VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

# DIPLOMOVÁ PRÁCE

Brno, 2019 Bc. Dominik Konečný


# VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

## FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

## ÚSTAV TELEKOMUNIKACÍ
DEPARTMENT OF TELECOMMUNICATIONS


# ADAPTIVNÍ REAL-TIME REVERB JAKO PLUGIN PRO HERNÍ ENGINE UNITY
ADAPTIVE REAL-TIME REVERB AS A PLUGIN FOR UNITY GAME ENGINE

**DIPLOMOVÁ PRÁCE**
MASTER'S THESIS


**AUTOR PRÁCE** — **Bc. Dominik Konečný**
AUTHOR

**VEDOUCÍ PRÁCE** — **Ing. Tomáš Kiska**
SUPERVISOR


**BRNO 2019**

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

# Diplomová práce

magisterský navazující studijní obor **Audio inženýrství**
Ústav telekomunikací

***Student:*** Bc. Dominik Konečný ***ID:*** 162641
***Ročník:*** 2 ***Akademický rok:*** 2018/19

**NÁZEV TÉMATU:**

## Adaptivní real-time reverb jako plugin pro herní engine Unity

**POKYNY PRO VYPRACOVÁNÍ:**

V rámci této práce bude vytvořen zvukový plugin v jazyce C++ za použití Unity Native Audio Plugin SDK a jiných knihoven pro herní engine Unity. Právě ten bude simulovat prvotní odrazy a difuzní část dozvuku v 3D prostoru. Plugin bude obsahovat nastavení času dozvuku a HF/LF ratio (oba závislé na materiálech a velikosti prostoru). Dále pak nastavení pre-delay, difuzního poměru, reflexního poměru a nastavení dry/wet signálu. Výstupem diplomové práce bude funkční plugin v Unity GUI. Bude vytvořen uživatelský skript v jazyce C#. Adaptace reverbu bude probíhat v reálném čase na základě uživatele v reálném prostoru. Součástí testování pluginu bude ukázková scéna v 3D prostoru, která bude a následné testovat funkčnost jak na velkosti prostoru, tak na různých materiálech (např. dřevo, beton, dlažba, záclony, …). V rámci této práce bude vytvořen reverb s požadovanými vlastnostmi. Dále pak bude rozšířen o jeho adaptivnost v 3D prostoru a následně otestován.

**DOPORUČENÁ LITERATURA:**

[1] HOCKING, Joseph. Unity in action: multiplatform game development in C#. Shelter Island, NY: Manning Publications Co., [2015]. ISBN 978-1617292323.

[2] ZÖLZER, Udo. DAFX: digital audio effects. 2nd ed. Chichester: Wiley, 2011. ISBN 0470665998.

***Termín zadání:*** 1.2.2019 ***Termín odevzdání:*** 16.5.2019

***Vedoucí práce:*** Ing. Tomáš Kiska
***Konzultant:***

**prof. Ing. Jiří Mišurec, CSc.**
*předseda oborové rady*

# Abstrakt

Tato práce se věnuje reverberátorům jako systémům sloužícím k simulaci poslechových prostor a jejich implementaci v rámci herního vývojového prostředí Unity. Cílem této práce bylo vytvořit funkční zásuvný modul pro toto prostředí, který bude schopný pracovat v reálném čase a taky své parametry v reálném čase měnit.

# Klíčová slova

dozvuk, reverberátor, Unity, hra, DSP

# Abstract

This work is about reverberators as system for simulating space properties of sound and their implementation in game development software Unity. The goal of this thesis is to create a working plug-in module for this development software, which is capable of working in real time and also can change it's parameteres in real time.

# Klíčová slova

reverberation, reverb, Unity, game, DSP

KONEČNÝ, Dominik. *Adaptivní real-time reverb jako plugin pro herní engine Unity.* Brno, 2019. Dostupné také z: https://www.vutbr.cz/studenti/zav-prace/detail/118136. Diplomová práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2019, 77 stran, 1 příloha. Vedoucí práce Ing. Tomáš Kiska

# Prohlášení autora o původnosti díla

Prohlašuji, že svou diplomovou práci na téma Adaptivní real-time reverb jako plugin pro herní engine Unity jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

V Brně dne . . . . . .

. . . . . .

podpis autora

# Obsah

# Seznam obrázků

# Úvod

Hry jako médium existují již přes třicet let. Za tu dobu se z okrajové zábavy pro malou specifickou skupinu staly celospolečenským fenoménem ale také jedním z nejrychleji rostoucích kulturních a zábavních odvětví. Současné mainstreamové hry, tvořené často týmy stovek lidí, v sobě dokáží spojovat nejnovější technologie s uměleckou tvorbou. Nezanedbatelnou roli ve výsledném celku tedy pochopitelně hraje zvuk a hudba. Hra, má oproti jiným médiím, například filmu nebo hudební nahrávce, zásadní rozdíl, kterým je *nelinearita*. Celý herní systém, tedy veškeré zpracování obrazu, herní logiky i zvuku musí být prováděno nejen v reálném čase, ale zároveň na základě naprosto nepředvídatelného chování hráče.

Cílem této práce je vytvořit zásuvný modul pro herní engine Unity, který dokáže simulovat zvukové vlastnosti poslechových prostor automaticky na základě informací v rámci herního enginu, a tím ušetřit vývojáři nutnost manuálního nastavování pro každý prostor.

V první části se tato práce věnuje problematice simulace poslechových prostor ve virtuálním prostoru v reálném čase. Věnuje se různým přístupům k simulaci poslechových prostor v číslicovém zpracování signálu a jejich specifikám. Také jsou zde popsány základní stavební prvky a struktury, kterých se při číslicovém zpracování signálu využívá.

V druhé části se tato práce věnuje hernímu enginu Unity. Unity (případně Unity3D) je v současné době nejpoužívanějším herním enginem, v kterém vznikají tisíce her ročně a to jak od velkých vývojářských studií tak malých nezávislých týmů a jednotlivců. Bude zde přiblížena historie tohoto vývojového prostředí a práce v něm. Blížší pozornost bude věnována práci se zvukem v rámci enginu a bude zde zmíněny vývojové nástroje Unity Native Audio SDK a Unity Spatializer SDK pro tvorbu vlastních zásuvných modulů.

Ve třetí části je již popsán návrh samotného zásuvného modulu v rámci jednotlivých struktur.

V poslední části se tato práce věnuje samotným zásuvným modulům, vytvořeným v jazyce C++. Budou zde popsány jednotlivé základní třídy a funkce, ale také důležité uživatelsky editovatelné parametry. Dále zde bude popsáno fungování skriptů v jazyce C# ovládající tyto zásuvné moduly. Také zde bude přiblížen ukázkový projekt v Unity, jeho obsah a fungování.

# 1 Reverberátory

## 1.1 Simulace poslechového prostoru

Každá reálná místnost má své zvukové vlastnosti. Každý zvuk, který slyšíme, je doplněn o jeho opožděné odrazy z mnoha různých směrů. Náš sluch je na tento vjem zvyklý, takže si jej většinou neuvědomujeme. Typicky si jej všimneme v extrémních případech velikých odrazivých prostor, jakým mohou být například jeskyně nebo kostely, kde časy jednotlivých odrazů mohou být v řádech stovek milisekund a celková doba dozvuku v řádech sekund. Ovšem v malých místnostech (typicky například koupelna), kde je rozdíl mezi přímým zvukem a jeho odrazem v řádu jednotek milisekund, nevnímáme odrazy samostatně, ale jako součást samotného zvuku. Typicky nám mění hlasitost a barvu zvuku.

Za zásadní parametry pro vlastnosti simulace tedy můžeme považovat velikost daného prostoru, uspořádání a dále pohltivost, případně odrazivost jednotlivých stěn místnosti. Dále je pro nás důležitá poloha zdroje zvuku a poloha posluchače. Pokud uvažujeme posluchače jako stereofonní (případně i váce kanálový) výstup, zajímá nás také jeho orientace v prostoru. Pro skutečně realistickou simulaci by nás zajímala ještě směrová vyzařovací charakteristika zdroje, my ale v této práci pro zjednodušení budeme všechny zdroje považovat za všesměrové.

Simulace dozvuku prostor je hojně využívána v hudbě. A to jak ve studiové praxi, tak při živé produkci. Zásuvné moduly a efektové jednotky dnes najdeme v každém DAW a mixážním pultu. Hojně se také využívá ve filmové postprodukci a hrách, pro simulaci zvukových vlastností virtuálních prostor. Absence dozvuku si ve velkých virtuálních prostorách všimneme velmi rychle jako něčeho velmi nepřirozeného. Obecně se dá říct že reprodukovaný zvuk bez jakéhokoliv dozvuku, ať už nahraného nebo simulovaného se dá považovat za suchý a nepřirozený.

### 1.1.1 Přístupy k simulaci poslechového prostoru

Z hlediska zpracování signálu můžeme každou místnost se zdroji zvuku a posluchači považovat za lineární systém se vstupy a výstupy. Pokud máme všesměrový zdroj zvuku jako vstup a výstupy jako akustický tlak v poloze uší, můžeme místnost považovat za lineární časově invariantní systém popsaný dvěma přenosovými funkcemi. Takovýto systém můžeme popsat binaurální impulsní odezvou [22]:

$$\begin{aligned} y_L(t) &= \int_0^\infty h_L(\tau)x(t-\tau)d\tau, \\ y_R(t) &= \int_0^\infty h_R(\tau)x(t-\tau)d\tau, \end{aligned} \tag{1.1}$$

kde $h_L(t)$ a $h_R(t)$ jsou impulsové odezvy pro levé a pravé ucho, $x(t)$ je zdroj zvuku a $y_L(t)$ a $y_R(t)$ jsou výsledné signály pro levé a pravé ucho [16].

Délka takovéto impulsové odezvy může ale být v řádu desítek až stovek tisíc vzorků (v závislosti na době dozvuku a vzorkovacím kmitočtu). Realizace takovéhoto filtru za pomoci FIR filtru by byla extrémně výpočetně náročná [2]. V případě kdy potřebujeme tento systém počítat v reálném čase, a navíc souběžně s renderováním obrazu a fyzikálními interakcemi v herním enginu, není možné tento přístup zvolit. Dále bychom také museli mít banku impulsových odezev pro každou kombinaci polohy hráče vůči zdroji zvuku.

V praxi se můžeme setkat s následujícími přístupy k simulaci:

- Fyzikálni přístup:
- Vjemový přístup
- Aproximační přístup

### Fyzikálni přístup

Cílem fyzikálního přístupu simulace je simulovat šíření zvuku v prostoru exaktně. Toho docílíme konvolucí signálu zdroje a impulsové odezvy.

Pokud prostor, který chceme simulovat není reálný a jeho impulsové odezvy není možné změřit, musíme použít jiné metody. Základním předpokladem je, že známe přesně geometrii prostoru, akustické vlastnosti stěn a objektů v prostoru a polohy a orientace zdroje a posluchače. Využívají se metody sledování paprsku (ray-tracingu), zrcadlení zdroje (image-source) nebo metody konečných prvků (finite-element, boundary-element) [17, 16]. Tento přístup se v literatuře označuje jako *auralizace*.

S rostoucím výkonem dnešních počítačů roste i obliba *konvolučních reverberátorů*, tedy systémů využívajích konvoluce k simulacím poslechových prostor. Výhodou je, že můžeme relativně přesně simulovat skutečná a známá místa, typicky například slavná nahrávací studia nebo katedrály. S rostoucí popularitou reverberátorů i roste dostupnost těchto impulsových odezev. V posledních letech také roste využití konvolučních reverberátorů v herním vývoji.

### Vjemový přístup

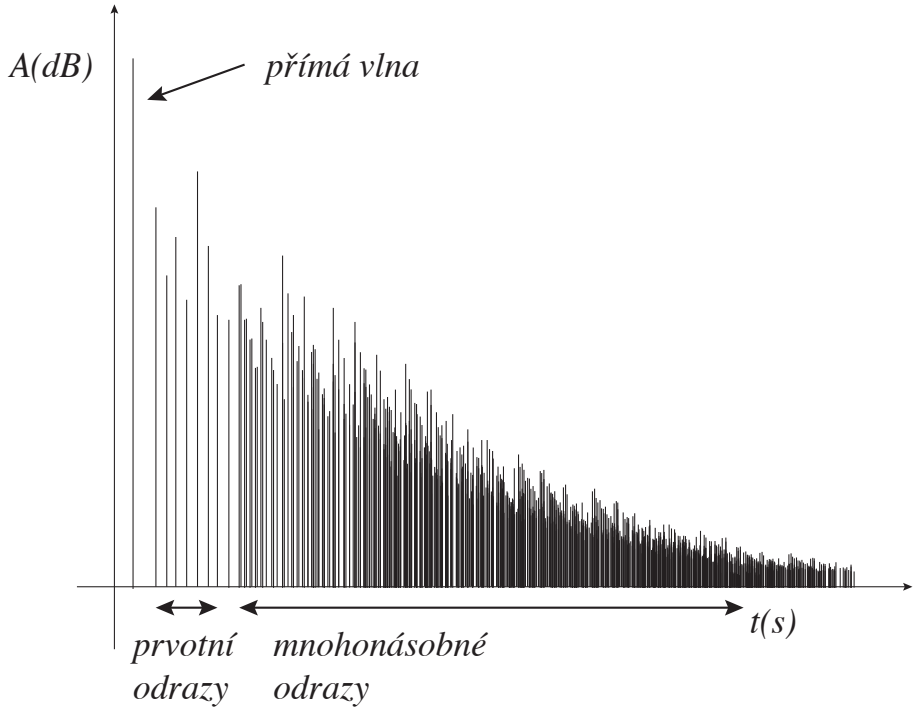
Je takový přístup, při kterém se snažíme reprodukovat pouze vjemově významné charakteristiky dozvuku na základě známých parametrů. V praxi se při implementaci využívají mnohem efektivnější IIR filtry (filtry s nekonečnou impulsovou odezvou). Nad takovýmto systémem máme kontrolu v reálném čase a můžeme v reálném čase

také jeho paramety měnit. Nejdůležitějším z těchto parametrů je pro nás doba dozvuku $t_{60}(f)$.
Pro zjednodušení si můžeme dozvuk rozdělit na více složek a to:

- Prvotní odrazy: simulace protních odrazů, řádově prvních 5-10. Ve složitějších simulacích můžeme zahuštěním těchto odrazů simulovat i následné odrazy.
- Mnohonásobné odrazy: simulují difúzní část dozvuku.

### Aproximační přístup

Aproximační přístup používá impulsové odezvy prostoru, které se blíží impulsové odezvě prostoru, který se snažíme simulovat. Hledaná odezva musí být taková, aby náročnost výpočtu byla menší než u fyzikálního přístupu [12].



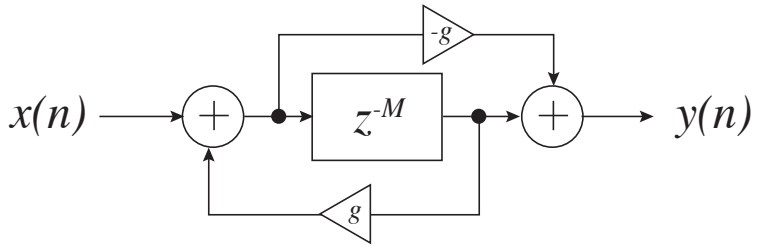

Obr. 1.1: Zjednodušený časový průbeh dozvuku

## 1.2 Základní stavební prvky

### 1.2.1 Zpožďovací linka

Zpožďovací linka je základním stavebním blokem. V číslicovém zpracování signálů zpožďuje výstupní signál oproti vstupnímu o $M$ vzorků.

$$\begin{aligned} &M = -3\ , \\ &x(n) = [1 \quad 3 \quad 5]\ , \\ &y(n) = x(n) * z^{-M}\ , \\ &y(n) = [0 \quad 0 \quad 0 \quad 1 \quad 3 \quad 5]\ . \end{aligned} \tag{1.2}$$



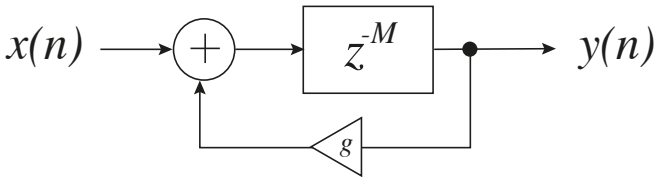

Obr. 1.2: Zpožďovací linka

### 1.2.2 Hřebenový filtr

Blokové schéma hřebenového filtru je na obrázku 1.3. Skládá se ze zpožďovací linky a jedné zpětné vazby. Přenosová funkce takovéhoto filtru je [4]:

$$H(z) = \frac{z^{-M}}{1 - gz^{-M}}\ , \tag{1.3}$$

kde $M$ je zpoždění ve vzorcích a $g$ je útlum zpětné vazby. Impulsová odezva takovéhoto filtru tedy bude obsahovat nenulové vzorky s exponenciálně klesající amplitudou $M$ vzorků od sebe. Všechny ostatní vzorky budou nulové. Frekvenční odezva takovéhoto filtru bude mít tvař hřebenového filtru s maximy na $\frac{1}{MT}$ kde $T$ je vzorkovací perioda [16].

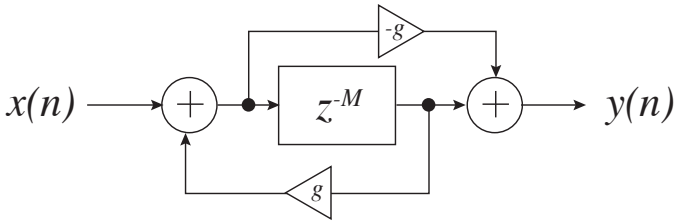



Obr. 1.3: hřebenový filtr

### 1.2.3 Fázovací článek

Pokud hřebenovému filtru přidáme dopřednou vazbu, viz obr 1.4 dostaneme *fázovací článek*, anglicky *allpass filter*. Frekvenční charakteristika fázovacího článku je rovna jedné. Neprovádí nám tedy žádné změny v modulovém spektru signálu [4].

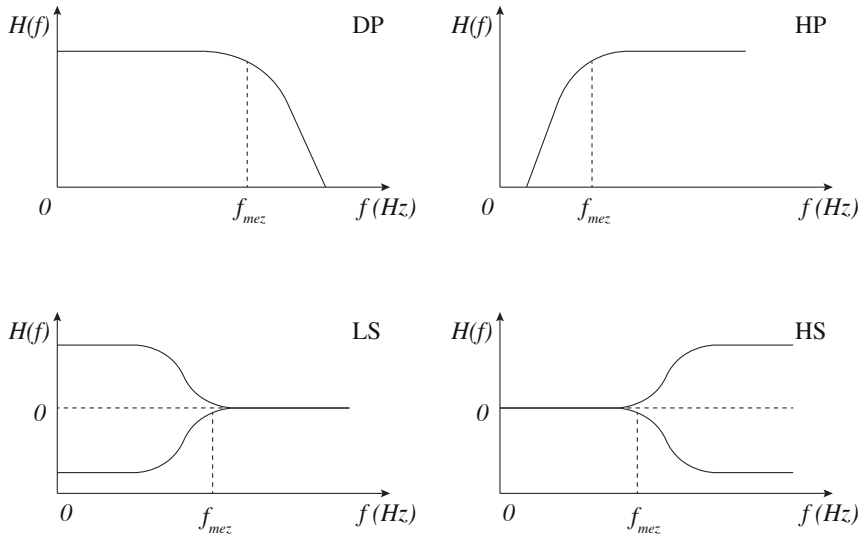
$$H(z) = \frac{z^{-M} - g}{1 - gz^{-M}} \, . \tag{1.4}$$




Obr. 1.4: fázovací článek

### 1.2.4 Kmitočtové filtry

Principem kmitočtových filtrů je jejich schopnost propouštět beze změny určité části spektra, a ovlivňovat útlum nebo zesílení v částech ostatních. Mezi základní kmitočtové filtry řadíme *dolní propust*, která beze změny propouští kmitočty pod mezním kmitočtem, zatímco vyšší tlumí. Dále *horní propust*, která funguje opačně, tedy tlumí kmitočty pod mezním kmitočtem. Kombinací těchto základních filtrů s dopřednou vazbou získáme filtry typu shelving. Kmitočtové charakteristiky filtrů pro názornost jsou na obrázku 1.5.

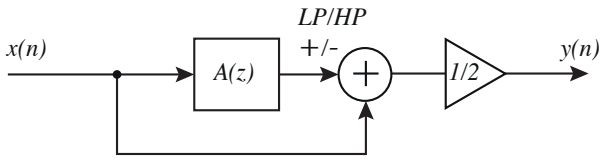



Obr. 1.5: Základní kmitočtové filtry

### 1.2.5 Filtry prvního řádu založené na fázovacích článcích

Mějme fázovací článek s funkcí [4]:

$$A(z) = \frac{z^{-1} + c}{1 + gz^{-1}} \,,$$
$$c = \frac{tan(\pi f_m / f_v z) - 1}{tan(\pi f_m / f_v z) + 1} \,, \tag{1.5}$$



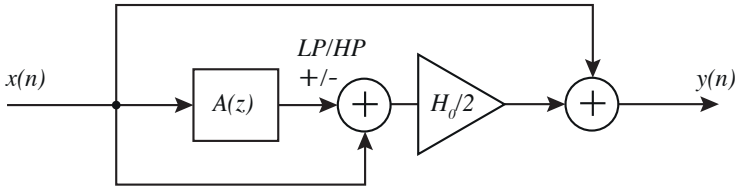

Obr. 1.6: horní/dolní propust realizovaná fázovacím článkem

kde $f_m$ je mezní kmitočet filtru a $f_{vz}$ je vzorkovací kmitočet. Dolní nebo horní propust prvního řádu za použití fázovacího článku lze zrealizovat jako součet, případné odečet, vstupního signálu a výstupu z fázovacího článku, jehož vstupem je vstupní signál, viz obrázek 1.6 Protože sčítáme 2 signály, je na výstupu potřeba výstupní signál váhovat hodnotou 0,5 [4].

