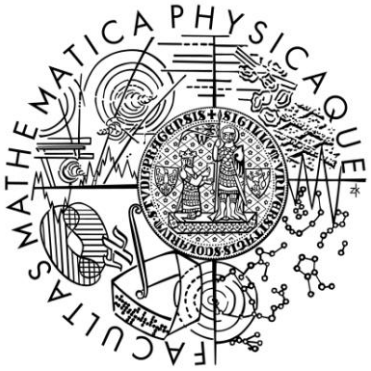
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Patrik Pasterčík

Pokročilý zvukový systém pro počítačové hry

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek, Ph.D.

Studijní program: Informatika

Studijní obor: Programování (IP)

Praha 2016

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 25.5.2016 Patrik Pasterčík

Název práce: Pokročilý zvukový systém pro počítačové hry

Autor: Patrik Pasterčík

Katedra / Ústav: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Důležitou součástí počítačových her je zvuk. Pro práci se zvukem existují speciální knihovny tzv. audio enginy. Ve srovnání s komerčními systémy jsou open source systémy výrazně omezeny. Cílem této práce je vytvoření audio enginu podobného komerčním řešením. Výsledkem je knihovna pro přehrávání zvuků ve hrách, jenž využívá knihovnu XAudio2 pro zpracovávání zvukových dat (aplikaci zvukových efektů a posílání dat zvukové kartě). Knihovna umožňuje přehrávání složených zvuků jako například zvuk motoru. Toto přehrávání může být ovlivňováno pomocí různých parametrů (například otáčkami motoru). Díky knihovně je možné na tyto zvuky postupně aplikovat různé zvukové efekty (echo, reverb, low-pass filter nebo high-pass filter). Knihovna též dovoluje napojit výstup jako zdroj jiného zvuku či předat výstup v bufferu vývojáři. Součástí práce je také editor, který vytváří složené zvuky. Editor zároveň slouží jako ukázka funkčnosti knihovny.

Klíčová slova: Audio engine, definice zvuku, editor

Title: Advanced Sound System for Computer Games

Author: Patrik Pasterčík

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek Ph.D., Katedra distribuovaných a spolehlivých systémů, pracoviště

Abstract: Sound is one of the important components of computer games. To work with sound, there are special libraries called audio engines. However in comparison with commercial systems, open-source systems are significantly limited. The goal of this thesis is to create an open audio engine with functionality similar to a commercial system. The result of this thesis is a library for playing sounds in games that uses the XAudio2 library for processing audio data (application of sound effects and sending data to the sound card). Our library enables playback of complex sounds such as the sounds of car engine. This playback can be influenced by various parameters (for instance engine RPM) and various sound effects can be applied to these sounds (echo, reverb, low-pass filter or high-pass filter). The library also enables to connect the output of another audio source or transmit output buffer developers. The thesis also includes an editor for creating complex sounds. Editor also serves as a demonstration of the functionality of the library.

Keywords: Audio engine, sound definition, editor

Rád bych poděkoval Mgr. Pavlu Ježkovi, Ph.D. za vedení této práce, cenné připomínky, rady, trpělivost a čas, který se mi během vytváření bakalářské práce věnoval. Dále bych poděkoval mým rodičům za podporu během mého studia. Nakonec bych poděkoval Michaele Jungové a Dominikovi Šmídovi, jež mi byli oporou při dokončování této práce.

Obsah

1 Úvod

Většina lidí někdy hrála nebo hraje jednu či více počítačových her. Díky tomu vydělávají firmy zabývající se tvorbou her miliony dolarů, viz např. článek o *růstu prodeje her o 33%* [1]. To firmy motivuje k tomu, aby vytvářely další komplexnější hry, kterými by zaujaly uživatele. Díky tomu mají vývojáři těchto firem stále více práce s tvorbou nových her. Kvůli složitosti her vývojáři rozdělují vývoj hry na tři základní části: grafiku, zvuk a herní logiku. Tyto části vyvíjejí paralelně a poté je spojí dohromady do jednoho celku. Aby firmy ušetřily čas a peníze, vývojáři používají pro vývoj určitých částí již existující řešení tzv. *enginy*, které následně poupraví podle požadavků konkrétní hry, a tak vytvoří danou hru. Nicméně lidé si mohou lehce všimnout herních a grafických chyb, zatímco zvukové chyby nemusí ani zaregistrovat (zdroj zvuku totiž není viditelný a uživatel nemusí být informován o nedostatečné kvalitě zvuku), takže se firmy v dnešní době více zaměřují na propracování herní logiky a grafiky, zatímco na zvuk nekladou takový důraz. Proto jsou herní a grafické enginy daleko propracovanější na rozdíl od zvukových (neboli audio) enginů, alespoň co se týče open source. Open source audio enginy tedy nejsou v současnosti příliš propracované, protože na jejich vývoj nebyl kladen velký důraz. Zato komerční audio enginy se i přes malý důraz na zvuk vyvíjely intenzivněji. Pokud se firmy zaměří na zvuk, dokážou díky těmto audio enginům vytvářet takové herní zvuky, které jsou hodně blízko zvukům v reálném světě, případně už lidé skoro nepoznají rozdíl. Kvůli tomuto rozdílu mezi open source akomerčními audio enginy si jako hlavní cíl této práce zvolíme vytvoření open source audio enginu, který bude obsahovat prvky podobné komerčním audio enginům.

Audio engine slouží pro přehrávání zvuku ve hrách, ale zatím nevíme, co vše by měl umět a jak bychom ho mohli vytvořit. Proto se podíváme na to, jak různé zvuky ve hrách jsou a jaké zvukové požadavky mají hry na audio engine. Poté si definujeme strukturu audio enginu, dále jak by měl vypadat z programátorského hlediska.

1.1 Zvuk ve hrách

Vývoj zvuku ve hrách můžeme rozdělit na tři části: zaprvé vývoj či úprava audio enginu, dále definování zvuků přehrávaných audio enginem a nakonec řízení přehrávání zvuků v aplikaci pomocí audio enginu. Pro první dvě části bývají vyčleněni speciální vývojáři. Těm, co definují zvuky, se říká designéři. U malých firem se o definování zvuků a řízení přehrávání může starat jen jeden vývojář.

Nyní se podívejme na to, jak se vyvíjel zvuk ve hrách, abychom si udělali představu, co všechno obsahuje zpracovávání zvuku a co se od zpracovávání zvuku očekává. Dopracujeme se k tomu, co vyžadují dnešní hry a na jaké úrovni by audio engine měl být.

1.1.1 Vývoj zvuku ve hrách

První počítačové hry, jež se objevovaly kolem poloviny sedmdesátých let, byly čistě textové, takže nepotřebovaly zvuk. S rozvojem dalších her ale vývojáři přemýšleli nad tím, jak hry obohatit, aby mohly podobně jako hry v herních

automatech obsahovat zvuk. Potíž byla v tom, že osobní počítač byl v té době považován za pracovní nástroj, tudíž měl jen malé reproduktory pro signalizaci různých chyb a stavů počítače, které nebyly určeny pro přehrávání zvuku.

Kolem osmdesátých let byla zvuková stránka her stále spíše vynechávána. Vývojáři ale už přišli na to, jak pomocí již zmíněných malých reproduktorů produkovat zvuk, a tak se začaly objevovat první hry obsahující zvuky. Tyto zvuky byly tvořeny jen bzučivými a pípavými zvuky, přesněji zvuk byl vytvářen tzv. softwarovým syntetizátorem, který umožňoval přehrávání omezeného počtu tónů. To vývojáře hodně omezovalo ve vytváření různých zvuků. Díky vývoji her, které obsahovaly zvuky, přišli výrobci počítačů s prvními zvukovými kartami, jež v sobě měly zabudovaný hardwarový syntetizátor a mohly tudíž produkovat více tónů různých nástrojů. Pomocí těchto zvukových karet už vývojáři mohli tvořit celkem složité melodie (viz např. hudba v *sérii her Monkey Island* [2]). Stále ale měli potíže s tvorbou některých zvuků, poněvadž jim k jejich tvorbě nestačily tehdejší syntetizátory.

Na přelomu osmdesátých a devadesátých let přišli výrobci počítačů s novou generací zvukových karet. Tyto zvukové karty kromě toho, že nově vytvářely zvuk pomocí syntetizátoru, umožňovaly také přehrávat již navzorkované zvuky (zvukové soubory nahrané pomocí mikrofonu nebo vytvořené pomocí zvukového editoru) a aplikovat na ně různé efekty (např. smyčky – přehrávání zvuku dokola, echo – ozvěna, reverb – ozvěna místnosti, low-pass filter – průchod zvuku zdí, atd.). Tak se opět zvýšila rozmanitost zvukové stránky her.

Pro plynulé přehrávání navzorkovaných zvuků bylo ovšem potřeba tyto zvuky načíst do paměti. To vedlo k problému, že se všechna zvuková data kompletně do paměti nevešla (kvůli malé kapacitě pamětí v té době). Někteří vývojáři malou kapacitu paměti řešili tím, že měli zvuková data uložená na CD a poté přehrávali data přímo z CD. I to ale mělo svá omezení: například kvůli tomu, že zvuk byl posílán rovnou na výstup, nebylo možné na tento zvuk aplikovat efekty. Proto vývojáři vytvořili knihovny (dalo by se říci první audio enginy) umožňující načítat do paměti jen části zvuku ze souboru, který se má přehrávat. Další části zvuku byly postupně načítány v průběhu přehrávání.

Složené zvuky

Rozmanitost zvuků, které mohly audio enginy přehrávat, byla stále malá. Vývojáře tedy napadlo, že by mohli vytvářet složené zvuky, jež by byly tvořeny pomocí několika navzorkovaných zvuků, které by určitým způsobem kombinovali dohromady. Abychom si udělali představu, co jsou složené zvuky, uvedeme si pár typických příkladů: zvuk motoru auta (označme *S1*), zvuk lesa (*S2*), zvuk kroků hráče (*S3*), hudba závislá na stavu hry (*S4*).

S1) Nyní si řekneme, co je potřeba pro zvuk motoru auta, dále jen zvuk motoru. U her, které obsahují jen přesně definované scény s auty, které nemůže uživatel ovlivňovat, stačí nahrát jen jeden zvuk motoru a ten pak přehrát. Ale když uživatel bude moci auto ovládat (neboli bude moci libovolně přidávat plyn), nestačí už nahrát jen jeden zvuk motoru. Je zapotřebí umět přehrávat celý rozsah otáček motoru. Pro simulaci motoru v různých otáčkách je potřeba mít několik vzorků zvuků v určitých otáčkách motoru (např. 1200 rpm – volnoběh, 3000 rpm, 5000 rpm), neboť pomocí modifikací jednoho vzorku zvuku bychom nedocílili různých charakteristik zvuku motoru v různých otáčkách motoru. Tyto zvukové vzorky poté budeme přehrávat

ve smyčce a pomocí interpolace rychlosti přehrávání (změny frekvence zvuku) a hlasitosti jednotlivých vzorků zvuku můžeme přehrávat celé spektrum otáček motoru. Motor má ovšem jiný zvuk, když je v zátěži (pod plynem) nebo bez zátěže, tudíž je nutné udělat další vzorky zvuků při zátěži a poté v závislosti na sešlápnutí (poloze) plynu interpolovat hlasitost mezi zvuky se zátěží motoru a bez zátěže motoru. Navíc může být potřeba i zvuk pro nastartování a zastavování motoru, a tak je nutné přidat další zvuky. Ale protože nastartování a zastavování motoru nezní stále stejně, je zapotřebí nahrát několik vzorků a poté při přehrávání vždy vybrat jeden z nich. Průběh přehrávání zvuku motoru bude probíhat tak, že při startování motoru se přehraje zvuk nastartování motoru, který plynule přejde do zvuku běhu motoru, při němž uživatel může ovlivňovat otáčky motoru. V momentě, kdy přijde pokyn uživatele na zastavení motoru, přejde zvuk běhu motoru do zvuku zastavení motor. Po jeho přehrání zvuk končí. To je základní zvuk motoru, jenž lze ještě dále zlepšovat (např. praskání výfuku, apod.).

S2) Podobným příkladem mohou být i zvuky lesa. Cílem je, aby hráč, uslyšel tím pestřejší zvuky okolní přírody, čím je hlouběji v lese. Tudíž nahrajeme zvuky, které mají simulovat okraj lesa, dále zvuky, které hráče obklopují hlouběji v lese a nakonec uprostřed lesa. V závislosti na tom jak je hráč hluboko v lese, interpolujeme hlasitost jednotlivých předpřipravených zvuků a tím vytvoříme požadovaný efekt.

S3) Dalším příkladem složeného zvuku je přehrávání zvuku kroků nějaké postavy ve hře. Postava ale většinou nechodí jen po jednom typu povrchu (např. tráva a beton), je tedy potřeba pro každý povrch nahrát odlišný zvuk kroku. Navíc, když člověk jde, každý jeho krok nevydává přesně stejný zvuk, tudíž i postava ve hře by neměla vydávat stejné zvuky kroků. Takže pro každý povrch vyskytující se ve hře je potřeba nahrát několik vzorků zvuku. Poté v závislosti na daném povrchu, na kterém se postava ve hře nachází, potřebujeme při každém jejím kroku přehrát jeden ze vzorků zvuku, patřící ke konkrétnímu povrchu. Přitom je nutné při každém dalším kroku mezi jednotlivými vzorky jasně přecházet (hráč by si totiž mohl všimnout stejně znějících kroků, což by považoval za nedokonalost).

S4) Posledním typickým příkladem je přehrávání hudby v závislosti na stavu hry (od předchozího příkladu *S3* se liší tím, že hudba, na rozdíl od kroků, hraje nepřetržitě). Pro jeden konkrétní stav hry máme vždy několik hudebních souborů, které se mají přehrávat a postupně střídat. Vždy když se změní stav hry, přejde se na jinou skupinu hudebních souborů.

Složené zvuky však přinášely problém při řízení jejich přehrávání v aplikaci. Místo toho, aby na struktuře, reprezentující složený zvuk, stačilo zavolat metodu pro spuštění přehrávání složeného zvuku, bylo nutné pro řízení přehrávání tohoto složeného zvuku vytvořit netriviální část kódu. Struktury se totiž skládaly z několika samostatných zvuků a neexistovala žádná struktura, která by tyto zvuky reprezentovala pomocí jednoho objektu. Metodu pro spuštění přehrávání složeného zvuku si ukážeme na příkladu řízení přehrávání složeného zvuku bez struktury složeného zvuku.

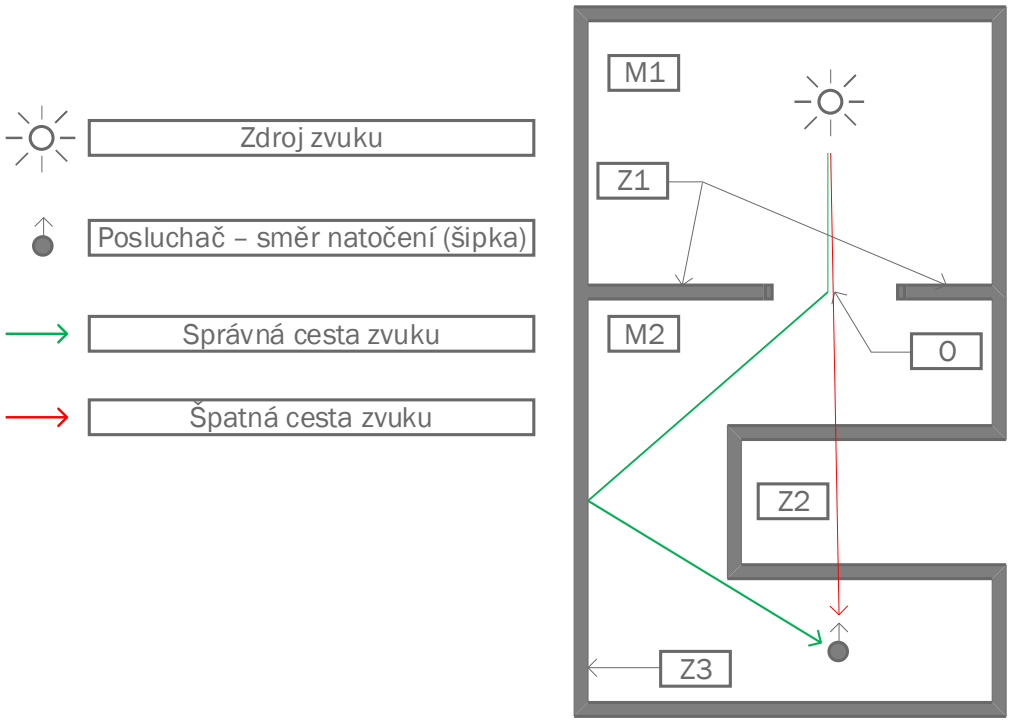
Popíšeme si postup používání zvuku na zdrojovém kódu herní logiky, jež bude přehrávat zvuk *S1* (zvuk motoru auta). V určité části kódu reaguje herní logika na klávesu, kterou hráč startuje auto. V kódu herní logiky se náhodně vybere jeden ze zvuků startování motoru (jelikož zvuk motoru auta obsahuje několik zvuků startování motoru) a zavolá se na něm metoda pro jeho spuštění. Kód herní logiky ale nemůže pokračovat dál, neboť je nutné zajistit, aby před skončením zvuku startování motoru tento zvuk přešel plynule do zvuku běhu motoru. To by se v kódu herní logiky mohlo zajistit tak, že se herní logika bude ve smyčce v malých intervalech dotazovat na pozici přehrávání startovacího zvuku. Až bude tento zvuk končit, tak na zvuku motoru ve volnoběhu zavolá metodu pro jeho spuštění. Tento cyklus ovšem nemůže být jako synchronní součást kódu herní logiky, protože do doby než by dohrál startovací zvuk motoru, zasekla by se herní logika na kontrole pozice zvuku startovacího motoru, tudíž je nutné zavolat asynchronně metodu, jež bude tento cyklus obsahovat. Další důležitou částí kódu herní logiky bude část, kdy kód reaguje na přidávání plynu (změnu otáček motoru) hráčem. V kódu se vypočtou odpovídající otáčky motoru, ale pak se musí vypočítat i hlasitost a rychlost přehrávání pro jednotlivé zvuky určitých otáček motoru a zavolat na zvucích běhu motoru metody pro nastavení hlasitosti a rychlosti přehrávání. Poté v kódu herní logiky se při splnění určitých podmínek (např. hráč stiskl tlačítko pro vypnutí motoru či došlo palivo) musí zajistit přechod zvuku z běhu motoru na jeho zastavení. Opět však nastává problém: nestačí zastavit zvuky běhu motoru pomocí kódu a spustit zvuk zastavení motoru, ale musí se pomocí asynchronního cyklu plynule přejít ze zvuku běhu motoru na jeho zastavení. Práce se zvukem se tedy sestává z mnoha řádků kódu, jež budou součástí kódu herní logiky, přičemž s každým dalším podobným zvukem bude přibývat dalších mnoho řádků kódu, jež budou zajišťovat přehrávání těchto zvuků.

Herní logika musela pro přehrávání zvuku obsahovat netriviální část kódu pro přehrávání zvuků, proto bylo potřeba vytvářet pro složené zvuky struktury, jež obsahovaly jednotlivé zvuky složeného zvuku a předem nadefinované hodnoty, jak má zvuk probíhat, případně jak se může zvuk ovlivňovat (např. událostmi, parametry). Tyto struktury v základě obsahovaly rozhraní jako jednoduché zvuky a staraly se automaticky o řízení průběhu přehrávaných zvuků. To vývojářům herní logiky umožňovalo soustředit se na svojí práci a nezabývat se detaily přehrávání těchto složených zvuků. Použití struktury složeného zvuku si ukážeme na příkladu.

Ukážeme si opět postup používání zvuku na příkladu zvuku *S1* (zvuk motoru auta) s využitím těchto speciálních struktur (dále jen struktura). Mějme zdrojový kód herní logiky, přičemž v určité části má kód herní logiky reagovat na klávesu, kterou hráč startuje auto. V kódu se tedy na struktuře zvuku motoru zavolá metoda pro spuštění přehrávání motoru a poté následuje další kód herní logiky. Stejně jako v příkladu bez použití struktury dále následuje část kódu, kdy má herní logika reagovat na to, jak hráč přidává plyn (mění otáčky motoru). Kód herní logiky podle polohy plynu vypočte odpovídající otáčky motoru a pak zavolá metodu struktury zvuku motoru, jež zvuku nastaví parametr otáček motoru. Poté v místě herní logiky, kde se má zajistit zastavení motoru, kód herní logiky zavolá metodu na struktuře zvuku motoru pro provedení události „zastav motor“. Jak je vidět, používání složeného zvuku pomocí speciálních struktur je o mnoho snazší.

Šíření zvuku virtuálním prostředím

S rozvojem 3D her vznikl další problém, kterým byla simulace šíření zvuku prostorem. Bylo potřeba, aby hráč slyšel zvuk přicházet z určitého směru (směrů), jenž byl určen tvarem virtuálního prostředí. Aby se docílilo tohoto efektu, bylo zapotřebí vypočítávat cestu šíření zvuku prostorem od jeho zdroje k hráči a podle cesty se následně určil směr, odkud zvuk přicházel (příklad nedokonalého určení směru zvuku ze hry Half-Life je uveden v následujícím odstavci). To ale nestačilo, neboť hráč očekával, že šíření zvuku ve hře se bude podobat šíření zvuku v reálném prostředí. Tudíž bylo potřeba přidat ke zvuku různé efekty (jako ozvěna, zkreslení) v závislosti na prostředí, kudy vede cesta šíření zvuku virtuálním prostředím. Díky tomu se šíření zvuku virtuálním prostředím přibližovalo šíření zvuku v reálném světě.



Obrázek 1.1 - Situace ze hry Half-Life

Jako příklad nedokonalého šíření zvuku prostorem mohu uvést situaci ze hry *Half-Life* [3] (viz obrázek *1.1*), kde máme dvě místnosti: první čtvercovou (*M1* – pro ilustraci horní místnost), ve které je umístěn zdroj zvuku (*S*) a druhou místnost (*M2* – dolní místnost) ve tvaru podkovy, napojenou zdola pomocí otvoru ve zdi (*O*) na první místnost. Hráč (*H*) je umístěn v dolní části druhé místnosti a je otočen směrem nahoru (k první místnosti), takže mezi hráčem a zdrojem zvuku je otvor ve zdi (*Z1* – která spojuje obě místnosti) a zeď (*Z2*) druhé místnosti (díky místnosti tvaru podkovy). Jenže hráč slyší zvuk zdroje zvuku zepředu (shora), který není utlumený zdí (*Z2*), ačkoliv by spíše měl slyšet zvuk zleva, jež prochází otvorem (*O*), kde by se měl lámat a poté se odrazit od zdi (*Z3*).

Simulování šíření zvuku virtuálním prostředím vedlo k stále větším nárokům na efektivitu aplikování efektů, neboť pro přesnější simulaci šíření zvuku bylo nutné aplikovat několik efektů na každý zvuk. Každá aplikace efektu ale zabírala určitý čas

pro zpracování zvukových dat, což se ve výsledku projevovalo na opožděném přehrávání zvuku. Tohoto zpoždění bychom se měli snažit vyvarovat. Bylo tedy potřeba vymýšlet nové algoritmy, kterými by se dokázali rychleji aplikovat efekty na zvuková data.

Dalším důležitým efektem při šíření zvuku prostorem je zpoždění zvuku. Šíření zvuku je totiž omezeno rychlostí jeho šíření, naproti tomu světlo se šíří rychleji. Realita je taková, že když se něco stane daleko od nás, vidíme to téměř okamžitě, zatímco zvuk bývá o chvíli opožděn. Těchto situací v reálném světě si však lidé všímají málo (typickým příkladem je, když někde vidí letět letadlo, zatímco zvuk přichází ze směru, odkud letělo). Zato v častých situacích, kdy je zdroj zvuku vzdálen do 100 m od pozorovatele, si lidé neuvědomují zpoždění zvuku (protože je velmi malé). Od her tudíž očekávají, že když vidí nějakou akci, měli by ihned slyšet i zvuk. Ukažme si to na příkladu výstřelu z pistole.

> Vezměme si příklad, kdy hráč vystřelí z pistole a očekává, že zvuk výstřelu bude slyšet v okamžiku, kdy uvidíme záblesk spalování střelného prachu u hlavně pistole. Zvuk výstřelu má být správně opožděn. Bude-li však opožděn, může to zmást hráče a kazit zážitek ze hry.

Rozdělení definování a používání zvuků

Díky rozvoji her se zvyšoval počet zvuků ve hrách a zároveň se zpřesňovalo prostředí pro šíření zvuku. Proto se definování složených zvuků a vlastností prostředí pro šíření zvuku stávalo náročnějším a vývojář musel znát o chování zvuku více informací. To vedlo firmy k tomu, že začaly vyčleňovat pro definování zvuků a prostředí speciální členy týmu tzv. *designéry*. Tím nezatěžovaly vývojáře, kteří měli na starosti vývoj herní logiky či grafiky. Nicméně u malých firem vývojáři stále zastávali obě funkce.

Vývojáři se měli soustředit jen na programování, designéři jen na definování zvuku. Designéři ale většinou nejsou programátoři, a tudíž nejsou zvyklí psát kód jako programátoři. Proto bylo potřeba vytvořit grafické editory, které umožní designérům soustředit se na svou práci a lépe definovat zvuky.

Hra více hráčů

S rozvojem 3D her se také začínaly vytvářet hry pro více hráčů. Při hře více hráčů na jednom počítači však nastal problém. Obraz se totiž dělil na několik částí, jehož každá část byla určena pro jednoho hráče a měla produkovat zvuk nezávisle na jiné části. Avšak dosavadní audio enginy umožňovaly přehrávat zvuk s podporou jen jednoho posluchače. Pro podporu více posluchačů bylo potřeba upravit audio enginy, jinak se musela ručně vytvořit nadstavba nad audio enginem, která to umožnila.

Prioritizace zvuků

Kvůli množství přehrávaných zvuků a potřeby na tyto zvuky aplikovat různé efekty, audio enginy narážely na nedostatečný výkon počítačů. To vedlo k tomu, že bylo zapotřebí, aby audio engine třídil zvuky na ty, které je potřeba přehrávat přednostně a na ty, u kterých to nutné není (neboli, které nemají na výsledný zvuk rozhodující vliv). K tomu audio enginy většinou používaly parametr priority k jednotlivým zvukům a též se rozhodovaly podle vzdálenosti zdroje od posluchače. Aby audio enginy měly jistotu, že se všechny zvuky stačí plynule přehrávat, jejich počet většinou omezovali konstantou a nenechávali to na konkrétní situaci náročnosti přehrávání zvuků. To vedlo k tomu, že hry uměly přehrávat dohromady jen malé

množství zvuků. Díky vývoji zvukových karet umožňujících rychlejší zpracování zvuků se toto množství zvyšovalo a už skoro nebylo poznat, že jsou nějaké zvuky utlumeny.

Tvorba univerzálního řešení

Na začátku vývoje her se každý audio engine odlišoval hlavně podle svého výrobce. Specifičnost zvuku záležela na tom, co konkrétní hra vyžadovala. To ale představovalo problém, protože každá firma si chránila svoje řešení a začínající nové firmy nebo skupiny vývojářů, které chtěli vytvářet hry, si museli vytvořit vlastní engine, jež některé věci uměl lépe a některé hůře. Proto vývojáři začali vytvářet audio enginy, které byly dostupné pro každého, což byl dobrý začátek. Díky množství platforem, jež má každá jiné rozhraní pro komunikaci se zvukovou kartou, vznikaly různé audio enginy. Postupem času vývojáři nechtěli hry tvořit jen pro jednu platformu, ale chtěli je mít dostupné na několika platformách a to představovalo problém kvůli různorodosti audio enginů. S nárůstem multiplatformních her (dostupných na více platformách) vývojáři vytvořili audio enginy, jež měly stejné rozhraní pro ovládání zvuku, ale dokázaly běžet na různých platformách.

1.1.2 Potřeby dnešních her – shrnutí

Výše jsme si popsali, co je vyžadováno od zvuku ve hrách. Pro lepší přehled si nyní shrneme předchozí text do několika bodů popisujících potřeby dnešních her.

R1) Základním požadavkem dnešních her na audio engine je, aby všechny výpočty týkající se zvuku probíhaly v reálném čase. Jde totiž o to, aby byly zvuky hry ve výsledku synchronizovány s grafikou hry. (Viz kapitola *1.1.1 Vývoj zvuku ve hrách – Šíření zvuku virtuálním prostředím*.)

