum Erzeugen der Hintergrundmusik verwendet, da Soundeffekte oft aus mehreren Elementen bestehen, die mit den wenigen Mitteln solcher Chips nur schwer nachgestellt werden können.

1980 gab es dann mit dem TMS5220 sogar einen Chip, der in der Lage war, Sprache synthetisch zu erzeugen. In diesem Chip sind Sprachdaten gespeichert, welche verwendet werden können, um eine menschenähnliche Stimme zu reproduzieren (Texas Instruments, 1981b).

Ende 1980 entstand die Idee, das zu dieser Zeit noch recht neu entwickelte Protokoll 'MIDI' für Spiele zu verwenden (Collins, Karren, 2008). Bei diesem Protokoll handelt es sich um ein System, welches es elektronischen Musikinstrumenten und Computern erlaubt, untereinander Befehle und Informationen auszutauschen (MIDI Manufacturers Association, 2009). Musikdaten konnten im MIDI-Format abgespeichert werden und der Spieler erhielt die Möglichkeit eine eigene Soundkarte oder Synthesizer zu verwenden, um die MIDI-Daten zu interpretieren. Der Vorteil und die Begründung hierbei war, dass diese Synthesizer meist bei weitem mehr Möglichkeiten boten als die bisherigen Soundchips. Zu diesen zählte beispielsweise auch, dass die Klänge echter Instrumente mithilfe von verwendet werden konnten. Allerdings gab es bei dieser Variante auch ein paar Nachteile. Wenn eine Musik auf einem bestimmten Instrument abgespielt werden sollte, mussten zunächst die Speicherdaten dieses aufgerufen werden. Da diese Speicherdaten allerdings noch nicht zu einem Standard generalisiert wurden, kam es vor, dass auf einer anderen Soundkarte an der selben Adresse Speicherdaten für ein vollkommen anderes Instrument hinterlegt waren. Das Klangergebnis variierte also sehr stark.

Um dieses Problem zu umgehen, wurde der Standard General MIDI entwickelt, was zwar noch nicht dazu führte, dass die Klangqualität der Soundkarten angeglichen wurde, aber zumindest gab es nun die Möglichkeit, die gleichen Klänge auf verschiedenen Soundkarten zu verwenden.

Der nächste Schritt für die auf diese Weise prozedural generierte Klänge war, Musik adaptiv zu komponieren. Entwickler hatten den Wunsch, die Instzenierung linearer Medien, welche die abgebildete Atmosphäre klanglich unterstützen konnten, ebenfalls zu ermöglichen. Vorreiter für diese Art des Komponierens

war das Studio Lucas Arts mit ihrem Spiel 'Monkey Island 2' (Collins, 2016). In diesem Spiel kann der Spieler viele verschiedene Plätze erkunden. Diese Vielfältigkeit wollten die Entwickler aufgreifen. Sie wollten beispielsweise nicht, dass auf dem Schiffsdock die selbe Musik spielt wie in Chucks Taverne. So war die Idee geboren, die Hintergrundmusik von der aktuellen Situation des Spielers abhängig zu machen.

Mit der Entwicklung der CD sah sich die Spieleindustrie aber einer neuen Herausforderung gegenüber gestellt. Die Qualität aufgenommener Samples und Musik wurde besser und somit stiegen auch die Erwartungen der Spieler. Die Nutzung dieses Mediums brachte mehrere Vorteile mit sich. Hierzu zählen beispielsweise der erweiterte Speicherplatz, die verbesserte Qualität und die Tatsache, dass das abzuspielende Audio nun auf allen Geräten, auf denen es wiedergegeben werden sollte, gleich klang. Mit der Implementierung dieses Mediums ging der Eisatz des generativ erzeugten Klanges aber sehr stark zurück (Collins, 2016).

Mit heute zur Verfügung stehender Computertechnik können die meisten der bisherigen Probleme recht einfach gelöst werden. Die Audioqualität ist sehr hoch, der benötigte Speicherplatz ist im Verhältnis zum verfügbaren sehr gering. Prozessoren sind leistungsfähig genug um nicht nur die Berechnung der Spielphysik und Graphik sondern auch der Sounds zu übernehmen. Aus diesen Gründen ist zu beobachten, dass das Prinzip, Klänge erst während der Laufzeit zu generieren, in Spielen wieder sehr viel häufiger aufgegriffen wird.

### 2.3.2 Fallbeispiele

**FractOSC** In dem Puzzlegame [2] FractOSC wandert der Spieler durch eine Welt, in der mehrere Puzzle gelöst werden müssen. Diese Puzzle bestehen in den meisten Fällen daraus, dass mehrere Kisten durch ein komplexes Terrain an die für sie vorgesehene Stelle geschoben werden müssen. Nähert sich der Spieler einem Puzzle, wird der Soundtrack dieses hörbar. Jedes Einzelteil des Puzzels (bsp. eine Kiste) stellt einen Part des Soundtracks dar und je nachdem wo sie plaziert sind, ändert sich dieser Part klanglich. Je näher der Spieler an der Lösung des Puzzels ist, desto mehr klingt der Soundtrack nach einer vollständigen Komposition. Ist das Puzzle gelöst, wird der Track in seiner ursprünglich komponierten Form abgespielt und mit zusätzlichen Elementen komplettiert. In einer der Welten verändert sich beispielsweise beim Schieben einer Kiste in x-Richtung durch den Raum die Hüllkurve eines Tons. Wird sie in y-Richtung durch den Raum geschoben, verändert sich die Tonhöhe. Würde man für jeden Ton und jede Veränderung ein extra Sample generieren, müsste hierfür sehr viel Speicherplatz reserviert werden und nur ein geringer Bruchteil der gespeicherten Samples würde tatsächlich verwendet werden. Wesentlich effizienter ist es hingegen, die Sounds zur benötigten Zeit von einem Synthesizer generieren zu lassen. Die Entwickler dieses Spiels haben sich dazu entschieden, für diese Aufgabe die Software PureData in die Spielengine Unity zu implementieren (Henk,

[2] Spiel mit dem Ziel, Rätsel verschiedener Art zu lösen („Puzzle video game“, 2019).

2013b). In diesem Beispiel ermöglicht prozedurales Sounddesign also, Musik zu synthetisieren die auf die Aktivitäten des Spielers reagiert. Schon von Beginn an wird damit gerechnet, dass sich die Klänge innerhalb des Spieles verändern werden (Henk, 2013a).

**No Man's Sky** Das open world action adventure [3] No Man's Sky verwendet ebenfalls an einigen Stellen prozedural generierte Sounds (Mongeau, 2017). Inhalt dieses Spieles ist, dass der Spieler verschiedene unbekannte Welten erkunden kann. Die Entwickler hatten das Ziel, eine Spielwelt zu generieren, die in ihrer Größe einer Galaxie gleichkommt. Für eine derartig große Welt ist es unmöglich, alle Welten manuell zu generieren. Aus diesen Gründen gibt es für jede Welt ein Set an zufällig generierten Eigenschaften wie Größe, Atmosphäre und Vegetation, nach denen diese erstellt werden kann. Auch die Lebewesen, welche auf diesen Planeten zu finden sind, werden entsprechend dieser Eigenschaften generiert. Deswegen ist es für die Entwickler des Spiels nur schwer vorherzusehen, wie diese Tiere aussehen werden. Es gilt nun, für diese unbekannten Tiere Klänge zu designen, die ihrem Aussehen entsprechen können. Für diesen Zweck wurde der Synthesizer 'VocAlien' entwickelt. Dieser erzeugt in Abhängigkeit der verschiedenen Eigenschaften der Tiere, wie zum Beispiel Gewicht oder Größe des Kopfes und Körpers, ein Modell, nach dem ein Klang generiert werden kann, welchen das Tier in verschiedenen Situationen von sich geben könnte (Weir, 2017). In diesem Beispiel wird also mit Hilfe des prozedural generierten Sounddesigns und einer gut ausgearbeiteten Parametrisierung eine praktisch unedliche Menge an Geräuschen erstellt. Vom zu vertonenden Objekt sind allerdings nur Eigenschaften, nicht aber das tatsächliche Aussehen bekannt. Dies hat nicht nur den Vorteil das sehr viel Speicherplatz gespart wird, sondern sorgt auch dafür, dass immer individuelle und zu den Tieren passende Klänge erzeugt werden können, ohne dass bereits gehörte Samples wieder verwendet werden müssen.

**Grand Theft Auto V** In dem open world action adventure GTA V lebt der Spieler in der fiktiven Stadt Los Santos. Eine Stadt wie diese ist voll von Ereignissen, Menschen, Haustieren und Autos. All diese Dinge geben Geräusche von sich und bilden die 'natürliche' Soundkulisse der Stadt. Hierzu zählen neben Gesprächen, Hundebellen und der Klang einer entfernten Autobahn beispielsweise auch Schüsse und Sirenen.

Alastair MacGregor, der Sounddesigner des Spiels, erwähnte in einem Vortrag, wie wichtig den Entwicklern neben dieser Kulisse auch impact sounds [4] waren und dass sie besonders viel Wert darauf legten, dass diese realistisch klingen (MacGregor, 2014a, S. 11-12). Weitere Beispiele für Sounds, die für dieses Spiel prozedural generiert wurden sind das Brummen von Klimaanlagen, Motorengeräusche, Rasseln von Fahrradketten und sogar Musik die aus Clubs zu

[3] Spiel, mit einer frei begehbaren Spielwelt, in der der sowohl das lösen von Rätseln (Adventure) also auch Geschicklichkeit (Action) gefordert sind („Action-Adventure", 2019).

[4] Ein Geräusch, welches entsteht, wenn ein Objekt auf ein anderes trifft. Dies können beispielsweise einfache Schritte sein, aber auch Fahrzeuge die gegen Absperrungen fahren oder Munition, die in Wände einschlägt.

hören ist(MacGregor, 2014b). Durch dieses prozedurale Sounddesign kann eine sehr authentische akustische Vielfalt erzeugt werden.

# 3 Arbeiten mit prozeduralem Sounddesign

## 3.1 Vortile von prozedural generiertem Sound

Der direkteste Ansatz, Klänge in Spiele zu bringen, beruht darauf, die gewünschten Samples vorzubereiten und dann zum rechten Zeitpunkt abzuspielen (samplebasiert). Mit dieser Methode lassen sich die benötigten Resourcen und das Verhalten der Klänge bereits zu Beginn sehr gut einschätzen. Dennoch bietet der etwas komplexere Weg einer prozeduralen Klangerzeugung ebenfalls einige Vorteile. Anhand der in Kapitel 2.3.2 genannten Beispiele lassen sich diese gegenüber einer herkömmlichen, samplebasierten Methode recht deutlich erkennen.