V případě, že signály bude sčítat, bude tento systém fungovat jako dolní propust, pokud odečítat, budeme systémem realizovat horní propust.

### 1.2.6 Shelving filtry

Jako základní stavební prvek pro shelving filtry použijeme dolní/horní propust popsané výše. Přibude jedna dopředná vazba, a váhujeme na základě požadovaného zesílení, viz obrázek 1.7 a vzorec 1.6 [4].




Obr. 1.7: high/low shelving filter

$$H_0 = V_0 - 1 \;\; kde \;\; V_0 = 10^{G/20} \,, \tag{1.6}$$

kde $G$ je požadované zesílení v decibelech.

Parametr $c$ fázovacího článku bude různý pro případ kdy shelving filtrem zesilujeme a zeslabujeme. Definujeme tedy $c_z$ pro zesílení a $c_u$ pro útlum. Pro low shelving tedy parametry získáme jako [4]:

$$\begin{aligned} c_z &= \frac{tan(\pi f_c/f_s) - 1}{tan(\pi f_c/f_s + 1} \,, \\ c_u &= \frac{tan(\pi f_c/f_s) - V_0}{tan(\pi f_c/f_s + V_0} \,, \end{aligned} \tag{1.7}$$

a pro high shelving:

$$\begin{aligned} c_z &= \frac{tan(\pi f_c/f_s) - 1}{tan(\pi f_c/f_s + 1} \,, \\ c_u &= \frac{V_0 tan(\pi f_c/f_s) - 1}{V_0 tan(\pi f_c/f_s + 1} \,. \end{aligned} \tag{1.8}$$

## 1.3 Prvotní odrazy

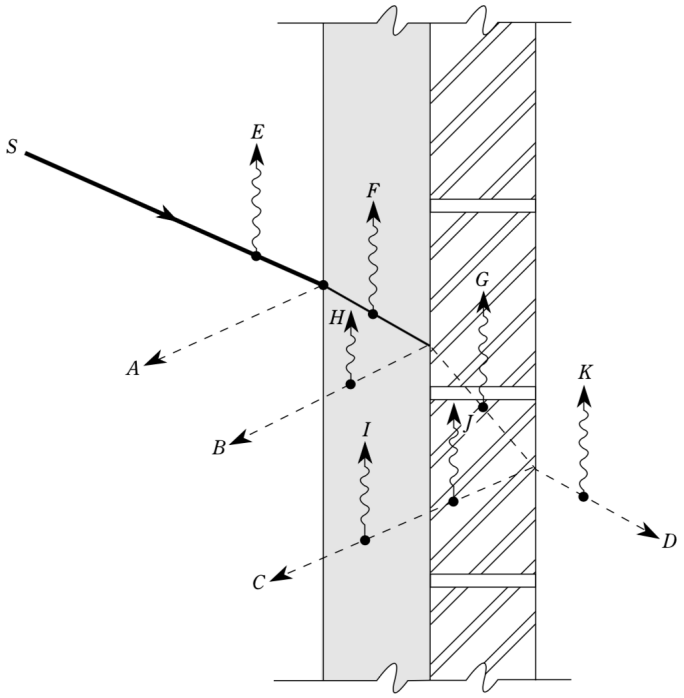
Prvotními odrazy při simulaci poslechových prostor rozumíme zdrojový signál, který k posluchači dojde se zpožděním, utlumen, kmitočtově filtrován a z jiného směru než signál původní. Nad všemi těmito parametry máme kontrolu, pokud máme o prostoru veškeré informace.

### 1.3.1 Odraz a absorpce

Na obrázku 1.8 je vidět situace kdy zvuková vlna *s*, narazí na překážku. Na obrázku uvažujeme překážku složenou ze dvou materiálů, jednoho s vyšší a druhého s nižší hustotou. Pro příklad můžeme uvést dřevenou desku potaženou vrstvou koberce.

Ještě než vlna *s* dorazí k překážce, ztratí část své energie, která se promění v teplo vlivem tření v prostředí (na obrázku vyznačeno *E*). V momentě kdy vlna narazí na překážku, část její energie se odrazí, (odraz *A*) a část pokračuje dále překážkou. Vzhledem k tomu, že prostředí překážky je hustší než vzduch, změní směr šíření. Další část energie se přemění v teplo, vlivem tření v prostředí. To je znázorněno šipkou *F*. Tato ztráta je větší než ztráta ve volném prostředí *E*.

Když vlna dorazí na rozhraní dvou materiálů, opět dochází k odrazu, část energie prochází a pokud je prostředí hustější, změní se opět směr šíření.




Obr. 1.8: Odraz a absorpce převzato z [6]

Tab. 1.1: Absorpční koeficienty vybraných materiálů

| | Frekvence ($Hz$) | | | | | | |
|---|---|---|---|---|---|---|---|
| | 125 | 250 | 500 | 1000 | 2000 | 4000 | poznámka |
| Záclony | 0,04 | 0,05 | 0,11 | 0,18 | 0,30 | 0,35 | hustota 0,338 $kg/m^2$ pověšený u zdi |
| Koberec | 0,02 | 0,06 | 0,14 | 0,37 | 0,60 | 0,65 | těžky koberec, položený na betonu |
| Dřevo | 0,28 | 0,22 | 0,17 | 0,09 | 0,1 | 0,11 | překližka, tloušťka 1 $cm$ |
| Beton | 0,02 | 0,03 | 0,03 | 0,03 | 0,04 | 0,07 | hrubý beton |
| Sklo | 0,35 | 0,25 | 0,18 | 0,12 | 0,07 | 0,04 | tabulka skla |
| Vata | 0,08 | 0,25 | 0,45 | 0,75 | 0,75 | 0,65 | skelná vata, tloušťka 25 $mm$ |

Absoprci je možno změřit. Zavádíme tedy pojem *absorpční koeficient*, značíme nejčastěji $a$. Ten udává schopnost materiálu pohlcovat zvuk. *Absorpční koeficient* se nejčastěji udává jako bezrozměrné číslo v rozmezí nula až jedna. Pokud bychom měli materiál s koeficientem $a = 0$, znamená to že třicet procent zvuku bude pohlceno. Není možné aby jakýkoliv materiál měl koeficient větší než jedna. Také jej můžeme zjednodušeně uvažovat jako poměr mezi zvukem odraženým a absorbovaným. Můžeme tedy říct že například otevřené okno v místnosti má $a_{okno} = 1$.

*Absorpční koeficient* je však závislý na mnoha faktorech. Nejvýraznější změny jsou s frekvencí a úhlem, pod kterým zvuk na překážku dopadá.

V praxi se s absorpcí potkáváme v *sabinech*. Tato jednotka nám udává množství absorpce na plochu. *Absorpci* místnosti tedy vypočteme následovně:

$$A = S_1 a_1 + S_2 a_2 + S_3 a_3 + \dots , \tag{1.9}$$

kde $A$ je *absoprce* s sabinech, $S_i$ plocha, zaujímaná materiálem $i$ a $a_i$ je absorpční koeficient tohoto materiálu.

#### Absorpční koeficienty vybraných materiálů

V tabulce 1.1 jsou uvedeny absorpční koeficienty vybraných materiálů. Nejčastěji se v praxi setkáme s koeficienty udaných v šesti oktávových pásmech se středními kmitočty 125, 250, 500, 1000, 2000 a 4000 $Hz$ [3].

### 1.3.2 Metoda zrcadlení zdrojů

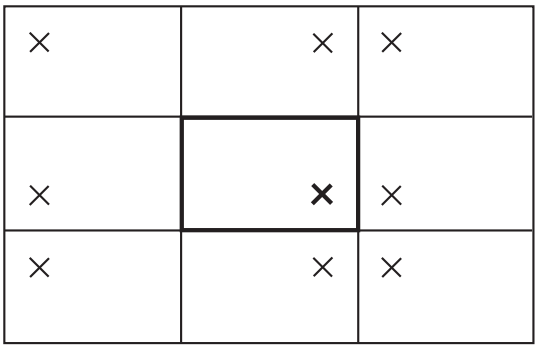
Metodu pro zrcadlení zdrojů lze využít pro zjištění polohy virtuálních zdrojů zvuku, kterými můžeme zastoupit odrazy. Pro pravoúhlou místnost můžeme použitím této metody zrcadlové zdroje zjistit jednoduše, viz obrázek 1.9. Tato metoda však lze

použít i pro složitější geometrii. Počet zrcadlených zdrojů odpovídá přibližně $N^k$ kde $N$ je počet stěn a $k$ je řád odrazů [16].

Každá stěna geometrie nám dává jeden zrcadlový zdroj. Stěny tohoto zrcadlového zdroje nám opět umožňují vypočítat další zrcadlové zdroje. Můžeme tedy použít rekurzivních algoritmů pro výpočet zrcadlových zdrojů vyšších řádů.
Pokud tedy víme vzdálenost mezi virtuálním zdrojem zvuku a posluchačem, můžeme vypočítat zpoždění a útlum daného odrazu pomocí vzorců 1.10 a 1.11 [19].

$$zpoždění = \frac{s}{c} f_s \ , \tag{1.10}$$

$$útlum = 20log_{10}\frac{s}{s_{max}} \ . \tag{1.11}$$



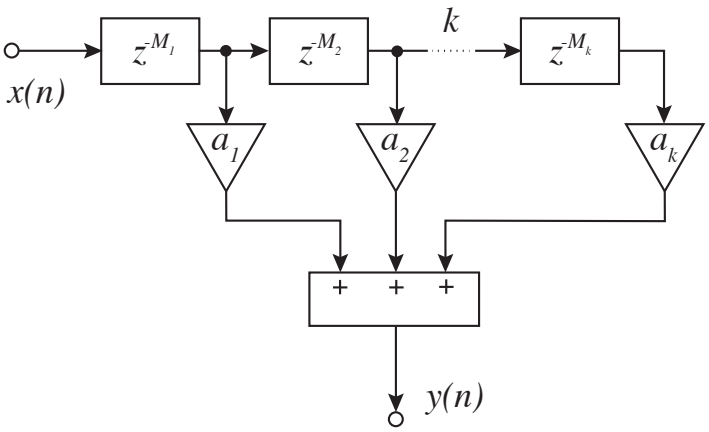
Obr. 1.9: Zrcadlené zdroje

### 1.3.3 Vliv prvotní odrazů

Prvotní odrazy mají velký vliv na vjem poslechového prostoru. Pokud čas těchto odrazů je více než $50\,ms$, vnímáme je jako jako oddělené zvuky, podobně jako bychom je získali pomocí efektu echo. Čím delší tato doba je, tím prostorněji na nás simulovaný prostor působí. Pokud čas odrazů je méně než $50\,ms$, vnímáme je jako součást samotného zvuku a mají vliv na barvu zvuku. Mohou také zvýšit jeho amplitudu a pro krátké časy také ovlivnit vnímaný směr, z kterého zvuk k posluchači přichází [16].

### 1.3.4 Simulace prvotních odrazů

Pro simulaci prvotních odrazů potřebujeme zpožďovací linku s více výstupy. Tedy tolika výstupy, kolik prvotních odrazů budeme simulovat.




Obr. 1.10: Blokové schéma zpožďovací linky s více výstupy

Na obrázku 1.10 je blokové schéma zpoždovací linky s $k$ výstupy. Každý výstup je oproti vstupu opožděn o $N_k$ vzorků a zároveň váhován koeficientem $a_k$. Na výstupu jsou všechny zpožděné signály sečteny.

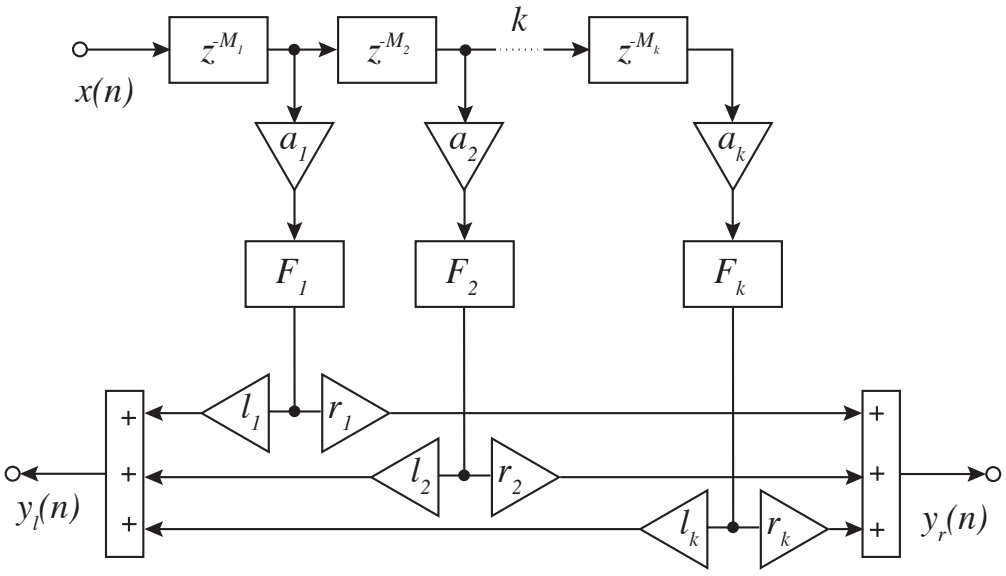
Pro naše potřeby je ale tento systém nedostatečný, protože nebere v potaz stereofonní poslech a absorpční vlastnosti prostoru. Proto jej musíme rozšířit o váhování pro každý z kanálů a kmitočtové filtry. V tomto případě uvažujeme pouze dva kanály, levý a pravý. Kmitočtovým filtrem můžeme rozumět filtr simulující kmitočtové absorpční materiálu. Pro zjednodušení můžeme uvažovat pouze dolní propust.

Na obrázku 1.11 tedy vidíme zpožďovací linku s více výstupy. Každý výstup zpožďovací linky má svoji váhu $a_k$, dále kmitočtový filtr $F_k$. Signál je poté rozdělen na 2 a váhami $l_k$ respektive $r_k$ váhován. Poté jsou zpožděné signály pro pravý a levý kanál nezávisle sečteny a poslány na výstup.

## 1.4 Panorámování

V této práci pro pozicování zvuku bude uvažován pouze stereofonní poslech pomocí dvou reproduktorů, případně sluchátek. Za typický ideální poslechový prostor považujeme dva reproduktory svírající úhel šedesát stupňů, tvořící spolu s posluchačem rovnostranný trojúhelník. Tato pozice posluchače je také známa jako *sweet spot*.

Přístupy k panorámování zdroje signálu jsou poté dva. Amplitudové panorámování, tedy různá intenzita signálu v obou kanálech a časově fázové panorámování, tedy signál přicházející z jednoho směru s časovým zpožděním oproti druhému směru [4].




Obr. 1.11: Blokové schéma zpožďovací linky pro simulaci k prvotních odrazů

### 1.4.1 Amplitudové panorámování

Amplitudovým panorámováním rozumíme různé intenzity zdrojového signálu v pravém a levém kanálu.

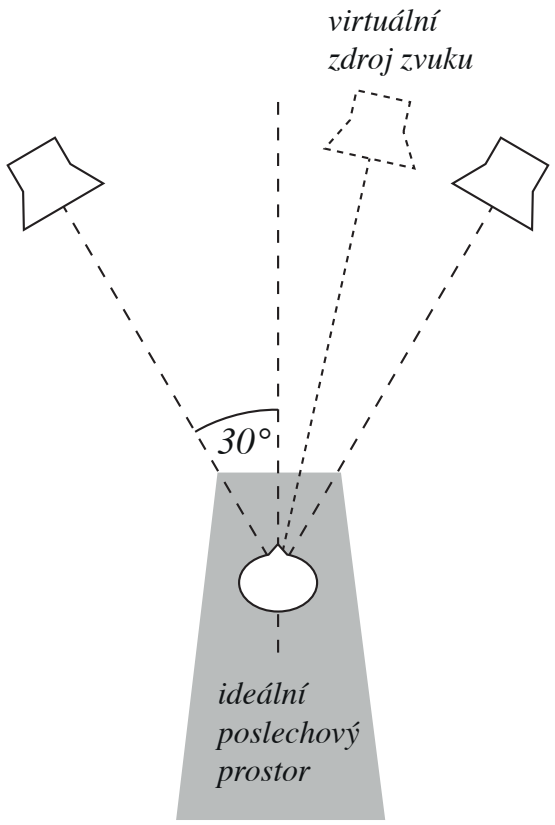
$$x_i(t) = g_i x(t), \quad i = 1, ...., N \ , \tag{1.12}$$

kde $x_i(t)$ je signál pro $i$-tý reproduktor a $g_i(t)$ je zasílení, případně útlum odpovídajícího kanálu. Pokud je posluchač od obou reproduktorů stejně vzdálen, poměr zesílení $g_i(t)$ nám vytvoří vjem zvuku přicházející z úhlu $\alpha$.
Problémem při poslechu na reproduktory je, že signál z pravého kanálu se dostane i k levému uchu. Tento fenomén, známý jako *přeslech* je nejvýraznější na nízkých frekvencích. Na vysokých frekvencích nám hlava vytváří akustický stín a přeslechy tedy nejsou tak výrazné. Zároveň nám přeslechy mohou vytvářet v pozicí uší hřebenové filtry a měnit barvu zvuku. Avšak při poslechu na standartní reproduktory tento efekt nebude příliš výrazný.[23]

Pro výpočet zesílení v jednotlivých kanálech na základě požadovaného úhlu mezi osou posluchače a virtuálním zdrojem zvuku využijeme tangentový zákon [4].

$$\frac{tan\ \alpha}{tan\ \alpha_0} = \frac{g_1 - g_2}{g_1 + g_2} \ . \tag{1.13}$$

Problém nastává pokud posluchač opustí *ideální poslechový prostor*, viz obrázek 1.12. Pokud by se například posunul více vlevo, i zdánlivá pozice virtuálního zdroje




Obr. 1.12: Stereofonní poslech

se posune více vlevo, protože bude v pozici posluchače výrazně převažovat zvuk přicházející z levého reproduktoru.

### 1.4.2 Panorámování pomocí časového a fázového zpoždění

Lidské ucho směr přicházejícího zvuku určuje jednak na základě rozdílu amplitud v obou uších, ale také na žákladě rozdílu času, v kterém zvuk dorazil na ušní bubínky. Tedy například zvuk přicházející zleva posluchače dorazí nejdřív do levého ucha a se zpožděním poté do ucha pravého. Pokud je tedy posluchač ve sweet spotu stereofonního poslechového prostoru, jako je na obrázku 1.12 a jeden kanál bychom zpozdili, pozice virtuálního zvukového zdroje se posune na stranu odpovídají nezpožděnému reprodukovanému kanálu.

Pokud ale máme signál, obsahující spíše nízké kmitočty, tento efekt nebude velmi výrazný. V extrémních případech se může stát že se v pozici posluchače dokonce na nižších kmitočtech dojde k odečtu. Panorámování pomocí zpoždění je tedy kmitočtově závislé.

Dalším postupem, kterým jde dosáhnout změny rozložení stereofonní báze signálu je otočením polarity signálu posílaného do jednoho z kanálů. Zatímco vyšší frekvence virtuálního zdroje zvuku zůstávají v takovémto případě ve středu stereofonní báze, pozice nižších frekvenci působí náhodně a šířší[4].

## 1.5 Přenosová funkce vztažená k hlavě (Head related transfer function – HRTF)

Zvuk putující prostorem mění své vlastnosti. Stejně jako prostor na něj má vliv fyziologie lidského těla. Zvuk, který dopotuje k ušním bubínkům, je v každém uchu jiný. Vzhledem k vzdálenosti mezi ušima může zvuk dorazit ke každému bubínku v jiný čas a s různou amplitudou. Stejně tak naše hlava může pro zvuky přicházející ze stran vrhat akustický stín. Ten je výrazný pro vyšší frekvence nad 2 *kHz*. Uspořádání ušních boltců a ramen, a odrazy zvuku od nich také ovlivňují výsledný zvuk. Soubor těchto přenosových funkcí se označuje v anglické literatuře jako *head related transfer function*, zkráceně HRTF, nebo *přenosová funkce vztažená k hlavě*. Je obecně závislá na poloze a orientaci posluchače vůči zdroji zvuku [23] [9].

Nevýhodou HRTF je v praxi nutnost použití sluchátek, abychom eliminovali přeslechy mezi ušima. Dále fakt, že každá hlava má přenosovou funkci hlavy mírně odlišnou. Pokud bychom ale HRTF chtěli použít při simulaci prvotních odrazů dozvuku, blokové schéma by se rozšířilo o bloky simulující HRTF. Takovéto blokové schéma je na obrázku 1.13.