R2) Je potřeba, aby se zvuky načítaly, až když jsou potřeba a zbytečně tak nezaplňovaly paměť (většina zvuků se totiž vyskytuje jen v určité části virtuálního prostředí, či určitém levelu hry).

R3) Dále je zapotřebí umět efektivně slučovat zvuky dohromady a umět efektivně aplikovat různé efekty jako echo, reverb, apod.

R4) Pak je nutné umět určovat, které zvuky se mají zpracovávat přednostně a které následně.

R5) Hry mají obsahovat složené zvuky (jako výše v kapitole *1.1.1 Vývoj zvuku ve hrách* příklady zvuků *S1*-*S4*). O definici těchto zvuků by se staral designer zvuků pomocí editoru. Následně by tyto zvuky měly být ve hře přehrávány stejně jednoduše jako jednoduché zvuky.

R6) Pro simulaci šíření zvuku virtuálním prostředím je potřeba vypočítávat cestu šíření zvuku virtuálním prostředím od jeho zdroje k posluchači. Podle cesty je nutné určit směr zvuku společně s efekty a jejich parametry, které se mají na zvuk aplikovat.

R7) Pro podporu více hráčů na jednom počítači či tvorbě složitějších efektů je potřeba umět vytvářet více posluchačů ve virtuálním prostředí, které lze ovládat (přesunovat) nezávisle na sobě.

R8) Multiplatformní řešení – díky němu může být audio engine provozován na různých platformách.

1.2 Definice struktury audio enginu

Hlavním cílem této práce je vytvoření audio enginu. Výše jsme si definovali požadavky, jaké hráči očekávají od zvuku ve hrách. Nyní si řekneme, jak to vypadá z programátorského hlediska, abychom získali přehled o struktuře audio enginu.

Zde ale vyvstává první problém. Při zkoumání audio enginů jsme nenalezli žádnou definici, co audio engine je a z jakých částí (vrstev) je tvořený. Tudíž se podíváme na již existující audio enginy. Z toho, co obsahují a umějí, se pokusíme sestavit popis audio enginu. Pro tyto potřeby si představíme typické audio enginy, jež se často používají při vývoji her. Jsou jimi *FMOD* [4], *WWise* [5], *irrKlang* [6], *OpenAL* [7] a *XAudio2* [8].

FMOD byl až do verze 3.75 považován přímo za knihovnu starající se o zvuk. Od vyšších verzí byl FMOD předělán a nyní tento název označuje skupinu čtyř částí, od editorů po knihovnu. Nás konkrétně bude zajímat část FMOD Ex (dříve jen FMOD), což je knihovna starající se o zvuk. WWise se stejně jako FMOD skládá z editoru a knihovny. Dalším je irrKlang, na rozdíl od FMODu a WWise se přímo jedná o knihovnu starající se o zvuk, k níž neexistuje editor. Přičemž tyto tři zmíněné audio enginy patří mezi komerční a jsou si svou funkčností více méně podobné. OpenAl a XAudio2 jsou zástupci open source audio enginů. Co se týče funkcí, jsou na rozdíl od těch komerčních omezené.

Horní a dolní vrstva audio enginu

Při zkoumání typických audio enginů jsme přišli na to, že knihovna se často dělí na základní dvě části tzv. dolní (low-level) a horní (high-level) vrstvu. Z jedné části je to dáno postupným vývojem zvuku ve hrách. Hlavním důvodem je ale přenositelnost na jiné platformy, na kterých se mohou hry spouštět. Neboť horní vrstva se skládá z částí, jež se starají o to, jak by měl být zvuk ovlivněn a pozměněn. Dolní vrstva obsahuje části, které se starají o zpracovávání a modifikaci zvukových dat, a často bývá závislá (z výkonových důvodů – rychlosti zpracování zvukových dat) na konkrétní platformě (např. díky API pro komunikaci se zvukovou kartou), na které běží. Aby mohl audio engine fungovat na více platformách, horní vrstva zůstává stejná, ale dolní vrstva se musí přeprogramovat pro použití na jiných platformách, případně audio engine musí obsahovat více dolních vrstev (každou pro jednu konkrétní platformu).

Nicméně hranice mezi horní a dolní vrstvou není tak přesně určena, jak by se mohlo zdát. Příkladem je načítání zvukových souborů, jež obsahuje knihovna FMOD Ex. Tato knihovna je firmou označena jako audio engine s dolní vrstvou. Zato knihovna audio enginu irrKlang řadí načítání zvukových souborů do horní vrstvy.

Definice podvrstev audio enginu

Výše jsme se dozvěděli, že neexistuje žádná přesná definice audio enginu a že knihovny existujících audio enginů se dělí na horní a dolní vrstvu. Pojďme si nyní definovat, z jakých podčástí (podvrstev) by se tyto vrstvy měly skládat a z jakého důvodu. Poté si ukážeme, jak by měl vypadat základní průběh zpracování zvukových dat.

Dolní vrstva by se měla skládat z těchto podvrstev:

- *Výstup* – stará se o předávání zvukových dat zvukové kartě na výstup, případně zpět aplikaci pro další zpracování.

- *Aplikace efektů, slučování zvuků* – stará se o aplikaci efektů jako echo, reverb, atd. Dále se stará o slučování více zvukových dat z více stop do jedné stopy, přičemž zvuková data může zpracovat softwarově na procesoru nebo pomocí zvukové karty. (Vyplývá z požadavku *R3* z kapitoly *1.1.2 Potřeby dnešních her – shrnutí*)
- *Vytvoření zvuku na dolní vrstvě* – stará se o posloupnost modifikací, které se mají pomocí dolní vrstvy udělat na zvukových datech.

Horní vrstva by se měla skládat z těchto podvrstev:

- *Načítání zvukových dat* – stará se o načítání zvukových dat ze zvukových souborů, které následně předává dolní vrstvě ke zpracování. Tuto vrstvu jsme umístili do horní vrstvy, protože rozhraní pro čtení souborů jsou už na většině platforem sjednocena. (Vyplývá z požadavku *R2* z kapitoly *1.1.2 Potřeby dnešních her – shrnutí*)
- *Načítání složených zvuků* – stará se o načítání struktur složených zvuků. (Vyplývá z požadavku *R5* z kapitoly *1.1.2 Potřeby dnešních her – shrnutí*)
- *Prioritizace přehrávání* – stará se o to, jaké zvuky mají být přehrány přednostně a které později, aby nedošlo k zahlcení procesoru. (Vyplývá z požadavku *R4* z kapitoly *1.1.2 Potřeby dnešních her – shrnutí*)
- *Výpočet šíření zvuku prostorem* – stará se o výpočet cest mezi zdrojem zvuku a posluchačem, přičemž podle cest vypočítává, které efekty s jakými parametry se mají na zvuk v dolní vrstvě aplikovat. (Vyplývá z požadavku *R6* z kapitoly *1.1.2 Potřeby dnešních her – shrnutí*)
- *Zdroje, posluchači* – stará se o umístění zdrojů a posluchačů ve virtuálním prostředí a spuštěné zdroje zvuku. Pokud se ve virtuálním prostředí vyskytuje více posluchačů, zároveň zajišťuje, které zdroje zvuku má slyšet který posluchač. (Částečně vyplývá z požadavku *R7* z kapitoly *1.1.2 Potřeby dnešních her – shrnutí*)

Nyní si popíšeme základní průběh zpracování zvukových dat (znázorněný ve vrstvách na obrázku *1.2*).

Pomocí vrstvy *Zdroje, posluchači* (*A*) si mohou vývojáři v enginu vytvářet jednotlivé zdroje zvuku a posluchače. V případě, kdy jsou potřeba souřadnice zdrojů a posluchačů, tyto souřadnice vývojáři aktualizují. Dále vrstva *Zdroje, posluchači* (*A*) umožňuje vývojářům řízení přehrávání zdrojů zvuku. Následně engine ve vrstvě *Výpočet šíření zvuku prostorem* (*B*) vypočítává cesty od spuštěných zdrojů zvuku

Obrázek 1.2 - Průběh zpracování zvuku

k posluchačům a na základě cest vypočítává efekty, které se mají aplikovat na zdroje zvuku. Pokud se nepřehrává zvuk v prostoru, tato vrstva se přeskakuje. Vrstva *Prioritizace přehrávání* (*C*) se pak stará o to, jestli daný zvuk se má opravdu přehrát, či nikoli (z výkonnostních důvodů). Poté, obsahuje-li zdroj zvuku složený zvuk, engine ve vrstvě *Načítání složených zvuků* (*D*) vytvoří strukturu složeného zvuku (např. FMOD je má založené na časové ose, zatímco Wwise je má založené na událostech), jenž se může skládat z několika zvukových souborů a dalších efektů. Ve vrstvě *Načítání zvukových dat* (*E*) se následně přiřadí zdroj zvukových dat, který při zpracovávání zvuku pomocí této vrstvy načítá zvuková data a posílá je vrstvě *Vytvoření zvuku na dolní vrstvě* (*F*), která se stará o sestavení průběhu zpracování zvuku na dolní vrstvě. Následně vrstva *Aplikace efektů, slučování zvuků* (*G*) se stará o samotné zpracování zvukových dat (neboli aplikuje různé efekty a spojuje zvuková data dohromady). Tato data se ve vrstvě *Výstup* (*H*) posílají na zvukovou kartu či případně zpět aplikaci.

Toto je typický průběh zpracování dat s využitím horní vrstvy. Vývojáři ale mohou přeskočit vrstvy až do vrstvy *Vytvoření zvuku na dolní vrstvě* (*F*) a využívat přímo jen dolní vrstvu audio enginu pro samotné zpracovávání zvukových dat, případně mohou používat jen některé části horní vrstvy.

Úprava cíle práce

Jako cíl této práce jsme si zvolili vytvoření audio enginu. Z důvodu rozsáhlosti audio enginu a toho, že všechny audio enginy jsou tvořeny dolní vrstvou (zpracováním zvukových dat), upravíme si cíl této práce na vytvoření horní vrstvy audio enginu s využitím dolní vrstvy již existujícího audio enginu.

1.3 Cíle

Nyní si v několika bodech uvedeme konkrétní cíle, které vedou k vytvoření horní vrstvy audio enginu:

C1) Nalezení vhodné dolní vrstvy při zachování přenositelnosti na různé platformy – přičemž vhodný způsob zachování přenositelnosti vybereme až při tvorbě této práce. Výběr plyne z potřeby *R3* a *R8* dnešních her.

C2) Správa zdrojů zvuků a posluchačů ve virtuálním prostředí – abychom mohli definovat jejich polohu (využít ji při výpočtu šíření zvuku prostorem) a dále uchovávat informace o konkrétních zdrojích zvuku a stavu jejich přehrávání. Vyplývá z potřeby *R7* dnešních her.

C3) Vhodné čtení souborů, správa prostředí (změna lokace – načítání dalších zvuků) – abychom zbytečně nezaplňovali paměť (např. zvuky v jiném levelu hry). Plyne z potřeby *R2* dnešních her.

C4) Možnost vytváření složených zvuků, jako *S1*-*S4* v kapitole *1.1.2 Potřeby dnešních her – shrnutí*. Bez této možnosti bychom mohli přehrávat jen jednoduché zvuky, což by velmi omezovalo použití audio enginu. Vyplývá z potřeby *R5* dnešních her.

C5) Editor pro vytváření složených zvuků – aby bylo možné jednoduše vyrábět složené zvuky, plyne z potřeby *R5* dnešních her.

C6) Výpočet šíření zvuku prostorem – je důležitým prvkem pro simulaci 3D zvuku, jenž se využívá pro výpočet směru, odkud má být zvuk slyšet. Je možné ho

použít též k výpočtu efektů (jako ozvěna, atd.), které se mají na daný zvuk aplikovat. Plyne z potřeby *R6* dnešních her.

C7) Prioritizace přehrávání zvuků – slouží k upřednostňování přehrávání zvuků, aby nedocházelo k zahlcení procesoru a tím ke zpoždění či sekání zvuku. Vyplývá z potřeby *R1* a *R4* dnešních her.

1.4 Struktura

V **druhé kapitole** probereme analýzu, co všechno má práce obsahovat, problémy a řešení jednotlivých částí. Z důvodu rozsahu práce ve **třetí kapitole** omezíme zadání pouze na významné body práce, přičemž probereme případné úpravy probraných částí a následně stanovíme nové cíle. Ve **čtvrté kapitole** se budeme zabývat technickými detaily audio enginu, zatímco v **páté kapitole** se podíváme na technické detaily editoru. V **šesté** a **sedmé kapitole** si ukážeme postupy ovládání audio enginu a editoru. Následně v **osmé kapitole** provedeme zhodnocení práce. Pak následují kapitoly se seznamy použité literatury a přílohy.

2 Analýza

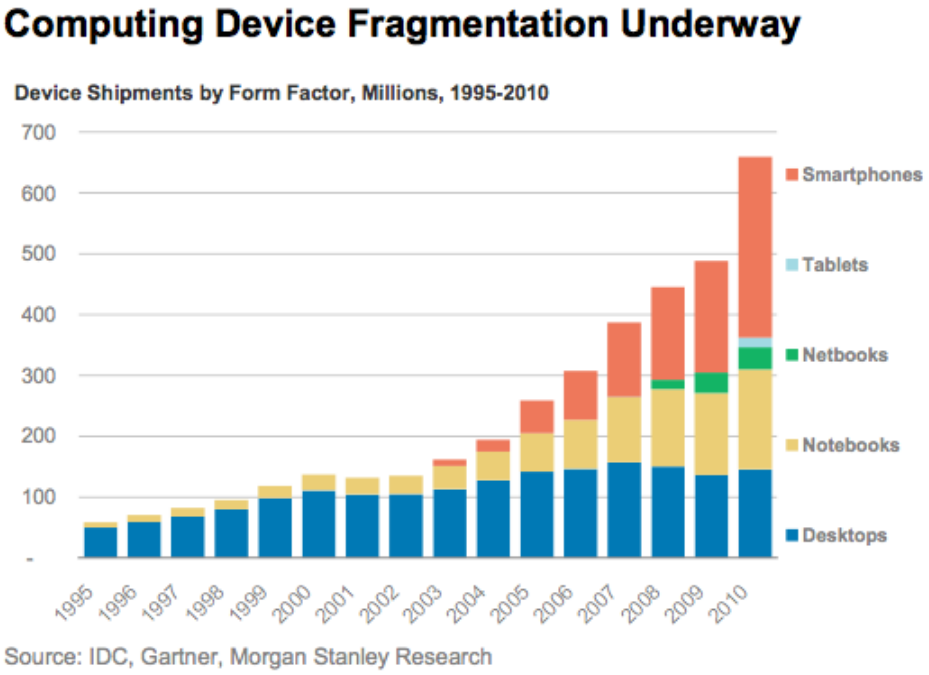
Tato kapitola se bude zabývat analýzou tvorby audio enginu, přičemž se zaměříme na různé problémy týkající se splnění cílů uvedených v kapitole *1.3 Cíle*. K problémům následně navrhneme některá řešení a vybereme to, které vyhovuje pro danou situaci nejvíce.

V průběhu analýzy zjistíme, že i když jsme se rozhodli pro dolní vrstvu použít již existující audio engine, abychom zredukovali rozsah práce, tak z analýzy této práce nakonec usoudíme, že bude potřeba zúžit cíle této práce, jež probereme v kapitole *3 Zúžení rozsahu práce*. Z toho důvodu některé části nebudou podrobně probrány, především ty pro práci s 3D zvukem, neboť se bude jednat o ty, které nakonec z cílů této práce vypadnou.

V následujících podkapitolách si probereme rozhodnutí týkající se výběru platformy pro implementaci audio enginů. Dále se zaměříme na konkrétní cíle této práce a s tím související problémy.

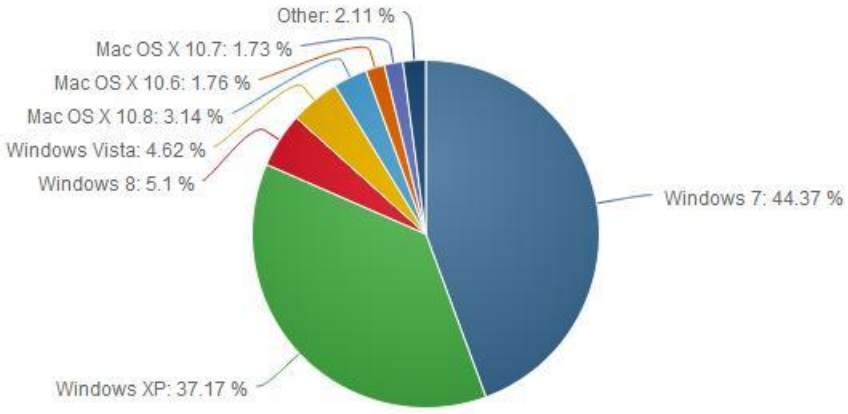
2.1 Výběr platformy

Začneme výběrem hardwarové platformy, pro kterou budeme audio engine programovat. Mezi hlavní hardwarové platformy patří osobní počítače, herní konzole, herní automaty a mobilní zařízení. Dříve byly jako hardwarová platforma nejrozšířenější osobní počítače, ale nyní už se mezi nejrozšířenější hardwarovou platformu spíše řadí mobilní zařízení, viz graf dělení výpočetních zařízení na obrázku *2.1*. Jako hardwarovou platformu však vybereme osobní počítače, protože autor práce tuto platformu preferuje a pro vývoj ostatních platforem nemá dostupné hardwarové prostředky.



Obrázek 2.1 - Dělení výpočetních zařízení, převzato z článku Tablet Marketing: 30 Must-Have Facts [Charts] [9]

Po výběru hardwarové platformy následuje další důležité rozhodnutí a tím je volba operačního systému. Z operačních systémů máme na výběr rodinu Windows, poté různé distribuce operačního systému Linux a také operační systémy OS X. V minulosti na počítačích převládaly operační systémy Windows, a i když se v dnešní době pomalu zvyšuje podíl ostatních operačních systémů, operační systém Windows stále převládá (viz obrázek *2.2*). To vede k tomu, že se většina her tvořila a tvoří pro operační systém Windows, přičemž v dnešní době se vývojáři snaží, aby byly hry dostupné pro co nejvíce operačních systémů. Proto jako operační systém pro vytvoření audio enginu vybereme operační systém Windows, konkrétně Windows 7, neboť je v době této práce nejrozšířenější.



Obrázek 2.2 - Podíl OS na osobních PC (červen 2013), převzato z článku Windows 8 Market Share Outpaces Vista, but Is Still Far Below Windows 7 and Windows XP [10]

Další důležitou volbou je volba programovacího jazyka. V dnešní době existuje mnoho programovacích jazyků, avšak pro programování her (mimo programování umělé inteligence a skriptů ve hrách) se většinou využívají jazyky C, C++, C# (.NET), Java. C a C++ jsou starší jazyky, hodící se pro rychlé výpočty, jejichž hlavní rozdíl je, že C++ na rozdíl od C umí objektově orientované programování. Dalším jazykem je C#, který patří mezi novější jazyky patřící do rodiny jazyků .NET, jež je především používán s operačním systémem Windows, ale v posledních letech se rozšířil i na jiné operační systémy a herní zařízení. Psaní C# kódu je na rozdíl od C a C++ z pohledu autora jednodušší, přičemž C# lze dobře kombinovat s ostatními .NET jazyky. Posledním ze zmíněných jazyků je Java, ta též patří mezi novější jazyky a navíc je dostupná na velkém množství platforem (jak softwarových, tak hardwarových). Javu ale nebudeme zvažovat, neboť ta se používá většinou jen pro vývoj her pro mobilní zařízení. Díky výše zmíněným informacím si vybereme C# jako programovací jazyk k implementaci audio enginu.

Nyní se dostáváme k prvnímu cíli naší práce a tím je cíl *C1* (Nalezení vhodné dolní vrstvy při zachování přenositelnosti na různé platformy). Zvolená platforma spolu s operačním systémem a programovacím jazykem nám ale omezuje, jaké existující audio enginy můžeme použít jako dolní vrstvu pro naši implementaci horní vrstvy audio enginu. Součástí cíle je i přenositelnost naší horní vrstvy audio enginu na jiné platformy. Toho můžeme docílit dvěma způsoby:

1. Vybereme si jako dolní vrstvu audio enginu audio engine, který je přenositelný na různé platformy.
2. Vytvoříme si tzv. *mezivrstvu*, která bude propojovat horní a dolní vrstvu audio enginu, přičemž rozhraní mezivrstvy pro horní vrstvu bude pevně definováno,

takže pokud budeme chtít zprovoznit naší horní vrstvu audio enginu na jiné platformě, přepíšeme jen mezivrstvu pro použití jiného audio enginu jako dolní vrstvy našeho audio enginu.

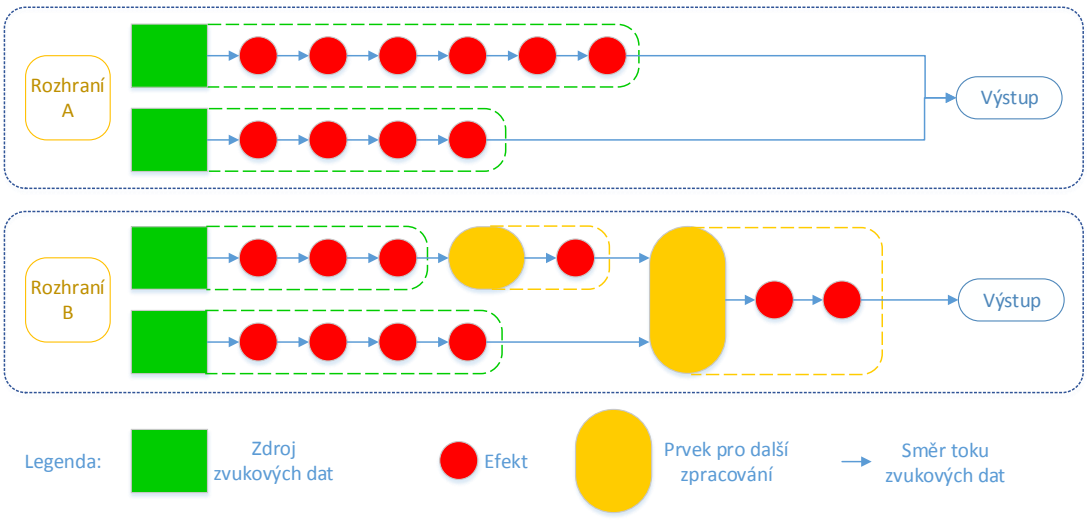
Z výše dvou uvedených způsobů, jak řešit přenositelnost audio enginu na jiné platformy, si vybereme druhý způsob, abychom nebyli závislí jen na jednom audio enginu, který poslouží jako dolní vrstva našeho enginu. Navíc díky tomuto způsobu bude možno v budoucnu vytvořit vlastní dolní vrstvu audio enginu bez předělávání horní vrstvy audio enginu.

2.2 Mezivrstva

V předchozí podkapitole jsme si řekli, že pro zachování přenositelnosti na jiné platformy použijeme mezivrstvu, což je částí cíle *C1* této práce. Audio enginy obsahující dolní vrstvu ale nemají jednotné rozhraní, tudíž je potřeba pro mezivrstvu zvolit vhodnou strukturu, kterou bychom mohli použít pro napojení různých dolních vrstev.

Každá dolní vrstva audio enginu má jiné rozhraní, přičemž při zkoumání různých enginů jsme narazili na dva obecné typy, jejichž princip si v následujících odstavcích zjednodušeně popíšeme.

První typ rozhraní (na obrázku *2.3 – rozhraní A*) je, že v základě vytváříme zdroje zvukových dat, jež slouží k řízení přehrávání jednotlivých zvuků. Tyto zdroje zvukových dat umožňují napojení na zvukový soubor, ale některé audio enginy umožňují i napojení na zvuková data dodávaná samotnou aplikací. Dále k jednotlivým zdrojům zvukových dat je možné přidávat různé efekty, jež určitým způsobem upravují zvuková data. Každý zdroj zvuku je tedy tvořen jedním zdrojem zvukových dat a poté seznamem efektů, které se mají na zvuk aplikovat. Příkladem použití tohoto typu rozhraní jsou audio enginy OpenAL a SDL [11].

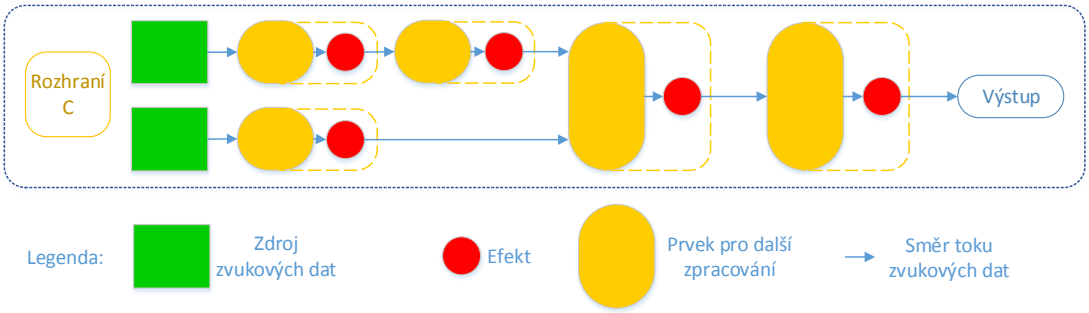


Obrázek 2.3 - Způsoby reprezentace zpracování zvukových dat

Druhý typ rozhraní (na obrázku *2.3 – rozhraní B*) se skládá ze dvou hlavních prvků: zdroje zvukových dat a prvku pro další zpracování dat. Oba prvky přitom mohou obsahovat seznam efektů, které se mají aplikovat na zvuková data daného prvku. Zdroj zvukových dat opět umožňuje napojení na zvukové soubory. Prvek pro další zpracování pak umožňuje mít jako vstup několik zdrojů zvuků či jiných

prvků pro další zpracování dat, jejich zvuková data sloučí do jedné stopy a poté po zpracování efektů je umožňuje poslat dále, ať už k dalšímu zpracování, tak na výstup. Příkladem použití tohoto typu rozhraní je audio engine XAudio2.

Jak je vidět druhý typ rozhraní je obecnější a lze je jednoduše použít i pro první typ rozhraní. Můžeme se ale zamyslet nad tím, že bychom mohli toto rozhraní ještě více zobecnit, že by i každý efekt byl samostatný objekt, jenž by přijímal zvuková data, zpracoval je a poslal dál (viz obrázek *2.4 – rozhraní C*, kde efekt jako samostatný objekt je tvořen prvkem pro další zpracování dat s přiřazeným jedním efektem).



Obrázek 2.4 - Způsob reprezentace zpracování zvukových dat

Nejlepší bude zvolit nejobecnější typ rozhraní, neboť nám umožní napojovat se na jiné typy rozhraní a zachová se vysoká míra přenositelnosti na různé dolní vrstvy audio enginů.

Vybrané rozhraní ale může přinášet problémy s pozastavováním zvuku. Představme si zvuk ve hře, třeba ránu do bubnu, na který je aplikován echo efekt. Necháme přehrát ránu do bubnu a poté hru pozastavíme. V tu chvíli by měly přestat hrát všechny zvuky a poté, co opět spustíme hru, měly by doběhnout ozvěny rány do bubnu. Jenže v druhém způsobu nastává problém s efekty po slučování zvuku, neboť při zastavení zdrojů zvuku, jež se slučují do jednoho, echo efekt nemá informace, že se všechny zdroje pozastavily a tak se efekt stále dohrává. Možné řešení by bylo, že bychom pozastavovali i samotné efekty, ovšem tuto možnost dostupné audio enginy neobsahují. Proto jediným způsobem, jak spolehlivě zastavit a znovu spustit zvuky je pozastavení celého audio enginu. To ale vede k dalšímu problému a tím je, že když hru pozastavíme, objeví se menu, které by mělo vydávat vlastní zvuky, ale kvůli tomu, že jsme pozastavili celý audio engine, zvuky menu se též nepřehrávají. Tento problém lze vyřešit vytvořením dvou instancí audio enginu, kde zvuky menu přehráváme pomocí jedné instance, zatímco zvuky hry pomocí druhé instance.