### 3.1.1 Erstellen von Inhalten

Klänge prozedural zu generieren bietet die Möglichkeit, verwendbare Spielinhalte (Assets), wesentlich schneller zu erzeugen. Mit nur wenigen Ausgangsmaterialien kann eine Prozedur beschrieben werden, die eine wesentlich größere Vielfalt bietet (Farnell, 2010, S. 321).

Ein gutes Beispiel hierfür bietet der Sounds von Schritten. In vielen Spielen in denen der Spieler sich frei bewegen kann, ist der Schrittsound der am häufigsten gehörte. Wird der Sound zu repetetiv ist es sehr warhscheinlich, dass dies dem Spieler negativ auffällt. Dies kann dazu führen, dass die Immersion des Spielers unterbrochen wird. Um dies zu verhindern sollte dafür gesorgt werden, dass jeder Schritt möglichst einzigartig und anders klingt. Natürlich ist eine Lösung, viele Sounds aufzunehmen und sie in zufälliger Reihenfolge wiederzugeben. Eine weitere ist jedoch, eine geringere Anzahl verschiedener Klänge, die einem Schritt eigen sind, aufzunehmen und diese auf unterschiedliche Weise zu kombinieren. Hierauf wird in Kapitel 3.4 näher eingegangen. So lassen sich mit nur wenigen Ausgangssamples eine wesentlich größere Anzahl an tatsächlich hörbaren Klängen erzeugen.

Im Allgemeinen ist diese Herangehensweise sehr gut für jegliche Art von Sounds die durch Einschläge passieren, da sie nur aus einer geringen Anzahl von Grundklängen bestehen, die sich aber zu einem komplexen Sound kombinieren lassen.

Auch bei Musik lassen sich die Vorteile der prozeduralen Erstellung von Inhalten deutlich erkennen. Ein sehr gutes Beispiel hierfür ist das Spiel ‘Fallout New Vegas‘. Der Spieler bewegt sich frei durch eine Welt und kann an verschiedenen Orten kleine Ansiedlungen betreten. Jede einzelne dieser Ansiedlung hat einen eigenen Soundtrack. Hierbei handelt es sich allerdings nicht um ein ausgearbeitetes Musikstück sondern vielmehr um ein Set an kleinen Melodien und Phrasen von verschiedenen Instrumenten. Durch diese prozedural generierte

Komposition kann für jedes Dorf ein individueller Soundtrack geschaffen werden, ohne dass auf ein fertiges Musikstück zurückgegriffen werden muss, welches dauerhaft wiederholt wird.

### 3.1.2 Verhalten

Ein weiterer Vorteil, Klänge prozedural zu erzeugen ist, dass auf das Verhalten der Spieler und Ereignisse in seiner Umwelt eingegangen werden kann. Gerade bei Rennspielen ist dies deutlich erkennbar. Der wahrschienlich wichtigste Sound für ein Rennspiel ist das Geräusch des Motors. Dem Spieler soll mit diesem Sound nicht nur das Gefühl von Geschwindigkeit übermittelt werden, sondern gerade bei realistischeren Simulatoren muss dieser dafür genutzt werden, bestimmte Verhaltensweisen und Zustände des Autos zu kommunizieren. Um dies zu gewährleisten ist es besonders wichtig, dass der Klang sich den Eingaben des Spielers anpasst. Somit ist ein solcher Klang ebenfalls am sinnvollsten durch prozedurale Generierung zu erzeugen. Auf diese Weise können nicht nur die aktuelle Drehzahl des Motors bedacht werden, sondern auch eventuelle Schäden oder Veränderungen können akkustisch wiedergegeben werden, ohne dass für alle Fälle ein extra Sample gestaltet werden muss.

Auf ähnliche Weise kann auch bei der Spielmusik auf die Aktionen und die Situation des Spielers eingegangen werden. Im bereits erwähnten ‘Fallout New Vegas‘ verändert sich die Menge der gleichzeitig oder sukzesiv gespielten Phrasen der einzelnen Ansieldungen in Abhängigkeit zur Entfernung des Spielers zu diesen Dörfern. Je näher der Spieler also einem Dorf kommt, desto mehr wird die Atmopshäre dieses Dorfes durch die ihm eigene Musik wiedergegeben.

Auch bei Gefahren oder Kämpfen kann der Spieler gewarnt werden. Wurde der Spieler von Gegnern bemerkt, so startet eine Musik, die Spannung erzeugt und den Spieler darauf aufmerksam macht, dass jemand nach ihm sucht. Kommt es zum Kampf, wechselst die allgemeine Hintergrundmusik in eine schnelle und actionreiche Kampfmusik.

## 3.2 Nachteile von prozedural generiertem Sound

Neben den Vorteilen, die diese Arbeitsweise mit sich bringt, müssen allerdings auch einige Nachteile in Betracht gezogen werden, die hierdurch vorallem bei der Implementierung in Computerspiele entstehen.

### 3.2.1 Resourcenkosten

Um prozedurale Klänge und Musik zu erzeugen, müssen diese zunächst jene Prozedur durchlaufen. Das bedeuted natürlich auch, dass wenn mehr Sounds vom Spiel benötigt werden, diese Prozeduren häufiger durchgeführt werden müssen. Das kann unter Umständen dazu führen, dass die Resourcenauslastung schwerer einschätzbar wird. Bei einem Sample-basiertem Ansatz lässt sich dies leichter kalkulieren, da ein Sound nur einmal geladen werden muss und von da an beim

Abspielen immer die gleiche Auslastung der Resourcen erfordert. Dies muss aber nicht ausschließlich ein Nachteil sein. Ein Beispiel bei dem dieser Faktor im prozeduralem Ansatz einen Vorteil mit sich bringt wäre die Vertonung einer Menschenmenge. Bei einem samplebasierten Ansatz müssten bei einer größer werdenden Menge mehr Samples geladen werden, was zu höherer Speicherauslastung führt. Mit einer prozeduralen Generierung würde die Auslastung des Prozessors und Speichers jedoch verhältnismäßig unverändert bleiben, da nur die Werte der Prozedur, nicht aber die benötigten Ausgansmaterialien angepasst werden. Deswegen sind gerade bei steigender Quantität aber gleichzeitig geringeren Anforderungen an die Qualität, Prozeduren oftmals wesentlich performanter als ein samplebasierter Ansatz (Farnell, 2010, S. 322).

### 3.2.2 Programmieraufwand

Bei der herkömmlichen samplebasierten Methode wird "nur" ein Sounddesigner benötigt, der die Samples erstellt. Diese Samples können dann ohne größeren Programmieruafwand an der entsprechenden Stelle aufgerufen und abgespielt werden. Möchte man aber nun die Sounds prozedural erstellen, ist es nicht nur erforderlich, die Prozeduren zu programmieren, sondern diese auch in die Spielengine einzubinden. Zudem muss die Implementierung so erfolgen, dass während des Spiels die Parameter der Prozedur angepasst werden können. Dies umzusetzen erfordert zum einen, dass der Sounddesigner mindestens geringe Programmierkenntnisse und der Programmierer einige sounddesigntechnische Kentnisse hat. Im Optimalfall sollten jedoch neben den Sounddesignern und Programmierern dedizierte Audioprogrammer oder Tooldeveloper eingesetzt, die die Schnittstelle zwischen beiden Seiten herstellen können.

## 3.3 Parametrisierung

Sound generativ zu erzeugen ist eine gute Art und Weise, den Klang an die Geschehnisse in seiner Umgebung anzupassen. Soll dies geschehen, müssen zwei Seiten miteinander kommunizieren, die in der realen Welt zwar miteinander verbunden, in der nachgestellten virtuellen Welt aber vollkommen voneinander getrennt sind. Auf der einen Seite steht die Physik, also die Programmierung der virtuellen Umgebung. Wie verhält sie sich, wie verändert sie sich, wie reagiert sie auf die Interaktion und wie verhält sich darauf hin das Objekt, welches den Sound abgibt. Auf der anderen Seite steht die Klangentwicklung, also das Sounddesign selbst. Wie müssen Filter verändert werden, um einen Sound klingen zu lassen, als wäre die Quelle hinter einer Wand versteckt, wie muss ein Sample verändert werden, um den Klang größer zu gestalten. Meist hat eine kleine Auswirkung in der virtuellen Welt eine große Folge auf die Gestaltung des Sounds.

Am Beispiel von Regen kann dies gut verdeutlicht werden. In der virtuellen Welt wird einprogrammiert, es soll mehr Regen fallen. Der Klang würde sich dabei auf verschiedene Weisen verändern. Je nach der Methode, mit der der Klang des Regens erzeugt wird, müssen mehr Samples für einzelne Highlights

[5] geladen werden. Um einen verwaschenen Klang in weiterer Ferne zu erzeugen könnte mehr Rauschen hinzugefügt werden. Außerdem ist es bei einer größeren Regenmenge wahrscheinlich, dass der Regen auf durchtränkten Boden fällt, wass durch das Hinzufügen von Obertönen wiedergegeben werden könnte. Zu guter Letzt müsste sehr wahrscheinlich im Vergleich zu schwächerem Regen die einzelnen Lautstärken angepasst werden. Sollen all diese Faktoren Seitens der Programmierung der virtuellen Welt bedacht werden, kann der gesamte Prozess schnell unübersichtlich werden. Um all diese Entscheidungen und Verknüfpungen in der Hand des Sounddesigners zu lassen, ist eine Parametrisierung ratsam. Mit dieser ist es dem Sounddesigner möglich, mehrere Designwerte zusammenzufassen, so dass der Programmierer nur einen Parameter anpasst und der Sound sich nach den Vorstellungen des Designers verändert.