## 1.6 Mnohonásobné odrazy

### 1.6.1 Doba dozvuku

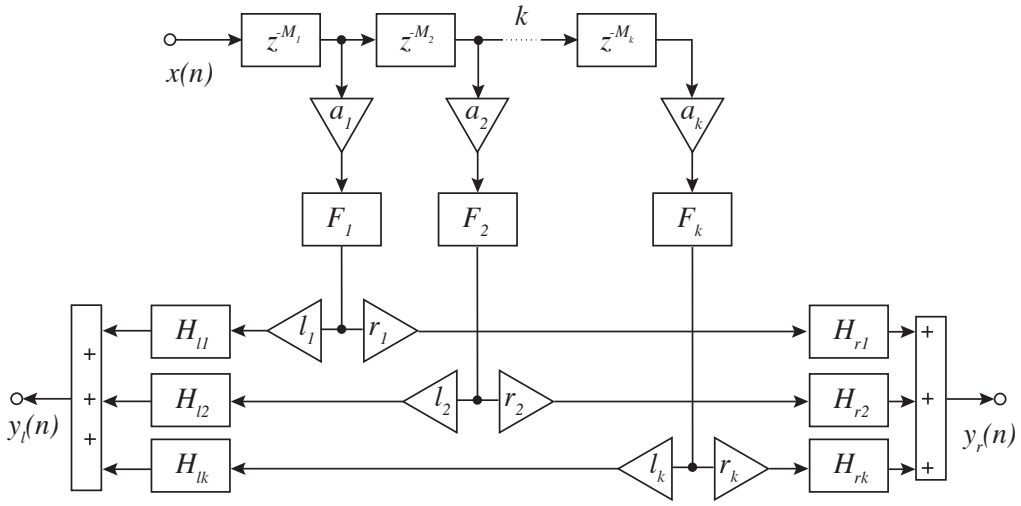
Časem dozvuku rozumíme čas, za který hladina akustického tlaku v místnosti po vypnutí zdroje zvuku klesne o 60 *dB*. Tento čas je úměrný velikosti prostoru a nepřímo úměrný celkové absorpci místnosti.

$$T_r \approx \frac{V}{A}\,, \tag{1.14}$$

kde $T_r$ je doba dozvuku, $V$ objem místnosti a $A$ celková absorpce místnosti.

Vzhledem k tomu že pohltivost je frekvenčně závislá veličina, typicky jsou materiály více pohltivé na vyšších frekvencích a podstatně méně učinné na nižších, je i doba dozvuku frekvenčně závislá veličina [16].

Doba dozvuku $T_{60}$ lze také vypočítat pomocí vzorce 1.15.




Obr. 1.13: Blokové schéma zpožďovací linky pro simulaci k prvotních odrazů s využitím HRTF

$$T_{60} = \frac{24ln(10)}{c_{20}} \frac{V}{Sa} \,, \tag{1.15}$$

kde $c_{20}$ je rychlost zvuku při 20 stupních Celsia, $V$ objem místnosti a součin $Sa$ celková absorpce v Sabinech [3].

### Útlumová křivka energie (Energy decay curve - EDC)

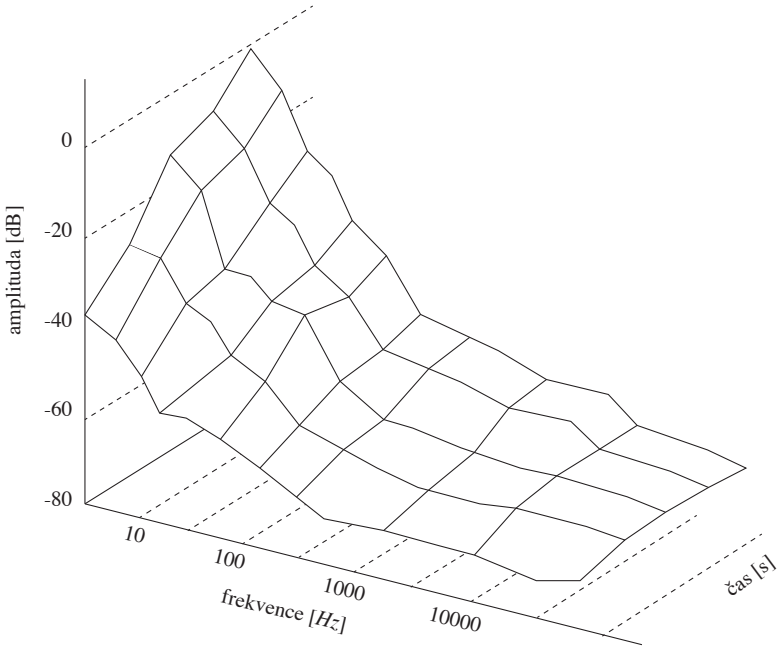
Pro měření času dozvuku $t_{60}$ lze použít EDC, což je integrál druhé mocniny impulsové odezvy v čase $t$. Tento vzorec je také známý jako Schroederův integrál [22] str 55.

$$EDC(t) = \int_t^\infty h^2(\tau) d\tau \,. \tag{1.16}$$

### Útlumový reliéf energie (Energy decay relief)

Rozšiřuje EDC o frekvenční závislost. Je funkcí času a frekvence. Je nejčastěji vykreslován jako 3D graf a efektivně nám znázorňuje úbytek energie v závislosti na čase na různých frekvencích [22] str. 55.

V reálných místnostech můžeme vždy pozorovat, že doba dozvuku na vyšších kmitočtech je kratší než na kmitočtech nízkých.




Obr. 1.14: Útlumový reliéf energie

### 1.6.2 Požadované kvality mnohonásobných odrazů

Pro přirozeně znějící mnohonásobné odrazy difúzní části dozvuku požadujeme:

- hladký postupný útlum
- hladkou frekvenční odezvu

Ideální pro nás je blížit se vjemově bílému šumu. Nežádoucí pro nás je takzvané dýchání, flutter nebo podobné aktefakty. Toho prakticky můžeme docílit vysokou hustotou jednotlivých odrazů [22]. Počet odrazů by měl s časem narůstat kvadraticky a nepřímo úměrně objemu.

## 1.7 Struktury pro simulaci mnohonásobných odrazů

Tato část se bude věnovat různým přístupům pro simulaci mnohonásobných odrazů. Přístupy zde popsané jsou seřazeny chronologicky od prvních pokusů s paralelními hřebenovými filtry, all pass filtry až k obecnějším přístupům za použití zpětnovazebních matic (FDN) [5].

### 1.7.1 Schroederův reverberátor

Manfred Schroeder byl jeden z prvních, kteří začali experimentovat s allpass filtry pro simulaci mnohonásobných odrazů. Právě na jeho práci je založena velká část následného vývoje na tomto poli.

Pro simulaci používal fázovací články v kombinaci s hřebenovými filtry [21].

#### Schroederův reverberátor

Nevýhodou hřebenového filtru je jeho slyšitelné zabarvení signálu. Pro krátké opakující se simulované odrazy bude zabarvení velmi výrazné. Mimo maxima hřebenového filtru bude úbytek energie velmi rychlý. Pokud bychom se snažili tento problém vyřešit delším zpožděním, budou slyšet jednotlivá opakování spíše jako *echo*.

Tento problém by měl být řešitelný použitím fázovacích článků, které mají frekvenční odezvu rovnou.

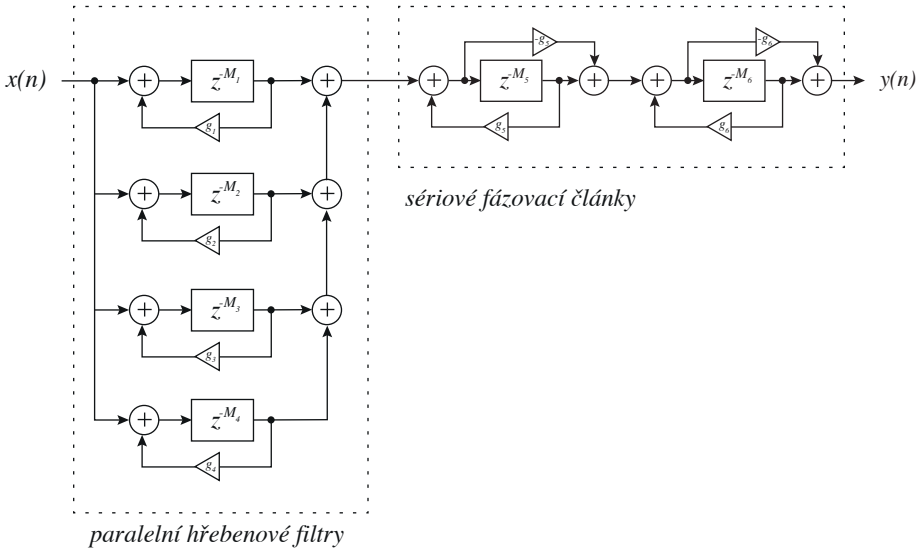
Pokud se snažíme zvýšit hustotu odrazů, je efektivní zařadit fázovací články do série. Docílíme tím toho, že následující články vytváří odrazy z odrazů které vytvořily články předchozí. Problémem je využití hřebenových filtrů v sérii. To z důvodu jejich filtrace kmitočtového spektra. Následující články pracují s již omezeným spektrem po průchodu články předchozími. Je ale možné zařadit je paralelně. Paralelní kombinace hřebenových filtrů s různými dobami zpoždění může vytvořit velmi hustý a přirozeně znějící základ pro další zpracování. Hustota odrazů je v tomto případě součtem hustot odrazů výstupů jednotlivých hřebenových filtrů.

Schroder navrhl reverberátor skládající se z paralelní kombinace hřebenových filtrů a následné sériové kombinace fázovacích článku, viz 1.15

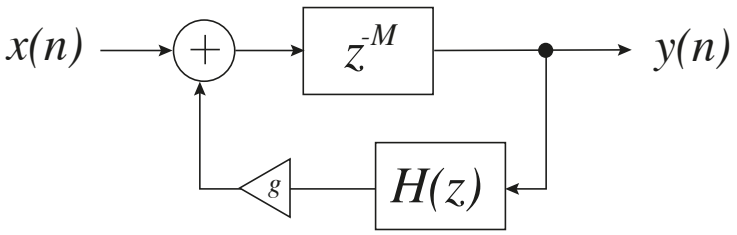
### 1.7.2 Moorerův reverberátor

Schroederův reverberátor je funčkní pro krátké časy, ale pro delší časy a impulsní zvukové zdroje může vytvářet nepříjemné a nepřirozené artefakty. Především pocit dýchání a třepotání zvuku. Pro delší časy dozvuku se také může zdát pozdější část dozvuku výrazně zabarvená. Největším problémem ale je, že hustota odrazů neodpovídá realitě a nenarůstá s časem. Stejně tak frekvenční spektrum dozvuku není časově závislé.

Moorerův reverberátor využívá principu Schroederova, ale přídává některá vylepšení. Aby vyřešil poslední zmiňovanou nevýhodu Schroederova přístupu, tedy absenci frekvenční závislé doby dozvoku, vložil do každé zpětné vazby hřebenových filtrů dolní propust prvního řádu. Tím tedy jednoduše docílil že kratší doby dozvuku na vyšších kmitočtech oproti kmitočtům nižším [16, 17].




Obr. 1.15: Schroederův reverberátor

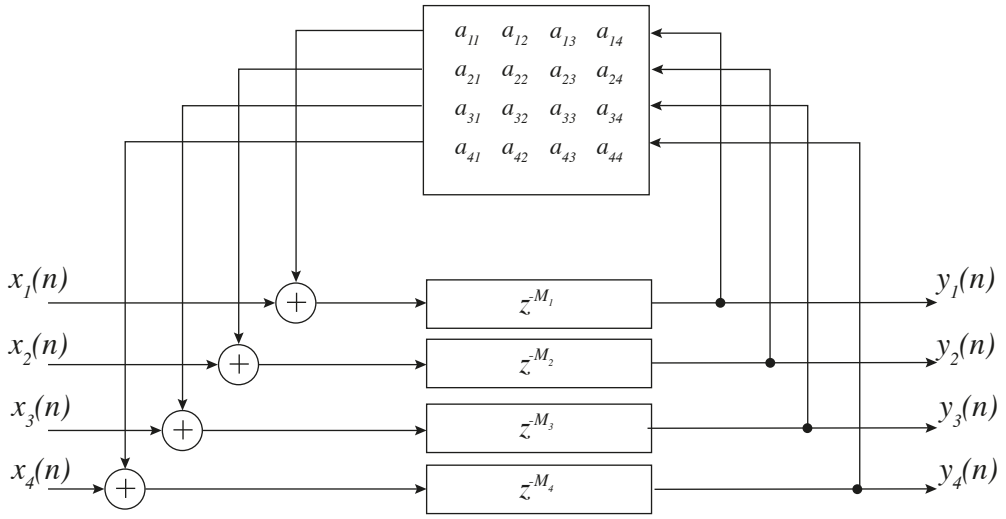



Obr. 1.16: Moorerův hřebenový filtr

### 1.7.3 Zpětnovazebná zpožďovací matice (Feedback delay network)

J. Stautner a M. Puckette navrhli strukturu pro simulaci poslechových prostor na základě *zpětnovazební zpožďovací sítě* (feedback delay network). Ta se skládá ze čtyř zpožďovacích linek a zpětnovazebné matice, viz obrázek 1.17

Tato struktura umožňuje aby výstup každé zpožďovací linky byl přiveden na vstup ostatních zpožďovacích linek, na základě parametrů daných maticí. Díky této struktuře je možné dosáhnout podstatně vyšší hustoty odrazů, než v případě Schroederova reverberátoru, pokud má matice dostatek nenulových prvků. Zajímavé je, že v případě jednotkové matice, nám FDN funguje identicky jako paralelní kombinace hřebenových filtrů [16, 4].




Obr. 1.17: Čtyřkanálová FDN

Stautner a Puckette navrhli následující matici:

$$\boldsymbol{A} = g\frac{1}{\sqrt{2}}\begin{bmatrix} 0 & 1 & 1 & 0 \\ -1 & 0 & 0 & -1 \\ 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \end{bmatrix}, \tag{1.17}$$

kde **A** je zpětnovazební matice, a $g$ je zpětnovazební koeficient. Pro stabilitu systému musí být splněno že $|g| < 1$.
Tento systém však nebude schopný produkovat dozvuk který je kmitočtově závislý. To je ale pro přirozeně znějící dozvuk klíčová vlastnost. Na práci Stautnera a Pucketta dále stavěl Jot [5].

### Jotův reverberátor

Jotovým záměrem bylo vytvořit reverberátor schopný simulovat absorpční vlastnosti. Tedy reverberátor schopný kmitočtově závislé doby dozvuku.

Toho docílil tím, že za každou zpožďovací linku FDN, obdobně jako Moorer, zařadil kmitočtový filtr [10].

### 1.7.4 Reverberátor založený na sametovém šumu (Velvet Noise Reverberation)

Jak již bylo zmíněno, v ideálním případě se difúzní část dozvuku, po odeznění prvotních odrazů, vjemově přibližuje šumu.

Z frekvenčního hlediska můžeme na odezvu místnosti pohlížet jako na odeznívající sinuoidy s hustotou $N_{mod}$ (počet módů místnosti na jeden Hz), která roste s frekvencí.

$$N_{mod}(f) = 4\pi V f^2/c^3 \ , \tag{1.18}$$

Z pohledu času, lze na odezvu místnosti pohlížet jako na sérii odrazů od jednotlivých ploch s rostoucí hustotou odrazů $N_o dr$ s časem.

$$N_{odr}(t) = 4\pi c^3 t^2/V \ , \tag{1.19}$$

Difúzní část dozvuku je tedy možno aproximovat sekvencí impulsů, vážených exponenciálním útlumem [11]. Pro fyzikální simulaci pomocí konvoluce signálu s impulsovou odezvou prostoru bychom potřebovali $N = f_s T$ počet vzorků, kde $N$ je počet vzorků, $f_s$ vzorkovací frekvence a $T$ požadovaná doba dozvuku. Lze však požít FIR s řídkou impulsovou odezvou, která bude podstatně efektivnější na zpracování. Experimentálně bylo zjištěno, že pro dostatečnou hustotu simulovaných odrazů je potřeba impulsová odezva obsahující alespoň 2000 nenulových hodnot za vteřinu [11].

2000 vzorků na vteřinu je však stále hodně. Při požadovaném dozvuku 3 vteřiny se dostáváme k šesti tisicům vzorků. Pokud bude používat vzorek délky 100 *ms* obsahující 200 nenulových koeficientů, zachováváme požadovanou hustotu 2000 nenulových koeficientů za sekundu. V zpětné vazbě je nutno mít zařazen zpožďovací článek odpovídajícího zpoždění. Dále je možno tento algoritmus rozšířit o dolní propust zařazenou ve zpětné vazbě, pro simulování frekvenční závislosti dozvuku [20].

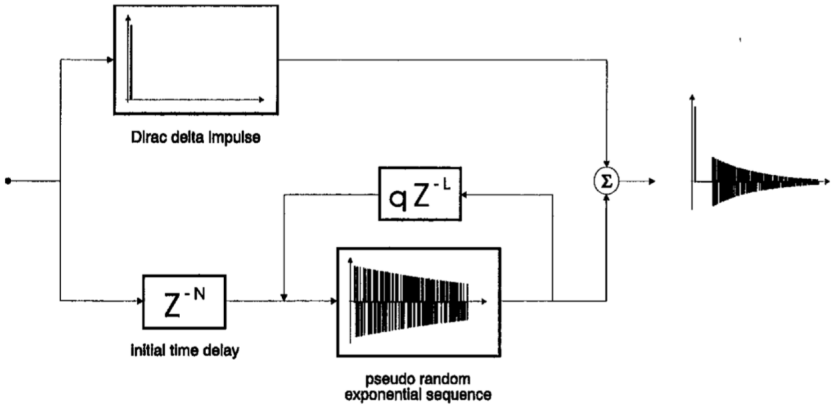
Diskrétní časová odezva takovéhoto systému tedy bude následující:

$$g[n] = h[n] + qh[n - L] + q^2 h[n - 2L] + .... \ , \tag{1.20}$$

kde *h[n]* je náhodný signál a q udává útlum zpětné vazby, tedy *0<q<1*;

#### Sametový šum

Bylo zjištěno, že některé řídké pseudo náhodné impulzní sekvence jsou subjektivně vnímány jako šum, který působí jemněji a přirozeněji než Gaussovský bílý šum.

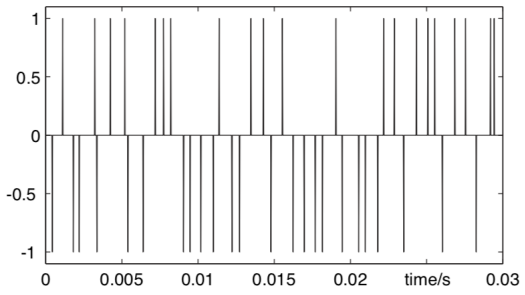



Obr. 1.18: Schéma systému pro pseudo náhodný generátor dozvuku, převzato [20]

Takováto posloupnost diracových impulsů s proměnlivými znaménky lze vygenerovat následovně:

$$n(k) = \sum_{m=1}^{M} a(m)u(k - \text{round}(\tfrac{T_d}{T_s}(m + \text{rnd}(m)))) \,, \tag{1.21}$$

kde *rnd(m)* je funkce generující náhodné číslo v rozmezí 0 až 1, *round(..)* zaokrouhluje argument funkce na celá čísla a funkce *a(m)* náhodně mění znaménko. $T_d$ je průměrná doba mezi impulsy, tedy obrácená hodnota impulsní hustoty) a $T_s$ je vzorkovací perioda. Pro ilustraci, na obrázku 1.19 je vidět třiceti milisekundová sekvence, generovaná pomocí vzorce 1.21.




Obr. 1.19: sekvence sametového šumu o délce 30 *ms* s hustotou 1500 špiček za sekundu, převzato [11]

# 2 Herní engine Unity

## 2.1 Herní engine

Pod pojmem herní engine rozumíme softwarové vývojové prostředí primárně určené pro tvorbu digitálních her. Nejdůležitějšími jeho vlastnostmi je podpora vykreslování dané scény v reálném čase, a to buď ve třírozměrném nebo dvourozměrném prostoru, simulaci fyziky a kolizí a podporu zpracování zvuku v reálném čase. Pro práci se zvukem nás nejvíce zajímá možnost automatizace parametrů. Základní je simulace poslechového prostoru, tedy panorámování a možnost ovládání hlasitosti jednotlivých zvukových zdrojů na základě jejich polohy ve virtuálních prostorách vůči poloze posluchače. Většina dnešních enginů ale nabízí i možnost realtime zpracování zvukového signálu pomocí efektových jednotek jak je známe z DAW softwarů. Nalezneme v nich typicky kmitočtové filtry, dynamické procesory, zvukové generátory a mnoho dalších. Ideálně ve vývojovém prostředí najdeme i možnost různých sběrnic, připomínajících architektury mixážních pultů pro zjednodušení práce. Dále v této práci budou popsány specifika herního enginu Unity.