Dolní vrstvy kromě různých rozhraní obsahují i různé efekty, které umějí aplikovat na zvuk, ale ne všechny audio enginy umějí aplikovat stejné efekty. Z tohoto důvodu by měla mezivrstva obsahovat i rozhraní, jež by informovalo, jaké efekty jsou na dané platformě k dispozici a které ne.

2.3 Nalezení vhodné dolní vrstvy

Z cíle *C1* této práce nám už zbývá jen výběr audio enginu, který použijeme jako dolní vrstvu k našemu audio enginu. To nám následně umožní testovat a provozovat

horní vrstvu našeho audio enginu. Proto si probereme požadavky, které má dolní vrstva splňovat, a podle těchto požadavků vybereme vhodný audio engine.

První požadavky plynou z námi vybrané platformy, pro kterou budeme audio engine vyvíjet. Dolní vrstva audio enginu by měla fungovat v operačním systému Windows 7 a měla by jít používat s pomocí jazyka C#, přičemž by měla být volně šiřitelná, abychom mohli naší práci bezplatně šířit.

Dále si probereme funkční požadavky, jež by měla dolní vrstva audio enginu splňovat. Základním funkčním požadavkem, co by měla dolní vrstva audio enginu splňovat, je umět přehrávat zvuková data, na která se dají aplikovat efekty. Z efektů by to pak měly být základní efekty, jež jsou ve většině audio enginů dostupné, jsou jimi: *echo* – pro vytváření ozvěny, *reverb* – pro vytváření dozvuku, *low-pass filter* – pro oříznutí vysokých tónů, *high-pass filter* – pro oříznutí hlubokých tónů. High-pass filter není moc využívaným efektem, proto jeho neexistence není zásadní překážkou. Dále by dolní vrstva měla umět spojovat zvuková data z více stop do jedné stopy z důvodu zmenšení výpočetní náročnosti, kdy je u složeného zvuku nutné na spojená data aplikovat nějaký efekt. Také je potřeba umět z mono zvuku vytvářet vícekanálový zvuk, abychom byly schopni přehrávat zvuk prostorově z více reproduktorů. Poslední částí, co by měla dolní vrstva umět je umět vracet zpracovaná zvuková data zpět hře, aby mohla dále zpracovávat zvuková data, či mohla dostávat zvuková data z mikrofonu.

Je důležité si uvědomit, že i když dolní vrstva umí aplikovat efekty jako echo, reverb, low-pass filter a high-pass filter, k jejich nastavení nemusí používat ty stejné parametry, proto bude potřeba si vybrat, jaké parametry budeme používat pro nastavování konkrétních efektů.

Výše jsme si shrnuli požadavky, které máme na dolní vrstvu. Začali jsme hledat dolní vrstvy pro náš audio engine. V následující tabulce je uveden seznam audio enginů, které jsme zvažovaly pro výběr jako dolní vrstvu, přičemž jsme se zaměřili na audio enginy dostupné pro operační systém Windows.

Název audio enginu	**Dostupné na OS**	**Programovací jazyky**	**Licence**	**Vícekanálový zvuk**
irrKlang	Win/Linux	C++/C#	Placené	Ano
BASS	Win/Linux	C/C++/VB/.NET	Placené	Ano
NAudio	Win	.NET	Microsoft Public License	Ano
SDL	Win/Linux	C/C++/C#	GNU LGPL	Ne
OpenAL	Win/Linux	C	LGPL	Částečně
OpenAL soft	Win/Linux	C	LGPL	Ano
XAudio2	Win	C++	Součástí DirectX licence	Ano

Jedním z požadavků je, abychom mohli naší práci bezplatně šířit, z toho důvodu pro nás nejsou vhodné audio enginy irrKlang a BASS [12], neboť se jedná o placené audio enginy. Dalším nevhodným je SDL, kvůli tomu, že nesplňuje požadavek

pro tvorbu vícekanálového zvuku. Nevhodný je i NAudio [13] z důvodů, že nemá implementované efekty, které požadujeme. OpenAl, OpenAl soft [14] a XAudio2 nám splňují požadavky ohledně operačního systému, programovacího jazyka, licence a podporují vícekanálový zvuk, tudíž si tyto audio enginy probereme podrobněji.

2.3.1 OpenAL

Audio engine OpenAL spadá pod licenci LGPL, je dostupný v operačním systému Windows a Linux a je napsaný v jazyce C. O vývoj OpenAL se stará firma Creative. Od verze 1.1 jeho kód firma uzavřela a není dále šířen jako open source.

Dále bychom chtěli objasnit, že název OpenAL označuje jak hardwarové rozhraní pro řízení zvuku, tak i implementaci audio knihovny od firmy Creative.

S příchodem Windows Vista ale implementace audio knihovny firmy Creative měla problém, že neumožňovala přehrávat vícekanálový zvuk na více reproduktorech, ale přehrávala jej jen jako stereo zvuk. Byl to důsledek toho, že s nástupem operačního systému Windows Vista operační systém neobsahoval Hardware Abstraction Layer (HAL), takže OpenAL místo hardwarového mixování (s využitím HAL) používalo softwarové, které umí jen stereo výstup. Proto firma vydala Creative ALchemy, jež řešil změnu rozhraní v operačním systému.

Kromě výchozí implementace od firmy Creative ale existuje více variant implementací OpenAL rozhraní, příkladem je implementace OpenAL soft, jež pro mixování zvuku používalo jiné knihovny než implementace od Creative. Díky tomu OpenAL soft implementace neměla problém se změnou operačního systému. Abychom se vyhnuli potížím s operačním systémem, zkoumali jsme rozhraní s implementací OpenAL softu.

Různé implementace rozhraní OpenAL znamenají také podporu jiné sady efektů, jež umějí aplikovat. Implementace OpenAL soft konkrétně umožňuje aplikovat efekty: reverb, echo, low-pass filtr a ring modulátor. K požadavkům nám tedy chybí efekt high-pass filtr, ale jak jsme si už uvedli v požadavcích, tak by to nevadilo.

Naší práci chceme psát v jazyce C#, ale jelikož je OpenAL napsané v jazyce C, hledali jsme C# wrappery pro OpenAL rozhraní a nalezli tato řešení:

- *Tao* [15]
  - Jedná se o wrapper pro skupinu knihoven zahrnující OpenAL
  - Je dostupný pod MIT licencí
  - Zastaralý – nahrazen OpenTK
- *OpenTK* [16]
  - Jedná se o wrapper pro OpenGL, OpenCL a OpenAL
  - Je dostupný pod MIT licencí
  - Lze použít jak na OpenAL, tak OpenAL soft (mají stejné rozhraní)

Volba C# wrapperu, který budeme používat pro testování OpenAL byla jasná z důvodu, že jeden je nástupcem druhého. Vybrali jsme tedy novější OpenTK.

Při zkoumání OpenAL rozhraní jsme ale narazili na obtížnou práci s vícekanálovým zvukem. Nedařilo se nám vytvářet z mono zvuku vícekanálový zvuk (který jsme potřebovali pro simulaci prostorového zvuku), tudíž jsme načítali

zvuk na začátku jako vícekanálový, ale to vedlo k dalšímu problému a tím bylo, že jsme nemohli určovat hlasitost jednotlivých kanálů, museli jsme vytvořit vlastní efekt, který upravoval zvuková data jednotlivých kanálů.

OpenAL také neumožňuje spojovat zvuková data z více stop do jedné stopy, kvůli typu rozhraní, jež je tvořeno zdrojem zvukových dat a na ně aplikovaných efektů.

2.3.2 XAudio2

Audio engine XAudio2 je součástí DirectX, tudíž má licenci společnou s DirectX, je dostupný v operačním systému Windows a je napsaný v jazyce C++.

Jak už bylo zmíněno v kapitole *2.2 Mezivrstva*, rozhraní XAudio2 se skládá ze dvou hlavních objektů: zdroje zvukových dat a objektu pro další zpracování. To nám umožní dobře vytvářet stromovou strukturu složených zvuků pro zpracování zvukových dat.

XAudio2 dále obsahuje tyto efekty pro zpracování zvuku: echo, equalizer, masteringlimiter, reverb, low-pass filtr, high-pass filtr, band-pass filtr. Přičemž nám dále umožňuje vytvářet vícekanálový zvuk (umožňuje manipulaci se zvukovými stopami při předávání zvukových dat mezi objekty).

XAudio2 je napsané v jazyce C++, přičemž naší práci chceme psát v jazyce C#, proto se blíže podíváme na C# wrappery pro XAudio2:

- *SlimDX* [17]
  - Jedná se o wrapper pro DirectX knihovny
  - Je dostupný pod MIT licencí
  - Neúplná dokumentace – chybí popisy některých tříd a metod
  - Poslední verze je z roku 2011 – dále se neopravují chyby
- *SharpDX* [18]
  - Jedná o wrapper pro DirectX knihovny
  - Je dostupný pod MIT licencí
  - Neúplná dokumentace – chybí popisy některých tříd a metod
  - Stále aktualizovaná v době této práce

I přes horší dokumentaci jsme si vybrali SharpDX wrapper, neboť je stále aktualizovaný.

Při zkoumání knihovny XAudio2 ale nastal problém, když jsme chtěli vracet zpracovaná data zpět aplikaci, což XAudio2 neumožňuje, ale obešli jsme to tím, že jsme si napsali vlastní efekt a místo, abychom upravili zvuková data, tak data čteme a posíláme zpět aplikaci.

Kromě problému s vracením zpracovaných dat zpět aplikaci, který jsme vyřešili, tak XAudio2 splňuje všechny podmínky, které máme na dolní vrstvu.

2.3.3 Shrnutí – výběr XAudio2

Podrobně jsme se zaměřili na audio enginy OpenAL a XAudio2. Zkoumali jsme, jak se používají a srovnávali s požadavky, co máme na dolní vrstvu. XAudio2 splnil

všechny požadavky a OpenAL nepodporoval jen efekt high-pass filtr, což jsme probrali v požadavcích, že by nebylo překážkou, nicméně XAudio2 nabízí z pohledu autora práce lehčí práci s vícekanálovým zvukem a navíc je tvořen typem rozhraní, které umožňuje lépe tvořit složené zvuky (popsané v kapitole *2.2 Mezivrstva*), díky tomu si jako dolní vrstvu naší práce vybíráme audio engine XAudio2.

2.4 Rozdělení podvrstev audio enginu

V předchozích kapitolách jsme probrali strukturu mezivrstvy a vybrali audio engine, který budeme používat jako dolní vrstvu této práce, proto se v této kapitole zaměříme na to, jak bude strukturovaná horní vrstva této práce, přičemž si probereme spojení jednotlivých částí s cíly práce a problémy těchto částí.

Tato kapitola bude pojednávat o částech: programátorské rozhraní (API), zdroje zvuků, posluchači, vrstva správa zdrojů zvuků a posluchačů, vrstva načítání zvukových dat, vrstva zdroje, posluchači, vrstva výpočet šíření zvuku, vrstva prioritizace přehrávání. Zmíněné vrstvy jsou definovány v kapitole *1.2 Definice struktury audio enginu*.

2.4.1 Programátorské rozhraní (API)

Základní věcí, kterou potřebujeme je, aby náš audio engine mohl komunikovat s programem. K tomu nám poslouží programátorské rozhraní (API), jež bude hra používat pro ovládání našeho audio enginu. Nyní si probereme jednotlivé části, které by mělo toto rozhraní obsahovat.

První část rozhraní souvisí s cílem *C3* (*Vhodné čtení souborů, správa prostředí*), konkrétně s načítáním souborů. Potřebujeme totiž umět nějakým způsobem načítat zvuky, tudíž první částí rozhraní je načítání zvukových prvků, které se mohou vyskytovat ve hře, konkrétně se jedná o zdroje zvuku a posluchače. Tyto prvky se ale většinou načítají při načítání herní úrovně hry, hodilo by se tedy mít seznamy těchto prvků a načítat přímo tyto seznamy. Neboť mohou být herní úrovně hry rozsáhlé, tak bychom mohli tyto seznamy prvků načítat i v průběhu hry, až když se hráč přesune do další části virtuálního prostředí. Kvůli tomu, že se hráč přesune do další části herní úrovně, tak původní zvukové prvky nemusí být potřeba, proto by se hodilo umět tyto prvky v průběhu hry i odnačítat, aby zbytečně nezaplňovaly paměť.

Další část rozhraní opět souvisí s cílem *C3* (*Vhodné čtení souborů, správa prostředí*), ale tentokrát se bude jednat o správu virtuálního prostředí. Je tedy potřeba umět toto virtuální prostředí načítat, proto další částí programátorského rozhraní je definování virtuálního prostředí. Opět, jako u zvukových prvků, by se zde hodilo načítání (odnačítání) virtuálního prostředí po částech, neboť levely hry mohou být rozsáhlé, ale není potřeba mít definované virtuální prostředí daleko od hráče, které by se pro výpočet šíření zvuků nepotřebovalo.

Kvůli cíli *C2* (*Správa zdrojů zvuků a posluchačů ve virtuálním prostředí*) je také potřeba mít informace o jednotlivých zdrojích zvuků a posluchačích ve virtuálním prostředí. Proto další částí programátorského rozhraní je definování zdrojů zvuků a posluchačů ve virtuálním prostředí, jež by obsahovaly informace o jejich umístění a v případě zdrojů zvuků i informace o jejich přehrávání.

Aby mohla aplikace získávat již zpracovaná zvuková data, případně data z mikrofonu, mělo by programátorské rozhraní obsahovat i callback, jenž by

průběžně předával daná zvuková data aplikaci s identifikací, o která zvuková data se jedná.

Hry mohou také vyžadovat různé nastavení audio enginu (např. nechce aplikovat nějaký druh efektů, případně chce hra dostávat zvuková data z mikrofonu), proto součástí programátorského rozhraní by měla být možnost nastavování audio enginu. Přičemž aby se mohl audio engine správně nastavit, tak je i důležité rozhraní, jež nám zajistí informace o dostupnosti jednotlivých částí (např. dostupnost echo efektu) audio enginu na dané platformě.

2.4.2 Zdroje zvuků

Zdroje zvuků jsou důsledkem cíle *C4* (*Možnost vytváření složených zvuků*), neboť potřebujeme umět tyto složené zvuky nějak interpretovat a cíle *C2* (*Správa zdrojů zvuků a posluchačů ve virtuálním prostředí*), kde budeme zdroje zvuku využívat. V audio enginu tedy chceme podporovat přehrávání složených zvuků *S1*-*S4* zmíněné v kapitole *1.1.1 Vývoj zvuku ve hrách*. Samotné zdroje zvuků nám poté definují, z jakých zvukových souborů a efektů se zvuky skládají a umožní nám přehrávat zvuková data.

Jak ale definovat strukturu složeného zvuku, jež se skládá z několika zdrojů zvukových dat a efektů? Zde se jedná o podobný problém jako v kapitole *2.2 Mezivrstva*, kde jsme řešili rozhraní dolních vrstev audio enginů. Opět nejvhodnějším způsobem bude nejobecnější řešení, kde budeme mít dva typy objektů: zdroje zvukových dat a objekt pro další zpracování. Zdroje zvukových dat budou dostávat zvuková data ze zvukových souborů, nebo z aplikace a budou je dál posílat buď na objekt pro další zpracování k aplikování efektů, případně na výstup. Objekty pro další zpracování budou umožňovat přijímat zvuková data od zdrojů zvukových dat případně jiných objektů pro další zpracování a po aplikování efektu na zvuková data pošlou zvuková data dále.

Strukturu složeného zvuku už máme definovanou, nyní ale potřebujeme umět řídit přehrávání zdrojů zvukových dat. Složené zvuky se totiž mohou skládat z více zdrojů zvukových dat, přičemž většinou nechceme přehrávat všechny najednou, ale aby se přehrávaly v určitý čas přehrávání složeného zvuku. Jak toho ale můžeme docílit?

1. První řešení je, že si zdroj zvuku zapamatuje herní čas, kdy došlo ke spuštění přehrávání složeného zvuku a poté pokaždé když herní čas dosáhne hodnoty, kdy se má spustit nějaký zdroj zvukových dat složeného zvuku, tak tento zdroj zvukových dat spustí a poté v určitém čase opět zastaví.
2. Druhým řešením může být vytvoření struktury tzv. časové osy, jež bude součástí zdroje zvuku. Časová osa by obsahovala čas přehrávání složeného zvuku, nezávislého na herním čase, přičemž zdroje zvukových dat by měli pevně stanovený čas, kdy se mají přehrávat a časová osa by se následně starala o jejich spouštění a zastavování.

Z těchto dvou řešení vybereme to druhé, neboť u prvního způsobu kvůli závislosti na herním čase by šlo obtížně vytvářet dopplerův efekt (zpomalování, zrychlování přehrávání), případně další efekty.

Samotná časová osa nám ale nemusí stačit, neboť když si vezmeme jako složený zvuk zvuk nějakého výrobního stroje, tak ten vytváří stále dokola stejný zvuk, dokud jednou neskončí, přičemž když bude končit tak se nejdříve dohraje poslední smyčka

zvuku. To je problém, neboť čas přehrávání zdrojů zvukových dat pomocí časové osy musíme přesně určit. Jenže počet opakování zvuků může být nekonečný, takže nemá smysl zkopírovat přehrávání stejných zdrojů zvukových dat po časové ose. Řešení se zdá být jasné, upravíme časovou osu tak, aby podporovala cykly, tudíž když dojde do určitého času, tak se čas přetočí zpět a budeme přehrávat zdroje zvukových dat opět od začátku, přičemž budeme mít možnost tento cyklus ukončit. Lepší ale bude použít místo cyklů obecně skoky, jež umožní skákat nejen zpět po časové ose, ale i dopředu, abychom mohli v určitém případě nějakou část přeskočit. Můžeme ale skočit i uprostřed přehrávání nějakého zdroje zvukových dat, tak je také potřeba, aby bylo možné nastavit, jak dlouho po skoku mají doznívat původní data a přitom nabíhat nová zvuková data, neboli aby mohl vzniknout plynulý přechod.

Jako další příklad složeného zvuku si vezmeme zvuk motoru auta, nejdříve se přehrává zvuk startování motoru, poté se přehrává zvuk běhu motoru a přehrávání končí zvukem zastavením motoru, přičemž zvuk běhu motoru může běžet libovolnou dobu. Tento příklad se zdá být podobný předchozímu příkladu s výrobním strojem, takže bychom mohli uvažovat o použití cyklů/skoků. Zvuk běhu motoru se ale může skládat z několika zdrojů zvukových dat, jejichž zvuková data nemusejí mít stejnou délku, přičemž zvuková data bývají připravena pro přehrávání v cyklu. Nemůžeme tudíž použít skoky, neboť bychom nepřehrávali (nevyužívali) všechna zvuková data, která máme k dispozici. Potřebujeme, aby se zvuková data dala přehrávat v cyklu nezávisle na čase časové osy. Pro řešení tohoto problému nás napadají dvě řešení:

1. První řešení spočívá v tom, že bychom mohli zastavit čas časové osy a přitom bychom nechaly hrát v cyklu ty zdroje zvuku, jež se mají v daný čas přehrávat, a poté na nějaký pokyn opět spustíme čas časové osy. Kvůli tomuto řešení nám ale čas časové osy nebude udávat celkový čas přehrávání složeného zvuku, proto bychom museli celkový čas přehrávání definovat zvlášť.
2. Druhá možnost je, že v určitý moment řekneme, že bychom chtěli přehrávat jen zvuky, které se nyní přehrávají. S tím, jak poběží čas časové osy, budeme tedy posunovat spouštění a zastavování dalších zvukových dat. To ale přináší nevýhodu v tom, že bychom museli stále přenastavovat časy spouštění a zastavování zvukových dat a při skocích bychom museli opět přenastavit časy.

Z těchto dvou řešení si vybereme to první, neboť je výpočetně méně náročné a nemění se časy přehrávání zdrojů zvukových dat, přičemž časům zastavení časové osy budeme říkat *cue body časové osy*.

Další věc, kterou potřebujeme u složených zvuků umět je jejich ovlivňování při přehrávání, příkladem je opět zvuk motoru auta, kdy při jeho běhu chceme měnit jeho otáčky. Toho docílíme tak, že budeme měnit rychlost přehrávání a hlasitost jednotlivých zdrojů zvukových dat běhu motoru. K tomu nás napadá jediné vhodné řešení a to nějaká struktura zdroje zvuku, jíž se předá název a hodnota toho, co chceme ovlivnit. Struktura následně podle hodnoty vypočte parametry zdrojů zvukových dat a efektů, jež jsou k tomuto názvu přiřazeni. Bude se v podstatě jednat o nastavování parametrů složeného zvuku, přičemž každý složený zvuk může mít svoje vlastní parametry.

Výše jsme si popisovali časovou osu a její prvky, nyní potřebujeme tyto prvky umět ovládat, jako aktivovat/deaktivovat skoky a umět znovu spustit cue body, případně umět okamžitě skočit po časové ose. K tomu bychom mohli využít

parametry složeného zvuku. Jenže ty se starají o nastavování parametrů prvků, z nichž se zvuk skládá, zatímco nyní potřebujeme upravovat přehrávání časové osy, hodilo by se tedy využít jinou strukturu, aby nás to nemátlo. Struktura bude podobná parametrům, ale místo ovlivňování prvků zvuku bude moci ovlivňovat časovou osu. Protože se bude jednat o jednorázové změny, tak tuto strukturu nazveme *událostmi*.

Poslední věcí, které chceme u složených zvuků je umět přehrávat i zdroje zvukových dat nezávisle na časové ose, příkladem jsou zvuky kroků, kdy potřebujeme jednorázově přehrávat zvuky kroků, přičemž chceme, aby se při každém přehrávání přehrál jiný zvuk kroku. Neboť se bude jednat o jednorázové změny zvuku, tak bychom k tomu mohli použít strukturu událostí, které bychom přidali události typu spusť/zastav zdroj zvukových dat nezávislých na časové ose.

Složené zvuky ale nejsou vždy potřeba, jsou totiž zvuky, jež se přehrávají jednorázově a nejdou zásadně ovlivňovat, proto by bylo vhodné vytvořit i odlehčenou verzi zdrojů zvuků, jež by byla tvořena jen jedním zdrojem zvukových dat s výchozí sadou efektů a parametry, jež by umožnily ovlivňovat parametry zdroje zvukových dat a efektů.

2.4.3 Posluchači

Pro splnění cíle *C2* (*Správa zdrojů zvuků a posluchačů ve virtuálním prostředí*) této práce je potřeba si definovat posluchače a to, co by všechno měli posluchači obsahovat. V základě se vlastně bude jednat o body ve virtuálním prostředí, do kterých budou směřovat všechny zvuky ze zdrojů zvuků ve virtuálním prostředí, a které budou posílat zvuková data na výstup.

Co bychom ale měli očekávat od zvukového výstupu? Měl by umět přehrávat data tak, že přicházejí z určitého směru. Pro přehrávání zvuku z určitého směru bude zapotřebí umět propojit posluchače a zdroj zvuku, abychom mohli vypočítat směr zvuku, přičemž posluchač bude získávat zpracovaná zvuková data ze zdroje zvuku a na ty následně aplikuje efekt pro vytvoření vícekanálového směrového zvuku, přičemž poté pošle zpracovaná data zvukové kartě.

Posluchač by nám tedy měl sloužit pro výstup dat, toho bychom mohli také využít k posílání zpracovaných zvukových dat zpět aplikaci. Tato zvuková data ale budou stačit v mono formátu, neboť se tyto zvuková data většinou využívají jako jiný zdroj zvukových dat (důvod bude popsán v kapitole *2.4.7 Vrstva výpočet šíření zvuku*), takže budeme posílat zvuková data ještě před aplikací efektu pro vytvoření vícekanálového zvuku. Budeme ale potřebovat sloučit všechna zvuková data všech zvuků, co přichází k tomuto posluchači, tudíž budeme potřebovat nejdříve spojit všechna data všech zvuků tohoto posluchače a až poté je poslat zpět aplikaci (pro výstup na zvukovou kartu to není potřeba, ta se o spojení zvuků postará sama).

Posluchač nám tedy bude sloužit pro výstup zvuku na zvukovou kartu a posílání zpracovaných zvukových dat zpět aplikaci.

2.4.4 Vrstva správa zdrojů zvuků a posluchačů

Vrstva správa zdrojů zvuků a posluchačů je implementací části načítání zvukových prvků programátorského rozhraní, souvisí tedy s cílem *C3* (*Vhodné čtení souborů*) této práce.

Jak bylo zmíněno v kapitole *2.4.1 Programátorské rozhraní*, tato vrstva by se měla starat o načítání/odnačítání seznamu zdrojů zvuků a posluchačů. Tyto seznamy zvukových prvků ale mohou obsahovat stejné prvky, neboť seznamy mohou být přiřazeny určité části virtuálního prostředí a stejný prvek se může vyskytovat v několika částech virtuálního prostředí, tudíž je potřeba zajistit, aby se stejné prvky nenačítaly dvakrát. Ten samý problém je i při odnačítání seznamu prvků, neboť když máme načtené dva seznamy zvukových prvků a oba obsahují stejný prvek, tak při odnačtení jednoho seznamu tento prvek musí zůstat načtený. Vhodným řešením tohoto problému je u každého načteného prvku si udržovat počet, kolikrát byl prvek načten a podle toho řídit načítání/odnačítání jednotlivých zvukových prvků.

Pak je tu otázka ohledně toho, jak by se měli tyto seznamy načítat, zda by k jejich načtení měla dát pokyn samotná hra (budeme tomu říkat manuální načítání), případně zda by je měl umět audio engine načítat automaticky (automatické načítání). U manuálního načítání řídí načítání zvukových prvků samotná hra, ta přitom má informace, kdy bude jaké zvuky potřebovat, takže by to pro hru neměl být problém. Ale díky automatickému načítání by hře stačilo načíst počáteční posluchače ve virtuálním prostředí a o zbytek načítání by se staral audio engine.

K automatickému načítání bychom ale potřebovali strukturu, jež by kontrolovala, jestli nejsou potřeba načíst nějaké zvuky a poté by je díky tomu načetla. To nás přivádí ke struktuře, jak bychom měli reprezentovat zvukové prvky, které jsou potřeba načíst. Buď bychom mohli mít seznamy prvků, jež by byly přiřazeny určité části virtuálního prostředí, pokud by se některý posluchač přiblížil této části, tak by audio engine tento seznam načetl a naopak, pokud by se posluchač od nějaké části vzdálil, tak by se seznam odnačetl. Nebo by mohli být zvukové prvky uloženy ve struktuře octree a vždy když by se posluchač přiblížil k jiné části tak by se načetly všechny prvky v dané části octree. Protože ale cílem práce nebude šíření zvuku prostorem, tak tuto část nebudeme moci vhodně používat, z důvodu, že nebudeme mít informace o pozicích zdrojů zvuků a posluchačů ve virtuálním prostředí, proto jako řešení této práce použijeme manuální načítání.

2.4.5 Vrstva načítání zvukových dat

V naší práci potřebujeme také vyřešit vhodné načítání zvukových souborů, jež je cílem *C3* (*Vhodné čtení souborů*) této práce. K tomu by měla sloužit vrstva načítání zvukových dat, jež je využívána vrstvou správa zdrojů zvuků a posluchačů.

Jedná se o to, jakým způsobem budeme načítat zvuková data ze zvukových souborů. Často používané zvukové soubory je nejlepší načíst celé do paměti, protože zdroje zvuku mohou požadovat zvuková data z různého času zvukového souboru a čtení dat z disku by zabíralo čas. Ale pak jsou zvukové soubory, které jsou využívány jen pro jeden konkrétní zdroj zvukových dat (například hudební soubory), přičemž tyto zvukové soubory bývají velké. Kvůli tomu, že jsou tyto zvuky využívány jako zdroj jednoho zdroje zvukových dat, tak není potřeba načítat tyto zvukové soubory celé do paměti, ale stačí nám po částech, neboť z času přehrávání zvukového souboru můžeme odvodit, kdy budou potřeba další zvuková data a včas načíst další část zvukových dat do paměti, abychom zbytečně nezabírali paměť nepotřebnými zvukovými daty.

Zvukové soubory ale nejsou jediným zdrojem zvukových dat, dalším zdrojem mohou být zvukové buffery, ať už ty, které vytváří náš audio engine tím, že posílá zpracovaná zvuková data zpět aplikaci, tak aplikace sama může dodávat zvukové

buffery. To nám přináší další problém a to, jak tyto buffery přehrávat, neboť se jedná o nekonečný zdroj zvukových dat, který se stále zvětšuje o nová data. Neboť ale nemáme nekonečnou velikost paměti, tak tyto zvukové buffery budeme muset časově omezit. Přičemž buffery nám přináší další problém a to při simulaci Dopplerova efektu, neboť nemůžeme data z těchto bufferů stále přehrávat zrychleně, protože by nám mohli data rychle dojít, proto přehrávání zvuku z bufferů bude mít omezení pro zrychlené (zpomalené) přehrávání.