Auch das Verhalten von Hintergrundmusik kann parametrisiert werden. Ein gutes Beispiel hierfür ist einem Spielprojekt, bei welchem es meine Aufgabe war, die Musik zu komponieren. In diesem wurde der Spieler mit einer immer größer werdenden Menge an Gegnern konfrontiert. Der Spielproduzent hatte den Wunsch, dass die Intesität der Musik sich in Abhängigkeit von der Gegnerzahl verändert. Damit die Programmiererin nicht die einzelnen Ebenen der Musik als Samples einprogrammieren, ineinander überblenden und gegebenenfalls Transitions zwischen einzelnen Abschnitten der Musik programmieren musste, haben wir uns dazu entschieden, die Audiomiddleware Wwise zu benutzen. Bei einer Audiomiddleware handelt es sich um ein Programm, welches einem Sounddesigner Werkzeuge zur Verfügung stellt, mit denen Prozeduren für die Klangerzeugung und interaktive Kompositionen entwickelt und diese für verschiedene Spielenginges bereitgestellt werden können. Mit dieser Audiomiddleware hatte ich die Möglichkeit, Teile der Musik zu arrangieren und diese Zusammensetzung anhand von realtime parameter controllern (RTPCs) zu verändern. Somit konnte die Programmiererin einfach angeben, wie viele Monster gerade vorhanden waren und aufgrund der in Wwise vorgeschriebenen Prozedur passte sich die Musik daran an.

Parameter sind bei generativem Sounddesign also die Brücke zwischen Programmierern und Sounddesignern. Die Programmierer bestimmen welche Werte Einfluss auf die Klangquelle haben und die Sounddesigner können die Prozedur entsprechend gestalten. Lehmann schreibt in seiner Arbeit: "[Parameter] entsprechen also einem wargenommenen Verhalten, das durch einen Prozess beschrieben werden kann."(Lehmann, 2014). Parameter sind also nicht als Werte zu sehen, die das Sounddesign selbst direkt verändern, sondern müssen als Werte verstanden werden, die die Prozedur mit welcher der Klang erzeugt wird beeinflussen.

[5] Einzelne akustische Ereginisse, die sich vom Grundklang eines Schallereignisses abheben. Einige Beispiele währen: Knacken in einem Feuer oder deutlich erkennbare Worte in einer Menschenmenge.

## 3.4 Arten von Prozeduren

In den in Kapitel 2.3.2 genannten Beispielen und anhand der Geschichte von Klängen in Computerspielen, lassen sich deutlich ein Paar der häufiger verwendeten Prozeduren erkennen. Diese sind den Prozeduren, welche Sounddesignern innerhalb von digital audio workstations [6] zur Verfügung stehen sehr ähnlich.

**Synthesizer** Der erste Ansatz, der auch mit den dedizierten Soundchips in frühen Spielkonsolen umgesetzt wurde, ist eine voll synthetischer Generierung. Durch additive und subtraktive Synthese werden die gewünschten Klänge von Grund auf neu erzeugt. Ein Satz an Regeln bestimmt, zu welcher Zeit welcher Oszilator mit welcher Frequenz und mit welcher Wellenform schwingen soll. In GTA V und No Man's Sky wurden jeweils eigenständige Synthesizer entwickelt, welche den Sounddesignern ermöglichten, die Regeln für die Prozeduren zu gestalten. Diese konnten dann exportiert und in das jeweilige Spiel eingebunden werden. Speziell in diesen beiden Beispielen wurden diese Tools so entwickelt, dass sie den für Sounddesigner bekannten Synthesizern Reaktor oder Max/MSP ähnlich sehen. Das sollte nicht nur dafür sorgen, dass die Sounddesigner in vertrauter Umgebung arbeiten konnten, sondern sie auch dazu animieren, neue Dinge auszuprobieren (MacGregor, 2014a, S. 21). Im Beispiel von FRACT OSC erstellte der Entwickler mithilfe der Software PureData (Pd) einen Synthesizer und implementierte diesen mithilfe des Tools libPd (Henk, 2013b).

**Samples** Samples stellen eine sehr effiziente Möglichkeit dar, Klänge aus der realen Welt in eine digitale Umgebung zu bringen. Sie verbrauchen wenig Speicherplatz und haben klanglich eine recht hohe Qualität. Außerdem ist es häufig wesentlich einfacher einen Klang aufzunehmen und diesen anzupassen, als den Gesamtklang synthetisch zu erzeugen. Mithilfe von einfachen Samples lassen sich ebenfallls einige Prozeduren erstellen. In verschiedenen Audiomiddleware ist dies ein bevorzugter Ansatz und es ergibt sich der folgende Arbeitsweg: Nachdem die Samples importiert wurden, können Kontainer erstellt werden, die diese Samples zufällig auswählen und innerhalb von definierten Grenzen in ihrer Abspielgeschwindigkeit, Tonhöhe und Lautstärke anpassen. (Audiokinetic, 2019) Komplexere Klänge können in kleinere Einzelteile zerlegt und auf zufällige Weise wieder zusammengemischt werden. Am Beispiel von Schritten ist dies sehr gut zu erkennen. Ein Schritt kann in zwei Teile separiert werden: Dem Aufsetzen der Ferse und das Abrollen des Fußballens. Verwendet man für beide Teile jeweils 5 verschiedene Samples und wählt beim Abspielen des Gesamtklanges aus diesen ein zufälliges Sample aus, ergeben sich 25 mögliche Ergebnisse. Diese können dann wiederum mit zufällig ausgwählten Geräuschen von Kleidung oder Rüstung vermicht werden. Auf diese Weise entstehen mit sehr wenig Ausgangsmaterial eine große Vielfalt an zu hörenden Schritten.

[6] Bei einer digital audio workstation (DAW) handelt es sich um eine digitale Umgebung, welche der Aufnahme und Bearbeitung von Audio dient. Mithilfe von Plugins können innerhalb dieser auch Klänge generiert werden.

**Granulare Synthese** Bei einer granularen Synthese wird ein Ausgangssample verwendet, welches in kleine Teile (Grains) unterteilt wird, die dann wiederum nach Bedarf gelooped, gestreched, gepitched und ineinander übergeblendet werden. Da bei diesem Ansatz die Klangcharakteristik sehr vom Ausgangssample abhängig ist, wird sie häufig in Fällen angewendet, in denen das Ergebnis einem bestimmtem Gegenstück der realen Welt ähnlich klingen soll. In einem solchen Fall kann ein Sample direkt vom Orginal aufgenommen und verwendet werden. Dies ist vorallem häufig bei Rennsimulatoren zu beobachten, die reale Fahrzeuge nachmodellieren. Das selbe Verfahren wäre für die Generierung von anderen mechanischen Klängen wie Helikoptern oder Generatoren oder das Erzeugen von Menschenmengen oder Wetter denkbar.

**Physical Modelling** Eine dritte, jedoch nur selten verwendete Methode ist physical modelling. In der Natur enstehen Klänge dadurch, dass Objekte zu Schwingungen angeregt werden. Diese Schwingungen breiten sich in der Luft aus und werden von unserem Ohr als Klang interpretiert. Da all dies auf physikalischen Gesetzen beruht, lässt sich, zumindest theoretisch, jeder Aspekt dieser berechnen und der entstehende Klang rekonstruieren. Hierzu wird zunächst ein Modell entwickelt, welches die physikalischen Eigenschaften eines Objektes beschreibt. Ausgehend von diesem Modell können dann Klänge berechnet werden, die unter verschiedenen Einwirkungen auf dieses Objekt entstehen.

Dieses Vorgehen wird beispielsweise, zusammen mit weiteren Syntheseverfahren, in dem VST-Plugin “Pianoteq 6‘ angewendet (Modartt, 2020).

In der Arbeit “Crumpling Sound Synthesis“ wird als Modell eine Animation verwendet, die das Zerknittern eines Objektes abbildet. Durch die Analyse dieser Animation kann der Sound, der beim Zerknittern entsteht, reproduziert werden (Cirio, Li, Grinspun, Otaduy & Zheng, 2016).

Um allerdings einen Klang auf rein pyhsikalischer Basis rekonstruieren zu können, müssen sehr viele Ausgangsfaktoren und Gegebenheiten in Betracht gezogen werden. Dies führt dazu, dass selbst auf leistungstarken Computern diese Berechnungen sehr viel Zeit und Resourcen in Anspruch nehmen können, Dadurch ist dieser Ansatz für eine Realzeitanwendung nur schwer möglich.

# Teil II
# Der Prototyp

## 4 Entwicklung des Prototypen

### 4.1 Anforderungen

Ein Klang kann praktisch gesehen endlos viele Ursachen haben. Zwei Objekte können aufeinander treffen, was sie zum Schwingen anregt. Dies passiert unter anderem bei Kollisionen, Schritten oder Regen. Bei der schnellen Expansion von Gasen können ebenfalls Klänge entstehen. Dies ist zu hören bei Schüssen, Explosionen, Fahrzeugen und Feuer. Ein weiteres Beispiel ist die Bewegung einer Luftsäule welche in oder um einem Objekt in Bewegung versetzt wird. Das geschieht bei einer Orgel, beim Sprechen oder Lüftergeräuschen. In fast allen Fällen sind wahrgenommene Klänge jedoch eine komplexe Mischung aus diesen und noch weiteren Ursachen. Auch wenn all diese theoretisch eine berechenbare, physikalische Grundlage haben, ist es praktisch gesehen mit heutiger Technik so gut wie unmöglich ein universelles Modell zu entwickeln welches alle Ursachen einbeziehen kann. Einer der Gründe hierfür ist, dass ständig sämtliche Berechnungen durchgeführt werden müssen, um das Modell und Klangergebnis realistisch zu gestalten. Dies ist mit einem hohen Rechenaufwand verbunden, der aber für viele Anwendungsbereiche aus praktischer Sicht nicht notwendig ist. Um den Rechenaufwand zu minimieren ist es effizienter, mehrere Modelle zu entwickeln, die auf verschiedene Bereiche spezialisiert sind und unnötige Faktoren außen vor lassen.

Für den Umfang dieser Arbeit soll ein solches Modell für den Klang von Feuer entwickelt werden. Hierbei soll die Grundlage eines Klangerzeugers entstehen, welcher einen Feuersound prozedural generiert und dem Nutzer Parameter wie in Kapitel 5.1.1 beschrieben zur Verfügung stellt. Um die Funktionalität und Anwendbarkeit dieses Generators anhand eines Beispieles zu untersuchen, soll dieser in einer exemplarischen Szene in der Spielengine Unity integriert werden.

### 4.2 Auswahl der Programmiersprache

**Unityskripte** Wie bereits erwähnt, ist es für diese Arbeit das Ziel, den Soundgenerator in Unity zu implementieren. Aus diesem Grund ist die naheliegendste Lösung, für die Programmierung Unityskripte zu verwenden. Die Implementierung wäre ohne weitere Umwege möglich und die Parameter könnten problemlos in allen anderne Skripte der Anwendung bereitgestellt werden. Außerdem könnte der Generator sehr einfach zwischen verschiedenen Unityprojekten ausgetauscht werden.