## 2.2 Historie

Herní engine *Unity*, původně *Unity3D*, byl poprvé představen v roce 2005 jako herní engine exkluzivní pro systém OS X, dnes MacOS. Už od počátku byl mířen na nezávislé vývojáře. Sami jeho tvůrci původně měli zájem o tvorbu her, ale chyběl jim dostatek dostupných kvalitních technologií. Všechny enginy v té době byly produkovány velkými herními společnostmi. Byly tedy drahé, často velmi specializované a pro začínající vývojáře vpodstatě nedostupné. Cílem Unity bylo vytvořit univerzální herní engine dostupný všem. Zároveň tento engine neslouží jen pro vývoj digitálních her. Více než třetina uživatelů jej využívá k jiným účelům, například architektuře nebo uměleckým instalacím.

Brzy po vydání byla přidána podpora pro systém Windows a také podpora Unity Web Player. Unity se velmi soustředilo na možnost hraní v internetovém prohlížeči, kde v této době vedl Flash Player. V současné době nabízi možnost exportu do HTML5 WebGL s velmi dobrou optimalizací.

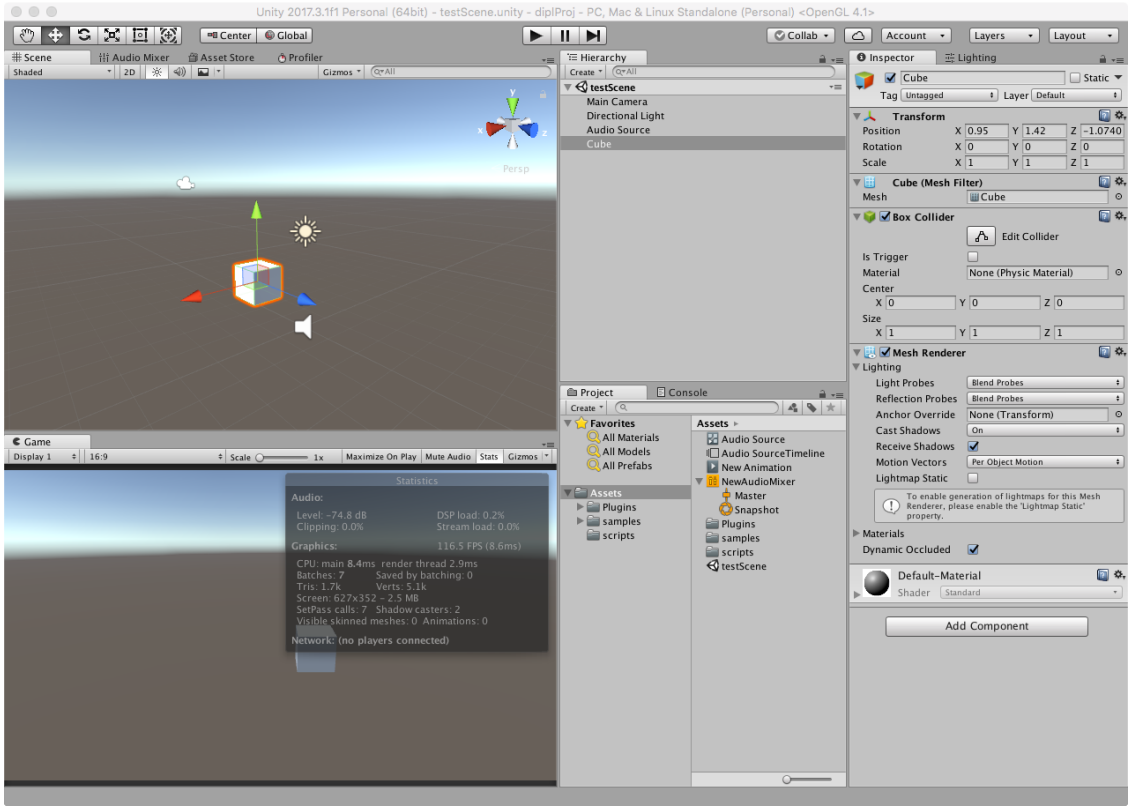
S příchodem smartphonů byla brzy přidána podpora systémů iOS a Adroid. Bylo to právě obrovskou popularitou těchto platforem, díky kterým se Unity stalo v současné době světově nejpoužívanějším herním enginem.

V současné době Unity podporuje všechny desktopové systémy, tedy *Windows (a Universal Windows Platform), MacOS* a *Linux*. Stejně tak mobilní platformy *iOS, Android, Windows Mobile* a všechny konzolové systémy (*Playstation 3/4, XBOX*

*360/ONE, Nintendo Switch,*). Také podporuje naprostou většinu systému pro virtuální realitu (*Oculus Rift, Gear VR*, atd.) a také již zmíněné WebGL, umožňující hraní v jakémkoliv moderním prohlížeči bez nutnosti zásuvných modulů [8].

Unity je zdarma volně ke stažení ve své plné verzi. Kompletní dokumentace je k dispozici na webových stránkách Unity.

## 2.3 Editor




Obr. 2.1: Unity Editor

Vývojové prostředí Unity, označované jako *Editor* je na obrázku 2.1. Toto je standartní rozložení vývojového prostředí. Oken je ovšem mnohem více, dají se libovolně přesunovat a zvětšovat.

V jeho levé horní části je žálozka *Scene*. Ta slouží pro tvorbu dané scény, tedy vidíme zde všechny objekty, které ve scéně máme, můžeme s nimi pohybovat, případně je rotovat a zvětšovat. Na obrázku 2.1 tedy ve scéně máme jen krychli, světlo, kameru a jeden zvukový zdroj.

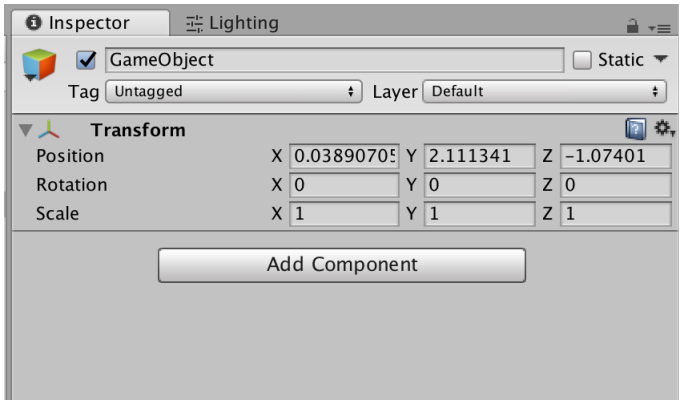
V levé spodní části je okno *game*, kde vidíme scénu v reálném čase pohledem z kamery, tedy jak hru uvidí hráč. V *editoru* lze přecházet mezi módy *edit* a *play*.

V *play* módu můžeme v okně hru přímo hrát, bez nutnosti buildu celé hry. Výhodou také je že v *play* módu máme možnost měnit veškeré parametry scény a objektů v ní.
Dále uprostřed dole je okno *Project*. To odpovídá *Finderu* v systému MacOS, případně *Exploreru* v systému Windows. Zde se nachází stromová struktura celého projektu. V případě hry tedy typicky veškeré *assety*, rozřazené do složek. Pojmem *asset* rozumíme jakýkoliv zdrojový soubor, který v projektu využíváme. Jde tedy o veškeré scény, modely, textury, zvukové soubory, ale také skripty, shadery, materiály a tak dále.

Uprostřed nahoře je okno *Hierarchy*. V tomto okně je výčet všech *GameObjectů* v právě načtené scéně, případně scenách. Na obrázku 2.1 zde tedy vidíme *GameObjecty* kamery, světla, zdroje zvuku a krychle stejně jako v okně *Scene*.

Poslední okno v pravé části obrazovky je okno *Inspector*. V tomto okně jsou veškeré *componenty* vybraného *GameObjectu*. V následující kapitole budou tyto pojmy blíže vysvětleny. V našem případě zde jsou componenty vybraného, tedy objektu krychle [14, 15].

### 2.3.1 GameObject

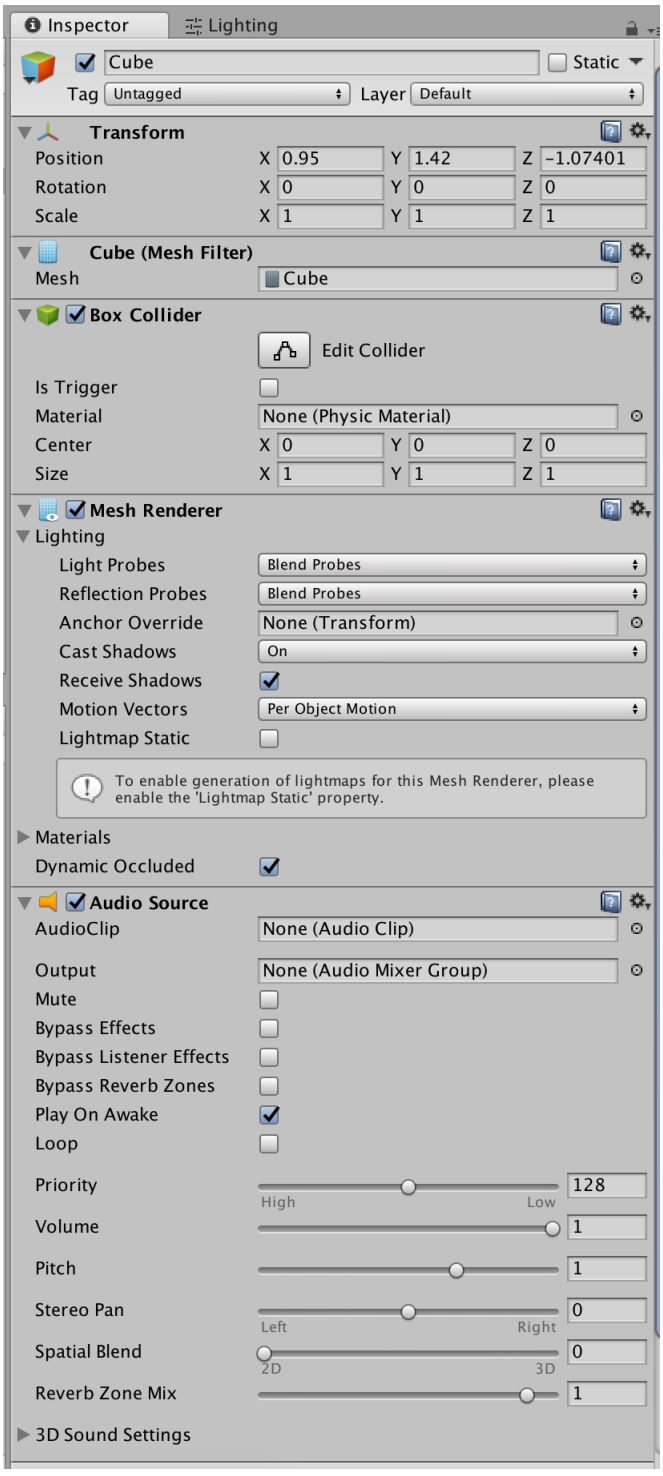



Obr. 2.2: GameObject

*GameObject* je základním stavebním prvkem scény. Každý *GameObject* má jasně definovánu svoji polohu, rotaci a velikost v prostoru scény. Ty jsou dány componentou *Transform*. Chování daného GameObjectu je poté dáno jeho componentami.

### 2.3.2 Component

*Componenty* jsou vpodstatě skripty ovlivňující chování daného *GameObjectu* v herním enginu. Uveďme si příklad na obrázku 2.3.




Obr. 2.3: Componenty

Tento *GameObject* má 5 component. První je *Transform*, který je nezbytný pro každý objekt. Jak už bylo zmíněno, určuje pozici, rotaci a měřítko objektu. Tyto tři hodnoty jsou definovány pomocí tří veřejně přístupných proměnných datového typu

`Vector3`.

Další je componenta *Mesh Filter*. Ta určuje geometrii daného objektu, jejím jediným parametrem je reference na zdrojový soubor obsahující mesh.

Componenta *Box Collider* nese informace pro simulaci kolizí. Má informace o geometrii krychlového collideru, daného opět dvěmi proměnnými *Center* a *Size* typu `Vector3`, dále referenci na fyzikální materiál, určující fyzikální vlastnosti daného objektu, a jeden parametr typu `bool` určující zda se jedná o skutečně kolizní objekt nebo jen zpouštěč událostí.

*Mesh Renderer* je componentou zajišťující vykreslování objektu na základě geometrie a dalších parametrů. Zde nastavujeme například materiály, interakci se světly atd.

Poslední componentou, která nás zajímá nejvíce, je *Audio Source*. Tuto componentu si probereme podrobněji.

### 2.3.3 Bázové třídy

V této části by bylo dobré zmínit také některé bázové třídy, z kterých mohou vycházet uživatelsky vytvářené skripty v jazyce C#. Ty slouží pro určení vlastní herní logiky v rámci enginu, editaci parametrů v reálném čase, zpracování vstupů atd. Enginem jsou tyto skripty vnímány na stejné úrovni jako *Componenty*, tedy mohou se na ně odkazovat a naopak.

#### MonoBehaviour

`MonoBehaviour` je bázová třída, z které defaultně dědí veškeré uživatelsky vytvářené skripty a je používána jako bázová třída pro většinu vytvořených ovládacích skriptů.

Jejími základními funkcemi jsou funkce `Awake()` a `Start()`. Ty jsou volány při vytvoření objektu, případně přidání componenty na objekt. V případě že objekt existuje ve scéně před jejím způštěním, tedy že nevzniká dynamicky během běhu programu, jsou tyto funkce volány při zpuštění [18].

Nejpoužívanější funkcí je funkce `Update()`, která je volána při každém snímku. Tzn pokud se hra vykresluje na frekvenci 60 snímku za vteřinu, tato funkce bude volána šedesátkrát za vteřinu. Podobnou funkcí je `FixedUpdate()`, která je volána s pevně danou periodou, využívá se především pro simulaci fyziky. Více informací lze najít pod heslem *MonoBehaviour* v dokumentaci, viz [27].

#### Editor

`Editor` je bázová třída umožňující změnu defaultního uživatelského rozhraní (GUI) skriptů dědících z `MonoBehaviour`.

V rámci inspektoru tedy můžeme na GUI skriptu vytvářet textová pole, posuvníky, zadávat křivky atd. Je nutné aby skript dědící z třídy `Editor` měl stejný název jako skript, jehož GUI přetěžuje, doplněn o slovo "Editor". Tedy například skript `ReverbVolume.cs` bude mít odpovídající skript `ReverbVolumeEditor.cs`. Pro správné fungování je dále nutné mít tyto skripty umístěny ve složce *Editor* v hlavní složce *Assets*. Více informací lze najít pod heslem *Editor* v dokumentaci, viz [27].

### 2.3.4 Audio Source

Audio Source componenta (a jí odpovídající třída `AudioSource`), je hlavní nástroj pro přehrávání zvuku ve scéně v Unity. Její uživatelské rozhraní je vidět na obrázku 2.4. Jeho hlavním parametrem je reference na zdrojový soubor typu AudioClip. To může být jakýkoliv zdrojový soubor podporovaného formátu, tedy .mp3, .ogg, .wav, .aiff. [26]

Dalším parametrem je cílový *Audio Mixer Group*. Co to je a k čemu slouží bude rozepsáno později. Pokud není žádný vybrán, signál jde přímo do hlavního výstupu. Dále zde je 6 parametrů typu `bool`. Ty není nutné podrobně vysvětlovat, jejich název je dostatečně vypovídající.

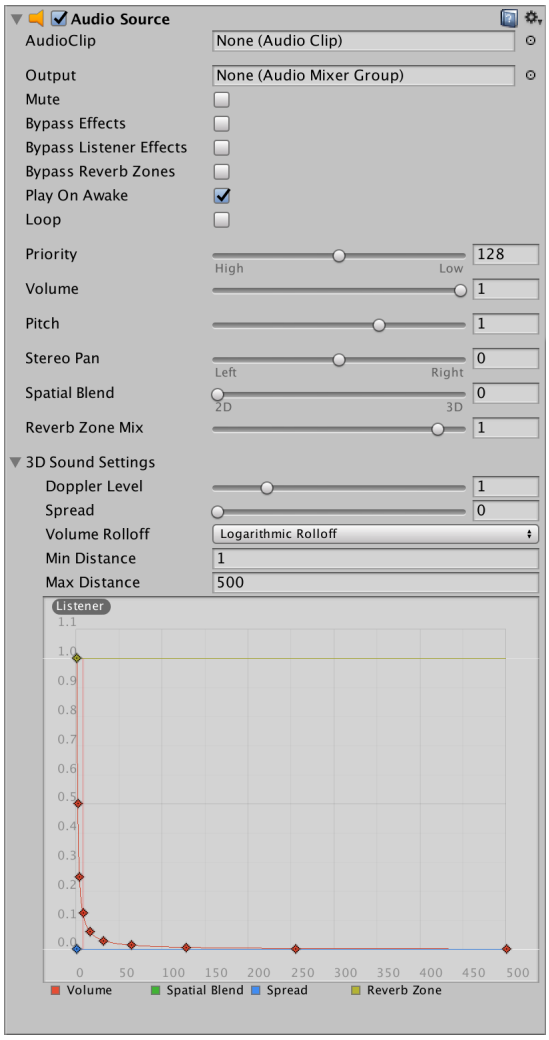
Unity je schopné přehrávat maximálně 32 instancí Audio Source zároveň. Parametrem *priority* jednoduše určujeme, které objekty engine vypne při překročení tohoto limitu. Priorita 0 je maximální. Typicky tedy například hudba bude mít prioritu 0. *Volume* udává hlasitost daného zdroje zvuku, *Pitch* rychlost přehrávání, *Stereo Pan* rozložení ve stereofonní bázi.

Parametr *Spatial Blend* určuje v jaké míře bude zvuk pozicován na. základě vzájemné polohy posluchače a zdroje zvuku. V případě že bude posuvník parametru vlevo u popisky *2D*, nebude ovlivňován nijak, tedy pouze na základě předchozího parametru *Stereo Pan*. Toto se využivá pro hudbu, ambientní zvuky, případně dialogy. V případě posuvníku zcela vpravo, bude zvuk ovlivňován plně, využijeme toho tedy pro všechny zvukové objekty ve scéně. Poslední parametr *Reverb Zone Mix* určuje jak moc bude zvuk ovlivňován nativní simulací prostor v Unity.

V záložce *3D Sound Settings* najdeme parametry pro simulaci dopplerova efektu, simulovanou šírku zdroje zvuku a dále automatizaci větší částí výše popsaných parametrů v závislosti na vzdálenosti od posluchače. Křivka této závislosti je uživatelsky editovatelná [13].

### 2.3.5 Audio Listener

Compomenta *AudioListener* a jí odpovídající datový typ `AudioListener` reprezentuje posluchače ve scéně. Nemá žádné vlastní parametry ale je bezpodmínečně nutné



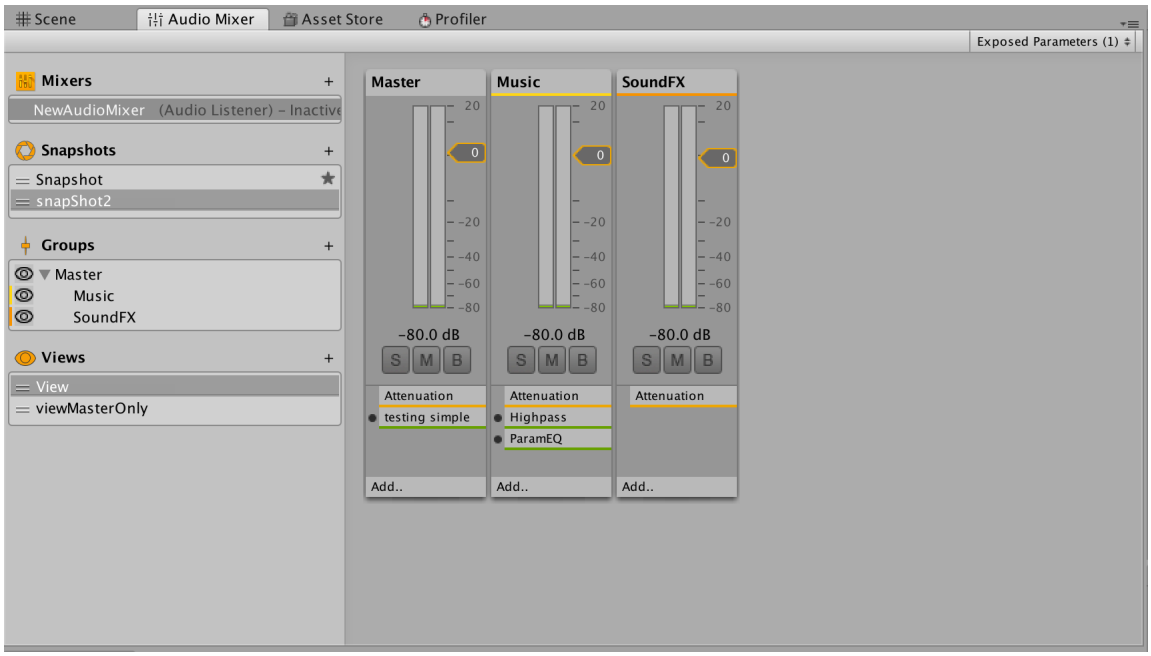

Obr. 2.4: Audio Source

ji ve scéně mít. Ve scéně musí být právě jedna instance této componenty. Defaultně je umístěna na hlavní kameře.