2.4.6 Vrstva zdroje, posluchači

V předchozích podkapitolách jsme si definovali, co jsou a k čemu slouží zdroje zvuků a posluchači, čehož využijeme v této vrstvě, jež řeší cíl *C2* (*správa zdrojů zvuků a posluchačů ve virtuálním prostředí*) této práce a je implementací části definování zdrojů zvuků a posluchačů ve virtuálním prostředí programátorského rozhraní. Tato vrstva by se tedy měla starat o umístění jednotlivých zdrojů zvuků a posluchačů ve virtuálním prostředí a následně z nich vytvářet zvuky.

Jak ale reprezentovat umístění zdrojů zvuků a posluchačů ve virtuálním prostředí? Buď bychom mohli ke každému typu zdroje zvuku a posluchači udržovat seznam zdrojů zvuků a posluchačů umístěných ve virtuálním prostředí. Nebo můžeme mít octree strukturu, v níž budou umístěné jednotlivé zdroje zvuků a posluchači podle umístění ve virtuálním prostředí. Přičemž umístění zdrojů zvuků a posluchačů v octree struktuře by nebylo neměnné, ale měnilo by se v závislosti na jejich pohybu ve virtuálním prostředí. K tomu jaká reprezentace bude lepší, ale potřebujeme vědět, jak se ze všech těch zdrojů zvuků a posluchačů vytvářejí samotné zvuky.

Máme dva směry pro vytváření zvuků, od zdrojů zvuků k posluchačům, nebo od posluchačů ke zdrojům zvuků. Měli bychom začít od posluchačů, neboť jejich pozice ovlivňuje, které zdroje zvuku uslyší (jsou v dosahu) posluchač. Tudíž k posluchači vybereme zdroje zvuků, které jsou v určitém dosahu (mohli by být slyšet) a spojíme instanci zdroje zvuku s instancí posluchače a toto spojení nám vytvoří zvuk. Z toho nám plyne, že lepší reprezentací umístění zdrojů zvuků a posluchačů bude použití octree struktury, abychom nemuseli procházet všechny kombinace zdroje zvuku a posluchače a zjišťovali jejich vzdálenost.

Další otázkou ale je, jak je to s instancemi zdrojů zvuků a posluchačů? Pro každé spojení zdroje zvuku s posluchačem bude zapotřebí zvláštní instance zdroje zvuku, neboť v závislosti na poloze a rychlosti zdroje zvuku k posluchači je zapotřebí přehrávat zvuk zdroje zvuku s určitými parametry, jež jsou pro každé spojení jiné. To samé platí i o instancích posluchačů, neboť při každém spojení zdroje zvuku a posluchače jde zvuk k posluchači z jiného směru, tudíž je potřeba mu nastavit odpovídající parametry pro konkrétní spojení.

Otázku, co za informace by měla tato vrstva obsahovat o umístění zdrojů zvuků a posluchačů si zodpovíme v další kapitole, neboť tam se dozvíme, které informace jsou potřeba pro výpočet šíření zvuku virtuálním prostředím.

2.4.7 Vrstva výpočet šíření zvuku

Dalším cílem této práce je cíl *C6* (*Výpočet šíření zvuku prostorem*), o to by se nám měla postarat vrstva výpočet šíření zvuku. Tato vrstva by nám měla určit efekty, jež by se měli aplikovat na každý zvuk určený propojením zdroje zvuku a posluchače

pomocí vrstvy *zdroj, posluchači*. A dále by se měla sloužit načítání/odnačítání virtuálního prostředí.

Nejdříve si řekneme o načítání/odnačítání virtuálního prostředí. Jak bylo zmíněno v kapitole *2.4.1 Programátorské rozhraní (API)*, tato vrstva by měla umět načítat virtuální prostředí po částech, neboť k šíření zvuku virtuálním prostředím nepotřebujeme mít definované virtuální prostředí, které je daleko od posluchače. K tomu náleží otázka ohledně manuálního a automatického načítání virtuálního prostředí podobně jako tomu bylo u načítání zdrojů zvuků a posluchačů v kapitole *2.4.4 Vrstva správa zdrojů zvuků a posluchačů*. Zde se bude hodit automatické načítání, kromě načítání dynamických objektů virtuálního prostředí, o jejichž nastavování se musí starat samotná hra.

To nás ale přivádí k otázce, jak budeme reprezentovat virtuální prostředí. Kvůli tomu, že virtuální prostředí se velmi podobá grafu, tak k jeho reprezentaci použijeme grafovou strukturu, přičemž uzly budou místnost (prostory) a hrany budou reprezentovat, jak jsou místnosti propojené, ať už pomocí dveří, oken, či zdí. Uzly místností by pak měly obsahovat informace, jež jsou důležité pro výpočet efektů, které se mají aplikovat na zvuk.

Co za informace k šíření zvuku ale budeme potřebovat od zdrojů zvuků a posluchačů? Základní informací by měla být jejich pozice ve virtuálním prostředí, dále by to měl být směr a nakonec i rychlost pohybu (pro simulaci Dopplerova efektu). Zdroje zvuků by navíc měli mít informace o tom, zda se přehrávají, jsou pozastaveny, případně zastaveny, abychom věděli, zda se musí počítat šíření zvuku virtuálním prostorem.

Další část je formát zvukových dat, jenž je vhodný pro aplikaci efektů, jež budou zajišťovat šíření zvuku virtuálním prostorem. Neboť cesta zvuku ve virtuálním prostředí bude reprezentována přímkami, a až jako poslední se aplikuje efekt pro vytvoření směru, odkud zvuk pochází, tak jako vhodný je jednokanálový (mono) formát.

Dále bude potřeba vybrat vhodný algoritmus, který bychom měli použít pro šíření zvuku virtuálním prostředím. Neboť virtuální prostředí bude reprezentováno grafovou strukturou, tak se nám zde bude hodit některý z grafových algoritmů pro hledání nejkratší cesty (zde myšleno cesty, podle níž bude zvuk nejhlasitější), přičemž jsme uvažovali nad algoritmem *A**, jež se též využívá pro výpočet pohybu jednotek ve 3D prostředí. Ale z důvodů, že jsme se rozhodli vynechat výpočet 3D zvuku, tak jsme tuto část dále neřešili.

2.4.8 Prioritizace zvuků

Na vrstvu výpočet šíření zvuků nám navazuje vrstva prioritizace zvuků, jež plní cíl *C7* (*Prioritizace přehrávání zvuků*) této práce. Jedná se o to, že hry mohou obsahovat velké množství zvuků, jež se vyskytují ve virtuálním prostředí a i přes dnešní výkon počítačů by nemusel stačit na jejich zpracování. Je tedy potřeba počet současně hrajících zvuků nějakým způsobem omezovat.

Vrstva Prioritizace zvuků by se tedy měla starat o omezování počtu současně hrajících zvuků. To lze udělat jediným způsobem a to tak, že audio engine zastaví přehrávání některých zvuků, dokud neskončí čas jejich přehrávání nebo se neuvolní výkon k jejich přehrávání. To vede k otázce, podle čeho máme určit, který zvuk zastavit a který nechat přehrávat. V každé hře mohou být důležitější jiné zvuky, je

tedy potřeba u každého zdroje zvuku udávat parametr, který určí prioritu přehrávání zvuku. Dále ale záleží na množství a typů efektů, neboť některé jsou méně či více náročné a pak ještě záleží na vzdálenosti zdroje zvuku a posluchače, neboli na tom, jak hlasitý bude výsledný zvuk, neboť nechceme přehrávat zdroj zvuku s vysokou prioritou, který je daleko od posluchače a není skoro slyšet, zatímco zdroj zvuku s nižší prioritou blízko posluchače by byl zastaven.

Díky tomu, že prioritizace zvuků spouští a zastavuje zvuk podle určitých parametrů, mohlo by se v určité situaci stát, že by se jeden zvuk dokola zastavoval a opětovně spouštěl, což by ve výsledku znělo špatně. Proto by se do parametrů, které určují, zda se zvuk bude přehrávat, mělo zaznamenávat, jak dlouho je zvuk spuštěný či zastavený, aby k této situaci nedocházelo.

2.5 Editor zvuků

Posledním cílem, který jsme ještě neřešili je cíl *C5* (*Editor pro vytváření složených zvuků*) této práce. Neboť bychom chtěli, aby se nám o definování zvuků a prostředí pro šíření zvuku starali designéři, přičemž designéři většinou nejsou programátoři, tak je potřeba vytvořit grafický editor, jež nám umožní definovat virtuální prostředí pro šíření zvuku spolu se zdroji zvuků a posluchačů.

Pro přehlednost editor dělíme na dvě základní části, na část pro vytváření zdrojů zvuků a posluchačů a na část pro definování virtuálního prostředí pro šíření zvuku.

2.5.1 Část pro definování virtuálního prostředí pro šíření zvuku

Část pro definování virtuálního prostředí pro šíření zvuku lze udělat dvěma způsoby. První je, že bude součástí editoru pro vytváření zdrojů zvuků a posluchačů, případně pro tuto část vytvoříme vlastní editor. Druhý způsob je ten, že použijeme některý z existujících editorů pro grafickou tvorbu virtuálního prostředí, ke kterému napíšeme plugin, jež by nám umožnil v daném editoru definovat virtuální prostředí pro šíření zvuku. Nakonec ale nebude šíření zvuku ve virtuálním prostředí součástí této práce, proto jsme se dál touto částí editoru nezabývali.

2.5.2 Část pro vytváření zdrojů zvuků a posluchačů

Tato část editoru by se měla starat o definování zdrojů zvuků a posluchačů, jež budeme moci využívat pro přehrávání pomocí našeho audio enginu. Dále si probereme, jaké části by měl tento editor obsahovat.

Rozdělení částí editoru pro vytváření zdrojů zvuků a posluchačů

Abychom mohli vytvářet zdroje zvuků a posluchače ke konkrétním hrám, tak by měl editor obsahovat část jako *správa projektu*. Dále pro možnost editování složených zvuků by měl obsahovat části jako *editor sestavení zvuku*, *editor grafů* a *editor časových os*.

Část nazvaná jako *správa projektu* by měla obsahovat seznam, ve kterém budou členitě zobrazeny skupiny zdrojů zvuků a posluchači společně s jejich parametry a událostmi, přičemž zdroje zvuků a posluchači se mohou vyskytovat ve více skupinách. Zároveň by tato část mohla obsahovat editaci parametrů a událostí jednotlivých zdrojů zvuku.

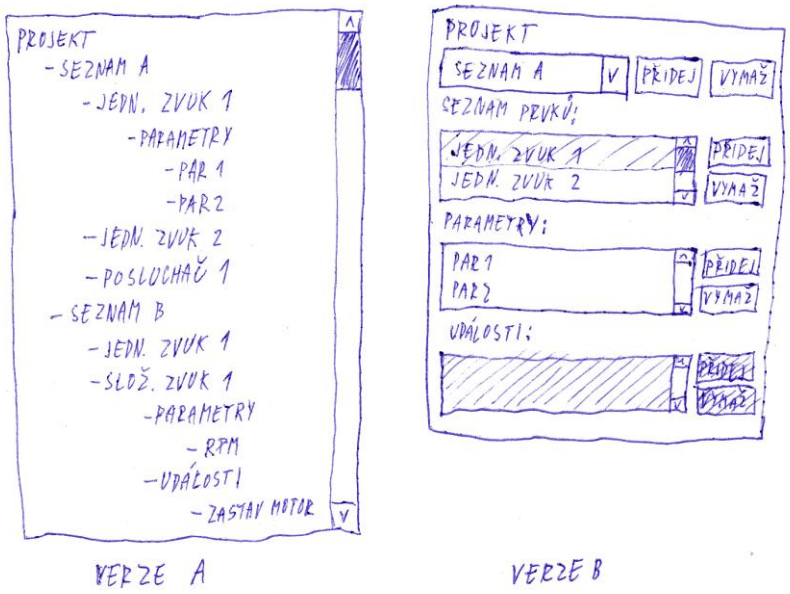
Další částí je *editor sestavení zvuku*, jenž by měl sloužit k sestavování průběhu zpracovávání zvukových dat. Měl by nám umožnit zvuk skládat ze zdrojů zvukových dat, slučování zvukových dat a efektů, jež bude končit výstupem.

Editor grafů by měl sloužit k výpočtu hodnot parametrů efektů či zdrojů zvukových dat z parametrů zvuku. Graf bude možné tvořit několika druhy čar (např. konstantní, lineární, atd.).

Poslední částí editoru je *editor časové osy*, ten bude určen pro editaci řízení přehrávání zdrojů zvukových dat. Zde by mělo být možné přidávat do časové osy zdroje zvukových dat a dále přidávat speciální prvky jako skoky a cue body.

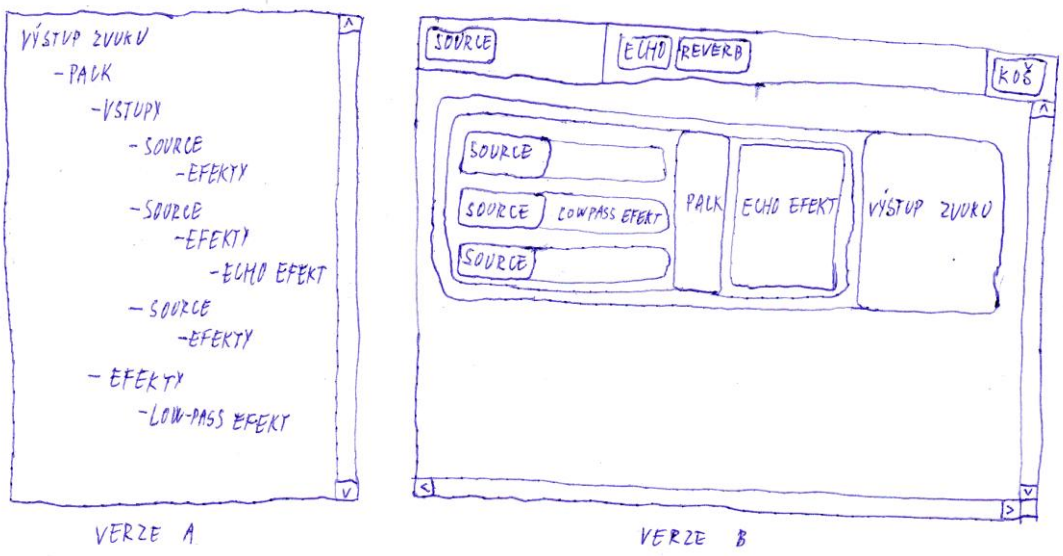
Návrh uživatelského rozhraní

V této části si probereme několik návrhů grafického rozhraní editoru, jak by si autor této práce představoval, že by mohlo vypadat.



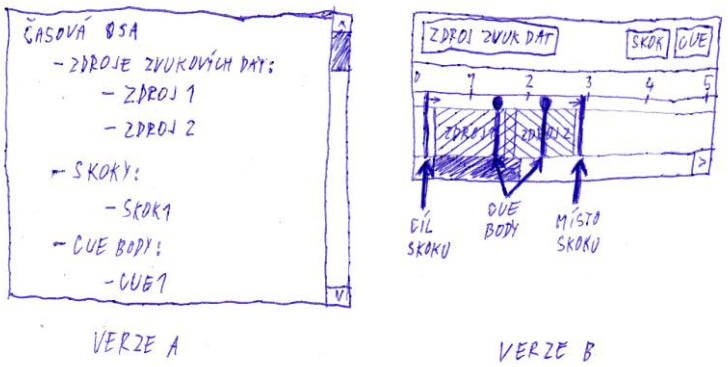
Obrázek 2.5 - Správa projektu

Pro část editoru *správa projektu* autor této práce navrhl dvě verze, jež jsou vyobrazeny na obrázku *2.5*. *Verze A* je vyobrazena s použitím odstupňovaného seznamu, kde jsou zobrazeny seznamy zvukových prvků a ve zvukových prvcích jsou další části jako parametry a události. *Verze B* je tvořena rozbalovacím seznamem seznamů zvukových prvků, přičemž jednotlivé zvukové prvky a následně jejich parametry či události jsou zobrazeny v listboxech.



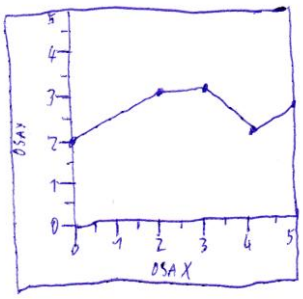
Obrázek 2.6 - Editor složeného zvuku

Další částí je *editor složeného zvuku*. Autor opět navrhl dvě verze zobrazení vyobrazené na obrázku 2.6. *Verze A* je odstupňovaný seznam, jenž tvoří stromovou strukturu složeného zvuku, přičemž zvukový výstup je kořenem tohoto seznamu. *Verze B* je grafické znázornění stromové struktury složeného zvuku, přičemž data v tomto grafickém stromě plynou zleva doprava, takže je to mnohem přirozenější. Přičemž přidávání jednotlivých zdrojů zvukových dat, či efektů se bude dělat přetažením požadovaného prvku do stromové struktury.



Obrázek 2.7 - Editor časové osy

Následuje část *časové osy*, jejichž grafické verze jsou znázorněny na obrázku *2.7*. *Verze A* je opět odstupňovaný seznam, kde jsou uvedeny jednotlivé zdroje zvukových dat přiřazené k časové ose a poté i seznamy skoků a cue bodů časové osy. *Verze B* je znázorněna graficky osou, na kterou můžeme přetažením přidávat jednotlivé zdroje zvukových dat, přičemž dobu a délku spuštění zdroje zvukových dat nastavíme umístěním na určitou pozici časové osy, přičemž dále umožňuje i přetažením přidávat skoky a cue body.



Obrázek 2.8 - Editor grafu

Poslední částí je *editor grafů*, přičemž autor této práce navrhnul jen jeden způsob (vyobrazený na obrázku *2.8*) a to, že osy grafu jsou umístěny na levém a dolním okraji grafu. Graf se bude skládat z minimálního počtu dvou bodů, přičemž nové body by mělo jít přidávat dvojklikem na již existující přímku grafu a pozice jednotlivých bodů by měla jít nastavovat jejich přetažením.

3 Zúžení rozsahu práce

Jak jsme si v úvodu předchozí kapitoly řekli, tak z výčtu jednotlivých částí této práce usuzujeme, že je potřeba zúžit cíle této práce, konkrétně některé cíle této práce vynecháme. V této kapitole tedy vybereme ucelenou část horní vrstvy a probereme úpravy některých částí, jež budeme muset upravit.

3.1 Výběr ucelené podčásti

Problémem nyní je, jak vybereme ucelenou podčást horní vrstvy, kterou implementujeme jako cíl této práce. Chceme, aby se jednalo o funkční celek, tudíž při sestavování ucelené podčásti nebudeme postupovat výběrem cílů, ale budeme postupovat od dolních podvrstev horní vrstvy a z toho pak vybereme cíle této práce.

Jako základ ucelené podčásti nám poslouží mezivrstva, jež bude určena k testování a provozování části horní vrstvy. Abychom mohli pomocí našeho audio enginu přehrávat zvuky, tak je potřeba mít zdroje zvuků a posluchače, jež nám budou vytvářet zvuky. Další částí díky tomu je vrstva pro načítání zdrojů zvuků a posluchačů, abychom je mohli snadno načítat do našeho audio enginu a nemuseli je definovat v kódu aplikace, přičemž na tuto část navazuje vrstva načítání zvukových souborů, bez které bychom nemohli přehrávat zvuková data ze zvukových souborů. K definování zdrojů zvuků a posluchačů budeme potřebovat také zachovat editor jako součást práce. Tyto části nám budou dohromady tvořit ucelenou podčást horní vrstvy. Vynecháme tedy vrstvu zdroje zvuků, posluchači, dále vrstvu šíření zvuku prostorem a vrstvu prioritizace zvuků, přičemž nemá smysl vybírat jen vrstvu zdroje zvuků, posluchači, neboť ta vyžaduje i zbylé dvě vrstvy, a kdybychom vybrali vrstvy pro šíření zvuku prostorem a prioritizaci zvuků, tak bychom rozsah této práce moc nezúžili. Navíc tyto části nemá smysl vyměňovat za jiné, neboť bez vybraných částí by nebyl audio engine jako celek dobře použitelný.

Zůstává nám tedy mezivrstva, která slouží k testování a provozování naší části horní vrstvy. Je navíc důležitou částí práce, jež nám bude zajišťovat přenositelnost na jiné platformy při používání jiných audio enginu jako dolní vrstvy této práce.

Další vybranou částí jsou zdroje zvuků, jež se skládají z jednoduchých a složených zvuků, přičemž hlavně podpora složených zvuků nám zaručí velkou zvukovou rozmanitost pro přehrávání zvuků ve hrách. Dále jsme vybrali posluchače, kteří nám budou sloužit k nastavování směru zvuku, odkud má zvuk přicházet. Kvůli výběru ucelené části bez části šíření zvuku prostorem, ale bude aplikace muset pro vytváření zvuků sama propojovat zdroje zvuku a posluchače a v případě 3D her bude muset aplikace sama zajišťovat výpočet šíření zvuku prostředím a pomocí přidávání efektů ke zvukům simulovat zvuk ve 3D prostředí.

Další částí je načítání zdrojů zvuků a posluchačů. Tato část nám slouží k tomu, abychom nemuseli načítat do audio enginu všechny zvuky, ale jen ty, které momentálně potřebujeme (např. zvuky pro konkrétní level, zvuky určité části virtuálního prostředí, atd.). Na to navazující část je načítání zvukových dat, jež nám zajistí načítání zvukových dat ze zvukových souborů.

Součástí práce nám též zůstane editor, jenž bude sloužit k definování zdrojů zvuků a posluchačů. Ale díky tomu, že nám odpadá část pro výpočet šíření zvuku

prostředím, tak nám odpadá i část editoru, jež by měla být zaměřena na definování virtuálního prostředí pro šíření zvuku.

3.2 Přepracované programátorského rozhraní

Díky zúžení rozsahu této práce se nám též mění programátorské rozhraní našeho audio enginu. Zachová se nám část pro načítání definic zdrojů zvuků a posluchačů, jež budeme moci využívat k produkování zvuku naším audio engine. Odpadají nám ale část pro definování virtuálního prostředí a část pro definování zdrojů zvuků a posluchačů ve virtuálním prostředí.

Kvůli tomu, že nám odpadla část pro definování zdrojů zvuků a posluchačů ve virtuálním prostředí, bude potřeba, aby hra měla možnost vytvářet zvuky. Další částí programátorského rozhraní tedy bude možnost vytváření instancí zvuků pomocí dvojic zdrojů zvuků a posluchačů, přičemž hra bude moci řídit přehrávání jednotlivých instancí zvuků.

V rozhraní pro nastavování audio enginu nám zůstane jen možnost pro nastavování, zda chce hra přijímat zvuková data z mikrofonu, neboť díky tomu, že hra si sama bude muset nastavovat efekty, tak nemá smysl mít možnost nastavování deaktivace aplikací nějakých efektů, neboť je hra sama o sobě nebude využívat. Zůstane nám ale rozhraní pro informování dostupnosti efektů a mikrofonu na dané platformě.

3.3 Vybrané cíle

C1) Nalezení vhodné dolní vrstvy při zachování přenositelnosti na různé platformy – přičemž vhodný způsob zachování přenositelnosti vybereme až při tvorbě této práce. Výběr plyne z potřeby *R3* a *R8* dnešních her.

C3) Vhodné čtení souborů, správa prostředí (změna lokace – načítání dalších zvuků) – abychom zbytečně nezaplňovali paměť (např. zvuky v jiném levelu hry). Plyne z potřeby *R2* dnešních her.

C4) Možnost vytváření složených zvuků, jako *S1*-*S4* v kapitole *1.1.2 Potřeby dnešních her – shrnutí*. Bez této možnosti bychom mohli přehrávat jen jednoduché zvuky, což by velmi omezovalo použití audio enginu. Vyplývá z potřeby *R5* dnešních her.

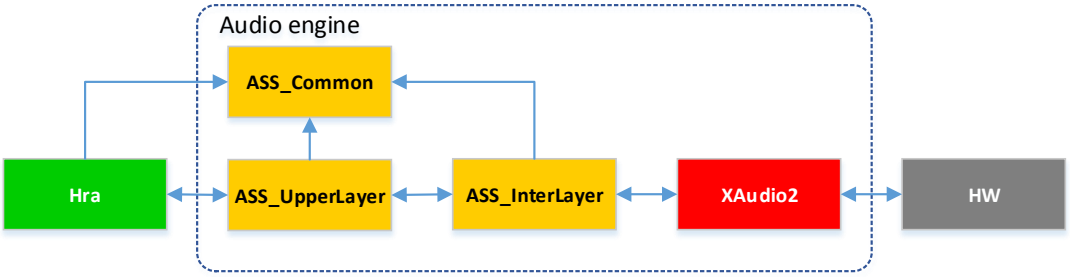
C5) Editor pro vytváření složených zvuků – aby bylo možné jednoduše vyrábět složené zvuky, plyne z potřeby *R5* dnešních her.

4 Vývojová dokumentace A – engine

V této kapitole si popíšeme implementaci hlavní části audio enginu *Advanced Sound System*, který je připraven ve formě několika knihoven. Všechny knihovny jsou zahrnuty v jednom solution, které se jmenuje *Advanced Sound System.sln*. Solution bylo otestováno ve *Visual Studiu 2010*.

Součástí solution je i visuální editor zvuků (*ASS_Editor.exe*), jež nám umožňuje vytvářet zvukové prvky a ty ukládat do souborů potřebných pro běh audio enginu. Popis implementace editoru je popsán v kapitole *5 Vývojová dokumentace B – editor*.

V následujících podkapitolách si popíšeme assembly *ASS_UpperLayer.dll*, *ASS_InterLayer.dll* a *ASS_Common.dll*, z nichž se audio engine skládá. Abychom si je mohli popsat, tak si nejdříve projdeme vztahy mezi jednotlivými assembly a daty, která assembly požadují. Nejdříve se podíváme na vztahy mezi assembly, které máme znázorněny na obrázku *4.1* níže:



Obrázek 4.1 - Vztah mezi assembly

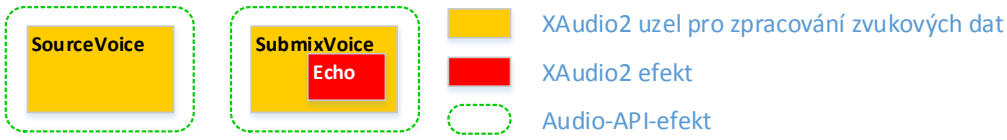
Audio engine *Advanced Sound System* se dělí na dvě základní vrstvy: horní vrstvu a dolní vrstvu. Horní vrstva je reprezentovaná assembly *ASS_UpperLayer*, která přímo komunikuje se hrou a je jediným vstupem hry do audio enginu. Pak je zde dolní vrstva, ve stávající implementaci je zastoupena externí knihovnou *XAudio2*, jež slouží k samotnému přehrávání zvuků. Chceme ale zajistit snadnou přenositelnost horní vrstvy na jiné dolní vrstvy nebo i jiné systémy, proto zde máme assembly *ASS_InterLayer*, která zajišťuje vazbu mezi dolní a horní vrstvou audio enginu a tak zajišťuje nezávislost horní vrstvy na dolní vrstvě. Následně tyto dvě části audio enginu (mimo dolní vrstvy), editor a hra potřebují využívat společné třídy (komponenty) a abychom se vyhnuli složitým závislostem, tak jsme tyto prvky vyčlenily do assembly *ASS_Common*.

Dále audio engine potřebuje ke svému běhu data, jež jsou uložena v souborech. Tato data jsou pak zpracovávána assembly *ASS_UpperLayer* a assembly *ASS_InterLayer*. Assembly *ASS_UpperLayer* vyžaduje soubory, v nichž jsou seznamy odkazů na popisy zvukových prvků a poté soubory se samotnými popisy zvukových prvků, přičemž základní struktura načítání těchto prvků je v assembly *ASS_Common*, neboť tato data také budeme využívat v editoru. Zato assembly *ASS_InterLayer* pracuje jen se zvukovými soubory. Tyto soubory ale potřebujeme propojit mezi sebou. Ručně by to bylo náročné, proto je součástí této práce editor.