Der größte Nachteil den dies allerdings mit sich bringt ist, dass der Generator auch nur in Unity selbst benutzt werden kann. Um in die generierten Klänge in anderen Anwendungen zu verwenden, sind zusätzliche Kommunikationswege

wie beispielsweise das MIDI- oder OSC-Protokoll [7] notwendig. Auf diese Weise können Befehle wie das Einstellen der Parameter gesendet werden. Um den Audiobuffer zu übergeben, muss ein virtueller Audiokanal eingerichtet werden. Bei einem solchen Setup müssen sehr viele Komponenten verwendet werden, was zu einer erhöhten Fehleranfälligkeit, erweitertem Programmieraufwand führt und verlängerter Einrichtungszeit führt.

**PureData** PureData (Pd) ist eine der unter Musikern und Sounddesignern bekanntesten visuellen Programmiersprachen (Puckette, 2020). Mit dieser können auf einfache Weise und mit wenig Quellcode Software für verschiedene Zwecke erstellt werden. Der Hauptanwendugsbereich ist jedoch das erstellen von Synthesizern.

Die libPd-library kann die in Pd erstellten Projekte, sogenannte Patches, dann in eine native Bibliothek kompilieren. Für diese Bibliothek kann dann wieder ein Wrapper für die jeweilige Software, in diesem Fall Unity, programmiert werden.

Um diese Wrapper umzusetzen gibt es bereits eine Open-Source Lösung. Die heavy compiler collection (hvcc). Die hvcc benutzt libPd, um aus Pd-Patches Plugins für verschiedene Umgebungen zu kompilieren. Zu diesen gehören unter anderem auch Unity 5 und VST2. Eine Notitz auf der GitHub-Seite des Kompilers weist jedoch darauf hin, dass dieser zur Zeit nicht weiter entwickelt wird (White & Roth, 2018). Offiziell gibt es noch keinen Support für neuere Unityversionen oder VST3.

**Faust** Functional Audio Stream (Faust) ist eine funktionale Programmiersprache [8]. Sie wurde speziell für Audioanwendungen wie Synthesizer, Instrumente und Effekte entwickelt. Faust bringt einen große Sammlung an Kompilern für sehr viele Zielplatformen mit sich. Zu diesen gehören Unity, CSound, PureData, node.js und Octave. Der Syntax von Faust ist sehr minimalistisch was sowohl Vor- als auch Nachteile mit sich bringt. Unter anderem führt dies auch dazu, dass komplexere und stark rekursive Algorithmen sehr schwer in Faust umzusetzen sind (Orlarey, Fober & Letz, 2020).

**Bibliotheken** Bibliotheken (dll für Windows, bundle für Mac) sind dazu gedacht, anderer Software verschiedenen Funktionalitäten bereit zu stellen. Sie stellen eine Schnittstelle bereit, durch welche andere Programme die in der Bibliothek geschriebenen Routinen und Algorithmen verwenden können. Genau wie bei PureData und Faust können dann für die Anwendungen in denen die Bibliothek verwendet werden soll Wrapper geschrieben werden.

[7] Open Sound Control (OSC) ist ein System, welches zur Kommunikation zwischen elektronischen Musikinstrumenten und Computern verwendet wird. Im Gegensatz zum MIDI-Protokoll sind bei diesen unter aderem die Parameter freier definierbar und stellen einen größeren Wertebereich zur Verfügung (Wright, 2002)

[8] Im Gegensatz zu einer imperativen Programmiersprache wird bei einer funktionalen nicht der Zustand sondern das Verhalten von Objekten beschrieben. (Hudak, 1989).

Als sogenannte managed plugins können diese Bibliotheken sehr einfach in Unity implementiert werden. Dort können sie auf ähnliche Weise verwendet werden wie in Unity geschriebene Skripte. Allerdings bringt diese Methode auch Nachteile mit sich. Managed plugins können von Unity nur als solche verwendet werden, wenn sie ausschließlich auf Elemente aus dem .NET Framework zurückgreifen. Andere Abhängigkeiten können nicht verwendet werden (Unity Technologies, 2018a). Eine Alternative für managed plugins sind sogenannte native plugins. Als native Plugin können alle Bibliotheken verwendet werden die in nativem Code (beispielsweise C, C++ oder Objective-C) geschrieben sind. Diese sind in der Lage auf systemeigene Funktionalitäten zuzugreifen, was allerdings zur Folge hat, dass sie auch für jedes Betriebssytem speziell kompiliert werden müssen (Apple Inc., 2017; Microsoft, 2018).

Alle genannten Wege verlaufen recht ähnlich. Zunächst wird mit einer bestimmten Programmiersprache der eigentliche Synthesizer erstellt. Um diesen dann in eine andere Software zu implementieren, muss in einer passenden Sprache ein Wrapper programmiert werden. Für die Umsetzung des Prototypen für diese Arbeit entscheide ich mich dazu, eine Bibliothek in C++ zu schreiben und diese als native plugin in Unity zu testen. Auch wenn die anderen genannten Methoden jeweils nur wenige Nachteile aufzeigen, ist diese im Hinblick auf Flexibilität die mit den geringsten Einschränkungen. Die auf diesem Weg entstandene Bibliothek kann nach der Entwicklung für andere Zielsysteme bereitgestellt werden. Diese sind beispielsweise ein VST-Plugin, ein Plugin für Audiomiddleware wie Wwise oder FMod, Plugins für andere Spielengines oder eine eigenständige Anwendung.

# 5 Analyse

## 5.1 Feuersound

Der Prototyp soll für den Klang von Feuer entwickelt werden. Die Entsccheidung hierzu ist damit begründet, dass dieser Klang aus mehreren einzelnen Elementen besteht, die von unterschiedlichen Faktoren beeinflusst werden. Diese Einflüsse können physikalisch beschrieben und zu verschiedenen abstrakten Parametern zusammengefasst werden. Somit ist der Klang von Feuer sehr gut für eine exemplarische Ausarbeitung des Arbeitsweges für die Entwicklung eines parametrisierten prozeduralen Klangerzeugers geeignet.

Zunächst muss für die Reproduktion des Klanges analysiert werden, aus welchen Elementen dieser besteht. Er kann nach Farnell (Farnell, 2010, S. 409-411) unter anderem auf folgende Geräusche heruntergebrochen werden.

**Flammensound** Feuer erwärmt Luft, welche daraufhin aufsteigt. Weiter vom Feuer entfernt jedoch ist die Luft etwas kälter. Die aufgestiegene warme Luft kühlt sich wieder ab und sinkt. Diese Luftzirkulation erzeugt um das Feuer herum eine Luftsäule. Durch die Geschwindigkeit, mit welcher die Luft aufsteigt

und die Unregelmäßigkeit des Feuers, welche durch Wind oder ungleichmäßige Platzierung der Brandquellen entsteht, wird diese Luftsäule in Schwingung versetzt. Dies ist durch ein tiefes Rauschen zu hören.

Ein Feuer hat in den meisten Fällen mehr als nur eine Brandquelle. Im Beispiel eines Lagerfeuers können dies mehrere Stücken Holz sein. Jede Brandquelle hat ihre eigenen Dimensionen und deswegen eigene Resonanzen. Dies wird dadurch hörbar, dass sich sowohl dieses Rauschen, als auch alle später beschriebenen Klänge sich mit verschiedenen unterschiedlichen Grundfrequenzen überlagern.

Ein weiterer bestimmender Faktor für dieses Geräusch ist die Größe der Brandquelle. Je kleiner sie ist, desto kleiner ist auch die Luftsäule. Dadurch wird die Grundfrequenz dieses Geräusches höher.

Auch die Wärme des Feuers spielt eine große Rolle bei diesem Klang. Bei einem heißeren Feuer wird die Luft mehr erwärmt, was dazu führt, dass sie schneller aufsteigt. Im Grundrauschen des Feuers sind dadurch mehr Obertöne zu hören.

**Zischen** In einer Brandquelle befinden sich häufig Flüssigkeiten wie Wasser und Harz (vor allem in Holz). Wenn diese Flüssigkeiten verdampfen und Austreten, wird ein Zischen hörbar. Besonders deutlich ist dieser Effekt zu hören, wenn man in eine Kerze ein wenig Wasser tropft und sie dann anzündet. Aufgrund der unregelmäßigen Verteilung und das strkutkurell bedingte ungleichmmäßige Austreten der Gase ist dieses Geräusch in sehr zufälligen Abständen zu hören.

Die Menge und Häufigkeit des Zischens ist aber nicht nur von der Menge an eingeschlossener Flüssigkeit abhängig. Hier spielt vorallem auch die Temperatur eine Rolle. Während kleinere Flüssigkeitsspeicher bei niedrigen Temperaturen recht schnell verdampft werden können, ist es wahrhscheinlich das größere Reserviours mehr Zeit und höhere Temperaturen benötigen, um auf diese Weise ein Zischen zu erzeugen.

**Knistern** Ein besonders charakteristisches Geräusch gerade für Kamin- oder Lagerfeuer ist das Knistern und Knacken. Dieses Geräusch entsteht, wenn in der Brandquelle eingeschlossene Gase explosionsartig freigelassen werden. Aus diesem Grund ist auch häufig zu hören, dass einem Knacken ein Zischen folgt. Das Gas sucht sich zunächst kraftvoll einen Weg ins Freie und entweicht dann nach und nach. Da dieses Geräusch ebenfalls von der Feuchtigkeit der Brandquelle beeinflusst wird, ergibt sich eine ähnliche Abhängigkeit der Werte wie beim Zischen. Allerdings ist zu beachten, dass für das explosionsartige Austreten von Gasen je nach struktureller Stabilität der Brandquelle mehr Druck erforderlich ist. Da für ein Knacken mehr Kraft gebraucht wird als für ein Zischen, ist zu beobachten, dass bei kälteren Feuern mehr Zischen und nur leise und kleine Knackser zu hören sind als bei heißerem Feuer.

Natürlich gibt es bei einem Feuer noch wesentlich mehr Geräuschquellen, wie zum Beispiel spritzende oder tropfende Flüssigkeiten, wenn diese aus der

Brandquelle austreten, ohne ganz gasförmig zu sein, oder das Geräusch, welches entsteht, wenn die Brandquelle unter der strukturellen Belastung zusammenbricht. Diese Geräusche bilden allerdings einen sehr geringen Anteil. Strukturelle Belastungen treten nur spontan, nach langer Zeit sehr kurz auf. Tropfende Flüssigkeiten sind sehr leise und im Gesamtklang nur schwer auszumachen, je nachdem auf was für einem Untergrund sich das Feuer befindet und wie laut die Umgebung oder das Feuer selbst sind. Spritzende Flüssigkeiten klingen dem Zischen sehr ähnlich das durch die austretenden Gase entsteht. Alles in allem muss später im konkreten Anwendungsfall beurteilt werden, ob die zusätzlich benötigte Rechenleistung für das gewünschte Ergebnis gerechtfertigt ist.