## 2.4 Audio Mixer

Pro obsah této práce je v rámci Editoru pro nás důležité ještě jedno okno, které nebylo zmíněno. Jedná se o okno Audio Mixer, viz obrázek 2.5. Toto okno nám umožňuje efektivní ovládání a routování zvuku v rámci editoru. V levé části vidíme jednotlivé instance a celkové nastavení daného Audio Mixeru.

V pravé části jsou jednotlivé AudioMixerGroup, které nápadně připomínají kanálové zobrazení typické pro mixážní pulty nebo DAW software. AudioMixerGroup již bylo v této práci zmíněno, jako parametr componenty Audio Source. Právě do




Obr. 2.5: Audio Mixer

těchto AudioMixerGroup můžeme směřovat signál z jednotlivých Audio Source pro následné zpracování.
V obrázku 2.5 tedy vidíme 3 instance AudioMixerGroup, zjednodušeně budou dále označovány jako *kanály*. Zleva to je *Master*, *Music* a *SoundFX*. V levé části spektra v záložce *Groups*, je k vidění jejich hierarchie. Tedy vidíme, že *Music* a *SoundFX* jsou podskupinami *Master*. V praxi to znamená že signál, který pošleme do *Music* dále poputuje o skupinu výš do *Master* [13].

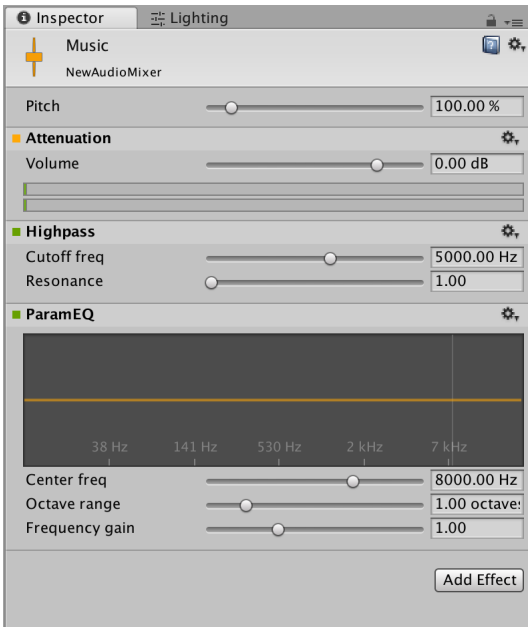
Každý kanál má své ovládací prvky. Hlavním je tedy tahový fader, ovlivňující hlasitost dané skupiny. Dále je to skupina tlačítek *Solo*, *Mute* a *Bypass*. Pod nimi se nachází prostor pro efekty.Unity má defaultně množství efektů, pro příklad *Lowpass*, *Compressor* atd. Pokud vybereme daný kanál, v okně Inspector se nám zobrazí nastavení efektů pro daný kanál, viz 2.6 [14].

## 2.5 Native Audio Plugin SDK

Native Audio Plugin SDK je vývojová sada poskytovaná Unity pro tvorbu zvukových zásuvných modulů. Zásuvný modul samotný se skládá z dvou částí.

První z nich je DSP zásuvný modul v jazyce C, případně C++, implementovaný jako .dll knihovna pro Windows nebo .dylib knihovna pro systém OSX.

Druhou částí je uživatelské rozhraní (GUI). Implementace GUI je pomocí jazyka C#, není však nutné. Unity ze zadaných parametrů v rámci DSP zásuvný mo




Obr. 2.6: Audio Mixer Group detail

automaticky generuje elementární GUI. Ukázky GUI efektů jsou například na obráku 2.6. Detailněji toto bude rozvedeno dále. V rámci jedné .dll, případně .dylib knihovny je možné mít více zásuvných modulů.

Celé SDK je vpodstatě pouze soubor `AudioPluginInterface.h`. V souborech SDK dále najdeme soubory `AudioPluginUtil.h` a `AudioPluginUtil.cpp`. Ty zde jsou pro jednoduší práci s parametry a podporu více zásuvných modulů v rámci jedné knihovny.

V rámci `AudioPluginUtil.h` je řada funkcí, nejdůležitější pro jsou však `InternalRegisterEffectDefition, CreateCallback` a `ProcessCallback`.

Fukce `InternalRegisterEffectDefition` slouží k inicializaci proměnných, které budou veřejné. Tedy ty, ke kterým Unity bude automaticky vytvářet GUI ovládácí prvky. V příloze C.1 je jedna takováto funkce pro názornost. Funkce je z ukázkového zásuvného modulu `PluginNoise.cpp`, který je součástí SDK. U každého parametru, voláme funkci `RegisterParameter`, která bere jako argumenty název této proměnné zobrazovaný v editoru, odpovídající proměnnou ale také maximální, minimální a defaultní hodnotu.

Funkce `CreateCallback` slouží k inicializaci vnitřních parametrů a proměnných na začátku jeho fungování. Funkce `ProcessCallback` je nejdůležitější pro fungování celého zásuvného modulu. Pro příklad ji najdeme v příloze C.2. Funkce je zavolána vždy když se naplní vstupní buffer, tedy právě zde probíhá samotné zpracování signálu. Parametry této funkce je proměnná `UnityAudioEffectState• state`. Tato

třída v sobě obsahuje pro nás důležité informace jako je například vzorkovací frekvence. Dalšími parametry jsou dva pointery `float* inbuffer` a `float• outbuffer`. Inbuffer je buffer se všemi vstupními vzorky, outbuffer naopak výstupní buffer, který je potřeba naplnit. Jak vstupní tak výstupní buffer je ve formě jednorozměrného pole obsahující všechny vzorky pro všechny kanály za sebou. Tedy pokud bychom měli délku bufferu na kanál 1024 vzorku, ve vstupním i výstupním bufferu bude prvních 1024 vzorků pro pravý kanál a následovat bude dalších 1024 vzorků pro kanál pravý.

Posledními parametry je `length` udávající počet vzorků pro kanál, `inchannels` určující počet vstupních kanálů a `outchannels` určující počet výstupních kanálů [25].

## 2.6 Audio Spatializer SDK

Audio Spatializer SDK nabízí nástroje pro tvorbu vlastní implementace auralizace (tedypanorámování a útlumu zdroje zvuku) v závislosti na vzájemné poloze hráče a posluchače. Zatímco zásuvné moduly vytvořené pomocí *Native Audio Plugin SDK* fungují v rámci kanálu AudioMixeru, implementace Spatializeru je nezávisle na každé componentě AudioSource. Výhodou Spatializeru je, že má defaultně informace o poloze zdroje a posluchače.

Audio Spatializer SDK vychází z Native Audio SDK, má však několik odlišností, které budou dále zmíněny. Abychom zdrojový soubor definovali jako Spatializer, je nutné toto v rámci funkce `InternalRegisterEffectDefinition` definovat voláním `definition.flags |= UnityAudioEffectDefinitionFlags_IsSpatializer;`. Pro správné fungování je poté nutné vybrat námi zvoleny Spatializer zásuvný modul v rámci nastavení Unity, záložka *Edit - Project settings - Audio*.

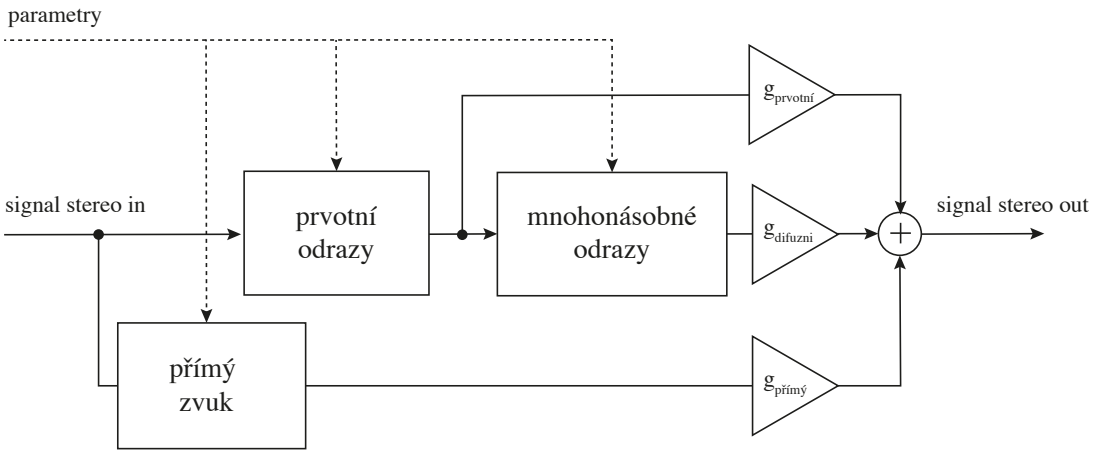
Jak již bylo zmíněno, Spatializer má informace o poloze zdroje a posluchače. Tyto informace jsou obsažené ve struktuře `UnityAudioSpatializerData`. Pro nás nejdůležitější jsou proměnné `listenermatrix` a `sourcematrix`, což jsou matice 4x4 určující vzájemnou polohu posluchače a zdroje zvuku.

Další funkce a parametry, jakými je například funkce `ProcessCallback` nebo `CreateCallback` jsou implementovány stejně jako v případě Native Audio Plugin SDK zmíněného výše v kapitole 2.5 [24].

# 3 Návrh

## 3.1 Přehled

Celkové blokové schéma zásuvného modulu je na obrázku 3.1

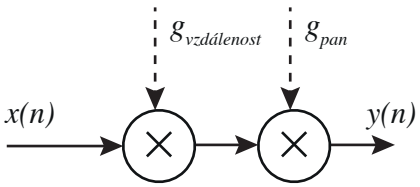



Obr. 3.1: Blokové schéma zásuvného modulu

Celá realizace se dá zjednodušit na 3 bloky. Blok simulující prvotní odrazy, blok simulující mnohonásobné (difúzní) odrazy a blok ovlivňující přímý zvuk. Dále budou tyto části rozebrány podrobněji.

### 3.1.1 Přímý zvuk

Pro přímý zvuk je realizace velmi jednoduchá. Vstupní signál je utlumen v závislosti na vzdálenosti od zdroje zvuku a dále je panorámován na základě úhlu který svírá posluchač se směrem, z kterého zvuk přichází. na obrázku 3.2 je znázorněna situace pro jeden z kanálů, tzn levý nebo pravý.




Obr. 3.2: Blokové schéma zpracování přímého zvuku

### 3.1.2 Prvotní odrazy

Realize pro simulaci prvotních odrazů vpodstatě odpovídá schématu na obrázku 1.11. Oproti tomuto schématu jsou však všechny parametry jednotlivých bloků variantní v čase a jsou pro každý odraz přidány další dva kmitočtové filtry, jeden high shelving a jeden low shelving. Lze měnit hodnoty zpoždění zpožďovacích linek, útlum jejich výstupů a mezní kmitočty filtrů. Také je možné v reálném čase měnit váhovací koeficienty pro panorámování daného odrazu. Zjednodušené schéma je na obrázku 3.3.

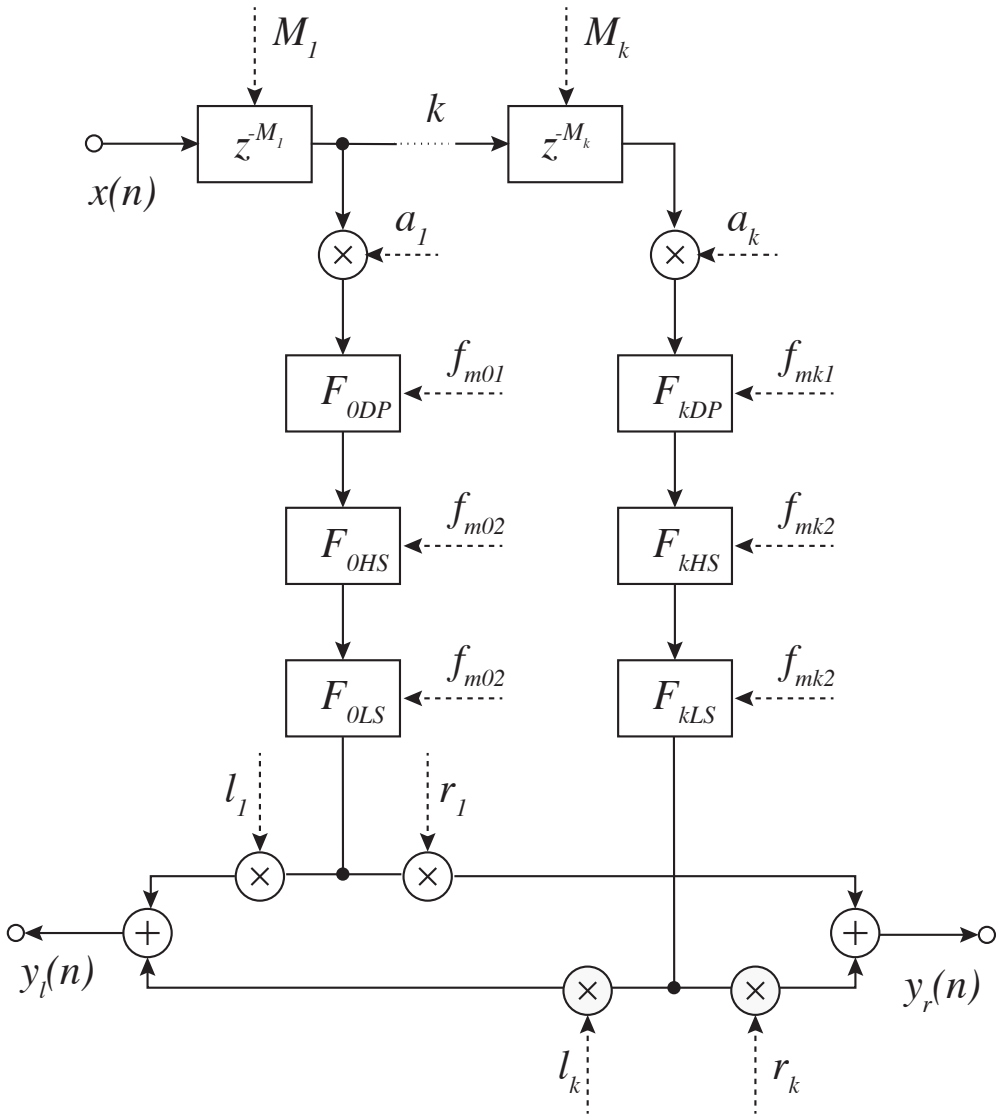
### 3.1.3 Mnohonásobné odrazy

Realizace systému pro simulaci mnohonásobných odrazů vychází z poznatku z kapitoly 1.7.4. Tento systém lze realizovat velkým množstvím zpožďovacích článků s narůstající dobou zpoždění, které jsou váhovány exponenciálně se snižující hodnotou zesílení. Schéma takového systému je na obrázku 3.4. S Pro parametry $M_k$ a $g_k$ platí:

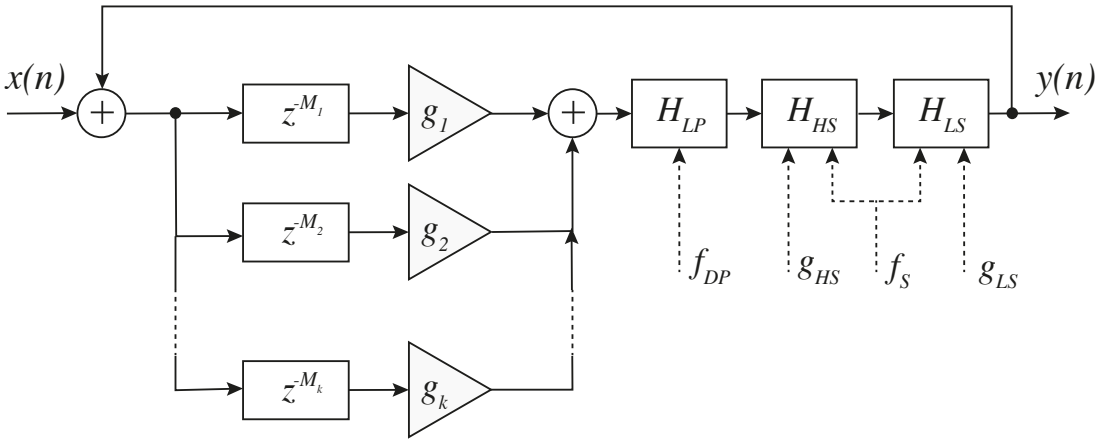
$$\begin{aligned} &M_1 < M_2 < M_3 < ... < M_k \ , \\ &g_1 > g_2 > g_3 > .. > g_k \ , \end{aligned} \tag{3.1}$$

kde $M$ je doba zpoždění a $g$ zesílení.

Dále jsou zde zařazeny kmitočtové filtry dolní propusti $H_{DP}$, high shelving $H_{HS}$ a low shelving $H_{LS}$. Jejich parametry, tedy mezní kmitočty případně zesílení je možné měnit v čase.




Obr. 3.3: Návrh realizace systému simulující k prvotních odrazů




Obr. 3.4: Návrh realizace systému simulující mnohonásobné odrazy

# 4 Výsledky

V rámci diplomové práce byl vytvořen zásuvný modul pro herní engine Unity ve formě knihovny `.dylib` pro systém MacOS. Samotný soubor se jmenuje *AudioPluginNoise.bundle* a nachází se ve složce `Plugins` v rámci složky `Assets` Unity projektu. Složka `Plugins` je složka, z které Unity při zpuštění projektu načítá veškeré zásuvné moduly, ať už pro práci se zvukem nebo jiná rozšíření.

Samotný C++ projekt byl vytvořen v rámci vývojového prostředí Xcode. Projekt se nachází ve složce `NativeCode` ve složce Unity projektu. Dále bude podrobněji popsán zdrojový kód zásuvného modulu v jazyce C++.

## 4.1 C++ Zásuvný modul

Samotná implementace zásuvného modulu v jazyce C++ se skládá ze zdrojových souborů `Plugin_Reverb.cpp` a `Plugin_Spatializer.cpp`. `Plugin_Reverb.cpp` využívá Unity Native Audio SDK, popsané v kapitole 2.5 a `Plugin_Spatializer.cpp` využívá Unity Spatializer SDK popsané v kapitole 2.6.

Dále byly vytvořeny ovládací skripty fungující v prostředí enginu Unity, a to `ReverbVolume.cs` a `ReverbUnit.cs`, které na základě vstupu upravují parametry dozvuku.

V rámci zdrojového souboru byly vytvořeny třídy implementující kmitočtové filtry dolní propusti a high shelvingu filtru a kruhový buffer, jmenovitě to jsou třídy `LowPass`, `HighShelving` a `BasicBuffer`.

Dále byly v rámci zdrojového souboru vytvořeny implementace jednotlivých funkcí pro zpracování signálu, tedy funkce `InternalRegisterEffectDefinition`, `CreateCallback` a `ProcessCallback`.

### 4.1.1 Lowpass

Zdrojový kód třídy `Lowpass` je v příloze D.1. Třída implementuje dolní propust prvního řádu popsanou v kapitole 1.2.5.

Třída má dva konstruktory, jeden bezparametrický a jeden parametrický. Parametrický konstruktor volá konstruktor bezparametrický ale také nastavuje hodnoty vzorkovací frekvence a mezního kmitočtu filtru. Dále zde najdeme funkce pro nezávislé nastavení frekvence `SetFrequency` a vzorkovacího kmitočtu `SetSampleRate`. Poslední funkce `Iterate` má jako parametr vstupní vzorek, na kterém provádí filtraci a vrací zpracovaný vzorek.

### 4.1.2 HighShelving

Zdrojový kód třídy `HighShelving` je v příloze D.2. Třída implementuje high shelving filter prvního řádu popsaný v kapitole 1.2.5. Struktura je obdobná jako v případě LowPass filtru. Navíc jsou zde proměnné pro hodnotu útlumu, případně zesílení. Pro nastavení parametrů je zde pouze jedna funkce `SetParameters` nastavující hodnoty vzorkovací frekvence, mezního kmitočtu a hodnoty zesílení nebo útlumu.

### 4.1.3 BasicBuffer

Zdrojový kód třídy `BasicBuffer` je v příloze D.3. Třída implementuje funkcionalitu kruhového bufferu.

Parametrický konstruktor `BasicBuffer` má parametry vzorkovací frekvence a délky bufferu. Dále implementuje funkci `GetValue` na získání vzorku se zpožděním o `delayInSamples` vzorků, funkce pro vložení dalšího vzorku `SetNextValue` a funkci pro iteraci bufferu `Iterate`, posunující čtecí a zapisovací ukazatele.

### 4.1.4 Plugin_Spatializer.cpp

Tento zdrojový soubor implementuje pozicování zvuku v rámci stereofonní báze a dále simuluje prvotní odrazy.

#### InternalRegisterEffectDefinition

Funkce sloužící pro definování veřejných proměnných, tedy proměnných ke kterým lze přistupovat pomocí funkcí `SetFloat`, viz 4.2.2. Těchto parametrů je více než padesát proto zde nebudou vypsány všechny, jejich seznam a zdrojový kód je k nahlédnutí v příloze E.1.

Najdeme zde parametry odpovídající jednotlivým prvotním odrazům jako je jejich zpoždění, úhel dopadu oproti úhlu přímého zvuku, útlum a nastavení jednotlivých kmitočtových filtrů. Dále zde najdeme parametry vnořených fázovacích článků simulujících mnohonásobné odrazy, celkový útlum, případně zesílení, prvotních odrazů a další.