Než si popíšeme jednotlivé assembly, tak si ještě nastíníme průběh komunikace mezi hrou a audio enginem. V první řadě bude hra inicializovat své části, tudíž hra

bude zjišťovat, jaké jsou dostupné možnosti pro nastavení zvuku. Tyto možnosti jsou pak využity při inicializaci audio enginu s konkrétním nastavením (jež nelze v průběhu dynamicky měnit). Poté následuje postup, který se bude v určitých intervalech (např. při dokončení herní úrovně) opakovat. V tomto postupu hra nejdříve v audio enginu načte soubory zvukových prvků, jež bude chtít do budoucna využívat. Následně hra z těchto zvukových prvků bude v audio enginu vytvářet instance jednotlivých zvuků, jež bude chtít hra přehrávat. Následuje část ovládání přehrávání jednotlivých instancí zvuků a poté když už je hra nevyužívá, tak tyto instance uvolní. Poté na konci postupu hra z audio enginu uvolní i jednotlivé zvukové prvky, které dosud využívala a začíná daným průběhem od začátku.

Pro další text kapitoly si zavedeme názvosloví ohledně efektů, neboť budeme využívat třídy reprezentující efekty v různých vrstvách. Pro zpracovávání zvuků knihovna *XAudio2* nabízí třídu `XAudio2.SourceVoice` (vstup zvukových dat) a třídu `XAudio2.SubmixVoice` (další zpracování zvukových dat), které zpracovávají zvuková data. Z těchto tříd pak lze sestavit řetězec prvků pro zpracování zvukových dat. Aby mohly tyto třídy modifikovat zvuková data, tak mohou mít přiřazeny různé druhy efektů. Abychom zjednodušili popis této struktury v následujícím textu, tak instance těchto tříd (obsahující efekty knihovny *XAudio2*) budeme označovat jako *audio-API-efekty*, viz obrázek *4.2* níže. Dále zde máme třídy reprezentující efekty na mezivrstvě, jež nám budou udávat výchozí množinu efektů, jež můžeme využívat v našem audio enginu. Efekty mezivrstvy budeme označovat jako *inter-efekty*. Nakonec zde máme efekty v horní vrstvě audio enginu, tyto efekty budeme označovat běžným způsobem jako *efekty*.



Obrázek 4.2 - Definování audio-API-efektu

I když jsou třídy dolní vrstvy umístěné ve jmenném prostoru `SharpDX.XAudio2`, tak pro zjednodušení budeme jejich jmenný prostor uvádět zkráceně jako `XAudio2`. Následující podkapitoly začneme popisem assembly *ASS_InterLayer*. Následně si popíšeme assembly *ASS_UpperLayer* a skončíme popisem assembly *ASS_Common*.

4.1 Assembly ASS_InterLayer

Assembly *ASS_InterLayer* slouží k propojení knihovny *XAudio2* s assembly *ASS_UpperLayer* jako mezivrstva. Tuto assembly generuje stejnojmenný projekt *ASS_InterLayer*, přičemž třídy této assembly jsou ve jmenném prostoru *ASS_InterLayer*.

V této práci je přizpůsobena knihovně *XAudio2*. Pokud bychom chtěli využívat jinou knihovnu jako dolní část audio enginu, tak tuto assembly budeme muset předělat podle případné knihovny.

Assembly *ASS_InterLayer* se skládá z následujících částí:

- Třída `InterLayer` sloužící jako hlavní instance mezivrstvy, jež se stará o nastavování dolní vrstvy audio enginu.

- Třídy `Support.SourcePool` a `Support.SubmixPool`, jež se starají o správu instancí audio-API-efektů.
- Třída `Support.BackBuffers`, jež slouží k získávání již zpracovaných zvukových dat.
- Třída `Source` sloužící jako zdroj zvuku (inter-efekt).
- Třídy `*Effect` představující inter-efekty, jež definují základní efekty, s nimiž audio engine bude pracovat.

4.1.1 Třída InterLayer

Třída `InterLayer` představuje instanci mezivrstvy, jež poskytuje přístup ke třídám dolní vrstvy. Třídy dolní vrstvy však potřebují k vytváření instanci třídy `XAudio2.XAudio2` (reprezentující dolní vrstvu), proto si třída `InterLayer` udržuje její instanci.

Mezivrstva je složena z několika celků, přičemž třída poskytuje položky a metody, jež umožní těmto celkům pracovat dohromady:

- Potřebovali jsme definovat efekt, jenž umožní směrovat zvuková data na zvukový výstup. K tomu jsme využili audio-API-efekt reprezentovaný třídou `XAudio2.MasteringVoice`. Aby mohly audio-API-efekty předávat zvuková data jiným audio-API-efektům, tak dolní vrstva potřebuje mít ke každému audio-API-efektu vytvořenou instanci třídy `XAudio2.VoiceSendDescriptor`. Z tohoto důvodu si třída `InterLayer` k efektů pro směrování dat na zvukový výstup udržuje instanci třídy `XAudio2.VoiceSendDescriptor` k tomuto audio-API-efektu. Pro nastavování hlasitosti výstupu na zvukový výstup pak třída `InterLayer` implementuje metodu `SetMasterVolume` (a pro získání hlasitosti metodu `GetMasterVolume`).
- Pro poskytování audio-API-efektů k inter-efektům třída `InterLayer` obsahuje instance tříd `SourcePool` a `SubmixPool`.
- Pro pravidelnou aktualizaci instancí tříd mezivrstvy třída `InterLayer` obsahuje metodu `Update`.
- Dolní vrstva pracuje v rámci tzv. *operačních setů*, kdy změny nastavení tříd dolní vrstvy se projeví až po potvrzení daného operačního setu, přičemž existuje i operační set (definovaný nulou), jež provádí změny okamžitě. Pro potvrzování operačního setu a získání dalšího identifikátoru operačního setu na třídě `InterLayer` slouží metody `CommitOperationSet` a `NextOperationSet`.
- K dodávání již zpracovaných zvukových dat třída `InterLayer` využívá instanci třídy `BackBuffers`. Pro registraci/uvolnění dodávání zpracovaných zvukových dat třída `InterLayer` obsahuje metody `CreateBuffer` a `DestroyBuffer`.
- Pro řešení odnačtení audio enginu si třída `InterLayer` ukládá seznamy instancí zdrojů zvuků a inter-efektů. K manipulaci se seznamy těchto instancí pak třída obsahuje metody `AddSource`, `RemoveSource`, `AddEffect`, `RemoveEffect`.

4.1.2 Třída SourcePool a SubmixPool

Vytváření instancí audio-API-efektů je časově náročné, proto vznikly třídy `Support.SourcePool` (zkráceně `SourcePool`) a `Support.SubmixPool` (`SubmixPool`). Program může vytvářet zvuky z více vláken, tudíž je potřeba mít tyto třídy *thread safe*, neboť každý zvuk se bude skládat z různých efektů, jež vyžadují inter-efekty a ty vyžadují audio-API-efekty.

Základní funkčnost těchto tříd je podobná a to, že třída `SourcePool` poskytuje instance tříd `XAudio2.SourceVoice` a třída `SubmixPool` poskytuje instance tříd `XAudio2.SubmixVoice`, přičemž si třídy udržují seznamy již použitých instancí audio-API-efektů k opětovnému využití. Z důvodu, že každá třída poskytuje jiný typ třídy, jež vyžaduje i jiné parametry, tak jsou tyto třídy `SourcePool` a `SubmixPool` napsané odděleně.

Třídy uchovávají seznamy již použitých audio-API-efektů, díky tomu se může stát, že třídy uchovávají řadu instancí audio-API-efektů, jež není potřeba využívat. Ve třídách je tedy definována řada konstant, jež definují, za jaký čas nečinnosti třídy (neposkytování, ani ukládání audio-API-efektů) se mají uvolnit již nepoužívané instance těchto audio-API-efektů. Uvolňování již nepoužívaných instancí audio-API-efektů následně probíhá pomocí metody `Update`, jež obě třídy implementují.

Různé instance audio-API-efektů jsou vytvořené s určitými parametry, jež nelze za běhu měnit, proto bylo potřeba tyto instance rozlišit a je potřeba uchovávat několik seznamů audio-API-efektů. K rozlišování seznamů jsme si definovali třídy `SourcePool.SourceDoubleKey` a `SubmixPool.SubmixQuatroKey` (kde „*Double*“ a „*Quatro*“ značí počet parametrů pro rozlišování seznamů). Dalším požadavkem pak bylo zajistit, aby se ve správný čas uvolňovali instance audio-API-efektů z již nepoužívaných seznamů *audio-API-efektů*. K tomuto účelu jsme si definovali třídy `SourcePool.SourceTimeQueue` a `SubmixPool .SubmixTimeQueue`, jež si pro seznam audio-API-efektů udržuje čas, jak dlouho se se seznamem audio-API-efektů nemanipulovalo.

Problémem bylo zjistit, jak se audio-API-efekty chovají, když se přestanou používat. Zda ještě zpracovávají nějaká data, či si vyprazdňují nějaké struktury. Z toho důvodu jsme potřebovali oddálit znovupoužití audio-API-efektů, proto jsme pro reprezentaci seznamu audio-API-efektů zvolili frontu.

Třídy `SourcePool` a `SubmixPool` pro získávání a vkládání instancí audio-API-efektů implementují metody `Fetch` a `Store`. Tyto metody obsahují parametry popisující parametry audio-API-efektu, jež chceme získat, či vložit, neboť nelze zjistit všechny parametry ze samotných instancí audio-API efektů.

4.1.3 Třída BackBuffers

Potřebovali jsme přístup horní vrstvy k již zpracovaným zvukovým datům, jež bude nezávislý na dolní vrstvě. Tento přístup nám zajišťuje třída `Support.BackBuffers` (zkráceně `BackBuffers`).

Třída umožňuje zpracovávaná zvuková data získávat pomocí několika oddělených bufferů, ty si označíme jako „*zpětné buffery*“. Pro definování (vytváření/rušení) zpětných bufferů třída obsahuje metodu `CreateBuffer` a metodu

`DestroyBuffer`, přičemž jednotlivé zpětné buffery jsou označovány kladným celým číslem. Číslo nula pak značí zpětný buffer pro data získaná ze zvukového vstupu (například mikrofonu). V této verzi ale podpora zvukového vstupu není implementována, neboť nám to neumožňuje knihovna *XAudio2*. Pro využívání zpětného bufferu slouží metoda `GetBuffer`, a pro uvolnění slouží metoda `ReleaseBuffer`. Zpětný buffer využíváme ve výstupním efektu, viz kapitola *4.1.4 Zdroje zvuku a inter-efekty – Ostatní inter-efekty*.

Pro reprezentaci zvukových dat (spolu s popisem formátu) posílané zpětným bufferem vyšší vrstvě audio enginu využíváme třídy `BackBuffers .BackBufferStructure`.

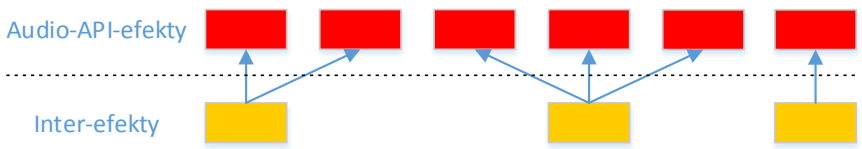
Dolní vrstva (knihovna *XAudio2*) však neumožňuje jednoduše získat zvuková data, z toho důvodu jsou zde třídy `BackBuffers.BackBufferEffect` a `BackBuffers.BackBufferParam`. Tyto třídy aplikujeme na třídu `XAudio2.SubmixSource` a tím si vytvoříme vlastní audio-API-efekt, pomocí nějž získáme zpracovávaná zvuková data a můžeme je předat vyšší vrstvě audio enginu pomocí zpětného bufferu.

4.1.4 Zdroje zvuku a inter-efekty

V analýze jsme probírali způsob reprezentace efektů v mezivrstvě, viz kapitola *2.2 Mezivrstva*. Jak jsme si popsali v úvodu kapitoly, tak zvuk bude v dolní vrstvě tvořen posloupností audio-API-efektů. Chceme ale mít základní strukturu efektů nezávislou na dolní vrstvě, z toho důvodu jsme si navrhli množinu inter-efektů, jež nám definuje efekty, které můžeme používat pomocí horní vrstvy. Inter-efekty tedy v základu slouží pro správné propojení a nastavení jednotlivých audio-API-efektů na dolní vrstvě, přičemž inter-efekty definují parametry pro jejich nastavování. Vytvořili jsme několik základních inter-efektů:

- zdroj zvuku – posílá zvuková data pro přehrávání dolní vrstvě,
- echo efekt,
- reverb efekt,
- efekt dolní propusti,
- efekt horní propusti,
- pack efekt - slučuje zvuková data z několika inter-efektů dohromady,
- výstupní efekt – posílá zvuková data na instanci `XAudio2.MasteringVoice`.

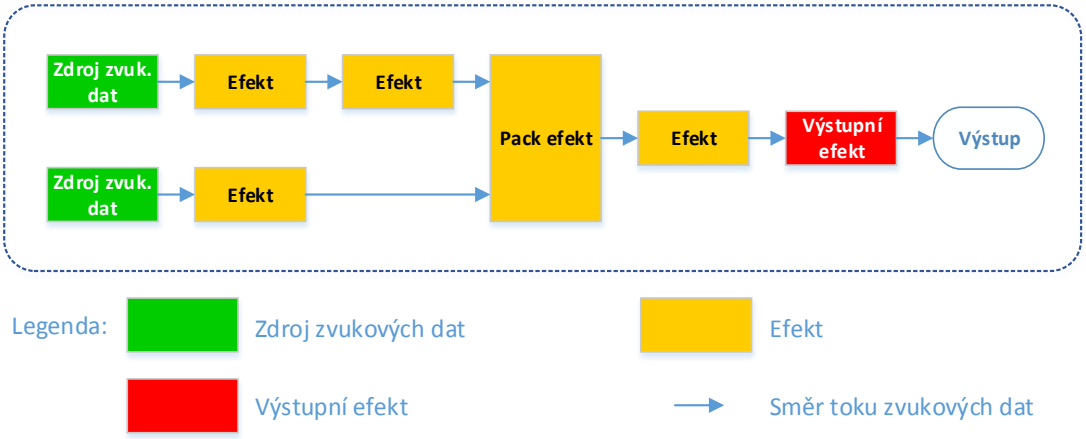
Inter-efekty můžou být v určitých případech tvořeny několika audio-API-efekty, viz obrázek *4.3*.



Obrázek 4.3 - Propojení inter-efektů s audio-API-efekty

Audio-API-efekty mají definované pořadí, ve kterém budou zpracovávat zvuková data, toto pořadí se nazývá „*process stage*“. Tudíž audio-API-efekty, které se budou aplikovat dříve, nemůžou mít process stage vyšší než audio-API-efekty, jež se budou aplikovat později. Inter-efekty mohou být složeny z více audio-API-efektů, tudíž inter-efekty mají definovaný rozsah process stage, jež budou využívat díky audio-API-efektům. Toho se pak využívá při budování řetězce inter-efektů (a následně audio-API-efektů, z nichž je zvuk vytvořen.

Zvuk se v mezivrstvě zpracovává řetězcem inter-efektů, přičemž určité druhy inter-efektů mají speciální význam. Zvuk se tedy bude skládat ze zdrojů zvukových dat (poskytují zvuková data), na ty budou dále navazovat různé inter-efekty. Zvuková data z inter-efeků pak lze sloučit dohromady do jednoho proudu zvukových dat pomocí pack efektu. Tyto inter-efekty budou končit výstupním efektem (pro směrování na výstup). Příklad složení zvuku z inter-efektů je na obrázku *4.4* níže:



Obrázek 4.4 - Složení zvuku z inter-efektů

Rozhraní IInputableProcess

Jako standartní rozhraní inter-efektů jsme navrhli rozhraní `IInputableProcess`. Toto rozhraní se skládá z následujících skupin metod:

- Metody `SetNewOutput`, `SetNewInput`, `GetOutput` a `RemoveThis`, jež zajišťují propojení inter-efektů mezi sebou.
- Metody `SetParameters`, `SetVolume`, `GetParameters` a `CommitParametersChanges`, sloužící pro nastavování parametrů inter-efektům.
- Metody `GetHighestProcessStage` a `GetLowestProcessStage` pro zjistění nevyšší a nejnižší process stage inter-efektu.

Rozhraní IInternalInputableProcess

Chtěli jsme mít na inter-efektech stejné metody, jež by navenek mezivrstvy nebyly viditelné, proto vzniklo rozhraní `IInternalInputableProcess`, které dědí od rozhraní `IInputableProcess`. Toto rozhraní pak implementují všechny inter-efekty.

V tomto rozhraní je definováva metoda `GetDescriptor`, jež využíváme pro získání instance `XAudio2.SendVoiceDescriptor` inter-efektu, umožňující propojení audio-API-efektů inter-efektu mezi inter-efekty.

Zdroj zvuku

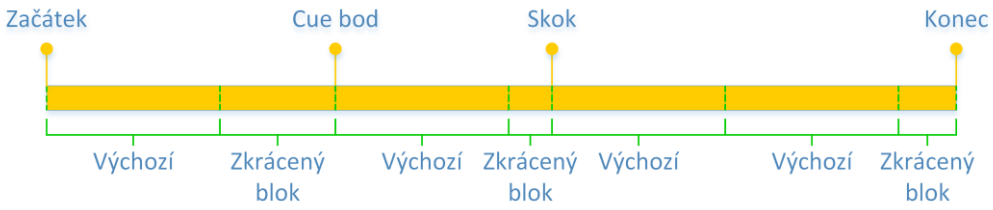
Zdroj zvuku je speciální inter-efekt, který je reprezentovaný třídou `Source`. Tato třída pak slouží k dodávání zvukových dat dolní vrstvě (neboť dolní vrstva si zvuková data sama nenačítá). Rozdíl mezi tímto a ostatními inter-efekty je ten, že zdroji zvuku nelze nastavit žádný vstupní inter-efekt, přičemž zdroj zvuku ani nelze odebrat z řetězce inter-efektů.

Zdroj zvuku lze ovlivňovat následujícími parametry:

- `FrequencyRatio` – jež umožňuje ovládat rychlost přehrávání zvukových dat,
- cue body – přičemž lze nastavit jen dva cue body (v průběhu je lze měnit) z důvodů rychlosti zpracování,
- smyčka – zda má zdroj zvuku dodávat zvuková data ve smyčce,
- konec – nastavuje konec dodávání zvukových dat (lze nastavit před koncem zvukového souboru)

Pro přehrávání zvukových dat pomocí dolní vrstvy třída `Source` využívá instanci audio-API-efektu, konkrétně třídy `XAudio2.SourceVoice`. Jako zdroj zvukových dat využívá třída `Source` instanci s rozhraním `IWaveReader` (viz kapitola *4.3 Assembly ASS_Common*). Pro průběžné načítání zvukových dat třída `Source` obsahuje vlákno, jež se zpracovává pomocí metody `LoadBuffer`.

Metoda `LoadBuffer` si na začátku definuje buffery pro načítání zvukových dat do instance audio-API-efektu. Přičemž velikost a počet bufferů lze definovat pomocí konstant (uvnitř třídy `Source`). Metoda implementuje smyčku, jež se stará o vlastní načítání zvukových dat. Zpracovávání smyčky čeká do doby, dokud není některý z bufferů pro načítání zvukových dat prázdný nebo se spustilo přehrávání zdroje zvuku na jiné pozici zvukových dat a je potřeba načíst nová zvuková data (v tomto případě se vyprázdní všechny načtené buffery zvukových dat). V základu načítáme jen konečná zvuková data (z nějakého zvukového souboru), pak zde máme experimentální načítání nekonečných zvukových dat (již zpracovaná zvuková data, či data dodávaná přímo aplikací), jež není zevrubně otestováno. Očekáváme, že pro finální verzi načítání nekonečných zvukových dat budeme potřebovat drobné architekturální úpravy! Při načítání konečných zvukových dat načítáme zvuková data do bufferu s ohledem na pozice cue bodů, smyčky a konce zvukových dat, aby pokud se některý z těchto prvků vyskytne, tak tyto prvky jsou vždy na rozmezí dvou bufferů (někdy je tedy buffer menší než jeho maximální povolená velikost), příklad načítání bufferů je znázorněn na obrázku *4.5*.



Obrázek 4.5 - Načítání bufferů zvukových dat

Toho se využívá pro správné zpracování těchto prvků. Při načítání nekonečných zvukových dat se hlídá, zda jsou k dispozici zvuková data a případně se nastaví stav přehrávání, že zdroj zvuku hladoví, či už nehladoví. V tomto případě nebere přehrávání zdroje zvuku ohled na cue body a na smyčky. Na konci smyčky se zvuková data uloží do bufferu a pošlou se instanci audio-API-efektu, jež tato zvuková data přehrává.

Třída `Source` dále obsahuje metody `sourceVoice_BufferStart`, `sourceVoice_BufferEnd` a `sourceVoice_ProcessingPassEnd`. Tyto metody slouží pro správné zpracování cue bodů, smyčky a konce zvukového souboru a jsou specifické pro knihovnu `XAudio2`, neboť se jedná o callbacky z instance audio-API-efektu (třídy `XAudio2.SourceVoice`). Metoda `sourceVoice_BufferStart` se využívá pro zjištění právě přehrávaného bufferu se zvukovými daty. Metoda `sourceVoice_ProcessingPassEnd` slouží pro zjišťování, zda se blíží nějaká událost a zda se má přehrávání zvuku ztlumit, či se má obnovit přehrávání zvuku. Metoda `sourceVoice_BufferEnd` zajišťuje reakci na nějakou událost, jako například zastavení přehrávání zvuku na cue bodě, nebo na konci zvukových dat.

Pro ovládání přehrávání zdroje zvuku jsou ve třídě `Source` metody `Start` a `Stop`. Metoda `Start` umožňuje kromě spuštění zvukových dat od jejich začátku spustit přehrávání zvukových dat z libovolné pozice.

Pro nastavování parametrů a hlasitosti zdroje zvuku slouží metody `SetParameters` a `SetVolume`. Zde bylo nutné vyřešit nastavování hlasitosti a cue bodů. Při nastavování hlasitosti by se nám mohlo stát, že si přenastavíme nulovou hlasitost při automatickém zastavení zvuku pomocí cue bodu. Proto si aktuální nastavení hlasitosti uložíme do kolekce `paramOperationVolume` podle operačního setu a hlasitost nastavíme až v metodě `CommitParametersChanges`, kde nastavíme hlasitost podle potvrzeného operačního setu (s ohledem na hlasitost při automatickém zastavení zvuku). Obdobně je řešené nastavování cue bodů, kde využíváme kolekci `paramOperation`. Pro aplikaci nastavení cue bodů je ve třídě `Source` metoda `ParametersChanges`, jež provede případné znovuspuštění přehrávání pozastaveného zdroje zvuku.

Ostatní inter-efekty

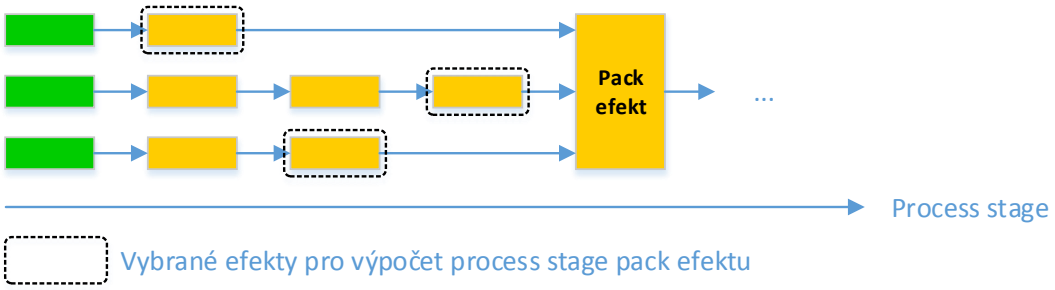
Na rozdíl od třídy `Source` ostatní inter-efekty dědí od třídy `Effects.AbstractEffect`, přičemž všechny jsou umístěny ve jmenném prostoru `Effects`. Většina inter-efektů využívá jen jeden audio-API-efekt, tudíž třída `AbstactEffect` obsahuje položku `submix`, což je instance audio-API-efektu a položku `descriptor`, jež je instance třídy `XAudio2.VoiceSendDescriptor` (jež slouží pro propojování audio-API-efektů). Dalšími jsou položky `input` a `output`, jež odkazují na souvislé inter-efekty a položky informující o hodnotě process stage. Každý inter-efekt obsahuje jako parametr jeho hlasitost, proto třída dále obsahuje položku Volume.

Třída `AbstractEffect` je přizpůsobena pro použití jednoho vstupního inter-efektu (či zdroje zvuku), čemuž je přizpůsoben konstruktor třídy, jež řeší i výpočet hodnoty process stage daného efektu.

Třída `AbstractEffect` implementuje většinu metod rozhraní `IInputableProcess` a `IInternalInputableProcess`, neboť zbytek

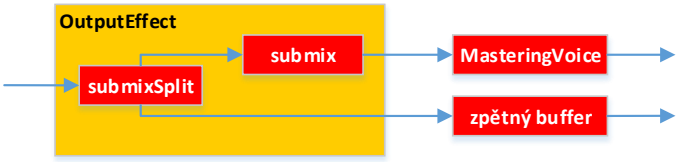
implementují jednotlivé třídy inter-efektů. Třídy inter-efektů implementují metody `SetParameters`, `GetParameters` a `CommitParametersChanges`. Je to důledek toho, že každý inter-efekt definuje vlastní strukturu parametrů `*Effect.Parameters`. Inter-efekty v konstruktoru třídy pak nastavují správné nastavení audio-API-efektu.

Potřebovali jsme inter-efekt, jenž bude umět sloučit zvuková data z několika inter-efektů dohromady do jednoho proudu zvukových dat, proto jsme vytvořili pack efekt. Tento inter-efekt je reprezentovaný třídou `Effects.Special.PackEffect` (zkráceně `PackEffect`). S tím jsme museli vyřešit problém nastavení správného process stage pro pack efekt. Toho je docíleno tak, že nastavíme process stage, jež následuje po vstupním inter-efektu s nejvyšším process stage, viz obrázek *4.6*.



Obrázek 4.6 - Určování process stage pack efektu

Dále jsme potřebovali inter-efekt, jenž bude umět posílat zvuková data na zvukový výstup, přičemž bude umět nastavit hlasitost jednotlivých kanálů na výstupu a bude umět posílat zvuková data do zpětného bufferu. Pro tento inter-efekt jsme si vytvořili třídu `Effects.Special.OutputEffect` (zkráceně `OutputEffect`). Pro nastavování hlasitosti jednotlivých kanálů definuje třída řadu metod podle rozložení výstupu definovaného výčtem `OutputEffect.OutputType`. Aby stále nemuselo při aktualizaci parametrů docházet k rozhodování, která metoda se má použít pro nastavení parametrů inter-efektu, tak jsme si definovali delegáta `OutputEffect.UpdateDelegate`, jež splňují všechny metody pro nastavování parametrů. Následně jsme si vytvořili položku update, jež využívá delegáta `OutputEffect.UpdateDelegate`, a té přiřadíme podle rozložení reproduktorů správnou metodu jen jednou v konstruktoru a metodu voláme pomocí položky update. V případě, že inter-efekt posílá zvuková data i na zpětný buffer, má třída kromě položky `submix` (audio-API-efekt) i položku `submixSplit` (audio-API-efekt). Audio-API-efekt `submixSplit` pak směřuje zvuková data na audio-API-efekt `submix` a na audio-API-efekt zpětného bufferu, dodaného třídou `Support.BackBuffers`. Směřování zvukových dat tohoto inter-efektu pomocí audio-API-efektů je znázorněno na obrázku *4.7*.



Obrázek 4.7 - Směřování zvukových dat inter-efektu OutputEffect

4.2 Assembly ASS_UpperLayer

Assembly *ASS_UpperLayer* slouží jako horní vrstva audio enginu. Je to část audio enginu, jež poskytuje hře potřebné třídy pro práci se zvuky. Tato assembly je součástí stejnojmenného projektu *ASS_UpperLayer*, přičemž její třídy jsou ve jmenném prostoru *ASS_UpperLayer*. Tyto třídy se poté starají o načítání zvukových dat a jejich přehrávání, jež může hra v průběhu předem danným způsobem ovlivňovat.