### 5.1.1 Parameter

Nachdem die Grundklänge beschrieben und die Faktoren, die diese beeinflussen ermittelt wurden, gilt es nun, Parameter für die Prozedur zu bestimmen. Wie im Kapitel 3.3 erläutert, ist es wichtig, dass diese Parameter nicht die Prozedur selbst verändern, sondern nur die äußeren Umstände beschreiben, von denen die Prozedur abhängig ist. Anders ausgedrückt sollen Parameter gefunden werden, mit denen ein reales Feuer ebenfalls beschrieben werden könnte.

**Größe** Der Flammensound ensteht durch die sich bewegende Luftsäule um das Feuer herum. Diese Säule hängt sehr stark von den Ausmaßen des Feuers ab. Je größer diese sind, desto tiefer ist der zu hörende Klang. Desweiteren liegt In einem größeren Feuer häufig mehr Brandmaterial vor. Eine größere Menge an Brandmaterial bedeuted, dass es zu mehr Zischen und Knacken kommen kann. Aber nicht nur die Menge, sondern auch der eigentliche Klang dieser beiden Sounds wird durch die Größe des Feuers beeinflusst. Je kleiner das Feuer ist, desto kleiner ist auch das Brennmaterial. Ein kleineres Stück Holz hat wesentlich weniger und kleinere Mengen an eingeschlossenen Flüssigkeiten. Die daraus resultierenden Knackser und Zischer werden sehr viel weniger, kürzer und höher ausfallen. Die Größe spielt also bei der Klangentwicklung des Feuers eine essenzielle Rolle.

**Feuchtigkeit** Speziell für Kaminfeuerholz wird aus verschiedenen Gründen oft Holz benutzt, welches schon eine Weile trocken gelagert ist. Bei einem solchen Feuer entsteht allerdings ein Klang, der von dem eines Lagerfeuers mit frisch gesammeltem Holz relativ gut zu unterscheiden ist. Je trockener das Holz ist, desto weniger Flüssigkeiten sind in ihm eingeschlossen. Dies führt zu einem verringertem Auftreten von Knacken und Zischen. Allerdings werden diese beiden Klänge unterschiedlich stark beeinflusst. Vergleicht man beide bei einem frischen Stück Holz, stellt man fest, das sie relativ gleich stark vertreten sind. Bei einem etwas trockenerem Stück Holz hingegen, sind die weniger stark eingeschlossenen Feuchtigkeitsreservoirs bereits verdunstet, was dazu führt, das weniger Zischen auftrit. Stärker eingeschlossene Reservoirs, welche für das Knacken verantworlich sind, sind hingegen noch vorhanden. Deswegen ist bei einem Kaminfeuer

mit getrocknetem Holz weniger Zischen, aber vergleichsweise mehr Knacken zu hören.

**Bewegung** Für Geräusche von Fackeln hat dieser Parameter eine besonders große Bedeutung, da sie, vom Spieler getragen, immer in Bewegung sind. Beim Bewegen einer Fackel entsteht ein sehr charakteristisches, lauteres Rauschen. Dieses Rauschen entsteht dadurch, dass beim Bewegen der Fackel die Luft um das Feuer herum wesentlich schneller durch die umgebende Luftsäule bewegt wird. Hierdurch entstehen mehr Obertöne und das Grundrauschen wird lauter. Das selbe Ergebnis ist auch zu hören, wenn eine Flamme durch Wind bewegt wird. Als gutes Beispiel hierfür dient eine Kerze. Eine Kerze die vollkommen ruhig ist produziert so gut wie keine Klänge. Dies liegt zum einen daran, dass die Flammensäule sehr klein ist und zum anderen daran, dass aufgrund der geringen Wärme die Luft sich ohnehin nur sehr langsam durch diese Säule bewegt. Pustet man jedoch vorsichtig auf die Kerze, so kann dieses Rauschen hörbar gemacht werden. Je kräftiger die Luft auf diese Weise bewegt wird, desto stärker ist das Rauschen und die dadurch entstehenden Obertöne.

**Weitere Parameter** Neben diesen Parametern gibt es natürlich noch einige Weitere, die unter Umständen von Relevanz sein können. Dazu gehören beispielsweise das Alter und die Wärme des Feuers. Diese haben allerdings nur eine indirekte Auswirkung auf den Klang. Je älter ein Feuer ist, desto länger brennt es bereits. Das bedeuted, dass das Brandmaterial bereits weniger geworden und deutlich trockener ist. Dies führt zu einer geringeren Feuchtigkeit. Außerdem kann dies auch zu einer erhöhten Hitze führen. Bei einem wärmeren Feuer wiederum steigt die Luft wesentlich schneller auf, was sich in einer schnelleren Bewegung wiederspiegelt. Desweiteren ist es sehr wahrscheinlich dass, je nach Alter des Feuers, trockeneres Brandmaterial vorliegt.

Alles in allem bedeuted das, dass diese Parameter sowohl untereinander als auch von anderen Parametern abhängig sind. Sie verändern nicht die Prozedur selbst, sondern nur die Parameter dieser. Aus diesem Grund ist es sinnvoller, sie nicht im Synthesizer selbst einzubinden, sondern bei Bedarf die Parameter, die einen direkten Einfluss auf die Prozedur haben, entsprechend im Wrapper zu kombinieren.

# 6 Programmierung

## 6.1 Programmierung des Synthesizers

Nachdem die Bestandteile und Einflüsse der Prozedur bestimmt sind, müssen diese wie in Kapitel 4.2 erörtert programmiert und in eine Bibliothek kompiliert werden. Um die mit der getroffenen Auswahl erreichte Flexibilität weiterhin zu gewährleisten, soll auf weitere externe Abhängigkeiten für den Synthesizer verzichtet werden. Somit kann gewährleistet werden, dass er auf allen Betriebs- und Zielssystemen verwendet werden kann.

Zunächst sollen die in Kapitel 5 analysierten Bestandteile programmiert werden. Für das Grundrauschen des Feuers ist lediglich ein gefiltertes weißes Rauschen notwendig. Die Grundbestandteile der anderen beiden Klänge sind ebenfalls ein Rauschen. Dieses wird in zufälligen Abständen mit einer Hüllkurve moduliert und ebenfalls gefiltert. Je nach Form der Hüllkurve kann somit entweder das Zischen oder das Knacken erzeugt werden.

Da das weiße Rauschen bereits eine Reihe an zufälligen Zahlen ist, können diese als Ausgang für die zufällige Bestimmung der Hüllkurven verwendet werden. Für das Knacken und das Zischen werden jeweils das selbe Rauschen und damit auch die selben Zufallszahlen verwendet. Da das Knacken durch das schlagartige öffnen von eingeschlossenen Flüssigkeitskammern entsteht ist es sehr wahrscheinlich, dass nach einem solchen Knacken ein Zischen folgt, welches durch das entweichen der restlichen Flüssigkeit entsteht. Durch die Verwendung der gleichen Zufallszahlen kann dieser Zusammenhang in der Prozedur ebenfalls hergestellt werden.

Nachdem die einzelnen Klangquellen erzeugt sind, werden mehrere Instanzen dieser erstellt und mit unterschiedlichen Filtern zusammengemischt. Somit können mehrere Brandquellen dargestellt werden und die klanglich wahrgenommenen Ausmaße des Feuers wirken größer.

Nachdem die Prozedur gestaltet ist, müssen die in Kapitel 5.1.1 festgelegten Parameter mit den entsprechenden Werten der Effekte verbunden werden. Die Größe beeinflusst hauptsächlich das Rauschen des Feuers. Hierfür muss lediglich der Filter verändert und die Lautstärke angepasst werden. Außerdem wird der Schwellwert für die Zufallszahlen verändert, der die Menge der anderen beiden Geräusche bestimmt. Je kleiner das Feuer ist, desto seltener sollen diese zu hören sein.

Auf ähnliche Weise wird der Parameter Feuchtigkeit eingebunden.

Um die Obertöne zu erzeugen, die durch die Bewegung des Feuers entstehen, kann das Rauschen übersteuert und dann in der Lautstärke angepasst werden.

## 6.2 API der Bibliothek

Um die Bibliothek nun in andere Umgebungen einzubinden, muss ein sogenanntes application programming interface (API) bereitgestellt werden. Diese API stellt die Schnittstelle zwischen der Bibliothek und der Zielplatform oder auch anderen Bibliotheken dar. In dieser Schnittstelle müssen verschiedene Möglichkeiten bereit gestellt werden.

Das reine Einbinden der Bibliothek garantiert noch nicht, dass die Objekte, welche die Bibliothek erzeugt, verwendet werden können. Zuerst wird also eine Mehtode benötigt, welche eine Instanz des eigentlichen Synthesizers erzeugt. [9]

Diese Instanz muss auf irgend eine Weise in der Programmiersprache des Wrappers gespeichert werden um sie später weiter verwendenden zu können.

[9] Da das Erstellen von Objekten Speicherplatz belegt, muss dieser auch wieder freigegeben werden, wenn er nicht mehr benötigt wird. An dieser Stelle ist es also auch sinnvoll, eine Methode zum Löschen des Objektes bereitzustellen, auch wenn viele moderne Programmiersprachen unbenutzte Objekte selbständig löschen.

Da aber nicht alle Programmiersprachen die selben Variablentypen unterstützen, muss hierfür ein kleiner Umweg genommen werden. In diesem Beispiel ist der Synthesizer vom Typ 'Fire'. Hierbei handelt es sich um einen sogenannten nicht-primitiven Datentyp, dass heißt, es ist ein Datentyp der nicht standardmäßig in der Programmiersprache vorhanden ist sondern von einem Programmierer erstellt wurde. Um die Kommunikation zwischen zwei Sprachen zu ermöglichen, sollte auf primitive Datentypen zurückgegriffen werden. Hierbei handelt es sich um Datantypen, die so grundlegend sind, dass sie in fast allen Programmiersprachen vorhanden sind. [10] Zu diesen primitiven Datentypen gehören unter anderem auch Referenzen. Im Gegensatz zum Objekt selbst handelt es sich bei einer Referenz um die Adresse des Speichers, in dem das Objekt abgelegt wurde („Reference (computer science)", 2020).