#### CreateCallback

V této funkci probíhá prvotní nastavení hodnot. Jsou zde tedy vytvořeny buffery a také volána funkce `SetShelfs` nastavující kmitočtové filtry. Zdrojový kód je v příloze E.1.

#### ProcessCallback

V rámci funkce ProcessCallback je nejprve spočítán z maticí poloh posluchače a zdroje, viz kapitola 2.6 vypočítán vektor směřující od posluchače ke zdroji a z tohoto vektoru dále vypočítán úhel pod kterým zvuk přichází k posluchači. Dále jsou nastaveny hodnoty všech kmitočtových filtrů dle vstupních parametrů.

Následuje samotné zpracování prvotních odrazů ve vnořených cyklech `for`. Ze zásobníku jsou získány hodnoty odpovídající zpoždění jednotlivých odrazů, které jsou dále filtrovány a panorámovány. Celková hodnota prvotních odrazů poté slouží jako vstup bloku simulující vícenásobné odrazy pomocí vnořených fázovacích článků.

Samotný zdrojový kód je k nahlédnutí v příloze E.3.

### 4.1.5 Plugin_Reverb.cpp

Tento zdrojový soubor implementuje simulaci mnohonásobných odrazů v rámci zásuvného modulu.

#### InternalRegisterEffectDefinition

Tato funkce slouží pro deklaraci veřejných proměnných, které budou editovatelné v rámci editoru. Zdrojový kód je v příloze F.1. Jedná se o parametry:

- Delay Time, tedy doba dozvuku $T_{60}$ popsaná v kapitole 1.6.1.
- Low Pass Frequency, mezní kmitočet dolní propusti.
- Crossover frequency, mezní kmitočet high a low shelving filteru
- Low shelf absorption, absorpce pro nízké kmitočty
- High shelf absorption, absorpce pro vysoké kmitočty

#### CreateCallback

Funkce `CreateCallback` je volána v momentě zpuštění Unity nebo momentu kdy je zásuvný modul přidán na daný kanál.

V rámci této funkce proběhne inicializace objektu typu `EffectData`, který dále drží veškerá data. Probíhá zde také inicializace bufferů. Zdrojový kód je v příloze F.2.

#### ProcessCallback

Funkce `ProcessCallback` byla už zmíněna v kapitole 2.5. v této funkci probíhá zpracování signálu v reálném čase. Parametry funkce jsou pointery `inbuffer` obsahující všechny vstupní vzorky a `outbuffer`, který obsahuje výstupní vzorky. Tento pointer ukazuje na jednorozměrné pole typu `float` kde jsou nejprve všechny vzorky

pro první kanál a za nimi všechny vzorky pro druhý kanál. Zdrojový kód je v příloze F.3.

Samotné zpracování signálu tedy probíhá ve vnořených cyklech `for` pro všechny vzorky vstupního bufferu. Využívá se zde poznatků z kapitoly 1.7.4 a návrhu popsaného v 3.1.3 . Samotný algoritmus vychází z implementace základního reverbu využívaného v Unity, je však rozšířen o kmitočtové filtry.

V rámci cyklu *for* je nejprve pro každy kanál naplněno pole typu `Tap`, které obsahuje hodnoty zpoždění a útlumu každého zpožďovacího prvku. Hodnoty zpoždění jsou rostoucí, avšak rostoucí, jak bylo zmíněno v 3.1.3.

Dále následuje druhý cyklus `for`, iterující přes všechny prvky vstupního zásobníku, kde se v rámci cyklu `while` vrací hodnoty ze zpožďovacího zásobníku. Tyto hodnoty jsou sečteny, zpracovány kmitočtovými filtry. Společně se vstupním signálem jsou poté vloženy zpět do zpožďovacího zásobníku. Hodnotou samotnou je také plněn výstupní zásobník.

## 4.2 Unity skripty

Funkčnost celého systému a především nastavování parametrů v reálném čase v rámci enginu Unity zajišťují 2 skripty: `ReverbVolume.cs` a `ReverbUnit.cs`. Pro správné fungování simulace je bezpodmínečně nutné je nejen do projektu zařadit, ale také správně nastavit. Dále je k dispozici skript definující souborý typ AcousticMaterial, který bude popsán dále.

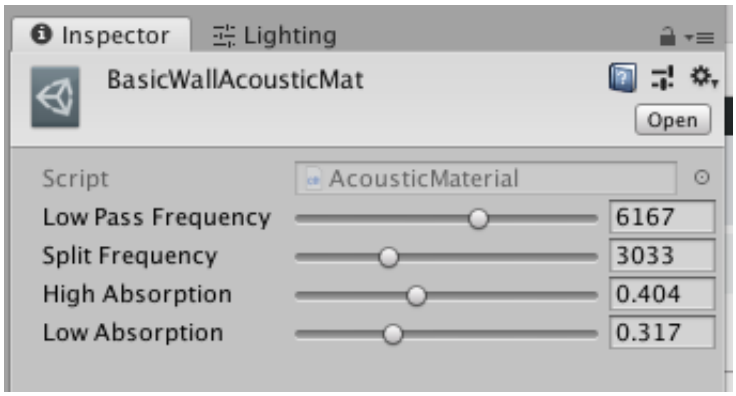
### 4.2.1 AcousticMaterial.cs

Skript `AcousticMaterial` dědí ze třídy `ScriptableObject`. Scritable object v rámci Unity fungují jako kontejnery na data. Není nutné je mít přiřazené objektu jako componentu, naopak je vytváříme mezi zdrojovými soubory, *assety*. Mají tedy pro nás stejnou funkci jako například zvukové soubory, textury atd.

Objekt *AcousticMaterial*, jak název napovídá určuje akustické vlastnosti materiálu. Uživatelsky jsou editovatelné následující parametry:

- *Low Pass Frequency*- frekvence dolní propusti v rozsahu 100 $Hz$ až 10 000 $Hz$
- *Split Frequency* - mezní kmitočet mezi vysokými a nízkými frekvencemi, také mezní kmitočet shelving filtrů v rozsahu 100 $Hz$ až 10 000 $Hz$
- *High Absorption* - absorpce materiálu na vysokých kmitočtech
- *Low Absorption* - absorpce materiálu na nízkých kmitočtech

V rámci projektu tedy můžeme vytvořit množství těchto materiálů pro různé povrchy s jinými vlastnostmi, například dřeva, betonu atd. Uživatelské rozhraní je vidět na obrázku 4.1.




Obr. 4.1: Uživatelské rozhraní skriptu Acoustic Material

### 4.2.2 ReverbVolume.cs

Skript `ReverbVolume.cs` určuje informace o velikosti akustického prostoru. Zde nastavujeme velikost poslechového prostoru pomocí parametru *Dimensions* typu `Vector3D`. Simulovaný prostor má tvar kvádru o stranách daných parametrem *Dimensions*. Velikost simulovaného prostoru není závislá na geometrii, která je vykreslována. Stejně tak není závislá na měřítku objektu k němuž je skript připojen. To je z důvodu větší volnosti při tvorbě prostředí. Prostor však je závislý na poloze a rotaci objektu.

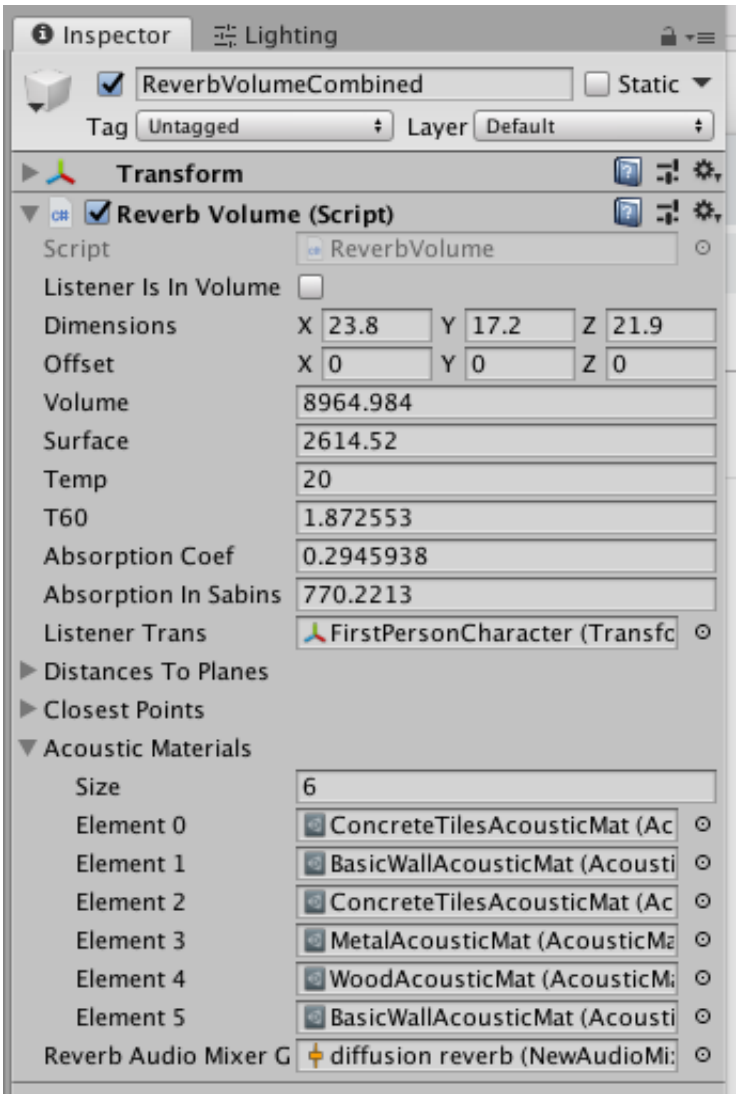
Dalším důležitým parametrem je pole objektů typu *Acoustic Material* o velikosti odpovídajícím počtu stěn, tedy 6. Pro každou stěnu tedy můžeme určit jiný akustický materiál.

Dále je zde nutné nastavit referenci na *Audio Mixer Group*, která má zařazen zásuvný modul reverbu, pro simulaci difúzního dozvuku.

Na základě zadaných parametrů tento skript počítá dobu dozvuku z objemu a pohltivosti jednotlivých materiálů, nastavení shelvingových filterů reverbu a dolních propustí a automaticky je nastavuje. Zároveň nese informaci zda se posluchač nachází v objemu nebo mimo něj. na základě toho se také parametry mění nebo ne.

#### Funkce ReverbVolume.cs

- `Awake()` je funkce volaná na začátku programu, avšak ještě před funkcí `Start`. Typicky slouží pro inicializaci veškerých proměnných a referenci, aby se ve funkci `Start()` mohla řešit herní logika. Pořadí volání této funkce napříč všemi skripty je náhodné [18]. v této funkci se vytváří nový statický `List` typu `ReverbVolume`, pokud ještě není inicializován. Pokud ano, je tento skript do tohoto dynamického pole přidán. Toto pole *reverbVolumes* tedy drží reference na všechny instance `ReverbVolume` v dané scéně.




Obr. 4.2: Uživatelské rozhraní ReverbVolume.cs

- `Start()` pouze určí referenci na objekt posluchače a volá funkci `ComputeAll()`
- `Update()` je funkce volaná každý vykreslovací snímek. Volá funkce `ComputeAll()` a `ComputeClosestPoint()`. Pokud se posluchač nachází v prostoru definovaném daným skriptem, volá se funkce `SetDiffusionParameters` nastavující parametry difúzního dozvuku.
- `ComputeAll()` pouze volá funkce počítající hlavní a doplňkové parametry. Také volá funkci `isInVolume`, jejíž návratovou hodnotu ukládá v proměnné typu `bool` *listenerIsInVolume*, určující zda je posluchač v daném prostoru.
- `ComputeVertices()` funkce která na základě polohy a rotace, definované `Transform` componentou a dále parametru *Dimensions* vypočte jednotlivé vrcholy stěn geometrie.

- `ComputeParameters()` vypočítává parametry objemu, plochy, času dozvuku, absorpčního koeficientu a absorpce v sabinech (*Volume*, *Surface*, *T60*, *AbsorptionInSabins*). Absorpce v sabinech se počítá dle kapitoly 1.3.1. Pomocí ní a hodnoty objemu místnosti můžeme vypočítat dobu dozvuku pomocí vzorce 1.15 zmíněného v kapitole 1.6.1
- `ComputePlanes()` slouží pro vypočet rovin, reprezentovaných datovým typem `Plane`. Roviny jsou definovány pomocí tří vrcholů pro každou stěnu prostoru a uloženy do pole *planes*
- `ComputeClosestPoint()` pomocí objektů rovin zmíněných výše počítá nejbližší body k posluchači na těchto rovinách a ukládá je do pole *closestPoints*.
- `GetDistancesToBounds()` má návratový typ `float[]`, tedy pole čísel typu `float`. v tomto poli jsou uloženy vzdálenosti posluchače od rovin *planes*.
- `GetClosestPointsToBounds()` má návratový typ `Vector3[]`, tedy pole třírozměrných vektorů. Vrací kopii pole *closestPoints*.
- `isInVolume()` vrací hodnotu typu `bool`, tedy `true` pokud se posluchač nachází uvnitř prostoru nebo `false` pokud se posluchač nachází mimo prostor.
- `GetAcousticMaterials()` vrací referenci na pole akustických materiálů popsaných v kapitole 4.2.1.
- `SetDiffusionParameters()` nastavuje parametry difuzního dozvuku. Pro správné fungování je potřeba mít nastavenou referenci na *Audio Mixer Group*, reprezentována proměnou *mix*. Volá se zde funkce `mix.setFloat(string name, float value)`, kde *name* je název parametru nastavený v rámci *Mixeru* a *value* je hodnota, kterou tomuto parametru přiřazujeme. Nastavuje se zde absorpce na nízkých a vysokých kmitočtech a mezní kmitočet mezi vysokými a nízkými frekvencemi, čas dozvuku *T60* a frekvence dolní propusti.

### 4.2.3 ReverbUnit.cs

Skript `ReverbUnit` slouží k nastavování parametrů Spatializer zásuvného modulu, componenty `AudioSource`. To znamená že vždy když chceme použít na zdroji zvuku rozšíření pro simulaci prvotních odrazů, je nutné zatrhnout argument *Spatialize* na samotné componentě, a poté na objekt přidat skript `ReverbUnit`, který parametry tohoto zásuvného modulu nastavuje.

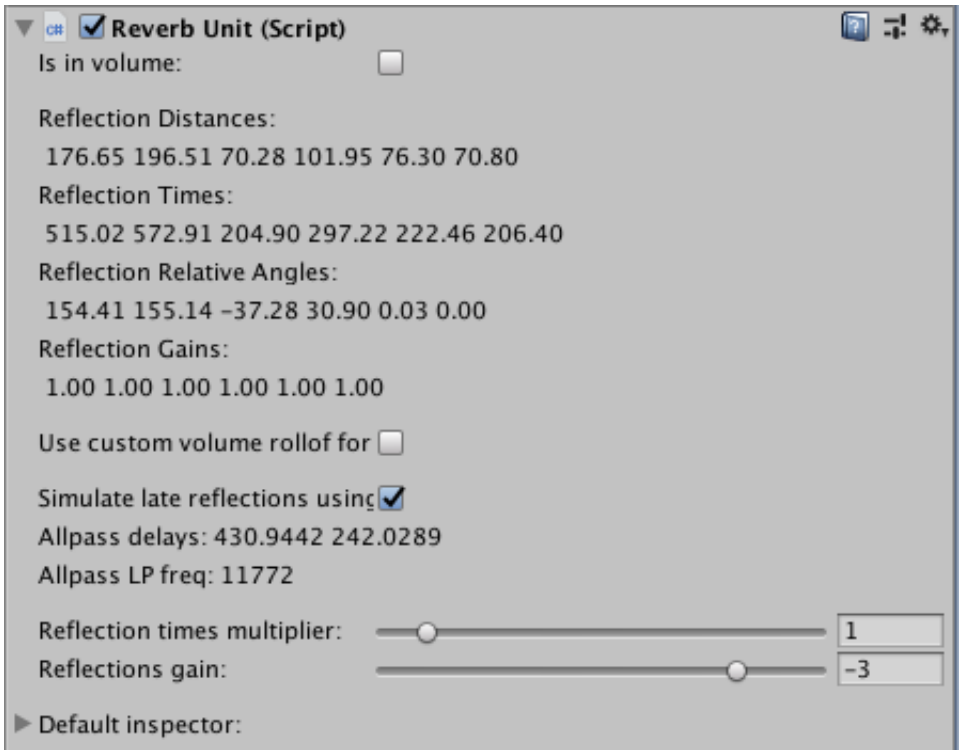
Skript počítá dráhy, respektive doby jednotlivých odrazů. Také počítá relativní zeslabení jednotlivých odrazů daných útlumem. Vstup zvukového signálu do *Spatializer* zásuvného modulu je již zeslaben na základě útlumové křivky určené componentou `AudioSource`. Je tedy nutné počítat při útlumu odrazů s tím, že pracujeme s již zeslabeným signálem. Avšak útlum signálu odrazu v závislosti na vzdálenosti je dán také touto křivkou

Pomocí parametru *Use custom volume rollof for reflections* je možné určit jinou křivku zeslabení v závislosti na vzdálenosti. v takovém případě se zobrazí okno s grafem, kde průběh můžeme pomocí křivek nakreslit.

Následují další parametry a to *Simulate late reflections using allpass filter*, pomocí nějž můžeme zapínat nebo vypínat zahušťování prvotních odrazů pomocí vnořených fázovacích článků.

Parametrem *Reflections times multiplier* pouze násobí časy vypočítaných prvotních odrazů. Parametr sloužil primárně pro testování, ale v některých případech se dá kreativně využít pro vytvoření dojmu většího, případně menšího poslechového prostoru. Z tohoto důvodu byl na componentě ponechán i ve finální verzi.

Posledním editovatelným parametrem je *Reflections Gain*, tedy parametr určující útlum nebo zesílení samotných odrazů oproti čistému zdrojovému signálu.




Obr. 4.3: Uživatelské rozhraní reverbUnit.cs

### Funkce ReverbUnit.cs

Dále buou popsány jednotlivé funkce obsažené v skriptu `ReverbUnit.cs`

- `Start()` je funkce volaná při zpuštění programu, nebo při vzniku samotného skriptu, pokud by byl vytvořen dynamicky během běhu programu. V ní skript volá funkci `CheckForVolume()`, nastavující odpovídající `AudioSource`

a také nastaví hodnoty pomocných proměnných. Dále je zde zpuštěn koprogram `InterpolateGainAndOcclusion()`.

- `Update()` je funkce volaná každý vykreslovací snímek. v rámci této funkce se počítá vzdálenost mezi odpovídajícím zdrojem zvuku a posluchačem, dále je volána funkce `ComputeAll()` a `SetSpatializerParameters()`.
- `CheckForVolume()` je volána na začátku skriptu. Slouží k tomu aby si skript `ReverbUnit` zjistil v kterém prostoru definovaném skripty `ReverbVolume` se nachází a přiřadil si na něj referenci.
- `OnDrawGizmos()` vykresluje v okně *Scene* Unity editoru dráhy jednotlivých odrazů. Tato funkce je pouze pro názornost a testování a není nezbytně nutná.
- `ComputeAll()` pokud má skript referenci na odpovídající skript `ReverbVolume` a componentu `AudioSource` tak tato funkce volá další funkce počítající nezbytné parametry a to jak v rámci tohoto skriptu tak funkce odpovídajícího `ReverbVolume` skriptu. Nastavují se zde hodnoty `closestPoints`, což je pole nejbližších bodů na stěnách a `closestDistances`, pole vzdáleností k těmto bodům.
- `ComputeMidpoints()` funkce hledá body, v kterých dochází k odrazu. Využívá se zde zákonu odrazu viz kapitola 1.3.1. v proměnné `closestPoints` již máme pole nejbližších bodů na geometrii pro zdroj zvuku, obdoně můžeme získat tyto body i pro posluchače, skript `ReverbVolume` má toto pole jako veřejnou proměnnou. Poměr vzdáleností od zdroje (nebo posluchače) k těmto bodům poté odpovídá poměrům, na které můžeme rozdělit úsečku spojující tyto dva body. v tomto bodě potom je místo odrazu. Místo odrazu se takto počítá pro každou stěnu a je uloženo v poli typu `Vector3` nazvaném *midPoints*.
- `ComputeReflecitonDistances()` počítá vzdálenost a zpoždění odrazů v milisekundách. Vzdálenost se spočte jednoduše jako velikost vektoru daného pozicí zdroje zvuku a místa odrazu a posluchače a místa odrazu.
  Zpoždění odrazů je potom $t = \frac{s}{c} * 1000.$, kde $s$ je vzdálenost a $c$ rychlost zvuku.
- `ComputeAngles()` počítá relativní úhel dopadu odrazu. Jedná se o úhel v horizontální rovině který svírá vektor daný pozicí posluchače a zdroje zvuku a místa odrazu a posluchače.
- `ComputeReflectionGains()` je funkce, která počítá útlum jednotlivých odrazů na základě jejich dráhy. Problémem je, že v rámci Unity můžeme nastavit libovolné křivky útlumu zvuku na základě vzdálenosti. a často se tohoto využívá pro větší kontrolu nad zvukovým prostředím. Proto je i útlum odrazů počítán za pomocí těchto křivek. Dalším problémem je, že zvukový signál, s kterým v rámci zásuvného modulu pracujeme, už je touto křivkou ztlumen. Pracujeme tedy již se ztlumeným signálem, a pokud bychom odrazy, které z toho signálu počítáme ztlumili hodnotou odpovídající vzdálenosti na křivce,

dostali bychom se na téměř nulové hodnoty. Musíme tedy počítat s rozdílovou vzdáleností, tedy odečíst od útlumu vypočítaného na křivce hodnotu útlumu, který již byl aplikován na zdrojový signál.