Tato assembly se skládá z několika hlavních částí. Za prvé je to třída `UpperLayer`, jež ve hře poté představuje instanci audio enginu. Další částí, tentokrát skládající se z více tříd, je část starající se o správné načítání dat ze souborů vytvořených editorem, jejíž součástí jsou i třídy pro načítání zvukových dat ze zvukových souborů. Poslední hlavní částí jsou třídy reprezentující instance jednotlivých zvuků, pomocí nichž hra může přehrávat jednotlivé zvuky.

4.2.1 Třída UpperLayer

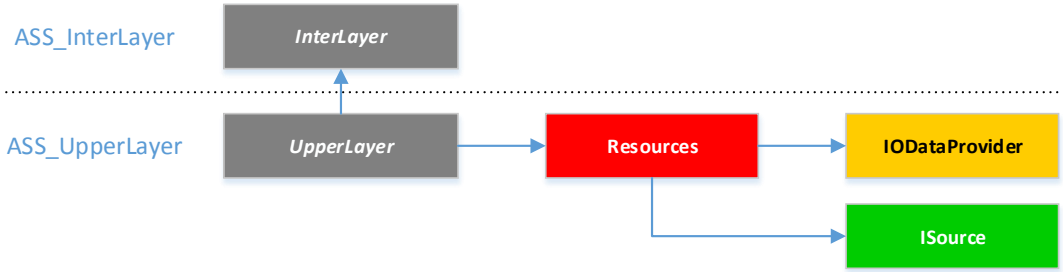
Třída `UpperLayer` je hlavní třídou audio enginu představující instanci audio enginu. Přesto tato třída není vytvořena jako singleton, neboť je možné, aby bylo vytvořeno více instancí, přičemž tyto instance se neovlivňují. Proto je důležité na to myslet, aby se nevytvářelo více instancí audio enginu najednou, pokud toho není zapotřebí.

Jednotlivé metody této třídy jsou v podstatě takovým přemostěním mezi hrou a jednotlivými třídami audio enginu, z nichž se audio engine skládá.

Metody této třídy můžeme rozdělit do několika skupin:

- Metody `Is*Available` a `GetAvailableChannels` poskytující informace o audio enginu, jako je dostupnost jednotlivých efektů, či počet zvukových kanálů, jež máme k dispozici na výstupu. Tyto metody jsou tvořeny jako statické, tudíž pro zjištění těchto informací nemusíme vytvářet instanci audio enginu.
- Metody `GetMasterVolume` a `SetMasterVolume`, ty slouží k nastavení hlasitosti celého audio enginu.
- Metody `LoadResources*`, `DestroyResources*` slouží pro načítání a odnačítání zvukových dat.
- Metoda `GetSource` vrací instanci konkrétního zdroje zvuku svázaného s určitým posluchačem (výstupem zvuku).
- Metoda `Update`, sloužící k aktualizaci částí audio enginu v průběhu hry.

Třída `UpperLayer` v základu zprostředkovává komunikaci se třídami `InterLayer` a `Resources`, viz obrázek *4.8*. Třída `Resources` pak využívá data dodávaná třídou `IODataProvider` a vrací třídě `UpperLayer` instance rozhraní `ISource`.



Obrázek 4.8 - Komunikace hlavních tříd horní vrstvy audio enginu

4.2.2 Instance zvuku

Hlavní součástí horní vrstvy jsou instance zvuků. Tyto zvuky může hra ovládat pomocí rozhraní `ISource`.

Rozhraní `ISource` implementují dvě třídy, třída `SimpleSource` a třída `MixedSource`, jež představují instance zvuku. Obě třídy obsahují podporu pro vnější efekty, posluchače a parametry zvuku. Třída `SimpleSource` dále obsahuje jen jeden zdrojový efekt, na který se mohou aplikovat různé efekty. Třída `MixedSource` obsahuje podporu pro několik zdrojových efektů, na které jdou aplikovat různé efekty. Navíc obsahuje speciální rekurzivní strukturu `MixedSource.PackClass`, jež umožňuje několik zvukových dat z různých zdrojů zvukových dat slévat do jednoho proudu zvukových dat, na která může aplikovat další efekty (viz *kód níže*). Přestože základní prvky obsahují stejné, přesto jedna nedědí od druhé a to z důvodu, že s těmito základními prvky tyto třídy nakládají jinak, proto obě implementují pouze rozhraní `ISource`.

```
class PackClass
{
    public PackClass[] packClass;
    public SourceEff source;
    public AbstractEff[] effects;
    public AbstractEff lastEffect;
}
```

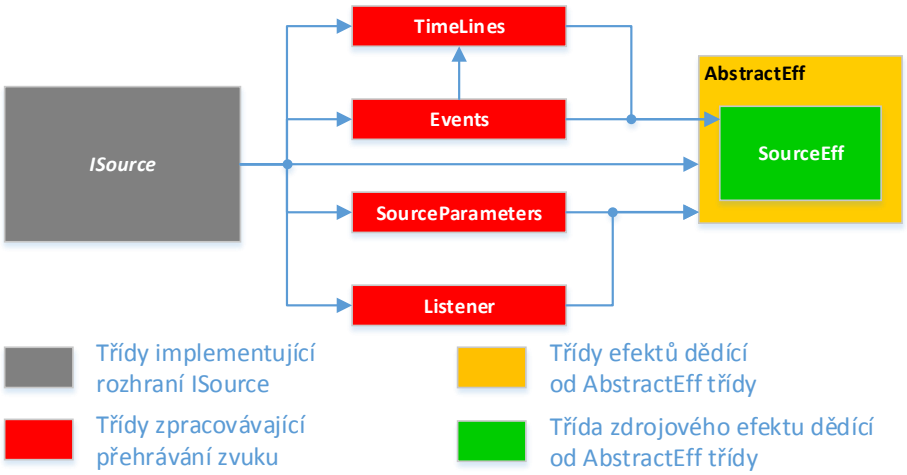
Rozhraní `ISource` se skládá z několika skupin metod:

- metody `PlaySource` a `StopSource` – sloužící k ovládání přehrávání zvuku,
- metody `*Effect` – spravující (přidává/nastavuje/ruší) vnější efekty,
- metoda `SetParameters` – sloužící pro nastavování parametrů,
- metoda `DoEvent` – pro volání událostí,
- metoda `Update` pro aktualizaci instance zvuku,
- metody pro výpis ladicích informací o instanci zvuku.

Instance zvuku se skládá z následujících prvků (tyto prvky si popíšeme v následujících kapitolách):

- Zdrojové efekty (viz kapitola *4.2.2.1 Zdrojové efekty a efekty*),
- efekty, jež se postupně aplikují na zvuková data (viz kapitola *4.2.2.1 Zdrojové efekty a efekty*),
- vnějšími efekty, jež může přidat samotná hra, přičemž tyto efekty nevycházejí z popisu zdroje zvuku - jejich počet je omezen na 20 z výkonových důvodů, přičemž tento počet je více než dostatečný,
- posluchačem, s výstupním efektem (viz konec této kapitoly),
- parametry zvuku (viz kapitola *4.2.2.3 Parametry*),
- časové osy přehrávání zdrojů zvukových dat (viz kapitola *4.2.2.2 Časové osy*),
- události zvuku (viz kapitola *4.2.2.4 Události*).

V následujícím textu si popíšeme vztah mezi prvky instance zvuku, přičemž pro ukázku použijeme třídu `MixedSource`, neboť třída `SimpleSource` je v podstatě zjednodušením třídy `MixedSource`. Jednotlivé vztahy jsou též znázorněny na *obrázku 4.9*.



Obrázek 4.9 - Vztahy mezi prvky instance zvuku

Přehrávání zvuku je řízeno zavoláním příslušné metody na instanci zvuku, jež spustí přehrávání časových os (třída `TimeLines`). Tyto časové osy pak řídí přehrávání jednotlivých zdrojů zvukových dat (třída `SourceEff`), z nichž je instance zvuku složena.

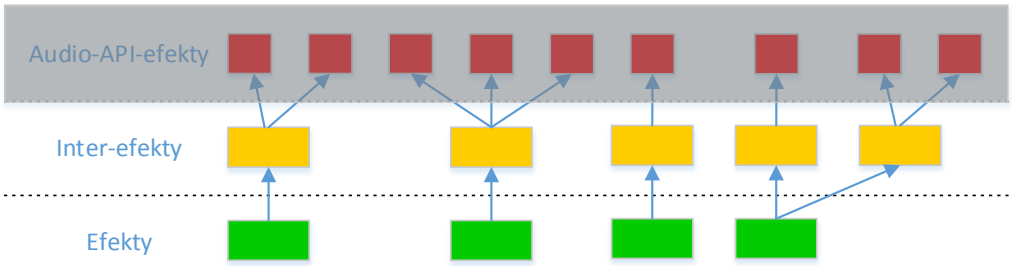
Nastavování parametrů zvuku probíhá skrze třídu `SourceParameters`, jíž se nastaví hodnota parametrů zvuku. Třída pak zpracuje hodnotu parametru, kterou následně nastaví zdrojům zvukových dat (třída `SourceEff`) a efektům (třídy dědící od `AbstractEff`) instance zvuku.

K volání událostí využívá instance zvuku třídu `Events`. Tato třída pak zpracuje událost a podle nastavení (dané popisu zvuku) pak upraví přehrávání časových os (třída `TimeLines`) instance zvuku, případně spustí/zastaví přehrávání určitých zdrojů zvukových dat (třída `SourceEff`) instance zvuku.

Pro nastavení výstupu zvuku pak slouží posluchač (reprezentovaný třídou `Listener`), jenž obsahuje efekt (třídy dědící od `AbstractEff`) pro výstup zvuku na zvukový výstup (či dalšímu zpracování).

4.2.2.1 Zdrojové efekty a efekty

Třídy efektů můžeme rozdělit na dva základní druhy: zdrojové efekty a efekty. Poskládáním těchto efektů za sebe, podobně jako inter-efekty v mezivrstvě (viz kapitola *4.1.4 Zdroje zvuku a inter-efekty*), vytváříme strukturu pro zpracování zvukových dat instance zvuku. Všechny zdrojové efekty a efekty (dále oboje budeme nazývat zkráceně *efekty*) vycházejí z abstraktní třídy `AbstractEff` a jsou umístěny ve složce projektu *Effects*. Každý efekt je zde tvořen jedním, či více inter-efekty (obdobně jako inter-efekty jsou tvořeny jedním či více audio-API-efekty), viz obrázek *4.10*.



Obrázek 4.10 - Propojení efektů s inter-efekty

Každý efekt má své určité spektrum parametrů a definovanou hlasitost na výstupu (dále jen parametry), proto je pro jejich manipulaci na abstraktní třídě definováno několik metod: `SetParameter`, `SetParameters`, `GetParameters`, `SetVolume` a `GetVolume`. Každý parametr se nicméně může skládat z několika hodnot. To je důsledkem, že potřebujeme zvuk přehrávat v určitém kontextu časové osy, parametrů instance zvuku a parametrů nastavovaných hrou (v případě vnějších efektů). K definici parametrů efektu z více hodnot používáme třídy `DynamicParameters` (viz níže *Třída DynamicParameters*). Tyto parametry mohou být nastavovány každý zvlášť, proto abstraktní třída definuje metodu `Change`, jež slouží pro výpočet konečné hodnoty parametrů efektu.

Třída DynamicParameters

Potřebovali jsme mít parametry, jež budou vypočítávány z více dílčích hodnot. K tomuto účelu jsme si vytvořili třídu `DynamicParameters`, která si tyto dílčí hodnoty uchovává a určitým způsobem je kombinuje a dodává výslednou hodnotu parametru.

Třída je tvořena polem hodnot, výslednou hodnotou a položkou `change`, jež udává, zda se některá hodnota z pole hodnot změnila a je potřeba přepočítat výslednou hodnotu.

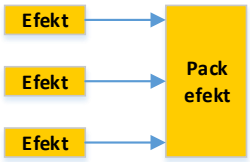
Od této třídy dědí celkem tři třídy, jež definují způsob kombinace hodnot:

- `DynamicParametersAdd` – kombinuje hodnoty sčítáním
- `DynamicParametersMulti` – kombinuje hodnoty násobením
- `DynamicParametersOne` – bere v potaz jen první hodnotu

Efekty

Třídy efektů tvoří obvykle jen jeden inter-efekt, jenž má parametry definované vlastní třídou v mezivrstvě. Z výše uvedeného důvodu, že efekt může být tvořen více inter-efekty, tak každý efekt má vlastní třídu (třída `<třída efektu>Parameters`) pro popis jeho parametrů. Pro správné přiřazení parametrů jednotlivým inter-efektům slouží výše zmíněná metoda `Change`. Mezi efekty patří:

- echo efekt (třída `EchoEff`),
- reverb efekt (třída `ReverbEff`),
- efekt dolní propusti (třída `LowPassEff`),
- efekt horní propusti (třída `HighPassEff`),
- efekt sloučení více zvukových stop (třída `PackEff`),
  - tento efekt je odlišný od ostatních v tom, že vstupem není jen jeden, ale více efektů, jejichž zvuková data slučuje do jednoho zvukového proudu (viz obrázek *4.11*),
- výstupní efekt (třída `OutputEff`),
  - na tento efekt by už neměl navazovat jiný efekt, neboť je to efekt přizpůsobený na zvukový výstup.



Obrázek 4.11 – Znázornění vsupů pack efektu

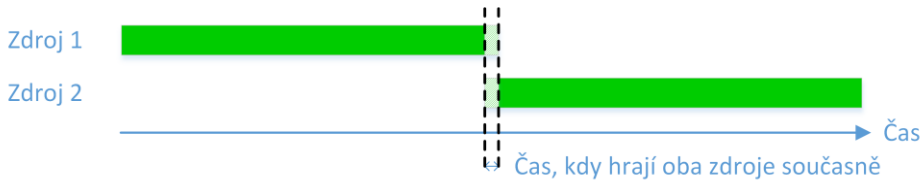
Zdrojové efekty

Zdrojové efekty jsou koncipované jako běžné efekty, s rozdílem, že nepřijímají zvuková data z žádných efektů, ale tato zvuková data dodávají. Tyto zdrojové efekty jsou zastoupeny třídou `SourceEff`. Zvuková data pak zdrojový efekt získává z instance `IReader`, jež mu byl při vytvoření přiřazen.

Kvůli časovým osám (viz kapitola *4.2.2.2 Časové osy*) jsme potřebovali umět přehrávat zdroj ve smyčce, nastavovat hlasitost přehrávání a nastavovat zdroji cue body.

Pro přehrávání zdroje ve smyčce jsme potřebovali možnost udělat zvukový přechod, kdy dohráváme zvuk z konce smyčky a začínáme přehrávat zvuk ze začátku smyčky (viz obrázek *4.12* na následující straně). Z toho důvodu zdrojové efekty obsahují dva inter-efekty zdroje zvuku (z nichž jeden je hlavní) a jeden inter-efekt pro sloučení těchto dat. K tomuto účelu pak slouží metody `SwitchActive`, kdy

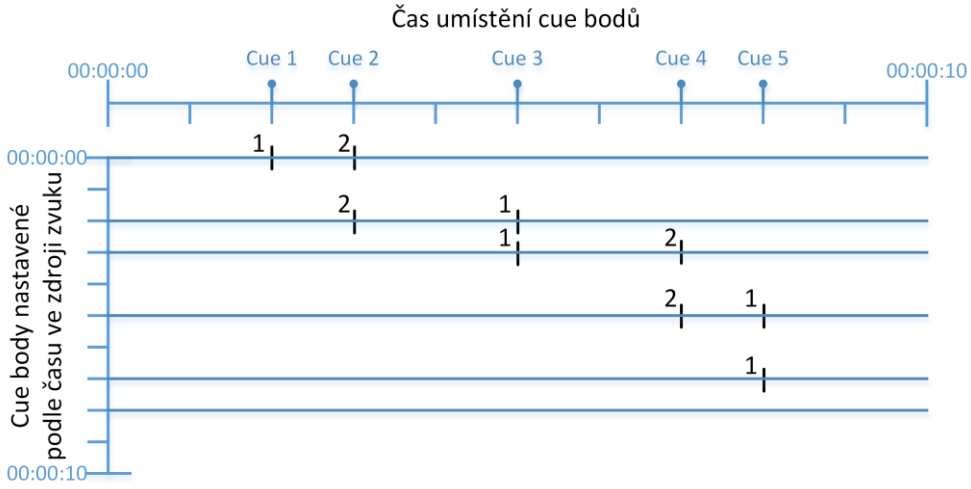
prohodíme ony dva inter-efekty zdroje zvuku a metoda `StopLoop`, která zastaví přehrávání vedlejšího efektu zdroje zvuku z mezivrstvy.



Obrázek 4.12 - Přehrávání zvuku ve smyčce pomocí dvou zdrojových efektů

K nastavování hlasitosti přehrávání z časových os zde máme metodu `SetTimeLineVolume`, jež ovlivní hlasitost zdrojového efektu nezávisle na nastavení parametrů zdrojového efektu. Také zde máme metodu `SetTimeLineVolumeLoop`, jež se využívá pro nastavení hlasitosti vedlejšího inter-efektu zdroje zvuku z mezivrstvy při přehrávání ve smyčce.

Zdrojový efekt pak obsahuje možnost nastavit cue body, tzn. časy, na kterých se daný zvuk zastaví bez vnějšího zásahu. Pro nastavení časů cue bodů a nastavení, že daným cue bodem můžeme projít, slouží metody `SetCuePosition` a `SetThroughCue`. Inter-efekty zdroje zvuku umožňují z výkonnostních důvodů nastavit jen dva cue body, proto zdrojový efekt řeší přidělování cue bodů inter-efektům zdrojů zvuků. Pro nastavení cue nejbližších možných cue bodů slouží metoda `SetNewCues`, jež podle času přehrávání nastaví dva nejbližší budoucí cue body, kdy cue bod se sudým indexem je nastaven jako 1. cue bodu a s lichým indexem jako 2. cue bod inter-efektu zdroje zvuku. Přičemž tuto metodu voláme při spuštění přehrávání zdrojového efektu. Druhá metoda `SetNextCue` slouží k nastavení dalšího cue bodu, kdy nový cue bod podle jeho indexu (sudý, lichý) nahradí cue bod na inter-efektu zdroje zvuku (jako výše přiřazení nových cue bodů). Průběh nahrazování cue bodů na inter-efektu je znázorněn na obrázku *4.13*. Kdy osa X představuje časovou osu s definovanými cue body a osa Y znázorňuje uložení cue bodů na daných pozicích v inter-efektu zdroje zvuku v závislosti na čase.



Obrázek 4.13 - Nastavování cue bodů v inter-efektu zdroje zvuku

Pro spouštění a zastavování přehrávání zdroje třída `SourceEff` obsahuje metody `PlaySource` a `StopSource`.

4.2.2.2 Časové osy

Instance zvuku se může skládat z několika zdrojových efektů a bylo potřeba ovládat přehrávání jednotlivých zdrojových efektů na předem určeném čase. K tomu nám slouží časové osy, jež ovládají přehrávání jednotlivých zdrojových efektů. Třídy týkající se časových os jsou v projektu ve složce *Effects/TimeLine*.

Boxy

Potřebovali jsme na časových osách seskupit určité zdrojové efekty do jednoho bloku, jenž by určitým způsobem přehrával zdrojové efekty. K tomuto účelu jsme si vytvořili abstraktní třídu `Box`. Každá instance boxu má definovanou hlasitost přehrávání pomocí třídy `Graph` (viz kapitola *4.2.2.3 Parametry*), která vypočítá hlasitost podle času přehrávání boxu (definovanou designérem zvuku).

Zpracovávání přehrávání boxu probíhá v metodě `Process`, která zjistí, zda daný box má spustit přehrávání zdrojových efektů, či zda je přehrává, nebo zda je má zastavit a zavolá příslušnou metodu `Process*`. Přičemž skrze třídu `Graph` také vypočítává hlasitost přehrávání zdrojů zvukových dat.

Popis metod volaných metodou `Process`, přičemž každá má v podtřídách svojí vlastní implementaci:

- `ProcessInput` – slouží k zahájení přehrávání zdrojů zvukových dat.
- `ProcessInside` – slouží k aktualizaci přehrávání zdrojů zvukových dat.
- `ProcessOutput` – slouží k ukončení přehrávání zdrojů zvukových dat.
- `ProcessFadeoutInside` – slouží k aktualizaci přehrávání dozvuku zdrojů zvukových dat (při skoku).
- `ProcessFadeout` – slouží k zahájení přehrávání dozvuku zdrojů zvukových dat (při skoku).
- `ProcessFadeoutEnd` – slouží k ukončení přehrávání dozvuku zdrojů zvukových dat (při skoku).

Konkrétní třídy boxů jsou následující:

- Třída `BoxSimple` – slouží pro přehrávání jednoho zdroje zvukových dat.
- Třída `BoxMulti` – slouží pro přehrávání více zdrojů zvukových dat najednou.
- Třída `BoxRandom` – slouží k náhodnému přehrávání jednoho z více zdrojů zvukových dat. Přičemž pokud je více než jeden zdroj zvukových dat, tak jako další zvuk přehrává náhodně jiný zdroj zvukových dat (aby se nepřehrával ten samý dvakrát za sebou).

Třída TimeLines

Třída `TimeLines` slouží k ovládání jednotlivých časových os a umožňuje jednotlivým časovým osám nastavovat:

- stav časových os (metodou `SetState`),
- cue body – kterými budou moci časové osy projít bez zastavení (metodou `SetCueThrough`),
- počet opakování skoků na časových osách (metodou `SetEnableCycles`).

Pro ovládání běhu časových pak používá dvě metody:

- metodu `Update`, jež aktualizuje přehrávání právě aktivní časové osy a tím aktualizuje její prvky,
- metodu `Jump`, jenž umožňuje skočit na stejnou, či jinou časovou osu (i doprostřed časové osy) a změnit tak průběh přehrávání časových os.

Třída TimeLine

Instance časové osy je tvořena třídou `TimeLine`. Tato třída se stará o ovládání přehrávání zdrojových efektů pomocí boxů. Zpracování běhu časové osy pak probíhá pomocí metody `Update`, jenž řídí přehrávání časové osy s ohledem na cue body a skoky (cykly). Třída dále obsahuje metodu `UpdateFadeOut`, ta slouží ke zpracování průběhu neaktivní časové osy (používá se při skoku pro doznění zdrojových efektů).

Cue body zastavují čas časové osy v určitém čase, přičemž zastavují přehrávání zdrojových efektů, které nemají nastavené přehrávání ve smyčce, ty naopak ponechá běžet dále. Pro nastavování pak třída `TimeLine` obsahuje metody `SetCueThrough`, `SetCue` a `TryConfirmCue`.

Skoky pak mění čas přehrávání časové osy, přičemž skok může být v rámci jedné časové osy, či může dojít ke změně aktivní časové osy. K provedení skoku třída `TimeLine` obsahuje metodu `Jump`. Třída pak dále obsahuje metodu `SetEnableCycles` pro nastavení cyklů.

4.2.2.3 Parametry

Potřebovali jsme umožnit nastavování parametrů instanci zvuku, jež by podle hodnoty nastavovalo určité parametry efektům instance zvuku. K definování jednotlivých parametrů jsme si vytvořili třídu `SourceParameters`. Pro výpočet hodnoty pro parametr efektu z hodnoty parametru instance zvuku používáme třídu `Graph`.

Třída `Graph` slouží pro výpočet hodnoty z dodané hodnoty, jedná se v podstatě o výpočet hodnoty Y z hodnoty X v definovaném grafu. Třída si udržuje strom uzlů, jež obsahují definice čar grafu, z nichž je graf poskládán. Definice čar grafu vycházejí ze třídy `AbstractGraphLine`, přičemž tato třída definuje metodu `Calculate`, jež zajistí výpočet hodnoty podle daného typu čáry. Máme celkem dva typy čar:

- konstantní – reprezentovaná třídou `ConstantGraphLine`,
- lineární – reprezentovaná třídou `LinearGraphLine`.

Třída `SourceParameters` slouží pro nastavování parametrů instanci zvuku. Pro nastavení parametrů obsahuje metodu `SetParameter`. Třída obsahuje seznam instancí jednotlivých parametrů reprezentovaných třídou `Parameter`, přičemž každý parametr má svůj unikátní název. Třída `Parameter` pak obsahuje seznam tříd `GraphAssignment`.

Třída `GraphAssignment` obsahuje instanci efektu, jemuž bude nastavovat určitý parametr, a instanci třídy `Graph`, jež zajistí výpočet hodnoty parametru efektu z hodnoty parametru instance zvuku. Nastavení parametru pak probíhá v metodě `Calculate`.

4.2.2.4 Události

Bylo potřeba umožnit volání událostí na instanci zvuku, jež by předem daným způsobem ovlivňovalo její přehrávání. K tomuto účelu jsme vytvořili třídu `Events`. Tato třída následně obsahuje seznam efektů reprezentovaných unikátním názvem a seznamem akcí, jež se mají provést (reprezentovaných abstraktní třídou `AbstractActionDescriptor`). K volání událostí třída `Events` obsahuje metodu `ProcessEvent`, jež zavolá pro událost daný seznam akcí.

Akce událostí pak dědí od abstraktní třídy `AbstractActionDescriptor`, jež definuje metodu `DoEvent`, jež následně v dané třídě implementuje danou akci.

4.2.3 Načítání dat

Třídy pro načítání dat se nacházejí ve složce *Management* projektu *ASS_UpperLayer*. Především se jedná o dvě třídy, třídu `Resources` a třídu `IODataProvider`.

Hlavní částí načítání dat je i hierarchie tříd, jež slouží k popisu jednotlivých částí zdrojů zvuků a jež jsou všechny umístěny ve složce *Loadings* tohoto projektu. V podstatě se jedná o vzájemně propojené popisné třídy s načtenými hodnotami ze souborů, přičemž každá třída obsahuje metodu pro vytvoření instance vlastní třídy dané popisné třídy pro audio engine. Přesný popis načítání samotných zdrojů zvuků je pak v kapitole *4.3 Assembly ASS_Common*, kde jsou definovány její základní části.

Třída Resources

Třída `Resources` slouží k načítání popisů zdrojů zvuků a posluchačů (viz výše kapitola *4.2.2 Instance zvuku*) ze souborů. Tyto popisy se seskupují do celků, které se načítají/odnačítají najedou, a jsou definovány v příslušném souboru. Většinou se popisy zdrojů zvuků a posluchačů v těchto celcích prolínají, takže stejný zdroj zvuku či posluchač může být obsažen v několika celcích. Proto tato třída pro načítání používá speciální třídy (tzv. *reference counting*), jež počítají počet načtení daného zdroje zvuku či posluchače, přičemž jejich popis třída nahrává jenom jednou a odnačítá ho, až bude odnačítán poslední celek, jež je obsahoval. Tím se zmenší velikost paměti potřebná pro uchovávání načítaných dat.

Třída `Resources` z načtených popisů zdrojů zvuků a posluchačů umožňuje vytvářet instanci zvuku, k tomu obsahuje metodu `GetSource`. Vytvářené instance si třída uchovává a následně pomocí metody `Update` aktualizuje jejich přehrávání. Zároveň uchování těchto instancí umožní jejich odnačtení v případě odnačtení samotného audio enginu.

Pro ladění audio enginu tato třída obsahuje několik metod pro výpis ladících informací, jako názvy načtených zdrojů zvuků a posluchačů s počtem jejich načtení.

Třída IODataProvider

Třída `IODataProvider` slouží ke správě zdrojů zvukových dat v audio enginu. Je součástí horní vrstvy audio enginu, neboť chceme mít nezávislost načítání dat na dolní vrstvě audio enginu.

Různé zvuky mohou být složeny ze stejných zdrojů zvukových dat, tudíž i zde je využito *reference countingu*, kde pro stejná data si pamatujeme počet jejich možného využití, ale data načítáme jen jednou. Zdroje zvukových dat se pak získávají

při načítání popisů zdrojů zvuků, proto se počítá jen s jejich možným využitím, neboť skutečný počet využití nás nezajímá z toho důvodu, že pokud není načtený popis zdroje zvuku s daným zdrojem zvukových dat, tak nemůže existovat instance zdroje zvuku, tudíž i využití daného zdroje zvukových dat.