Beim Erstellen einer Instanz des Synthesizers gibt die Bibliothek somit nicht das gesamte Objekt, sondern eine Referenz auf dieses an die Zielplatform zurück. Diese Referenz kann dann innerhalb des Wrappers gespeichert werden. Soll vom Wrapper aus nun ein Wert am Synthesizer verändert oder ein Audiobuffer mit Hilfe des Synthesizers befüllt werden, muss lediglich diese Referenz mit übergeben werden, um der Bibliothek mitzuteilen, für welche Instanz des Synthesizers diese Änderung vorgenommen werden soll. Die Referenz dient sozusagen als Erkennungszeichen zwischen dem Wrapper und dem Synthesizer. Ein weiterer Vorteil der auf diese Weise entsteht ist, dass problemlos mehrere Instanzen des Synthesizers erstellt werden können, die voneinander vollkommen unabhängig sind. Es ist nicht notwendig, die Bibliothek mehrfach zu importieren.

Wenn das Objekt instanziiert wurde, wird allerdings noch kein Klang erzeugt. Damit dies geschieht, muss seitens der Platform ein Audiobuffer vom Synthesizer angefordert werden. Hierzu stellt die Bibliothek eine Methode zur Verfügung, die die gewünschte Länge des Buffers übergeben werden kann. Der Synthesizer kann nun einen Buffer erstellen und füllen und gibt diesen an die Zielplatform zurück. Auf diese Weise liegt die Verwaltung des Audiomaterials in der Hand der Zielplatform und muss nicht erst innerhalb der Bibliothek konfiguriert werden. Außerdem können weitere Verarbeitungsmethoden wie Spatialisierung oder diverse Effektgeräte ebenfalls außerhalb der Bibliothek verwendet werden. Ein weiterer Vorteil, der dadurch entsteht ist, dass das Audiomaterial nur zur benötigten Zeit generiert wird, wodurch nicht benötigte Resourcen frei bleiben können.

Zuletzt werden noch Methoden benötigt, welche von der Zielplatform verwendet werden können, um die Parameter der Prozedur zu verändern.

Mit einer solchen Schnittstelle steht nun eine Bibliothek bereit, die in ein Spiel eingebunden werden kann. Für erfahrene Audiodesigner ist es allerdings sehr warhscheinlich, dass diese Parameter nicht ausreichend sind, und sie mehr Kontrolle über den Synthesizer möchten. Soll diese Bibliothek als VST-Plugin

[10] Es ist äußerst schwer, für diese Tatsache eine Quelle anzugeben, da es sich nicht um einen definierten Standard handelt, sondern um eine Gegebenheit, die sich im Laufe der Zeit so entwickelt hat. Viele Foren und Dokumentationen zu Programmiersprachen verweisen auf diese Tatsache und in allen modernen Programmiersprachen sind Vertreter dieser sogenannten primitiven Datentypen zu finden.

oder Plugin für Audiomiddleware bereitgestellt werden, können noch Methoden in die API eingebunden werden, die einen direkten Einfluss auf die Prozedur oder einzelne Klänge ermöglichen. Zu diesen zählen beispielsweise Filter- und Hüllkurvenparameter. Außerdem können dem Audiodesigner Möglichkeiten zur Verfügung gestellt werden, den Einfluss der Prozedurparameter auf die Klangentwicklung anzupassen.

Für den beispielhaften Prototypen sieht diese API nun wie folgt aus:

Listing 1: API

```
1
2       __declspec(dllexport) Fire* createFire(float sampleRate)
            {
3           Fire* fire = new Fire(sampleRate);
4           fire->init(sampleRate);
5           return fire;
6       }
7
8       __declspec(dllexport) std::vector<float>* getAudio(int
            numberOfSamples,Fire* fire){
9           auto a = new std::vector<float> (fire->getAudio(
                numberOfSamples));
10          return a;
11      }
12
13      __declspec(dllexport) float getBufferContent(std::vector
            <float>* buffer, int position){
14          return buffer->at(position);
15      }
16
17      __declspec(dllexport) void setMoist(Fire* fire, float
            value){
18          fire->setMoist(value);
19      }
20
21      __declspec(dllexport) void setSize(Fire* fire, float
            value){
22          fire->setSize(value);
23      }
24
25      __declspec(dllexport) void setMovement(Fire* fire, float
             value){
26          fire->setMovement(value);
27      }
```

## 6.3 Beispielhafte Implementierung in Unity

Um die Bibliothek in Unity zu verwenden, müssen folgende Schritte vorgenommen werden:

1. Implementierung der Bibliothek in Unity
2. Erstellung/Importierung des Wrapperskripts um den Generator in der Engine bereit zu stellen.
3. Anhängen des Wrappers an das Objekt im Spiel (GameObject) um dieses als Soundquelle zu definieren.
4. Einstellung der Parameter auf gewünschte Ausganswerte und Verknüpfung mit Spielvariablen.

### 6.3.1 Implementierung der Bibliothek in Unity

Nachdem die Bibliothek für das Zielsystem kompiliert wurde, kann sie in Unity wie ein normales Asset importiert werden. Um allerdings auf die Inhalte der Bibliothek zugreifen zu können, muss erst noch deklariert werden, was die Inhalte dieser Bibliothek überhaupt sind. Die Deklaration der verfügbaren Funktionen funktioniert in C# wie folgt: Zunächst wird mit dem Attribut [DllImport(libraryname)] angegeben, dass die in der darauf folgenden Zeile beschriebene Methode in einer Bibliothek (dll) mit dem namen "libraryname" gefunden werden soll. Diese Methode kann wie gewohnt mit Modifizierern und dem zusätzlichen Schlüsselwort 'extern' definiert werden. Wichtig ist hierbei das der Name, Rückgabewert und Parameter der Funktion mit der in der API überein stimmt (Unity Technologies, 2020b). Die Deklaration der Methode zum Erstellen einer Instanz des Synthesizers sieht demnach wie folgt aus:

Listing 2: Einbindung der Bibliothek

```
1    private const string LibraryName = "FireGeneratorDyLib";
2
3    [DllImport(LibraryName)]
4    private static extern IntPtr createFire(float sampleRate
         );
```

Im Rest des Scripts kann die Funktion 'createFire(sampleRate)' nun wie gewohnt aufgerufen werden.

### 6.3.2 Erstellung des Wrapper Scripts

GameObjects können in Unity als Schallquellen (AudioSource) verwendet werden. Unity bietet mehrere Möglichkeiten, auf von AudioSources generierte Audimaterial zuzugreifen. Eine dieser Methoden ist, die 'OnAudioFilterRead' Callbackfunktion zu verwenden. Als Parameter stellt sie neben der Anzahl der gefragten Kanäle auch den aktuellen Audiobuffer zur Verfügung. Dieser kann hier direkt beschrieben werden.

Das Problem mit dieser Methode ist, dass die Berechnung des Einflusses der Positionierung des Objektes auf das Audiosignal bereits *vor* dem Aufruf dieses Callbacks geschiet (Unity Technologies, 2019). Das bedeuted, wenn der Audiobuffer innerhalb dieser Methode überschrieben wird, geht die akkustische Positionierung verloren.

Neben dieser Callbackfunktion bietet Unity auch ein SDK an, welches speziell für Audioplugins entwickelt ist (Unity Technologies, 2018b). Allerdings handelt es sich bei Audioplugins für Unity um Effekte welche in einen Kanalweg implementiert werden müssen. Sie beziehen sich also nicht auf einzelne GameObjects sondern auf Signalpfade. Dies führt zum einen dazu, dass ein wesentlich komplexeres Setup benötigt wird, um einem einzelnen GameObject eine Schallquelle zuzuweisen und zum anderen, dass ebenfalls die Positionierung dieses GameObjects verloren geht.

Eine dritte Möglichkeit ist das Verwenden einer Callbackfunktion innerhalb eines Samples (in Unity als AudioClip verwendet). In Unity können AudioClips in Runtime erstellt und manipuliert werden. Hierzu wird die Funktion 'AudioClip.Create' angeboten und unter anderem kann an diese eine Callbackfunktion übergeben werden, welche jedes mal aufgerufen wird, wenn der Inhalt des AudioClips verwendet wird (Unity Technologies, 2020c).

Mit dieser Methode ergibt sich der folgende Arbeitsweg: Beim Instanzieren des Synthesizers wird an das GameObject, welches diesen verwendetn soll ein AudioSource angehängt. Diese wird von Unity als Schallquelle betrachtet und entsprechend der Umgebung spatialisiert. Daraufhin kann ein AudioClip erstellt und an diese AudioSource übergeben werden. Beim Erstellen des AudioClips wird eine Callbackfunktion geschrieben, in der mithilfe der Biblothek ein AuidoBuffer generiert werden kann, der das Sample im AudioClip überschreibt.

Dieses Verhalten betrachtet ein GameObject als Schallquelle und ermöglicht somit, die Spatialisierung zu erhalten und gegebenenfalls weitere Effekte auf den Klang anzuwenden. Aus diesem Grund soll für die Programmierung des Wrappers diese Methode angewendet werden.

Als nächstes muss das Wrapperskript so angepasst werden, dass es das GameObject entsprechend vorbereitet. Zudem wird die Callbackfunktion an den AudioClip übergeben.

Listing 3: Synthesizerkomponente

```
1       private void Start()
2       {
3           //Create Fire
4           _sampleRate = AudioSettings.outputSampleRate;
5           _campFire = createFire(_sampleRate);
6
7           //Configure AudioSource
8           var tempClip = AudioClip.Create("FireSample",
                _sampleRate, 1, _sampleRate, true, OnAudioRead);
9           if( (_audioSource = GetComponent<AudioSource>())==
                null)_audioSource = gameObject.AddComponent<
```

```
            AudioSource>();
10          _audioSource.clip = tempClip;
11          _audioSource.loop = true;
12          _audioSource.spatialBlend = 1.0f;
13          _audioSource.Play();
14
15          //initialize Fire
16          setMoist(_campFire, fireMoist);
17          _lastMoist = fireMoist;
18
19          setSize(_campFire, fireSize);
20          _lastSize = fireSize;
21
22          setMovement(_campFire, fireMovement);
23          _lastMovement = fireMovement;
24      }
```

Listing 4: Callbackfunktion

```
1       private void OnAudioRead(float[] data)
2       {
3           var numberOfSamples = data.Length;
4           var tempBuffer = getAudio(numberOfSamples, _campFire
                );
5           for(var i = 0; i<numberOfSamples; i++)
6           {
7               var sample = getBufferContent(tempBuffer, i);
8               data[i]=sample;
9           }
10      }
```

Zuerst wird eine Instanz des Synthesizers erstellt und die Referenz gespeichert (Listing 3, Zeile 5). Daraufhin wird ein leerer AudioClip erstellt, an welchen die Callbackfunktion ‘OnAudioRead‘ (Listing 4) übergeben wird (Listing 3, Zeile 8)

Um diesen AudioClip als Teil des GameObjects zu verwenden, muss eine AudioSource vorhanden sein. In dieser können alle relevanten Parameter (Spatialisierung, Routing, Lautstärkeanpassung) für das Abspielen des AudioClips angepasst werden. Sollte noch keine AudioSource vorhanden sein, wird diese erstellt und an das GameObject angehängt (Listing 3, Zeile 9). In den Zeilen 10 - 13 (Listing 3) wird die AudioSource für diesen Anwendungsfall wie folgt konfiguriert:

- loop wird auf true gesetzt. Dies ist notwendig, damit der Klang dauerhaft zu hören ist.
- spatialBend wird auf 1.0f gesetzt, damit Unity die Positionierung des GameObjects bei der Wiedergabe des Klanges einbezieht.