- `ComputeAllpassParameters()` počítá zpoždění vnořených fázovacích článků pro simulaci vícenásobných odrazů. Zpoždění jsou vypočítány jako průměr lichých a sudých prvků pole obsahujích zpoždění jednotlivých odrazů.
- `InterpolateGainAndOcclusion()` je funkce datového typu IEnumerator, a v rámci Unity se tedy jedná o koprogram (anglicky coroutine). Tyto speciální funkce umožňují zastavit běh funkce na daný časový okamžik [18], v případě této implementace příkazem `"yield return new WaitForSeconds(Time.fixedDeltaTime)"` zastavíme běh funkce na dobu definovanou fyzikálním engine Unity. v případě ukázkového projektu je to 20 ms. Tato funkce slouží k pozvolné změně celkového útlumu a také frekvence dolní propusti simulující akustický stín, pokud se posluchač nachází mimo prostor definovaný `ReverbVolume` skriptem.
- `SetSpatializerParameteres()` nastavuje veškeré vypočítané parametry samotnému zásuvnému modulu `Plugin_Spatializer`. K tomu slouží funkce objektu typu `AudioSource` `SetSpatializerFloat(int id, float value)`. Parametry tedy nastavujeme pomocí daného id a hodnoty. Pro jednoduší práci s parametry je v rámci skriptu definovaný výčtový typ (*enum*) *spatializerParams*, který kopíruje odpovídající výčtový typ *enum* v samotném zásuvném modulu. Je možné hodnoty toho výčtového typu přetypovat na hodnotu typu `int` a tu poté použít jako parametr.

  S některými hodnotami se zde provádí další operace, například hodnoty zpoždění fázovacích článků jsou převedeny z milisekund na vzorky a poté zaokrouhleny na nejbližší prvočísla funkcí `GetClosestPrime()`.

### 4.2.4 samplePicker.cs

Skript `samplePicker.cs` není nezbytně nutný pro fungování simulace poslechového prostoru. Slouží pouze pro testování celého systému. v ukázkové scéně jej najdeme na objektu *master*. Jeho veřejnými proměnnými jsou dvě pole. První je pole component *AudioSource* (v ukázkové scéně jsou zde všechny zdroje zvuku ve scéně) a druhé je pole typu *AudioClip*. Do tohoto pole můžeme umístit požadované testovací zvukové vzorky.

Při zpuštění scény můžeme klávesami 0 až 9 přepínat mezi jednotlivými vzorky. Klávesy odpovídají vzorkům v poli, tedy nula odpovídá prvnímu (nultému) vzorku, devítka odpovídá poslednímu vzorku v poli.

Pomocí mezerníku (klávesy *space*) je také možné znovu restartovat přehrávání

všech zdrojů zvuku.

Klávesou *L* je možné přepínat zda se mají vzorky přehrávat ve smyčce.

### 4.2.5 Nastavení Audio Mixer Group

Pro správné fungování reverbu je nutné v mixeru používaném pro přehrávání aktuální banky zvuku vytvořit *Audio Mixer Group*, která bude obsahovat zásuvný modul reverbu. Dále pochopitelně přidat na tuto group *receive* a odpovídající *send* componenty na ostatní kanály.

Na samotném reverbu je nutné nastavit všechny parametry jako *exposed*, pravým kliknutím na všechny parametry. Dále je nutné v okně mixeru nastavit názvy těchto parametrů následovně:

- Delay Time -> "DelayTime"
- Diffusion -> "Diffusion"
- Low pass freq -> "LowFreq"
- Crossover freq -> "CrossFreq"
- Low shelf absorp -> "LowAbs"
- High shelf absorption -> "HighAbs"

Tyto úkony je nutno provést manuálně, z toho důvodu, že Unity neumožňuje vytváření routingu a zařazování nových zásuvných modulů na kanály pomocí kódu za běhu programu.

## 4.3 Unity projekt

Ukázkový projekt se nachází mezi soubory přiloženými k této práci. Nejdůležitější jsou složky *Assets* a *NativeCode*.

Složka *Assets* obsahuje veškeré soubory pro samotnou ukázkovou scénu. V podsložce *Plugins* najdeme samotné zásuvné moduly pro simulaci poslechových prostor. Jedná se o soubor *AudioPluginNoise.bundle*. V podsložce *Scripts* se nachází všechny výše zmíněné skripty v jazyce C#. Dále zde najdeme testovací zvukové soubory ve složce *Samples*. Samotná scéna je přímo v kořenové složce a jde o soubor *testScene.unity*.

Ve složce *NativeCode* se nachází projekt vývojového prostředí Xcode, v němž najdeme zdrojové soubory `Plugin_Reverb.cpp` a `Plugin_Spatializer.cpp`.

Scénu lze ve vývojovém prostředí zpustit. V ukázkové scéně je poté možno se pohybovat pomocí šipek případně kláves WASD. Klávesami 1-6 je možno přepínat mezi jednotlivými testovacímy zvukovými vzorky. Klávesou L lze přepínat zda se vzorky přehrávají ve smyčce nebo pouze jednou.

# 5 Závěr

V rámci této práce byla rozebrána problematika reverberátorů jako systémů pro simulaci poslechových prostor a jejich implementace v rámci herního enginu Unity.

V první kapitole byly popsány základní přístupy pro simulaci poslechových prostor, byly představeny základní stavební prvky, které se používají při číslicovém zpracování signálu a dále byly nastíněny struktury pro simulaci prvotních a mnohonásobných odrazů.

Druhá kapitola se věnovala hernímu enginu Unity. Krátce byl definován herní engine a popsána jeho historie. Dále bylo popsáno rozhraní a důležité pojmy pro práci v tomto vývojovém prostředí. V poslední části této kapitoly byla podrobněji popsána práce se zvukovým signálem v rámci enginu a také sada vývojových nástrojů (SDK) pro tvorbu vlastních DSP zásuvných modulů.

V třetí kapitole byl podrobněji popsán návrh samotného zásuvného modulu v blokových schématech, vycházející z poznatků v první kapitole.

Ve čtvrté kapitole je shrnuta dosavadní práce. Je zde popsáno fungování samotných zásuvných modulů v jazyce C++ a podrobněji popsány jednotlivé funkce, postupy a použité parametry. Dále jsou v této kapitole popsány skripty v jazyce C#, zajišťující změny parametrů zásuvných modulů v reálném čase. Jsou zde popsány jednotlivé uživatelsky editovatelné parametry, významy jednotlivých funkcí a celková provázanost mezi skripty. V poslední části kapitoly je blíže popsán ukázkový projekt vytvořený v herním enginu Unity, jeho obsah a fungování.

Samotný zásuvný modul je schopen simulovat poslechové prostory na základě rozměrů definovaných uživatelem. Je schopen simulovat prostory různých velikostí, které jsou od sebe vzájemně rozeznatelné. Nedosahuje však zvukových kvalit, na které jsme zvyklí ze současných komerčně dostupných reverberátorů. Také při rychlém pohybu posluchače může docházet k slyšitelným zvukovým artefaktům. Systém, který byl vytvořen, se snaží simulovat prvotní odrazy co nejpřesněji, vzhledem k tomu, že je ale počítá ze zjednodušené geometrie, neposkytuje výsledný zvukový signál příliš výhod oproti klasickým vjemovým reverberátorům. Samotná implementace je náročná na výpočetní výkon a vyžadovala by více optimalizace.

Výhodou tohoto zásuvného modulu je jeho jednoduchost pro samotné nastavení v rámci Unity. Uživateli pouze stačí definovat velikost rozměrů a vlastnosti materiálů a samotné parametry jsou poté vypočítány a nastaveny samy. Nevýhodou je menší jedinečnost samotného zvukového vjemu, oproti například současným konvolučním reverberátorům, které jsou schopné pracovat v reálném čase. Zde je však nevýhodou nutnost hledání impulsových odezev odpovídajících virtuálnímu prostoru, případně jejich měření.

# Literatura


[1] ALLEN, Jont B. a David A. BERKLEY. *Image method for efficiently simulating small-room acoustics.* Bell Laboratories, Murray Hill, New Jerse, 1978. Acoustics Research Department, 9.

[2] BALÍK, Miroslav. *Číslicové zpracování akustických signálů* Fakulta elektrotechniky a komunikačních technologií, 2014. Vysoké učení technické v Brně.

[3] COX, Trevor J.; Peter D'ANTONIO. *Acoustic absorbers and diffusers: theory, design and application.* 2nd ed. New York: Taylor & Francis, c2009. ISBN 0-203-89305-0, 495.

[4] *DAFX: digital audio effects.* 2nd ed. Chichester, West Sussex, England: Wiley, 2011. ISBN 978-0-470-66599-2, 614.

[5] DATTORRO, Jon. *Effect Design: Reverberator and Other Filters.* Audio Engineering Society Journal Vol. 45, No 9. Stanford, 1997, 26.

[6] EVEREST, F. Alton. *The master handbook of acoustics.* 4th ed. New York: McGraw-Hill, c2001. ISBN 0-07-139974-7. 641.

[7] HAAS, John. *A History of the Unity Game Engine.* Worcester, 2012. Dostupné také z: `https://web.wpi.edu/Pubs/E-project/Available/E-project-030614-143124/unrestricted/Haas_IQP_Final.pdf`

[8] HOCKING, Joseph. *Unity in action: multiplatform game development in C#.* Shelter Island, NY Manning Publications Co., [2015]. ISBN 978-1617292323.

[9] JOT, Jean-Marc, WALSH M.;PHILP A. *Binaural Simulation of Complex Acoustic Scenes for Interactive Audio.* AES 121st Convention. 2006

[10] JOT, Jean-Marc. *DIgital Delay Networks for Designing Artificial Reverberators.* 90th AES convention. 1991.

[11] KARJALAINEN Matti; JÄRVALÄINEN Hanna . *Reverberation Modeling Using Velvet Noise* AES 30th International Conference. 2007

[12] Jeon, KWANG MYUNG, Kim, HONG KOOK. *Approximation of a Virtual Reverberation Filter For Handheld Devices*, 2011

[13] LANHAM, Micheal. *Game Audio Development with Unity* 5.X. UK: Packt Publishing, 2017. ISBN 978-1-78728-645-0

[14] LAVIERI, Edward. *Getting Started with Unity 2018.* Third Edition. Birmingham: Packt Publishing, 2018. ISBN 978-1-78883-010-2.

[15] LINOWES, Jonathan. *Unity Virtual Reality Projects.* Second Edition. Birmingham: Packt Publishing, 2018. ISBN ISBN 978-1-78847-880-9.

[16] MARK KAHRS, KARLHEINZ BRANDENBURG, *Applications of Digital Signal Porcessing to Audio and Acoustics*, 2002, ISBN: 0-3064-7042-X

[17] MOORER, James A. *About This Reverberation Business.* Computer Music Journal, Vol. 3, No. 2. The MIT Press, 1979

[18] OKITA, Alex. *Learning C# programming with Unity 3D.* Boca Raton: CRC Press, [2015]. ISBN 978-1466586529.

[19] OLIVEIRA, A.; CAMPOS G.; DIAS P., MURPHY D.; VIEIRA J.; MENDONCA C.; SANTOS J. *Real-Time Dynamic Image-Source Implementation For Auralisation.* Conference on Digital Audio Effects (DAFx-13). 2013.

[20] RUBAK P.; JOHANSEN L., *Artificial reverberation based on a pseudo-random impulse response II*, AES 106th Convention, (Munich), 1999.

[21] SCHROEDER, Manfred *Natural Sounding Artificial Reverberation.* Audio Engineering Society Journal. 1962.

[22] SMITH, Julius O. *Physical Audio Signal Processing: for Virtual Musical Instruments and Digital Audio Effects.* Lexington: W3K Publishing, 2010. ISBN 0974560723.

[23] *The technology of binaural listening.* New York: Springer, 2013. ISBN 978-3-642-37761-7.

[24] *Audio Spatializer SDK [online].* Unity Technologies, 2018. Dostupné také z: `https://docs.unity3d.com/Manual/AudioSpatializerSDK.html`

[25] *Native Audio Plugin SDK [online].* Unity Technologies, 2018. Dostupné z: `https://docs.unity3d.com/Manual/AudioMixerNativeAudioPlugin.html`

[26] *Unity Audio Documentation [online].* Unity Technologies, 2018. Dostupné z: `https://docs.unity3d.com/Manual/Audio.html`

[27] *Unity Scripting Reference [online].* Unity Technologies, 2019 [cit. 2019-05-01]. Dostupné z: `https://docs.unity3d.com/ScriptReference/`

# Seznam příloh

# A Obsah přiloženého CD

Na přiloženém CD se v kořenové složce nachází následující soubory:

- *UkazkaScena.mp4* je video ilustrující fungování simulace poslechových prostor v ukázkové scéně. V tomto videu hráč projde několika místnostmi s různými materiály zdí a různými objemy. Jsou zde slyšitelné změny mezi jednotlivými prostory a panorámování zdroje zvuku v závislosti na poloze zdroje zvuku.
- *UkazkaEditor.mp4* je video ukazující základní nastavení uživatelských skriptů `ReverbVolume.cs` a `ReverbUnit.cs`. Dále je zde ukázáno nastavení reference na hlavní AudioMixer a vzájemnou referenci mezi `ReverbVolume.cs` a `ReverbUnit.cs`.
- složka *build* obsahuje hratelnou ukázkovou scénu jako zpustitelný soubor typu `.app` pro systém MacOS.
- složka *diplProj* obsahuje veškeré zdrojové soubory. Obsahuje více podsložek z nichž nejdůležitější jsou:
  - složka *Assets* obsahující veškeré zdrojové soubory pro Unity projekt.
  - složka *NativeCode* obsahující projekt ve vývojovém prostředí XCode
- *DiplomovaPraceDominikKonecny.pdf* je elektronickou verzí této práce ve formátu PDF

# B Návod na zprovoznění zásuvných modulů

1. V kořenové složce *Assets* musí být vytvořena složka s názvem *Plugins* do ní bude umístěn soubor *AudioPluginNoise.bundle* obsahující samotné C++ zásuvné moduly. Unity tyto moduly načítá při startu, proto bude pravděpodobně nutné Unity restartovat.
2. Dále do složky *Assets* umístíme skripty *ReverbVolume.cs* a *ReverbUnit.cs*
3. Ve složce *Assets* dále vytvoříme soubor typu *Audio Mixer*. Vybereme jej a přepneme se do okna Audio Mixer. Zde vytvoříme novou *Group* a tlačítkem *Add..* pridáme efekt typu *Reverb*. Dále je potřeba všechny jeho parametry nastavit jako *Expose* pravým kliknutím na název parametru v okně *Inspector*. Tyto parametry je poté nutno přejmenovat následovně:
   - Delay Time -> "DelayTime"
   - Diffusion -> "Diffusion"
   - Low pass freq -> "LowFreq"
   - Crossover freq -> "CrossFreq"
   - Low shelf absorp -> "LowAbs"
   - High shelf absorption -> "HighAbs"
4. Dále opět ve složce *Assets* vytvoříme nový objekt typu *Acoustic Material* a v okně *Inspector* mu nastavíme parametry.
5. Ve scéně vytvoříme prázdný *GameObject*, vybereme jej a v okně *Inspector* klikneme na tlařítko *Add Component*. Ze seznamu vybereme položku *ReverbVolume*
6. Posledním parametrem je pole materiálů, námi vytvořený materiál tedy přetáhneme ze složky *Assets* do prvního prvku tohoto pole.
7. Vytvoříme nový *GameObject*, přidáme mu componenty typu *Audio Source* a *ReverbUnit*. Objektu *AudioSource* nastavíme zdrojový soubor a další požadované parametry. Je nutné zatrhnout pole *Spatialize*.
8. Na componentě *ReverbUnit* přesuneme objekt se skriptem *ReverbVolume* do prázdného pole *Reverb Volume Script*. Tím vytvoříme vzájemnou referenci.

# C Native Audio SDK

Výpis C.1: Inicializace přístupných proměnných v Unity Native Audio pluginu

```
                                                                    1
int InternalRegisterEffectDefinition                                2
  (UnityAudioEffectDefinition& definition)                          3
  {                                                                 4
      int numparams = P_NUM;                                        5
      definition.paramdefs =                                        6
          new UnityAudioParameterDefinition [numparams];            7
      RegisterParameter(definition, "Gain Multiplier", "",          8
                        0.0f, 10.0f, 1.0f,                          9
                        1.0f, 1.0f,                                 10
                        P_GAIN);                                    11
      return numparams;                                             12
  }                                                                 13
```

Výpis C.2: Inicializace přístupných proměnných v Unity Native Audio pluginu

```
UNITY_AUDIODSP_RESULT UNITY_AUDIODSP_CALLBACK
ProcessCallback(
       UnityAudioEffectState* state,
       float* inbuffer, float* outbuffer,
       unsigned int length,
       int inchannels, int outchannels)
{
    // Grab the EffectData struct we
    // created in CreateCallback()
    EffectData *data = state->GetEffectData<EffectData>();

    // a gain multiplier to silence
    the plugin when not in play mode or muted
    float wetTarget = ((state->flags &
    UnityAudioEffectStateFlags_IsPlaying) &&
    !(state->flags & (UnityAudioEffectStateFlags_IsMuted
    | UnityAudioEffectStateFlags_IsPaused))) ? 1.0f : 0.0f;

    // For each sample going to the output buffer
    for(unsigned int n = 0; n < length; n++)
    {
        // For each channel of the buffer
        for(int i = 0; i < outchannels; i++)
        {
            // Generate a random float in the range
            [-1.0, 1.0] (fixed-point arithmetic)
            int rInt = rand() % 20000;
            rInt -= 10000;
            float r = rInt / 10000.0f;

            // Write the sample to the buffer
            outbuffer[n * outchannels + i] = r *
            wetTarget * data->p[P_GAIN];
        }
    }

    return UNITY_AUDIODSP_OK;
}
```

# D Pomocné třídy a funkce

Výpis D.1: Třída `Lowpass` implementující číslicovou dolní propust

```
class LowPass
    {
    private:
        float c;
        float f;
        int sampleRate;
        float lastIn;
        float apOut;
    public:

        LowPass()
        {
            lastIn = 0;
            apOut = 0;
        }

        LowPass(float samplerate, float frequency)
        {
            LowPass();
            sampleRate = samplerate;
            SetFrequency(frequency);
        }

        void SetSampleRate(float rate)
        {
            sampleRate = rate;
        }

        void SetFrequency(float frequency)
        {
            f = frequency;
            c = ( tan( kPI * f / sampleRate ) -1 ) /
                ( tan( kPI * f / sampleRate ) + 1  );
        }

        float Iterate(float sample)
        {
            apOut = c* sample + lastIn - c* apOut;
            lastIn = sample;
            return (sample + apOut) /2 ;
        }
    };
```

Výpis D.2: Třída `HighShelving` implementující číslicový high shelving filtr

```
class HighShelving
    {
    private:
        float c, f, g, H0, lastIn, apOut;
        int sampleRate;
    public:
        HighShelving()
        {
            lastIn = 0;
            apOut = 0;
        }
        HighShelving(int samplerate, float frequency,
                    float gain)
        {
            HighShelving();
            SetParameters(samplerate, frequency, gain);
        }

        void SetParameters(float rate, float frequency,
                        float gain)
        {
            sampleRate = rate;
            f = frequency;
            g = gain;
            float V0 =pow(10,g/20);
            H0 = V0 - 1;
            if(g > 1)
                c =( tan( kPI * f / sampleRate ) -1 ) /
                      ( tan( kPI * f / sampleRate ) + 1  );
            else
                c = ( V0*tan( kPI * f / sampleRate ) -1 ) /
                       ( V0* tan( kPI * f / sampleRate ) + 1  );
        }
        float Iterate(float sample)
        {
            //výstup fázovacího článku
            apOut = c* sample + lastIn - c* apOut;
            lastIn = sample;
            return (sample - apOut) * H0/2 + sample ;
        }
    };
```