Zdrojů zvukových dat je několik typů, proto jsme vytvořili rozhraní `IWaveReader` (je definováno v assembly *ASS_Common*, viz kapitola *4.3 Assembly ASS_Common*), jež využijeme k definování tříd jednotlivých typů zdrojů zvukových dat.

Třída `IODataProvider` obsahuje metody `LoadReaders` a `DestroyReaders`, pomocí nichž načítá, či odečítá zdroje zvukových dat. Dále třída obsahuje metodu `GetReader`, jež je využívána při vytváření instance zvuku z jeho popisu a vrací instanci zvuku konkrétní zdroj zvukových dat.

Typy zdrojů zvukových dat:

- „*File*“ označuje soubor. Zvuková data se v tomto případě načítají celá do paměti. Jako zdroj zvukových dat se využívá třída `FileWaveReader`.
- „*FileOneUse*“ je také soubor s rozdílem, že zvuková data jsou streamovaná. Jedná se o to, že celá do paměti, ale nahrávají se postupně, aby zvuková data zbytečně nezabírala příliš paměti (hodí se např. pro hudbu). Z toho ale plyne, že nemůže být používán více zdroji najednou, nebo alespoň by neměl. Zde je potřeba obejít *reference counting*, toho je docíleno tak, že každý zdroj zvuku má tento soubor pojmenovaný vlastním identifikátorem, je ale potřeba si dát pozor a nevytvářet více instancí tohoto zdroje zvuku. Zdrojem zvukových dat je zde třída `DynamicFileWaveReader`.
- „*BufferIn*“ je buffer pro zvuková data z vnějšku audio enginu. Je podobý „*FileOneUse*“ s dvěma rozdíly. První je zdroj, zde to není soubor ale přímo zvuková data dodaná z vnějšku audio enginu. Druhý rozdíl je, že nemá definovanou délku zvukových dat. Zdrojem (bufferem) zvukových dat je třída `BufferIn`. K plnění dat bufferu je ve třídě `IODataProvider` metoda `FillBufferIn`, kde se mají dodávat data v definovaném WAV formátu. Jedná se o experimentální implementaci!
- „*BufferOut*“ je buffer pro zvuková data, jež jsou produkována audio enginem, tato data lze tedy použít jako opětovný vstup zvukových dat. Je podobný „*BufferIn*“ s rozdílem, že zde se data dodávají kontinuálně a nehrozí výpadek zvukových dat. Zdrojem (bufferem) zvukových dat je třída `BufferOut`. Zvuková data pak jsou audio enginem dodávána pomocí metody `FillBufferOut`. Tento typ zdroje zvuku navíc umožňuje posílat data zpět hře a to pomocí delegáta `BufferOutMethod`, jež se specifikuje při vytváření audio enginu. Také se jedná o experimentální implementaci!

4.3 Assembly ASS_Common

Assembly *ASS_Common* obsahuje prvky, jež se využívají v knihovnách audio enginu a u nichž je dobré je mít definované na jednom místě, abychom nemuseli z důvodu více implementací zajišťovat konverzi dat.

Nejvíce používaným prvkem, jež má přímou souvislost s hrou, je struktura `Time`. Tato struktura v sobě nese informace o uběhlém čase hry od jejího spuštění

a uběhlém čase mezi voláními metody `Update` audio enginu, k čemuž je tato struktura především určena.

Assembly ASS_*Common* také obsahuje rozhraní `IWaveReader`, jež definuje metody pro načítání zvukových dat ze zvukových souborů. Spolu s tímto rozhraním je v této assembly struktura `WaveFormatSpec`, jež slouží k uchovávání informací o formátu načítaného zvukového souboru typu WAV. Pro práci s pozicí zvukových dat tato struktura obsahuje pomocné metody `ConvertMilisecondToPosition` a `ConvertPositionToMilisecond`.

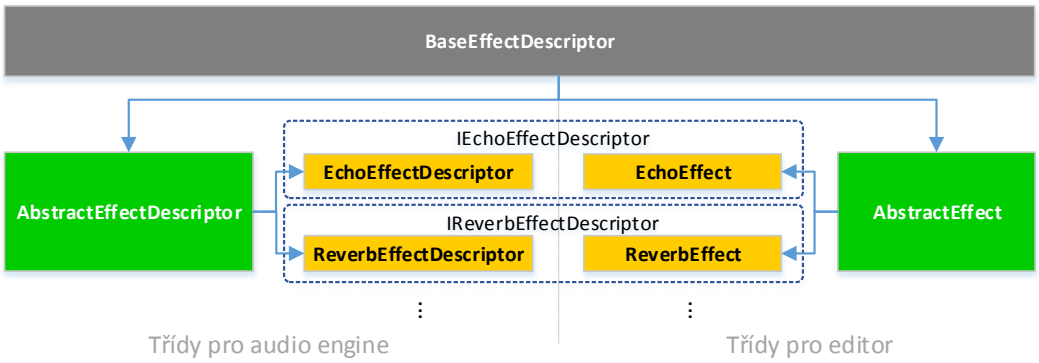
Další částí jsou pomocné metody definované ve statické třídě `CommonMethods`. Metoda `Clamp` slouží k omezení hodnoty do určitého rozsahu, tu poté využíváme především při nastavování hodnot parametrům efektů, neboť hodnoty parametru jsou zdola a shora omezeny. Metoda `GetRelativePath` pak slouží k získání relativní cesty mezi dvěma soubory. Tato metoda se používá především v editoru k následnému sestavení souborů s popisy jednotlivých zvuků.

Součástí assembly je dále třída `NameGroup`, jež slouží jako jmenný prostor, jež umožňuje vytvářet unikátní názvy. K tomu slouží základní tři metody: `NewName`, `ChangeName` a `RemoveName`. Přičemž první dvě vracejí unikátní název v rámci jmenného prostoru k požadovanému novému názvu. Pro lepší práci s touto třídou a využívání více nezávislých jmenných prostorů zde máme statickou metodu `GetAssignedNameGroup`, jež nám podle objektu a typu vrátí (či vytvoří novou) instanci třídy `NameGroup`, s kterou následně můžeme pracovat.

Struktury pro načítání zdrojů zvuků a posluchačů

Poslední a největší částí assembly *ASS_Common* jsou rozhraní a třídy pro načítání souborů popisujících zdroje zvuku a posluchače. Jsou součástí assembly *ASS_Common* z důvodu, že data zdrojů zvuků a posluchačů mají stejný základ jak pro audio engine, tak pro editor a tudíž aby se nemuseli pro ně zvlášť definovat.

Třídy pro popis zdrojů zvuků a posluchačů pak tvoří hierarchickou strukturu. Toho jsme využili pro návrh rozhraní a tříd pro načítání těchto popisů zdrojů zvuků a posluchačů. V této assembly jsme si tedy definovali základní třídy pro popis zdrojů zvuků a posluchačů, tyto třídy se jmenují `Base*`. Dále pak pro jednotlivé druhy těchto tříd jsme specifikovali různá rozhraní. Popisem části zdroje zvuku je pak třída, která v assembly audio enginu či editoru dědí od třídy `Base*` a daného rozhraní, viz obrázek *4.14* s příkladem tříd a rozhraní pro efekty.



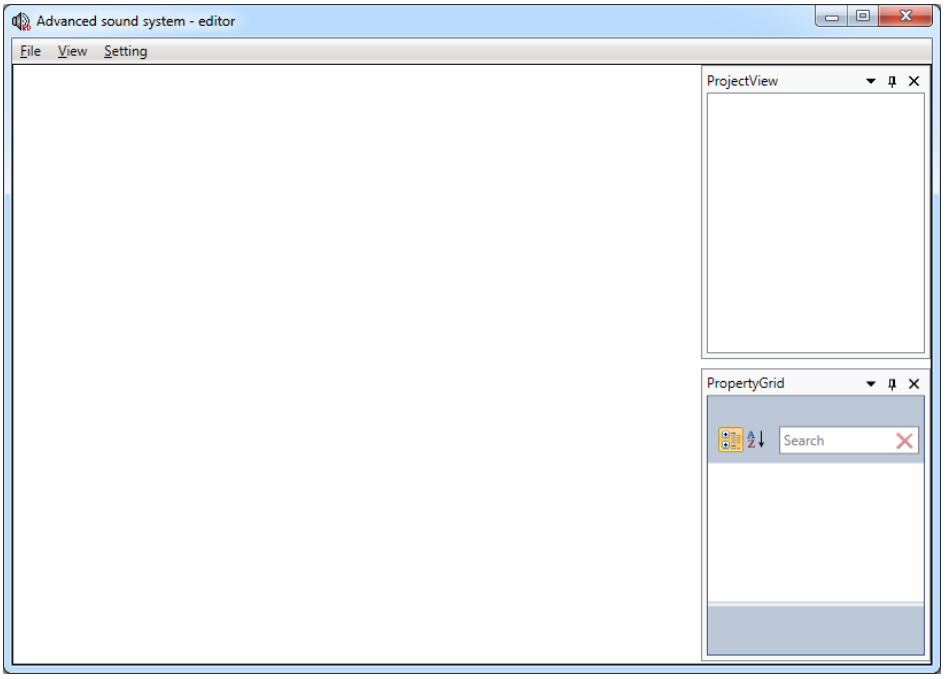
Obrázek 4.14 - Příklad hierarchie tříd a rozhraní pro načítání efektů

K načítání těchto popisů zdrojů zvuků a posluchačů jsme použili návrhový vzor tzv. továrnu (factory pattern), neboť potřebujeme zajistit, aby se v audio enginu vytvářely třídy pro audio engine a v editoru, aby se vytvářely třídy pro editor. Máme zde třídy `*Loader`, jež jsou u každé úrovně dat popisů zdrojů zvuků a posluchačů. Tyto třídy slouží k načítání tříd dané úrovně dat. Při jejím vytváření se jako parametr předá instance s rozhraním `I*Factory`, jež nám bude vytvářet instance tříd dané úrovně pro audio engine, či editor. Pro načítání tříd dané úrovně pak na třídě `*Loader` slouží metoda `Load`, jež vytváří pomocí třídy `I*Factory` třídy dané úrovně, dále načítá vnořenou úroveň dat (pokud existuje), následně načítá data specifická pro daná rozhraní úrovně dat a nakonec načítá data specifická pro danou `I*Factory` třídu.

5 Vývojová dokumentace B – editor

V této kapitole si popíšeme implementaci editoru pro audio engine *Advanced Sound System*. Konkrétně se jedná o assembly *ASS_Editor.exe*, jež je výstupem projektu *ASS_Editor* v solution s audio enginem.

Základní myšlenkou pro vznik editoru bylo usnadnění vytváření zvuků pro náš audio engine. Chtěli jsme, aby zvuky mohli navrhovat designéři pro to určení, proto náš editor je navrhnutý ve visuálním stylu, viz obrázek *5.1*. Designéři tedy nadefinují pomocí visuálních prvků dané zvuky pro danou aplikaci a editor pak uloží informace do souborů, jež jsou pak načítány naším audio enginem. Detailní pohled okna a jednotlivých podčástí je uveden v uživatelské dokumentaci v kapitole *7 Uživatelská dokumentace B – editor*. Zde se tedy zaměříme na implementaci jednotlivých celků.



Obrázek 5.1 - Základní okno editoru

Náš editor se skládá z několika částí, proto jsme hledali vhodné rozmístění prvků v okně editoru. Pro zobrazování jsme použili *AvalonDock* [19]. Ten nám umožní vytvářet a umisťovat panely a okna v okně editoru.

Základní okno editoru je tvořeno třídou `MainWindow`. V této třídě je pak pomocí *XAMLu* definován *AvalonDock* a menu editoru. Třída sama pak obsahuje metody pro zpracování menu.

5.1 Okna a panely

Samotná práce v editoru bude probíhat pomocí oken a panelů. Pro práci s okny jsme si vytvořili třídu `WindowManager`, jež nám bude zajišťovat umístění oken v *AvalonDocku* a také zajistí případné přejmenování oken v případě potřeby.

Součástí práce s okny jsou také rozhraní `IWindowOwner` a `IWindowContent`, jež definují metody, které informují dané prvky o tom, že uživatel zavřel jejich okno.

Další důležitou částí je panel *PropertyGrid* [20] (tento panel můžeme znát i ze samotného Visual Studia). Panel slouží k nastavování příslušných hodnot k daným objektům.

Nestačily nám ale základní upravovatelné položky a potřebovali jsme si vytvořit vlastní. Všechny třídy rozšíření *PropertyGridu* jsme umístili do složky projektu *Common/PropertyGrid*. Rozšíření *PropertyGridu* pak vypadá následovně:

- Rozhraní, jež třídě editované v *PropertyGridu* definuje metodu pro získání editoru dané položky.
- Třída, která dědí od rozhraní `Xceed.Wpf.Toolkit.PropertyGrid.Editors.ITypeEditor` a jež definuje metodu `ResolveEditor`, v níž zavolá metodu definovanou rozhraním na editované třídě.
- Třída s vlastním editorem, jež dědí od `UserControl`. V této třídě pak řešíme vlastní editaci položky, přičemž provázání s proměnnou daného objektu řešíme pomocí třídy `System.Window.Data.Binding`.

Tyto tři prvky pak zajišťují, abychom mohli v *PropertyGridu* editovat položky pomocí vlastního editoru. Pokud nám nevadí, že budeme při každém přiřazení třídy k editaci do *PropertyGridu*, tak se můžeme obejít bez rozhraní a ve třídě dědící od `ITypeEditor` můžeme vytvářet novou instanci editoru.

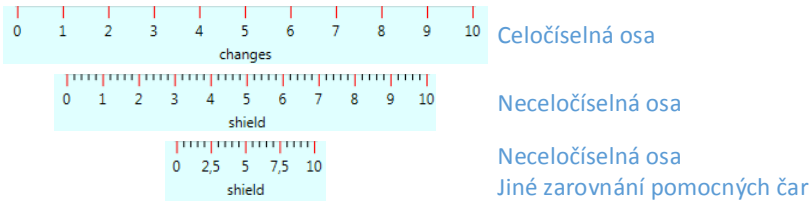
5.2 UserControl Axis

V editoru využíváme různé prvky, jež v některých případech vyžadují znázornění číselné osy, ať už se jedná o osu s hodnotami, či časovou osu. Vytvořili jsme si tedy vlastní prvek, jež je reprezentovaný třídou `Axis` ve jmenném prostoru `Common`.

Osu jsme přizpůsobili tak, aby mohla být na jakékoliv straně nějakého prvku (nahoře, dole, vlevo a vpravo). Dále lze definovat, zda může nabývat celočíselných či neceločíselných hodnot. Pro zjištění nejbližší možné hodnoty k požadované hodnotě pak třída definuje tyto metody:

- `GetAvailableValue` – vrátí nejbližší dostupnou hodnotu zadané hodnoty,
- `HasNextAvailableValue` – vrátí, zda je dostupná následující hodnota,
- `GetNextAvailableValue` – vrátí další dostupnou hodnotu,
- `HasPreviousAvailableValue` – vrátí, zda je dostupná předcházející hodnota,
- `GetPreviousAvailableValue` – vrátí předcházející dostupnou hodnotu.

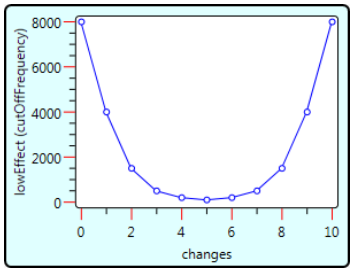
Pro určení minimální velikosti prvku osy pak využíváme metodu `MeasureOverride`, v níž podle orientace osy vypočítáme velikost jejího zobrazení. Pro samotné vykreslování používáme metodu `DrawLine`, jež zajišťuje vykreslení jednotlivých čar a popisků časové osy v závislosti na její orientaci a typu, viz obrázek *5.2*.



Obrázek 5.2 - Přehled zobrazení UserControl Axis

5.3 UserControl GraphEditor

V rámci nastavování zvuků jsme potřebovali vytvořit visuální prvek pro konverzi hodnoty z jednoho rozsahu hodnot do jiného rozsahu hodnot, využívané především pro výpočet určité hodnoty efektu podle hodnoty parametru. Pro tento účel jsme si vytvořili prvek stylizovaný do podoby 2D grafu (viz obrázek *5.3*), jež převádí hodnotu X na hodnotu Y, neboť data pro konverzi hodnot jsou uložena v podobě čar grafu (dále budeme tuto datovou strukturu nazývat grafem). Tento prvek je reprezentovaný třídou `GraphEditor`. Většina tříd pro tento prvek je umístěna ve složce projektu *GraphEditor* a stejnojmenném jmenném prostoru `GraphEditor`.



Obrázek 5.3 - UserControl GraphEditor

Každý prvek (třída `GraphEditor`) má definovaný rozsah osy X a osy Y (využívající pro definici os třídu `Axis`) v závislosti na editovaném grafu (hlavním grafu). Kromě hlavního grafu, jenž definuje rozsah editoru, tak umožňuje třída přidávat i vedlejší grafy, jež nám v určitých případech ulehčí nastavování hlavního grafu v závislosti na vedlejších, nicméně jdou upravovat i zobrazené hodnoty vedlejších grafů. Kromě grafů ještě umožňuje zobrazení vodicích čar, jež ulehčují editování jednotlivých grafů.

Jednotlivé grafy (seznamy čar grafu) jsou reprezentovány třídou `GraphLine`. Tato třída slouží pro propojení dat reprezentující graf s visuálními prvky editoru. Kromě napojení na samotný editor si třída `GraphLine` udržuje seznam segmentů čáry grafu vycházejících ze třídy `GraphVisualLine`, přičemž zároveň řeší jejich přidávání a odebírání z grafu.

Segmenty grafu vycházející z třídy `GraphVisualLine` slouží k jejich vykreslování v grafu, k jejich manipulaci a k nastavování v *PropertyGridu*. Pro vykreslování segmentu čáry třída definuje metodu `UpdatePosition`, kde

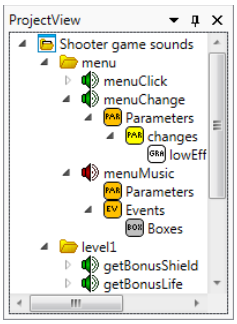
v podtřídách nastavíme vykreslení čáry v závislosti na nastavených hodnotách segmentu a případně i v závislosti na okolních segmentech sousedících s tímto segmentem. Další částí třídy `GraphVisualLine` jsou metody pro grafickou editaci segmentu, kde využíváme řadu metod pro manipulaci se segmentem:

- metoda `segment_MouseEnter` – nastaví zvýraznění segmenu,
- metoda `segment_MouseLeave` – odnastaví zvýraznění segmentu,
- metody `point_MouseRightDown` a `point_MouseRightUp` – při dvojkliku pravým tlačítkem myši na bod segmentu tento segment odeberou,
- metody `point_MouseLeftDown`, `path_MouseLeftDown`, `segment_MouseMove` a `segment_MouseLeftUp` – starají se o vytvoření nového segmenu při dvojkliku levého tlačítka myši na daný segment, případně přesouvají počáteční bod segmentu.

Pro nastavování hodnot v *PropertyGridu* třída `GraphVisualLine` definuje metody `SetToPropertyGrid` a `GetPropertyDefinitions`. Metoda `SetToPropertyGrid` slouží k nastavení editovaného prvku do *PropertyGridu*. Metoda `SetPropertyDefinitions` pak definuje jaké položky má *PropertyGrid* obsahovat pro editaci.

5.4 UserControl ProjectView

Základní práce s editorem probíhá skrze projekty, kde každý projekt obsahuje definované načítací celky a zvuky. Vytvořili jsme si tedy panel *ProjectView* viz obrázek *5.4*. Tento panel obsahuje hierarchický strom položek pro práci s projektem a jeho zvuky. Většina tříd pro panel *ProjectView* je umístěna ve složce projektu *ProjectView* a stejně pojmenovaném jmenném prostoru `ProjectView`.



Obrázek 5.4 - UserControl ProjectView

Samotný panel je reprezentovaný třídou `ProjectView`. Jednotlivé prvky hierarchického zobrazení v panelu jsou definovány v *XAMLu* pomocí třídy `System.Windows.HierarchicalDataTempate`. Tuto třídu pak propojujeme s třídou obsahující data pomocí pomocné třídy `*ViewModel`, jež se stará o manipulaci s prvky v panelu `ProjectView` a nastavuje potřebná data třídě obsahující daná data, viz obrázek *5.5*.

Obrázek 5.5 - Vztahy mezi třídami editoru v UserControl ProjectView

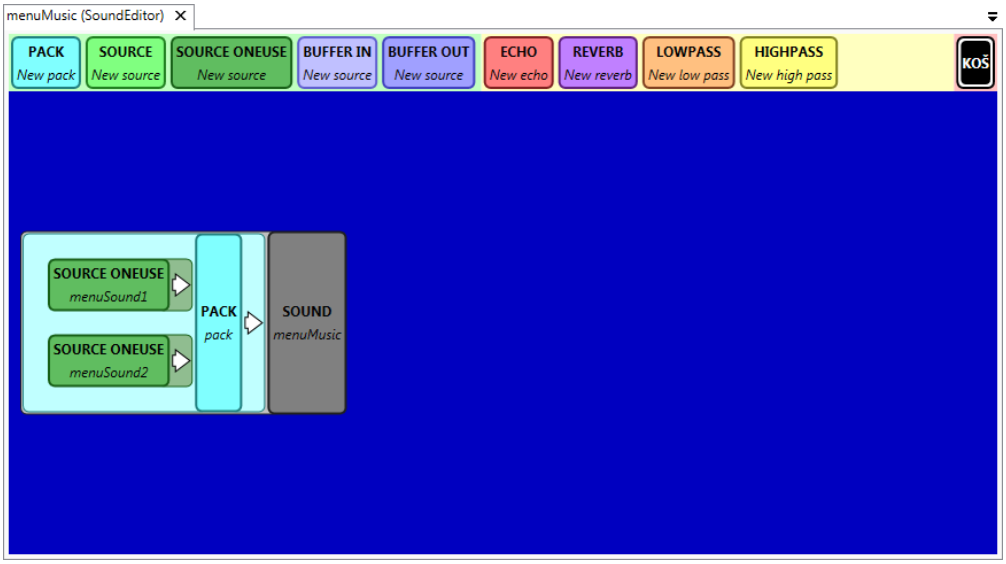
Všechny třídy `*ViewModel` dědí od třídy `TreeViewModel`, jež obsahuje položky pro umístění prvku v hierarchii panelu (rodičovský prvek, seznam potomků). Dále třída `TreeViewModel` pro práci s *PropertyGridem* definuje dvě metody, metodu `SetToPropertyGrid` a metodu `SetPropertyDefinitions` (podobně jako *GraphEditoru* viz výše *UserControl GraphEditor*).

Pro definování kontextového menu pak využíváme třídu `System.Windows.Controls.ContextMenu`. Vlastní definování kontextového menu a reakce na akce kontextového menu pak obstarává každá třída zvlášť v konstruktoru dané třídy.

Další věcí, co jsme museli řešit, bylo, že některé prvky tohoto panelu umožňují otvírat další okna v *AvalonDocku*. Pak se mohlo stát, že jsme prvek z panelu odebrali, ale zůstalo nám okno daného prvku v *AvalonDocku*, jež nám už k ničemu nebylo, neboť nebylo provázané s žádným prvkem v panelu. Tudíž když tento prvek odebíráme z panelu, tak musíme zkontrolovat, zda není otevřené jeho okno a případně toto okno zavřít.

5.5 UserControl SoundEditor

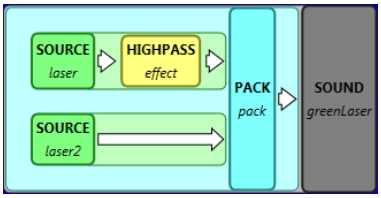
Pro editaci zvuku jsme navrhli visuální prvek (viz obrázek 5.6) reprezentovaný třídou `SoundEditor` implementující `IWindowContent` pro použití pomocí *AvalonDocku*. Většina tříd prvku je umístěna ve složce projektu *SoundEditor* a stejně pojmenovaném jmenném prostoru `SoundEditor`.



Obrázek 5.6 - UserControl SoundEditor

Jedná se o editor pro editaci zvuku, z jakých prvků se zvuk bude skládat (zdroje zvukových dat a efekty). Rozhodli jsme se to pojmout ve stylu průběhu zpracování zvukových dat, kde jednotlivé prvky editoru jsou zdrojové efekty a ostatní efekty.

Průběh zpracování zvukových dat necháme plynout zleva doprava (znázorněno pomocí šipek), kde nejpravější prvek bude symbolizovat výstup daného zvuku, viz obrázek *5.7*.



Obrázek 5.7 - Plynutí zvukových dat

Každý prvek editoru je zobrazovaný jako obdélník, který kromě hlavní části může mít i levou, či pravou část. Levá slouží ke vkládání prvků dodávajících zvuková data, tedy zdrojových efektů a pack efektů (ten jsme sem zařadili, neboť se jedná o speciální efekt). Pravá pak slouží pro vkládání efektů, jimiž se zpracuje daný proud zvukových dat generovaný zdrojovým efektem či pack efektem.

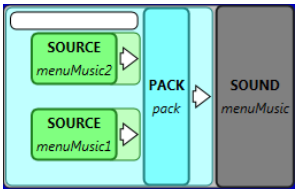
Zdrojové efekty (mezi které zde řadíme i pack efekt) vycházejí ze třídy `PatternAudioSourceComplete`, jež kombinuje všechny tři části dohromady, kde hlavní část je reprezentovaná samotnou třídou `PatternAudioSourceComplete`, neboť vychází ze třídy `PatternAudioSource`. Levá část je reprezentovaná třídou `AudioSourceIn` a pravá část je reprezentovaná třídou `AudioSourceOut`. Ostatní efekty pak vycházejí ze třídy `PatternAudioEffect`.

Třídy `PatternAudioSource` a `PatternAudioEffect` definují metody `SetToPropertyGrid` a `GetPropertyDefinitions` pro nastavení políček v *PropertyGridu* po kliknutí na daný prvek. Třídy prvků dědících od těchto tříd jsou pak napojeny na datové třídy pro ukládání nastavených dat.

Tyto prvky jsme potřebovali někde definovat, abychom mohli vytvářet nové instance těchto prvků. Vytvořili jsme si tedy obdélníkové prvky, jež představují definice nových prvků a umístili je do horní části okna, kde vlevo jsou zdrojové efekty a vpravo jsou ostatní efekty a úplně vpravo je koš pro mazání efektů. Vytváření nových prvků pak probíhá přesunutím definice prvku pomocí myši na místo, kde chceme vytvořit nový prvek. Definice prvků vycházejí ze tříd `PatternAudioSourceDEF` pro nové zdrojové efekty a ze třídy `PatternAudioEffectDEF` pro nové ostatní efekty. Tyto třídy pak při stisku tlačítka myši a pohybu předávají třídě `System.Windows.DragDrop` informace o přesouvaném prvku (nově vytvářeném prvku). Kromě vkládání nových prvků můžeme přesouvat i již vložené prvky, proto i třídy `PatternAudioSource` a `PatternAudioEffect` implementují přesun při stisku tlačítka myši a následném pohybu. Tyto prvky (nové či přesouvané) se pak vkládají do tříd `AudioSourceIn` nebo `AudioSourceOut` (levé, či pravé části obdélníku prvků) v závislosti na typu prvku. Třídy `AudioSourceIn` a `AudioSourceOut` pro účel přesouvání prvků implementují následující metody:

- metodu `*DragEnter` – vstup přenášeného prvku nad cílový objekt,
- metodu `*DragMove` – pohyb přenášeného prvku nad cílovým objektem,
- metodu `*DragLeave` – opuštění přenášeného prvku z cílového objektu,
- metodu `*OnDrop` – potvrzení vložení prvku do cílového objektu.

Kde metody `*DragEnter`, `*DragMove` a `*DragLeave` zpracovávají informace, že nad objektem se nachází přesouvaný prvek a těmito metodami nastavujeme, jak se zobrazují případné „našeptávače“ na jaké místo by se případný prvek vložil (viz obrázek *5.8*) a jak se má chovat myš (ikona přesunu, či kopírování prvku).