- playOnAwake wird auf true gesetzt, um sicher zu stellen, dass der Sound wiedergegeben wird, wenn das GameObject in der Szene aktiv ist.
- Play() wird aufgerufen um den AudioClip zu starten.

## 6.4 Einstellung der Parameter und Zuweisung der Werte

Die letzten Schritte der Implementierung bestehen darin, die Parameter der Prozedur in Unity zur Verfügung zu stellen. Je nach Notwendigkeit für das Projekt gibt es hierfür zwei Möglichkeiten. Um während der Entwicklung des Projektes die Werte eines Skripts einfach zur Verfügung zu stellen, bietet Unity die Möglichkeit, eigene grafische Oberflächen (GUIs) innerhalb dieser Skripte zu erstellen (Unity Technologies, 2020a). Eine solche GUI ist nach der Erstellung im Inspektor in Unity zu sehen und ermöglicht es, Werte innerhalb des Skriptes sowohl beim editieren als auch zur Laufzeit anzupassen, ohne dass das Skript neu kompiliert werden muss. Um die Parameter allerdings generativ und im Zusammenhang mit dem gesamten Projekt verwenden zu können müssen zunächst Wrapperfunktionen im öffentlichen Gültigkeitsbereich (public) erstellt werden.

Listing 5: Bereitstellung der Parameter

```
1     public void SetFireMoist(float value)
2     {
3         fireMoist = value;
4         setMoist(_campFire, fireMoist);
5         _lastMoist = fireMoist;
6     }
```

Diese Methode speichert zunächst den übergebenen Wert. Als nächstes wird, in diesem Beispiel, mit 'setMoist(_campFire, fireMoist)' der gewünschte Parameter innerhalb der Bibliothek angepasst. Hierbei ist '_campFire' die Referenz auf die von der Bibliothek erstellte Instanz des Synthesizers und 'fireMoist' der Zielwert des Parameters.

Nachdem das Wrapperskript einmal erstellt wurde, kann es innerhalb von Unity als Asset behandelt werden. Soll ein weiteres GameObject als Klangquelle für Feuer dienen, muss dieses Skirpt nur angehängt werden. Alle weiteren Einstellungen werden automatisch vorgenommen.

Desweiteren kann dieses Skript nun zwischen verschiedenen Unityprojekten ausgetauscht werden. Hierzu muss lediglich die Bibliothek in das Projekt importiert werden.

# 7 Fazit

**Performance** Einer der ausschlaggebensten Punkte bei der Verwendung von prozedural generiertem Sound ist das Einsparen von Speicherplatz auf Kosten

erhöhter Prozessorbeanspruchung. Dieser Punkt hat sich bei dem Prototypen sehr stark gezeigt. Was bei dem Prototypen das eigentliche Problem darstellt, ist zum einen die Tatsache, dass um einen komplexeren Klang zu bilden, mehrere Brandquellen mit unterschiedlichen Grundklängen simuliert wurden und vorallem dass alle diese Quellen eine sehr große Menge an Filtern verwenden. Diese Filter sind verhältnismäßig rechenintensiv. Bei der Analyse der Laufzeitperformance des Synthesizers ist zu sehen, das über 70% des Rechenzykluses nur für das Anwenden der Filter aufgebracht werden muss. Außerdem werden, um ein variables Klangergebnis zu erzeugen, die Koeffizenten einiger Filter zur Laufzeit verändert. Da der entstandene Klang (durch das Rauschen simuliert) hauptsächlich durch die Umgebung und den Erzeuger (durch die Filter simuliert) beeinflusst wird, kann durch diesem Arbeitsweg der Prozess so wie er in der Realität stattfindet wesentlich naturgetreuer nachgebildet werden. Um allerdings ein performanteres und ausgeglicheneres Ergebnis zu erzielen, sollten noch andere Synthesemethoden verwendet werden. Für diesen Prototypen wäre beispielsweise Wavetable- und granulare Synthese denkbar. Für den Flammensound könnte ein Rauschen mit einer bereits gefilterten Grundfrequenz erzeugt werden. Durch das verändern der Abspielgeschwindigkeit könnte dann zu einem gewissen Grad diese Grundfrequenz verändert werden. Auf ähnliche Weise können sowohl die Knackser als auch in bestimmtem Umfang die Zischer nachgebildet werden. Somit könnten einige der rechenintensiven Filter eingespart werden, ohne zu viel an Soundqualität und vorallem Prozedurkompatibilität zu verlieren.

Innerhalb der exemplarischen Szene ist jedoch anhand einer Performanceanalyse zu erkennen, dass der Rechenaufwand für den Synthesizer im Vergleich zu allen anderen seitens der Engine benötigten Prozessen sehr gering ist. Somit können für mehrere verschiedene Arten von Feuer (Lagerfeuer, Fackeln oder Brände) gleichzeitig die selbe Bibliothek verwendet werden. Das Importieren mehrerer vorgefertiger Samples und Reservieren von mehr Speicherplatz entfällt.

**Designprozess** Die Grundlage der von dem Prototypen durchgeführten Prozedur bildet nach wie vor, wie bei nahezu allen künstlich erzeugten Klängen ein Synthesizer. Um mit einem solchen Synthesizer den gewünschten Klang zu erzeugen, muss dieser vorerst konfiguriert und später auf die prozedurbeeinflussenden Parameter angepasst werden. Der gesamte Designprozess des Klanges hat sich für diesen Prototypen als relativ umständlich erwiesen. Auch wenn ich ein Interface erstellt habe, mit welchem einzelne Werte ausprobiert und angepasst werden konnten, mussten diese Werte dann noch von Hand in den Syntheszier eingetragen und dieser neu kompiliert werden. Gerade die Wertebereiche zu bestimmen in welchen die Prozedurparameter die Designparameter beeinflussen sollten, beinhaltete sehr viel ausprobieren. Aus rein zeitlicher Sicht, war der gesamte Vorgang recht ineffizient. Eine Lösung hierfür, wäre ein Interface zu erstellen, welches nicht nur die Prozedurparameter, sondern auch Designparameter zur Verfügung stellt. Bei der Erstellung der Bibliothek können

somit wesentlich einfacher Wertebereiche gefunden werden, in denen sich die Parameter bewegen sollen. Diese Werte können dann in einer zusätzlichen Konfigurationsdatei gespeichert werden, welche im letzten Schritt in die Bibliothek kompiliert werden kann. Während der Programmier- und Designprozess bisher Hand in Hand gingen und gleichzeitig stattfanden, könnten auf diese Weise die beiden Prozesse voneinander getrennt werden. Das erleichtert nicht nur den Arbeitsfluss, sondern bietet auch die Möglichkeit, mit mehreren Leuten an dieser Bibliothek und der Prozedur zu arbeiten.

**Implementierung** Das Ziel dieses Prototypen war es, eine Prozedur bereitzustellen, mit der ein bestimmter Klang erzeugt und beeinflusst werden kann. Die testweise Implementierung in Unity zeigte, dass das Einbinden der Bibliothek mithilfe eines Skriptes sehr einfach gelang. Nach der Importierung der Bibliothek kann die gesamte Funktionalität dieser mit einem Wrapperskript bereitgestellt werden. Die in der Bibliothek zur Verfügung stehenden Parameter können problemlos an alle anderen Objekte in Unity übergeben werden. Somit kann die gesamte Prozedur zur Laufzeit beeinflusst werden.

Durch die Entscheidung, den Synthesizer in einer Bibliothek zu Programmieren erhöht sich die Einsetzbarkeit ebenfalls sehr stark. Vorallem in C-ähnlichen Programmiersprachen, aber auch in Java und sogar JavaScript ist es recht einfach, native Bibliotheken zu importieren. Auf ähnliche Weise wie in der beispielhaften Integration in Unity mithilfe von C# gezeigt, kann die Bibliothek in andere Wrapper wie VST-Plugins oder als Standalone implementiert werden. Aufgrund der simpel gehalten API stellt es auch kein Problem dar, sie in Spiele zu integrieren, die keine bereits fertige Engine verwenden. Es wäre sogar begründet zu behaupten, dass die Implementierung in eine solche Engine wesentlich einfacher ist. Wie in Kapitel 6.3.2 bereits erwähnt, muss bei der Implementierung in Unity ein kleiner Umweg genommen werden, um den Audiobuffer an der richtigen Stelle verwenden zu können. Dieser Umweg kann, je nach Programmierung der Engine, in einer anderen wegfallen.