Výpis D.3: Třída `BasicBuffer` implementující kruhový buffer

```
  class BasicBuffer
{
private:
    int bufferSize; // Maximum delay lenght
    float* reverbBufferL ={0};
    float* reverbBufferR ={0};
    int sampleRate ;
    int reverbBufferPosL;
    int reverbBufferPosR;
public:
    BasicBuffer(int sampleRateX,
        float bufferLengthInSeconds)
    {
        sampleRate = sampleRateX;
        bufferSize = bufferLengthInSeconds*sampleRate;
        reverbBufferPosL = 0;
        reverbBufferPosR = 0;
        free(reverbBufferL);
        free(reverbBufferR);
        reverbBufferL = (float*)
            malloc(bufferSize*sizeof(float));
        reverbBufferR =(float*)
            malloc(bufferSize*sizeof(float));

        for(int i = 0 ; i< bufferSize; i++)
        {
            reverbBufferL[i] = 0;
            reverbBufferR[i] = 0;
        }
    }

    void SetNextValue(float value, int channel)
    {
        if(channel == 0)
        {
            reverbBufferL[reverbBufferPosL] = value;
        }
        if(channel == 1)
        {
            reverbBufferR[reverbBufferPosR] = value;
        }
    }

    float GetValue(int delayInSamples, int channel)
    {
        int id = 0;
```

```
            if(channel == 0)
                id = reverbBufferPosL - delayInSamples;
            if(channel == 1)
                id = reverbBufferPosR - delayInSamples;
            if(id < 0)
                id+= bufferSize;
            if(channel == 0)
                 return(reverbBufferL[id]);
            if(channel == 1)
                return(reverbBufferR[id]);
            else
                return 0;
        }

        void Iterate(int channel)
        {
            if(channel == 0){
            reverbBufferPosL ++;
            if(reverbBufferPosL >= bufferSize)
                reverbBufferPosL = 0;
            }
            if(channel == 1){
                reverbBufferPosR ++;
                if(reverbBufferPosR >= bufferSize)
                    reverbBufferPosR = 0;
            }
        }
    };
```

# E Plugin_Spatializer.cpp

Výpis E.1: funkce InternalRegisterEffectDefinition pro Plugin_Spatializer.cpp

```
int InternalRegisterEffectDefinition(UnityAudioEffectDefinition& definition)
{
    int numparams = P_NUM;
    definition.paramdefs = new UnityAudioParameterDefinition[numparams];
    RegisterParameter(definition, "AudioSrc Attn", "", 0.0f, 1.0f, 1.0f, 1.0f, 1.0f, P_AUDIOSRCATTN, "
        AudioSource distance attenuation");
    RegisterParameter(definition, "Fixed Volume", "", 0.0f, 1.0f, 0.0f, 1.0f, 1.0f, P_FIXEDVOLUME, "Fixed
        volume amount");
    RegisterParameter(definition, "Custom Falloff", "", 0.0f, 1.0f, 0.0f, 1.0f, 1.0f, P_CUSTOMFALLOFF, "Custom
        volume falloff amount (logarithmic)");
    RegisterParameter(definition, "Delay of first reflection", "ms", 0, 1000, 100, 1.0f, 1.0f, P_DELAY1);
    RegisterParameter(definition, "Delay of second reflection", "ms", 0, 1000, 100, 1.0f, 1.0f, P_DELAY2);
    RegisterParameter(definition, "Delay of third reflection", "ms", 0, 1000, 100, 1.0f, 1.0f, P_DELAY3);
    RegisterParameter(definition, "Delay of fourth reflection", "ms", 0, 1000, 100, 1.0f, 1.0f, P_DELAY4);
    RegisterParameter(definition, "Delay of fifth reflection", "ms", 0, 1000, 100, 1.0f, 1.0f, P_DELAY5);
    RegisterParameter(definition, "Delay of sixth reflection", "ms", 0, 1000, 100, 1.0f, 1.0f, P_DELAY6);
    RegisterParameter(definition, "Relative angle of first reflection", "", -180, 180, 0, 1.0f, 1.0f, P_ANGLE1)
        ;
    RegisterParameter(definition, "Relative angle of second reflection", "", -180, 180, 0, 1.0f, 1.0f, P_ANGLE2
        );
    RegisterParameter(definition, "Relative angle of third reflection", "", -180, 180, 0, 1.0f, 1.0f, P_ANGLE3)
        ;
    RegisterParameter(definition, "Relative angle of fourth reflection", "", -180, 180, 0, 1.0f, 1.0f, P_ANGLE4
        );
    RegisterParameter(definition, "Relative angle of fifth reflection", "", -180, 180, 0, 1.0f, 1.0f, P_ANGLE5)
        ;
    RegisterParameter(definition, "Relative angle of sixth reflection", "", -180, 180, 0, 1.0f, 1.0f, P_ANGLE6)
        ;
    RegisterParameter(definition, "Gain by distance of first reflection", "", 0, 1, 1, 1.0f, 1.0f, P_GAIN1);
    RegisterParameter(definition, "Gain by distance of second reflection", "", 0, 1, 1, 1.0f, 1.0f, P_GAIN2);
    RegisterParameter(definition, "Gain by distance of third reflection", "", 0, 1, 1, 1.0f, 1.0f, P_GAIN3);
    RegisterParameter(definition, "Gain by distance of fourth reflection", "", 0, 1, 1, 1.0f, 1.0f, P_GAIN4);
    RegisterParameter(definition, "Gain by distance of fifth reflection", "", 0, 1, 1, 1.0f, 1.0f, P_GAIN5);
    RegisterParameter(definition, "Gain by distance of sixth reflection", "", 0, 1, 1, 1.0f, 1.0f, P_GAIN6);
    RegisterParameter(definition, "Interpolation speed", "", 0, 1, 0.5f, 1.0f, 1.0f, P_INTERP);
    RegisterParameter(definition, "Reflections time multiplier", "", 0,10,1,1.0f,1.0f,P_MULT);
    RegisterParameter(definition, "LowPass of first Reflection", "", 10, 20000, 100, 1.0f, 1.0f, P_LPFREQ1 );
    RegisterParameter(definition, "LowPass of second Reflection", "", 10, 20000, 100, 1.0f, 1.0f, P_LPFREQ2 );
    RegisterParameter(definition, "LowPass of third Reflection", "", 10, 20000, 100, 1.0f, 1.0f, P_LPFREQ3 );
    RegisterParameter(definition, "LowPass of fourth Reflection", "", 10, 20000, 100, 1.0f, 1.0f, P_LPFREQ4 );
    RegisterParameter(definition, "LowPass of fifth Reflection", "", 10, 20000, 100, 1.0f, 1.0f, P_LPFREQ5 );
    RegisterParameter(definition, "LowPass of sixth Reflection", "", 10, 20000, 100, 1.0f, 1.0f, P_LPFREQ6 );
    RegisterParameter(definition, "Mid frequency of first Reflection", "", 10, 20000, 100, 1.0f, 1.0f,
        P_SHELFFREQ1 );
    RegisterParameter(definition, "Mid frequency of second Reflection", "", 10, 20000, 100, 1.0f, 1.0f,
        P_SHELFFREQ2 );
    RegisterParameter(definition, "Mid frequency of third Reflection", "", 10, 20000, 100, 1.0f, 1.0f,
        P_SHELFFREQ3 );
    RegisterParameter(definition, "Mid frequencyof fourth Reflection", "", 10, 20000, 100, 1.0f, 1.0f,
        P_SHELFFREQ4 );
    RegisterParameter(definition, "Mid frequency of fifth Reflection", "", 10, 20000, 100, 1.0f, 1.0f,
        P_SHELFFREQ5 );
    RegisterParameter(definition, "Mid frequency of sixth Reflection", "", 10, 20000, 100, 1.0f, 1.0f,
        P_SHELFFREQ6 );
    RegisterParameter(definition, "Low shelf absorption of first","",0.001f,1.0f , 0.3f, 1.0f, 1.0f, P_SHLOW1);
    RegisterParameter(definition, "Low shelf absorption of second","",0.001f,1.0f , 0.3f, 1.0f, 1.0f, P_SHLOW2)
        ;
    RegisterParameter(definition, "Low shelf absorption of third","",0.001f,1.0f , 0.3f, 1.0f, 1.0f, P_SHLOW3);
    RegisterParameter(definition, "Low shelf absorption of fourth","",0.001f,1.0f , 0.3f, 1.0f, 1.0f, P_SHLOW4)
        ;
    RegisterParameter(definition, "Low shelf absorption of fifth","",0.001f,1.0f , 0.3f, 1.0f, 1.0f, P_SHLOW5);
    RegisterParameter(definition, "Low shelf absorption of sixth","",0.001f,1.0f , 0.3f, 1.0f, 1.0f, P_SHLOW6);
    RegisterParameter(definition, "High shelf absorption of first","",0.001f,1.0f , 0.3f, 1.0f, 1.0f, P_SHHIGH1
        );
    RegisterParameter(definition, "High shelf absorption of second","",0.001f,1.0f , 0.3f, 1.0f, 1.0f,
        P_SHHIGH2);
    RegisterParameter(definition, "High shelf absorption of third","",0.001f,1.0f , 0.3f, 1.0f, 1.0f, P_SHHIGH3
        );
    RegisterParameter(definition, "High shelf absorption of fourth","",0.001f,1.0f , 0.3f, 1.0f, 1.0f,
        P_SHHIGH4);
    RegisterParameter(definition, "High shelf absorption of fifth","",0.001f,1.0f , 0.3f, 1.0f, 1.0f, P_SHHIGH5
        );
    RegisterParameter(definition, "High shelf absorption of sixth","",0.001f,1.0f , 0.3f, 1.0f, 1.0f, P_SHHIGH6
        );
```

```
    RegisterParameter(definition, "Q␣of␣shelf␣filters","",0.2f,10.0f , 1.0f, 1.0f, 1.0f, P_SHQ);                    52
    RegisterParameter(definition, "Feedback␣of␣allpass␣filter", "", 0.0f, 1.0f, 0.5f, 1.0f, 1.0f, P_VAPFB );       53
    RegisterParameter(definition, "Delay␣of␣allpassfilter␣1", "ms", 0, 1000, 100, 1.0f, 1.0f, P_VAPDELAY1);         54
    RegisterParameter(definition, "LowPass␣of␣VAP", "", 10, 20000, 100, 1.0f, 1.0f, P_VAPDELAY2);                   55
    RegisterParameter(definition, "Delay␣of␣allpassfilter␣2", "ms", 0, 1000, 100, 1.0f, 1.0f, P_VAPLPFREQ);         56
    RegisterParameter(definition, "frequency␣of␣lowpass␣when␣listener␣is␣not␣in␣volume", "Hz", 10, 15000,          57
        10000, 1.0f, 1.0f, P_OCLFREQ);
    RegisterParameter(definition, "Gain␣of␣reflections", "dB", -60, 12, 1.0f, 1.0f, 1.0f,  P_REFLECTIONSGAIN);     58
    RegisterParameter(definition, "simulate␣reflections␣on/off,␣threshhold␣0.5f", "", 0.0f, 1.0f, 1.0f, 1.0f,       59
        1.0f, P_SIMULATE);
                                                                                                                   60
    definition.flags |= UnityAudioEffectDefinitionFlags_IsSpatializer;                                             61
    return numparams;                                                                                              62
}                                                                                                                  63
```

Výpis E.2: funkce CreateCallback pro Plugin_Spatializer.cpp

```
UNITY_AUDIODSP_RESULT UNITY_AUDIODSP_CALLBACK CreateCallback(UnityAudioEffectState* state)
    {
        EffectData* effectdata = new EffectData;
        memset(effectdata, 0, sizeof(EffectData));
        effectdata->EarlyBuffer = new BasicBuffer(state->samplerate,20);
        effectdata->VapBuffer1 = new BasicBuffer(state->samplerate,20);
        effectdata->VapBuffer2 = new BasicBuffer(state->samplerate,20);
        state->effectdata = effectdata;

        SetShelfs(effectdata, state);

        /*
         if (IsHostCompatible(state))
         state->spatializerdata->distanceattenuationcallback = DistanceAttenuationCallback;
         */
        InitParametersFromDefinitions(InternalRegisterEffectDefinition, effectdata->p);
        return UNITY_AUDIODSP_OK;
    }
```

Výpis E.3: algoritmus prvotních odrazů v Plugin_Spatializer.cpp

```
                                                                    1
UNITY_AUDIODSP_RESULT UNITY_AUDIODSP_CALLBACK ProcessCallback(      2
    UnityAudioEffectState* state, float* inbuffer,
  float* outbuffer, unsigned int length, int inchannels, int        3
      outchannels)
{                                                                   4
    // Check that I/O formats are right and that the host API       5
       supports this feature
    if (inchannels != 2 || outchannels != 2 ||                      6
        !IsHostCompatible(state) || state->spatializerdata == NULL) 7
    {                                                               8
        memcpy(outbuffer, inbuffer, length * outchannels * sizeof(  9
            float));
        return UNITY_AUDIODSP_OK;                                   10
    }                                                               11
                                                                    12
    EffectData* data = state->GetEffectData<EffectData>();          13
                                                                    14
    SetShelfs(data, state);                                         15
                                                                    16
    static const float kRad2Deg = 180.0f / kPI;                     17
                                                                    18
    float* m = state->spatializerdata->listenermatrix;              19
    float* s = state->spatializerdata->sourcematrix;                20
                                                                    21
    float px = s[12];                                               22
    float py = s[13];                                               23
    float pz = s[14];                                               24
                                                                    25
    float dir_x = m[0] * px + m[4] * py + m[8] * pz + m[12];        26
    float dir_y = m[1] * px + m[5] * py + m[9] * pz + m[13];        27
    float dir_z = m[2] * px + m[6] * py + m[10] * pz + m[14];       28
                                                                    29
    float azimuth = (fabsf(dir_z) < 0.001f) ? 0.0f : atan2f(dir_x,  30
       dir_z);
    if (azimuth < 0.0f)                                             31
        azimuth += 2.0f * kPI;                                      32
    azimuth = FastClip(azimuth * kRad2Deg, 0.0f, 360.0f);           33
                                                                    34
    float gain = pow(10, data->p[P_REFLECTIONSGAIN]/20);            35
                                                                    36
    for(int i = 0; i < numberOfReflections;i++)                     37
    {                                                               38
        reflectionRelativeAngles[i] = data->p[9+i];                 39
        reflectionsGains[i] = data->p[15+i];                        40
        for(int j = 0; j <2;j++) {                                  41
```

```
            data->EarlyFilters[j][i].SetSampleRate(state->
                samplerate);
            data->EarlyFilters[j][i].SetFrequency(data->p[P_LPFREQ1
                +i]);

            if(i==0)
            {
                data->VAPFilter[j].SetSampleRate(state->samplerate)
                    ;
                data->VAPFilter[j].SetFrequency(data->p[P_VAPLPFREQ
                    ]);

                data->OcclusionFilter[j].SetSampleRate(state->
                    samplerate);
                data->OcclusionFilter[j].SetFrequency(data->p[
                    P_OCLFREQ]);
            }
        }
    }

    float in;
    float g1;
    float g2;
    int M1;
    int M2;
    float VAP = 0;

    for(unsigned int n = 0; n < length; n++)
    {
        SetReflectionTimes(data);
        // For each channel of the buffer
        for(int i = 0; i < outchannels; i++)
        {
            in =inbuffer[n*outchannels + i];
            if(data->p[P_SIMULATE] > 0.5f)
            {
                earlyReflections = 0.0f;
                reflection = 0.0f;
                data->EarlyBuffer->SetNextValue(in, i);
                for(int num = 0; num< numberOfReflections; num++)
                {
                    reflection = data->EarlyBuffer->
                      GetValue((data->reflectionsTimesSamples[num]*
                          data->p[P_MULT]), i)* reflectionsGains[num
                          ] ;
                    reflection *= GetPanGain(azimuth +
                        reflectionRelativeAngles[num], i);
```

```
                    reflection = data->EarlyFilters[i][num].Iterate
                        (reflection);
                    reflection = data->HighShelves[num][i].Process(
                        reflection);
                    reflection = data->LowShelves[num][i].Process(
                        reflection);

                    earlyReflections += reflection;
                }
                earlyReflections = earlyReflections / (
                    numberOfReflections );
                data->EarlyBuffer->Iterate(i);

                g1 = data->p[P_VAPFB];
                g2 = data->p[P_VAPFB];
                M1 = vapDelay1;
                M2 = vapDelay2;
                data->VapBuffer1->SetNextValue(in, i);
                VAP = g2* earlyReflections + g1*g2* data->
                    VapBuffer1->GetValue(M1,i)
                    + g1*data->VapBuffer1->GetValue(M2,i) + data->
                        VapBuffer1->GetValue(M1+M2,i)
                    - g1*data->VapBuffer2->GetValue(M1,i) - g1*g2*
                        data->VapBuffer2->GetValue(M2,i)
                    - g2*data->VapBuffer2->GetValue(M2+M1,i);
                VAP = data->VAPFilter[i].Iterate(VAP);
                data->VapBuffer2->SetNextValue(VAP, i);
                data->VapBuffer2->Iterate(i);
                data->VapBuffer1->Iterate(i);
            }
            if(gain < 0.1f){
                in = data->OcclusionFilter[i].Iterate(in);
            }
            outbuffer[n*outchannels + i] =  in*GetPanGain(azimuth,
                i)  + ((earlyReflections + VAP)*0.5f)*gain ;
        }
    }
    return UNITY_AUDIODSP_OK;
}
```

# F Plugin_Reverb.cpp

Výpis F.1: funkce `InternalRegisterEffectDefinition` v `Plugin_Reverb.cpp`

```
                                                                    1
int InternalRegisterEffectDefinition(UnityAudioEffectDefinition&    2
    definition)
    {                                                               3
        int numparams = P_NUM;                                      4
        definition.paramdefs = new UnityAudioParameterDefinition[   5
            numparams];
        RegisterParameter(definition, "Delay Time", "", 0.0f, 15.0f 6
            , 2.0f, 1.0f, 1.0f, P_DELAYTIME);
        RegisterParameter(definition, "Low pass frequency", "", 10, 7
             20000, 100, 1.0f, 1.0f, P_LPFREQ);
        RegisterParameter(definition, "Crossover frequency", "",    8
            10, 20000, 100, 1.0f, 1.0f, P_SHELFFREQ);
        RegisterParameter(definition, "Low shelf absorption",""     9
            ,0.001f,1.0f , 0.3f, 1.0f, 1.0f, P_LOWGAIN);
        RegisterParameter(definition, "High shelf absorption",""    10
            ,0.001f,1.0f , 0.3f, 1.0f, 1.0f, P_HIGHGAIN);
                                                                    11
        return numparams;                                           12
    }                                                               13
```

Výpis F.2: funkce `CreateCallback` v `Plugin_Reverb.cpp`

```
   UNITY_AUDIODSP_RESULT UNITY_AUDIODSP_CALLBACK CreateCallback(    1
      UnityAudioEffectState* state)
    {                                                               2
        EffectData* effectdata = new EffectData;                    3
        memset(effectdata, 0, sizeof(EffectData));                  4
        state->effectdata = effectdata;                             5
        InitParametersFromDefinitions(                              6
            InternalRegisterEffectDefinition, effectdata->p);
        return UNITY_AUDIODSP_OK;                                   7
    }                                                               8
```

Výpis F.3: algoritmus pro simulaci mnohonásobných odrazů v Plugin_Reverb.cpp

```
                                                                    1
UNITY_AUDIODSP_RESULT UNITY_AUDIODSP_CALLBACK ProcessCallback(      2
    UnityAudioEffectState* state,
    float* inbuffer, float* outbuffer, unsigned int length,         3
        int inchannels, int outchannels)
{                                                                   4
    if (inchannels != 2 || outchannels != 2)                        5
    {                                                               6
        memcpy(outbuffer, inbuffer, length * outchannels *          7
            sizeof(float));
        return UNITY_AUDIODSP_OK;                                   8
    }                                                               9
    EffectData* data = state->GetEffectData<EffectData>();          10
    float in = 0;                                                   11
    const float delaytime = data->p[P_DELAYTIME] * state->          12
        samplerate + 1.0f;
    //const int numtaps = (int)(data->p[P_DIFFUSION] * (MAXTAPS     13
        - 2) + 1);
    const int numtaps = (int)(0.25f * (MAXTAPS - 2) + 1);           14
    data->random.Seed(0);                                           15
                                                                    16
    for (int c = 0; c < 2; c++)                                     17
    {                                                               18
        SetEqParameters(data,state);                                19
                                                                    20
        InstanceChannel& ch = data->ch[c];                          21
        const InstanceChannel::Tap* tap_end = ch.taps + numtaps     22
            ;
                                                                    23
        float decay = powf(0.01f, 1.0f / (float)numtaps);           24
        float p = 0.0f, amp = (decay - 1.0f) / (powf(decay,         25
            numtaps + 1) - 1.0f);
        InstanceChannel::Tap* tap = ch.taps;                        26
        while (tap != tap_end)                                      27
        {                                                           28
            p += data->random.GetFloat(0.0f, 100.0f);               29
            tap->pos = p;                                           30
            tap->amp = amp;                                         31
            amp *= decay;                                           32
            ++tap;                                                  33
        }                                                           34
                                                                    35
        WriteImpulseToConsole(data);                                36
                                                                    37
        float scale = delaytime / p;                                38
        tap = ch.taps;                                              39
```

```
            while (tap != tap_end)
            {
                tap->pos *= scale;
                ++tap;
            }

            for (int n = 0; n < length; n++)
            {
                in = inbuffer[n * 2 + c] + reverbmixbuffer[n * 2 +
                    c];

                float s = 0.0f;
                const InstanceChannel::Tap* tap = ch.taps;
                while (tap != tap_end)
                {
                    s += ch.delay.Read(tap->pos) * tap->amp;
                    ++tap;
                }
                s = data->Lowpass[c].Iterate(s);
                s = data->HighShelf[c].Process(s);
                s = data->LowShelf[c].Process(s);

                ch.delay.Write(in + s);

                outbuffer[n * 2 + c] = s;
            }
        }

        memset(reverbmixbuffer, 0, sizeof(reverbmixbuffer));

        return UNITY_AUDIODSP_OK;
    }
```