# 8 Nachwort

Während der Entwicklung der Bibliothek wurde ich öfter gefragt, warum ich nicht einen einfacheren Weg, wie zum Beispiel das Entwickeln in PureData nehme. Der Grund hierfür war meine Überzeugung, dass, auch wenn in PureData ein einfacherer Designprozess möglich ist, die eigentliche Implementation in andere Umgebungen viel mehr Arbeit mit sich zieht als der von mir gewählte Weg. Der Schwerpunkt dieser Arbeit lag allerdings auf dem gesamten Workflow der Programmierung und nicht auf dem eigentlichen Sounddesign des Generators. Wenn auch das klangliche Ergebnis zur Zeit nicht so ansprechend ist, wie es mit anderer Generatoren oder gar eines aufgenommenen Samples möglich gewesen wäre, ist dies nur eine Frage des Aufwands der in den eigentlichen Designprozess, nicht aber in die gesamte Bibliothek gesteckt wird. Mit den im Kapitel 7 erläuterten Erkenntnissen über den Designprozess, kann die Bibliothek so umstrukturiert werden, dass sie diesen Prozess wesentlich vereinfacht und möglicherweise sogar für weitere Generatoren generalisiert werden kann. Denkbar wäre es auch, eine grafische Umgebung zu entwickeln, welche Sounddesignern ermöglicht, auf intuitive Weise Prozeduren für alle Arten von Klängen zu gestalten, zu parametrisieren und diese dann automatisch als native Bibliothek zu exportieren. Der Arbeitsaufwand würde sich für das Sounddesign und die Implementierung in vielen Bereichen nicht nennenswert erhöhen. Das Ergebnis jedoch würde gegenüber einem samplebasiertem Ansatz eine große Menge an Vorteilen und erweiterter Einsetzbarkeit bieten. Der in dieser Arbeit beschriebene Weg zur prozeduralen, parametrisierten Klangerzeugung könnte also mit einer solchen Umgebung sehr stark vereinfacht werden.

Meiner Ansicht nach wäre dies ein Schritt in eine Richtung, der bereits vor einiger Zeit in früheren Heimkonsolen und Arcadeautomaten genommen, dann aber mit steigender Qualität von Sampleaufnahmen wieder zurückgegangen wurde. Genau wie sich eine virtuelle Umgebung und die Interaktion mit dieser *optisch* weiter entwickelt, sollte auch der akustische Aspekt solcher Welten nicht auf einer Technik stehen bleiben, die vor über 20 Jahren entwickelt wurde.

# Literatur


*Action-Adventure*. (2019, 17. November), In *Wikipedia*. Page Version ID: 194132219. Zugriff 20. Januar 2020 unter https://de.wikipedia.org/w/index.php?title=Action-Adventure&oldid=194132219

Apple Inc. (2017, 27. März). About Bundles. Zugriff 25. März 2020 unter https://developer.apple.com/library/archive/documentation/CoreFoundation/Conceptual/CFBundles/AboutBundles/AboutBundles.html

Audiokinetic. (2019). Creating a random container. Zugriff 28. März 2020 unter https://www.audiokinetic.com/library/edge/?source=Help&id=creating_random_container

Bain, M. (2010, 22. Januar). BUG – tuned city. Zugriff 6. Januar 2020 unter http://www.tunedcity.net/?page_id=191

Cirio, G., Li, D., Grinspun, E., Otaduy, M. A. & Zheng, C. (2016). Crumpling Sound Synthesis. *ACM Trans. Graph.*, *35*(6).

Collins, Karren. (2008). *Game Sound - An Introduction to the History, Theory and Practice of Video Game Music and Sound Design*. Massachusetts: MIT Press books.

Collins, Karen. *Beep_ A Documentary History of Game Sound*. 2016.

Duden. (2019, 7. Oktober). Duden | prozedural | Rechtschreibung, Bedeutung, Definition, Herkunft. Zugriff 7. Oktober 2019 unter https://www.duden.de/rechtschreibung/prozedural

Farnell, A. (2010). *Designing sound*. London, England: The MIT Press.

Henk. (2013a). FRACT | An Indie Adventure Game by Phosfiend Systems. Zugriff 20. Januar 2020 unter http://fractgame.com/

Henk. (2013b, 21. Mai). FRACT audio tech: Getting started | FRACT [FRACT OSC dev-blog]. Zugriff 20. Januar 2020 unter http://fractgame.com/news/126-fract-audio-tech-getting-started

Huang, X. & Deng, L. (2010, 22. Februar). *Handbook of natural language processing* (2. Aufl.). Taylor & Francis Ltd.

Hudak, P. (1989). Conception, evolution, and application of functional programming languages, 53. Zugriff 26. März 2020 unter http://www.dbnet.ece.ntua.gr/~adamo/languages/books/p359-hudak.pdf

Kühne, L. (2015). Das Sibelius-Monument wird hörbar [@GI_weltweit]. Zugriff 6. Januar 2020 unter https://www.goethe.de/ins/fi/de/kul/mag/20741439.html

Lehmann, E. (2014). Parametrisierte Atmosphärengestaltung mit synthetischen Klangerzeugern am Beispiel der Software AudioWeather und fLOW, 59.

lev Team. (2017, 26. Juni). Das Loop-Ensemble als interaktive Klanginstallation | lev – die elektronische Musikschule in Berlin. Zugriff 6. Januar 2020 unter https://www.lev-berlin.de/projekte-koorperationen/interaktive-klanginstallationen/das-loop-ensemble-als-interaktive-klanginstallation/1411/

MacGregor, A. (2014a). The Sound of Grand Theft Auto V. Zugriff 24. Oktober 2019 unter https://www.youtube.com/watch?v=L4GuM15QOFE

MacGregor, Alastair. *The Sound of Grand Theft Auto V*. San Francisco. 2014.

meriam-webster. (2019, 7. Oktober). Definition of PROCEDURE. Zugriff 7. Oktober 2019 unter https://www.merriam-webster.com/dictionary/procedure

Meyer, L. & Schmidt, G.-D. (2005). *Physik*. Mannheim: Duden.

Microsoft. (2018, 31. Mai). Dynamic-link libraries (dynamic-link libraries) - win32 apps. Zugriff 25. März 2020 unter https://docs.microsoft.com/en-us/windows/win32/dlls/dynamic-link-libraries

MIDI Manufacturers Association. (2009). *An introduction to MIDI*. MIDI Manufacturers Association. Zugriff 27. März 2020 unter https://www.midi.org/images/easyblog_articles/43/intromidi.pdf

Modartt. (2020). Modartt - pianoteq 6. Zugriff 20. Januar 2020 unter https://www.modartt.com/pianoteq

Mongeau, A.-S. (2017, 28. März). Behind the sound of 'no man's sky': A q&a with paul weir on procedural audio. Zugriff 20. Januar 2020 unter https://www.asoundeffect.com/no-mans-sky-sound-procedural-audio/

Orlarey, Y., Fober, D. & Letz, S. (2020, 1. Februar). Faust Programming Language. Zugriff 1. Februar 2020 unter https://faust.grame.fr/doc/manual/index.html#quick-start

*Procedural generation*. (2019, 28. November), In *Wikipedia*. Page Version ID: 928346717. Zugriff 8. Dezember 2019 unter https://en.wikipedia.org/w/index.php?title=Procedural_generation&oldid=928346717

Puckette, M. S. (2020). Pure Data — Pd Community Site. Zugriff 1. Februar 2020 unter http://puredata.info/docs/manuals/pd/

*Puzzle video game*. (2019, 27. Dezember), In *Wikipedia*. Page Version ID: 932689661. Zugriff 20. Januar 2020 unter https://en.wikipedia.org/w/index.php?title=Puzzle_video_game&oldid=932689661

*Reference (computer science)*. (2020, 18. Februar), In *Wikipedia*. Page Version ID: 941445464. Zugriff 1. März 2020 unter https://en.wikipedia.org/w/index.php?title=Reference_(computer_science)&oldid=941445464

Schweitzer, D. C. (2004). Ton & traum: A critical analysis of the use of sound effects and music in contemporary narrative film, 87.

Stache, E. (2012, 13. Januar). Eine interaktive Klanginstallation in der Bonner Fußgängerzone art-in.de [art-in.de]. Zugriff 6. Januar 2020 unter https://www.art-in.de/incmeldung.php?id=2863

Texas Instruments (Hrsg.). (1978). *SN76477 Datenblatt*. MOS Technology Inc. Zugriff 13. Januar 2020 unter http://experimentalistsanonymous.com/diy/Datasheets/SN76477.pdf

Texas Instruments. (1981a). *MOS6581 datenblatt*. MOS Technology Inc. Zugriff 13. Januar 2020 unter http://archive.6502.org/datasheets/mos_6581_sid.pdf

Texas Instruments. (1981b, Juni). *Tms5220.pdf*. MOS Technology Inc. Zugriff 14. Oktober 2019 unter http://sprow.co.uk/bbc/hardware/speech/tms5220.pdf

Unity Technologies. (2018a). Unity - manual: Managed plug-ins. Zugriff 1. Februar 2020 unter https://docs.unity3d.com/Manual/UsingDLL.html

Unity Technologies. (2018b). Unity - manual: Native audio plugin SDK. Zugriff 1. März 2020 unter https://docs.unity3d.com/Manual/AudioMixerNativeAudioPlugin.html

Unity Technologies. (2019, 14. Oktober). Unity - scripting API: MonoBehaviour.OnAudioFilterRead(float[], int) [OnAudioFilterRead API]. Zugriff 14. Oktober 2019 unter https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnAudioFilterRead.html

Unity Technologies. (2020a). Unity - manual: Custom editors. Zugriff 1. März 2020 unter https://docs.unity3d.com/Manual/editor-CustomEditors.html

Unity Technologies. (2020b). Unity - manual: Native plug-ins. Zugriff 1. März 2020 unter https://docs.unity3d.com/Manual/NativePlugins.html

Unity Technologies. (2020c). Unity - scripting API: AudioClip.create. Zugriff 1. März 2020 unter https://docs.unity3d.com/ScriptReference/AudioClip.Create.html

Weir, P. (2017). The Sound of No Man's Sky. Zugriff 24. Oktober 2019 unter https://www.youtube.com/watch?v=zKJ_XuQjjiw

White, J. & Roth, M. (2018, 21. September). Enzienaudio/hvcc [GitHub]. Zugriff 1. Februar 2020 unter https://github.com/enzienaudio/hvcc

Wright, M. (2002, 26. März). The Open Sound Control 1.0 Specification | opensoundcontrol.org [opensoundcontrol]. Zugriff 28. März 2020 unter http://opensoundcontrol.org/spec-1_0

Hiermit erkläre ich, dass ich die Bachelor-Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle aus der Literatur und sonstigen Quellen (insbesondere auch Internettexten) übernommenen Zitate und Gedanken wurden kenntlich gemacht.

Ferner erkläre ich, dass die eingereichte Bachelor-Arbeit nicht schon ganz oder teilweise bei einer anderen Prüfung oder Präsentation vorgelegt wurde.

Ich bin damit einverstanden, dass meine Bachelor-Arbeit zur Einsichtnahme in der Bibliothek bereitgestellt wird.

Ich bin damit einverstanden, dass der Titel meiner Bachelor-Arbeit in ein Verzeichnis (zum Beispiel im Rahmen der Internet-Präsenz der Hochschule) aufgenommen wird.

Berlin, den 30. März 2020, 

Benjamin Feder