Obrázek 5.8 - Zobrazení „našeptávače“ při vkládání prvku

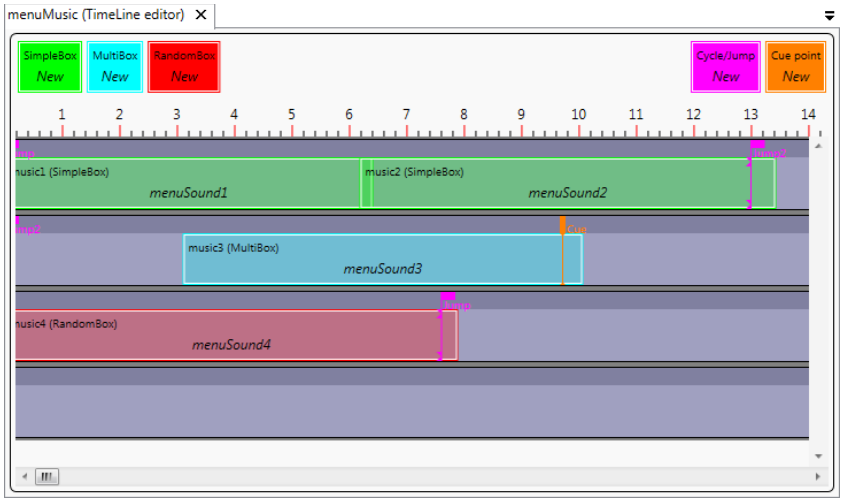
Metoda `*OnDrop` se pak stará o samotné vložení prvku na dané místo, přičemž v případě přesunu musí zajistit jeho odstranění z původního umístění. Toho pak využívá prvek koše, jenž prvek nikam nevkládá, ale jenom maže.

Implementaci vytváření a přesouvání prvků jsme se rozhodli řešit pomocí *Drag&Drop metody*, neboť jsme chtěli umožnit, aby bylo možné kopírovat, či přesouvat prvky, ze kterých je složen zvuk, mezi jednotlivými zvuky.

Kromě těchto prvků jsme si chtěli vytvořit pomocný prvek šipky pro znázorňování směru zpracování zvukových dat. Zde jsme ale řešili, jak nastavovat její šířku. Chtěli jsme, aby se nastavila podle dostupné šířky, ale pokud se dostupná šířka zmenší, tak jí chceme také zmenšit (nechceme, aby ona sama zabírala větší prostor, než je nezbytné). Pro tento účel jsme přepsali metody `MeasureOverride` a `ArrangeOverride`. V metodě `MeasureOverride` definujeme šířku šipky, tudíž nastavíme, že zabírá velmi málo prostoru a v metodě `ArrangeOverride` pak dostaneme dostupnou velikost prostoru pro naši šipku.

5.6 UserControl TimeLines

Potřebovali jsme zpracovat editaci časových os, pomocí nichž bychom řídili přehrávání jednotlivých zdrojových efektů, z kterých je zvuk složen. Pro tento účel jsme si vytvořili prvek reprezentovaný třídou `TimeLinesEditor` viz obrázek *5.9*. Většina tříd prvku je umístěna ve složce projektu `TimeLinesEditor` a stejně pojmenovaném jmenném prostoru `TimeLinesEditor`.



Obrázek 5.9 - UserControl TimeLines

Třída `TimeLinesEditor` se stará o vlastní manipulaci se všemi prvky. Definuje, že v editoru nahoře jsou umístěny prvky s definicemi prvků, jež můžeme umístit do časových os. Tyto prvky pak vycházejí ze třídy `BaseDefBox`, jež definuje jen jejich vzhled a slouží jen k odlišení různých prvků.

V dolní části pak třída `TimeLinesEditor` obsahuje instance tříd `TimeLineVisual`, jež představují jednotlivé časové osy zobrazované odshora (první časová osa) dolu (poslední časová osa). Třída `TimeLineVisual` se pak stará o ukládání dat jednotlivých prvků datové třídě představující časovou osu. Třída `TimeLinesEditor` se stará o jejich údržbu ve smyslu, že po každé manipulaci s prvky časových os tyto prvky zkontroluje a odstraní z konce postupně všechny prázdné časové osy, přičemž tam nechá, případně vytvoří jednu prázdnou a tím řešíme přidávání časových os v editoru.

Jednotlivé prvky časové osy pak implementují rozhraní `ITimeLineElement`, jež obsahuje metody pro jejich vkládání a posouvání po časové ose. Součástí těchto prvků je pak prvek skoku, ten se od ostatních liší tím, že je složen ze dvou částí, samotného skoku a cíle skoku. Tyto části jsou vlastně definované jako samostatné prvky (třídy `JumpPoint` a `JumpTargetPoint`), s rozdílem, že jsou provázané dohromady. Hlavní částí tohoto prvku je třída `JumpPoint`.

Přesouvání prvků pak probíhá tak, že po stisku levého tlačítka myši na definici prvku si třída `TimeLinesEditor` vytvoří instanci příslušného prvku. Při pohybu po editoru pak aktualizuje umístění prvku v jednotlivých časových osách, případně odebere z časových os, pokud se kurzor nachází mimo časové osy. Po uvolnění tlačítka pak potvrdí umístění příslušného prvku (případně prvek vymaže). S tím jsme museli řešit posouvání části okna časových os, neboť vidíme jen určitý výřez časových os. Zaregistrovali jsme si tedy metodu pro pohyb myši ve výřezu okna `scrollView_MouseMove`, v této metodě řešíme pozici myši vzhledem k výřezu. Pokud je pozice myši blízko stěny výřezu, tak nastavíme proměnné na pohyb obsahu výřezu požadovaným směrem a spustíme timer `dispatcherTimer`, jež volá metodu ve vlákně vykreslování okna a zajistí posun obsahu výřezu daným směrem. Timer `dispatcherTimer` pak zastavíme při opuštění výřezu. Posun se též zastaví při uvolnění tlačítka myši (timer `dispatcherTimer` v tomto případě ale běží dál, což nám ale nevadí).

6 Uživatelská dokumentace A – engine

V této dokumentaci si projdeme všechny kroky, jež jsou potřeba k přehrávání zvuku ve hře pomocí našeho audio enginu. Konkrétně si přehrávání zvuku předvedeme na hře *„Shooter game“*, kterou jsme pro tento účel vytvořili.

Potřebné složky a soubory

Na DVD jsou všechny soubory, jež jsou potřebné pro tuto ukázku:

- Složka `Ukazka_hra\Shooter_game_v1` – obsahuje solution hry (do Visual Studia 2010) bez implementace zvuku
- Složka `Ukazka_hra\Shooter_game_v2_base` – obsahuje solution hry (do Visual Studia 2010) se zapojeným audio enginem (zvuk není implementován)
- Složka `Ukazka_hra\Shooter_game_v3_menu` – obsahuje solution hry (do Visual Studia 2010) s implementovanými zvuky menu
- Složka `Ukazka_hra\Shooter_game_v4_level_base` – obsahuje solution hry (do Visual Studia 2010) s implementovanými základními zvuky herních úrovní
- Složka `Ukazka_hra\Shooter_game_v5_flying_objects` – obsahuje solution hry (do Visual Studia 2010) s předpřipraveným kódem pro implementaci zvuků létajících objektů
- Složka `Ukazka_hra\Shooter_game_v6_final` – obsahuje solution hry (do Visual Studia 2010) s hotovou implementací zvuků
- Složka `Ukazka_hra\Zvuky` – obsahuje složkovou strukturu se zvukovými soubory a soubory popisujícími jednotlivé zvuky
- Složka `Ukazka_hra\Zvukova_data` – obsahuje soubor se třídou MusicData
- Složka `Ukazka_hra\Knihovny` – obsahuje knihovny audio enginu

6.1 Popis hry pro zapojení audio enginu

Jedná se o klasickou vesmírnou střílečku podobnou hrám *„Ravage“* nebo *„Space Shooter Ultimate“*. V této hře hráč ovládá vesmírnou loď, která letí vesmírem. V jednotlivých herních úrovních na hráče útočí protivníci, přičemž na konci herní úrovně bývá speciální protivník, tzv. *boss*, jež má proti běžným protivníkům více života a lepší zbraně.

Ovládání hry

Hra obsahuje menu. V menu můžeme vybírat mezi jednotlivými položkami buď pomocí myši (stačí najet na tlačítko), pomocí kláves `W` a `S` anebo pomocí šipek *nahoru* a *dolu*. Pro potvrzení volby menu slouží *levé tlačítko myši*, případně klávesa `Enter`, nebo `Space`. Pro ukončení hry bez vybrání položky menu pro ukončení hry můžete použít klávesu `Escape`.

Po spuštění hry se v levé části obrazovky objeví vesmírná loď, kterou můžete posunovat ve vertikálním směru a to za pomoci kláves `W` a `S`, anebo pomocí šipek

nahoru a *dolu*. Pro střílení pak slouží klávesa `Enter` nebo `Space`. Pokud bude chtít hráč opustit předčasně hru, může stisknout klávesu `Escape` pomocí níž se vrátí do menu hry.

Zvuky hry

V této hře chceme ukázat implementaci přehrávání zvuků pomocí našeho audio enginu. Nyní si tedy popíšeme, jaké zvuky bychom chtěli, aby hra obsahovala.

Zvuky jsme definovali pomocí editoru, jenž je součástí této práce, konkrétní popis vytvoření zvuků je popsán v kapitole *7 Uživatelská dokumentace B – editor*. Níže je uveden seznam zvuků, jež chceme přidat do hry a co jednotlivé zvuky znamenají (kdy se přehrávají).

V menu budeme chtít tyto zvuky:

- `menuChange` – změna položky menu, zvuk se bude při další změně mírně lišit, použijeme parametr `change` s hodnotami od 0 do 10
- `menuClick` – potvrzení volby menu
- `menuMusic` – hudba na pozadí

Ve hře budeme chtít tyto zvuky:

- `levelEnd` – zvuk na konci herní úrovně a zvuk konce hry
- `playerShip` – zvuk obsahující zvuk ztráty života, zvuk zásahu nepřítelem (poškození lodi), a zvuk štítu, jež se mění podle hodnoty parametru *„shield“* (hodnoty štítu lodi)
- `playerShipEngine` – zvuk vesmírné lodě hráče
- `getBonusShield` – zvuk při sebrání bonusu štítu
- `getBonusLife` – zvuk při sebrání bonusu života
- `greenLaser` – zvuk výstřelu zeleného laseru
- `redLaser` – zvuk výstřelu červeného laseru
- `enemySound` – zvuk výbuchu nepřátelské lodi
- `level1Music`, `level2Music`, `level3Music` – hudba jednotlivých levelů, skládají se z hudby pro samotný level a hudby pro boj s bossem

Kód a třídy hry

Základní třídou naší hry je třída `MainWindow`, jež se stará o vykreslování a běh hry. K tomu využívá pomocné třídy a struktury, jež nejdůležitější z nich je třída `GameLoop`, která v cyklu (držícího se kolem 60 FPS) volá metody `Update` a `Draw` ve třídě `MainWindow` a tvoří tak herní smyčku.

Všechny další třídy reprezentující prvky hry implementují metody `Update` a `Draw`. V těchto metodách dochází k aktualizaci stavu třídy, konkrétně v `Update` metodě se aktualizuje logický stav třídy, zatímco `Draw` metoda slouží k aktualizaci vykreslování třídy. Z těchto metod jsou pak volány stejné metody vnořených tříd a tím dochází k aktualizaci všech částí hry.

Základními třídami hry jsou `Menu` a `Game`, jež jsou ve jmenných prostorech stejných názvů. Třída `Menu` slouží k vykreslování menu hry, přičemž jednotlivé

položky jsou reprezentovány třídou `MenuItem`. Třída `Game` pak obsahuje třídu `GameStats`, jež uchovává stav hráče (skóre, život) napříč herními úrovněmi. Dále se třída `Game` stará o načítání a běh herních úrovní reprezentovaných třídou `Level` (také ve jmenném prostoru `Game`). Třída `Level` pak obsahuje instance tříd odvozených od třídy `FlyingObject` ve jmenném prostoru `Game.FlyingObjects`, jež tvoří létající objekty ve hře. Třída `Level` se pak stará o jejich vytváření a destrukci (i kolizemi). Třídy `FlyingObject` mohou obsahovat třídy odvozené od třídy `ObjectPath` (ve jmenném prostoru `Game.FlyingObjects.Paths`) pro ovládání pozice objektu, či třídy typu `ObjectWeapon` (ve jmenném prostoru `Game.FlyingObjects.Weapons`) umožňující objektům střílet. Data k herním úrovním lze získat ze tříd definovaných ve jmenném prostoru `Game.Data`.

6.2 Zapojení audio enginu do projektu hry

Nyní si ukážeme zapojení audio enginu do projektu hry. Zkopírujeme si nejdříve solution hry bez audio enginu ze složky `Ukazka_hra\Shooter_game_v1` na DVD třeba na `C:\Shooter_game`. Složka s projektem hry tedy bude `C:\Shooter_game\Shooter_game`.

Přidání reference audio enginu

Abychom mohli používat náš audio engine, tak potřebujeme do projektu přidat knihovny audio enginu. To uděláme tak že zkopírujeme všechny knihovny audio enginu (které nalezneme ve složce `Ukazka_hra\Knihovny` na DVD), do složky `Libs` s projektem hry. Následně otevřeme projekt pomocí Visual Studia 2010 (postup pro Visual Studio 2012 je podobný). Zobrazíme si `Solution Explorer` a přidáme referenci na knihovny `ASS_Common` a `ASS_UpperLayer`, které jsme si zkopírovali do složky projektu. Od této chvíle můžeme využívat jmenné prostory `ASS_Common` a `ASS_UpperLayer` a jejich třídy.

Zapojení audio enginu do hry

Nyní si ve hře rozběhneme instanci audio enginu. Základní okno hry a jeho kód je napsaný v souboru `MainWindow.xaml.cs` v projektu hry. V tomto souboru si nejdříve deklarujeme, že budeme využívat jmenné prosotry `ASS_Common` a `ASS_UpperLayer`, to uděláme přidáním následujících řádků za poslední `using`:

```
using ASS_UpperLayer;
using ASS_Common;
```

Nyní už můžeme využívat třídy audio enginu bez prefixace jmennými prostory. Definujeme si tedy proměnnou `audioEnine` třídy `UpperLayer` na začátku třídy `MainWindow`, která bude reprezentovat instanci audio enginu. Nyní přejdeme na inicializaci audio enginu. Hra se inicializuje v konstruktoru třídy `MainWindow`, přidáme tedy následující kód pro inicializaci audio enginu (přidáme na začátek inicializace hry, neboť ostatní prvky budou možná využívat audio engine hned zpočátku):

```
InitializeComponent();

audioEngine = new UpperLayer(UpperLayer.GetAvailableChannels());

menu = new Menu.Menu();
```

Kde první parametr kontruktoru třídy `UpperLayer` určuje, na kolik reproduktorů bude směřován výstup z audio enginu, přičemž statická metoda `GetAvailableChannels` na třídě `UpperLayer` nám vrací počet reproduktorů, které máme k dispozici.

Audio engine nyní máme ve hře inicializovaný, nyní je ale potřeba ještě při ukončení hry audio engine hry odnačíst. To uděláme přidáním kódu do metody `Dispose`, jež se volá po ukončení běhu herní smyčky (už se neaktualizují herní prvky):

```
public void Dispose()
{
	isDisposing = true;

	if (audioEngine != null)
		audioEngine.Dispose();
	…
}
```

Nyní zbývá jen průběžně aktualizovat audio engine, toho docílíme přidáním kódu na konec `Update` metody. Kde proměnná `audioEngineTime` je struktura `Time` audio enginu, která definuje celkový a uběhlý (od posledního updatu) čas hry. Tuto proměnnou pak předáme metodě `Update` na instanci audio enginu.

```
	switch (state)
	{
		…
	}

	Time audioEngineTime = new Time(
		gameTime.totalTime,
		gameTime.elapsedTime
	);
	audioEngine.Update(audioEngineTime);
}
```

Přidání zdrojových souborů zvuků

K tomu, abychom mohli přehrávat zvuky, tak potřebujeme zdrojové zvukové soubory. Ty jsou předpřipraveny ve složce `Ukazka_hra\Zvuky` na DVD. Zkopírujeme tedy všechny soubory z této složky do složky `Content` projektu hry. Poté si v `Solution Exploreru` zobrazte všechny soubory (ikonka nahoře `Solution Exploreru`), vyberte složku *Sounds* a soubor `sounds.xml` ze složky `Content` a vložte je do projektu. Poté je ještě nutné nastavit všem vloženým souborům položku `Copy to Output Directory` na `Copy if newer` a jako `Build Action` nastavit, že se jedná o `Content`. Nyní můžeme v projektu hry využívat zvukové soubory.

První změna bude při vytváření instance audio enginu v konstruktoru třídy `MainWindow`, konkrétně přidáme parametr, jehož hodnotou bude cesta k souboru `sounds.xml`, který jsme přidali do obsahu hry. Díky tomuto budeme moci načítat zdrojová data podle jejich názvů. Konkrétní úprava kódu bude vypadat takto:

```
InitializeComponent();

audioEngine = new UpperLayer(
       UpperLayer.GetAvailableChannels(),
       System.IO.Directory.GetCurrentDirectory() + @"/Content/sounds.xml"
);

menu = new Menu.Menu();
```

Nyní už máme zapojený audio engine do hry, hru v tomto stavu můžete nalézt na DVD ve složce `Ukazka_hra\Shooter_game_v2_base`.

Zvuky menu

Nyní si vytvoříme zvuky menu, které následně budeme ve vhodnou chvíli přehrávat. Menu je tvořeno třídou `Menu`, ve které je i logika jeho ovládání. Abychom mohli ve třídě `Menu` používat náš audio engine, tak si upravíme konstruktor menu, který bude přijímat instanci audio enginu a následně v konstruktoru třídy `MainWindow` předáme konstuktoru třídy `Menu` instanci audio enginu, neboli vytvoření instance menu bude vypadat takto:

```
menu = new Menu.Menu(audioEngine);
```

Nyní už předáváme instanci audio enginu. Pro ovládání zvuků ale potřebujeme mít dostupné prvky, proto si vytvoříme proměnné, pomocí nichž budeme ovládat zvuky.

```
public partial class Menu : UserControl
    {
        ISource clickSound;
        ISource changeSound;
        ISource musicSound;

        public List<MenuItem> items;
```

Pro vytvoření instancí zvuků potřebujeme nejdřív načíst potřebné zdrojové soubory těchto zvuků. Zdrojové soubory načteme zavoláním metody `LoadResourceByName` na instanci audio enginu, které předáme jako parametr název zdrojů, zde je to název `menu`. Poté už můžeme získat instance zdrojů zvuků pomocí metody `GetSource`, jejíž první parametr je název zdroje zvuku a druhý parametr je název posluchače. Jako posluchače zde zvolíme `stereoListener`, který slouží k přehrávání výstupu na přední reproduktory, i když máme třeba 5.1 reproduktory. Do konstuktoru třídy menu vložíme na začátek za inicializaci komponent menu tyto řádky:

```
InitializeComponent();

audioEngine.LoadResourcesByName("menu");
this.clickSound = audioEngine.GetSource("menuClick", "stereoListener");
this.changeSound = audioEngine.GetSource("menuChange", "stereoListener");
this.musicSound = audioEngine.GetSource("menuMusic", "stereoListener");

changeParameterOfChangeSound = 0;
```

Nyní už můžeme přehrávat zvuky v menu, musíme je ale spouštět v ten správný čas. Zvuk pro kliknutí budeme chtít přehrát při kliknutí na *tlačítko myší*, případně při stisku klávesy `Enter`, `Space`, či `Escape`. Neboť se jedná o jednoduchý zvuk, tak přehrávání zvuku spustíme zavoláním metody `PlaySource` na instanci zvuku, kde parametr udává, od jakého času chceme zvuk spustit. Tento zvuk budeme chtít vždy přehrávat od začátku, tak jako parametr zadáme `0`. Konkrétně přidáme kód na tři místa, jež provádějí dané operace (všechny jsou v metodě `Update` třídy `Menu`), neboli první místo je při kliknutí tlačítka myši:

```
if (mouseClick != -1)
{
        …
        clickSound.PlaySource(0);
}
```

Dalším místem je stisknutí klávesy `Enter`, či `Space`:

```
if (
        GameKeyboard.IsKeyPressed(Key.Enter) ||
        GameKeyboard.IsKeyPressed(Key.Space)
)
{
        …
        clickSound.PlaySource(0);
}
```

A posledním místem je stisknutí klávesy `Escape`:

```
if (GameKeyboard.IsKeyPressed(Key.Escape))
{
	…
	clickSound.PlaySource(0);
}
```

Dalším zvukem je změna položky menu, jež chceme přehrávat při změně položky menu. U tohoto zvuku ale chceme docílit, aby zněl při každé změně jinak, přičemž k tomu máme na zvuku definovaný parametr `change` s povolenými hodnotami od `0` do `10`. Proto si vytvoříme integerovou proměnnou `changeParameterOfChangeSound`, kterou budeme při každé změně položky menu zvyšovat o `1`. Touto proměnnou pak budeme nastavovat parametr `change` zvuku. To uděláme zavoláním metody `SetParameters` na instanci zvuku, jejíž první parametr je název parametru a druhý parameter je hodnota parametru. Pro nastavení parametru a spuštění zvuku přidáme následující kód do metody `UpdateSelected` třídy `Menu`, jež se stará o změnu položky menu.

```
protected void UpdateSelected()
{
	…

	changeParameterOfChangeSound++;
	if (changeParameterOfChangeSound > 10)
		changeParameterOfChangeSound = 0;
	changeSound.SetParameters("change", changeParameterOfChangeSound);
	changeSound.PlaySource(0);
}
```

V menu nám zbývá ještě zvuk, jenž zajišťuje přehrávání hudby. Ten bude o něco složitější, neboť budeme chtít, aby se hudba při náběhu postupně zesílila a při kliknutí na tlačítko menu, aby se postupně ztlumila. Kód přehrávání hudby je tedy rozdělen na tři části. První částí je puštění zvuku, ten budeme pouštět, když se menu poprvé aktualizuje (proběhne poprvé `Update` metoda), neboť chceme zvuk spouštět opakovaně, když například skončíme hru a přejdeme zpět do menu. Kód spuštění přehrávání je umístěn v `Update` metodě. Přičemž je zde umístěna podmínka, že `processTime` (čas běhu událostí menu – zde čas zobrazení menu) je nulový (menu se zobrazilo) a vypadá takto:

```
if (clicked != -1)
{
	…
}
else
{
	if (processTime == 0)
	{
		musicSound.PlaySource(0);
	}
	if (time.elapsedTime.TotalSeconds < 0.2f)
```

Druhá část zajišťuje jeho postupné zesílení, které probíhá pomocí metody `SetParametersToEffect` na instanci zvuku hudby. Kód je obalen podmínkou, že probíhá jen první dvě sekundy zobrazení menu. Kód je dále rozdělen na část, kdy se hudba postupně zesiluje (do jedné sekundy zobrazení menu) a na část, kdy už je zvuk zesílen (nastavena normální hlasitost - 1):

```
if (clicked != -1)
{
	…
}
else
{
	…
	if (time.elapsedTime.TotalSeconds < 0.2f)
		processTime += time.elapsedTime.TotalSeconds;

	if (processTime < 2)
	{
		if (processTime < 1)
		{
			musicSound.SetParametersToEffect(
				musicSound.GetOutputEffectIndex(),
				(float)processTime
			);
		}
		else
		{
			musicSound.SetParametersToEffect(
				musicSound.GetOutputEffectIndex(),
				(float)1
			);
		}
	}

	if (mouseSelect != -1)
```

Poslední část se stará o postupné ztlumení zvuku a po uplynutí jedné sekundy zajišťuje zastavení přehrávání zvuku (aby se nepřehrával, když není slyšet), kde čas je reprezentovaný proměnnou `processTime` (čas od potvrzení volby menu):

```
if (clicked != -1)
{
	…
	if (processTime > 1)
	{
		musicSound.StopSource();
		…
	}
	else
	{
		musicSound.SetParametersToEffect(
			musicSound.GetOutputEffectIndex(),
			(float)(1 - processTime)
		);
	}
}
```

Nyní když zkompilujeme a spustíme hru, tak po zobrazení menu začne hrát hudba a při jeho ovládání (změna vybrané položky menu, potvrzení volby menu) slyšíme dané zvuky menu. Pro kontrolu si můžete porovnat kód se solution hry ve složce na DVD `Ukazka_hra\Shooter_game_v3_menu`, který by měl být ve stejném stavu.

Základní zvuky herních úrovní hry

Zvuky samotné hry se definují a ovládají podobně, jako jsme si ukázali výše na menu hry. Samotnou hru pak definuje třída `Game`, jenž se stará o vytváření a přechod mezi jednotlivými herními úrovněmi. Zpřístupníme si tedy audio engine i pro třídu `Game`.

V konstruktoru třídy `Game` si přidáme parametr pro instanci audio enginu, přičemž na začátku si tuto instanci audio enginu uložíme do proměnné, kterou jsme si pro ni připravili. Kód třídy `Game` bude vypadat takto:

```
class Game
{
        UpperLayer audioEngine;
        GameStats gameStats;
        …

        public Game(UpperLayer audioEngine, Canvas canvas)
        {
                this.audioEngine = audioEngine;

                gameLevel = 0;
```

Nesmíme ale zapomenout přepsat vytvoření instance třídy `Game` v `Update` metodě třídy `MainWindow` a to takto:

```
game = new Game.Game(audioEngine, canvas);
```

Nyní už můžeme používat audio engine ve třídě `Game`. Využijeme toho pro načítání zvukových dat, které budeme využívat ve všech herních úrovních. Jako jsme si popisovali v menu, tak využijeme metody `LoadResourceByName` k načtení dat. V konstruktoru třídy `Game` tedy načteme příslušná data, přidáme tedy následující řádek do konstruktoru třídy `Game`:

```
this.audioEngine = audioEngine;
audioEngine.LoadResourcesByName("levelCommon");

gameLevel = 0;
```

Na rozdíl od menu zde budeme chtít data i odnačítat, k tomu nám poslouží metoda `DestroyResourceByName`. Kód umístíme do metody `GetState`, neboť ta vrací status, zda hra skončila (vrací nenulové číslo), či ne. Do této metody napíšeme následující kus kódu:

```
if (returnState != 0)
{
	audioEngine.DestroyResourcesByName("levelCommon");
	int tempState = returnState;
```

Základní zvuková data nyní máme k dispozici po dobu běhu hry. Přesuneme se nyní dál a to do herních úrovní, jež jsou definována pomocí třídy `Level`. Ve třídě `Level` si zpřístupníme instanci audio enginu, jako jsme udělali výše u třídy `Game`, předáním parametru v konstuktoru (kód je podobný a tak ho přeskočíme). Dalším krokem je načtení a odnačítání zdrojových dat zvuků konkrétního levelu. To uděláme zavoláním metody `LoadResourceByName` a `DestroyResourceByName` jako ve třídě `Game`. Kód pro načítání dat levelu bude umístěn v konstruktoru třídy a bude vypadat následovně:

```
public Level(…)
{
	this.audioEngine = audioEngine;
	audioEngine.LoadResourcesByName("level" + levelNumber);

	this.canvas = canvas;
	…
}
```

Odnačítání dat pak bude probíhat v metodě `Unload`, jež je volána v případě destrukce třídy `Level`, vložíme tedy kód pro odnačítání zdrojových dat na začátek metody `Unload` ve třídě `Level`, podobně jako ve třídě `Game`.

Nyní máme zvuková data pro dannou herní úroveň dostupná a tak si vytvoříme instance zvuků herní úrovně, konkrétně hudby herní úrovně (zdroj zvuku `level{N}Music`, kde `{N}` je označení levelu) a zvuků, jež se přehrávají při úspěšném či neúspěšném (konci hry) dokončení herní úrovně (zdroj zvuku `levelEnd`). Potřebné proměnné si nadefinujeme na začátku třídy (jsou typu `ISource`). Kód umístíme do konstruktoru třídy `Level`:

```
audioEngine.LoadResourcesByName("level" + levelNumber);
levelEndSound = audioEngine.GetSource("levelEnd", "stereoListener");
levelMusic = audioEngine.GetSource(
	"level" + levelNumber + "Music",
	"stereoListener"
);

this.canvas = canvas;
```

Nejdříve zprovozníme zvuky při dokončení herní úrovně. Na rozdíl od zvuků menu, je tento zvuk odlišný a to tím, že obsahuje více zvuků v jedné instanci zvuku. Konkrétně obsahuje jeden zvuk pro úspěšné dokončení herní úrovně a jeden zvuk pro signalizaci konce hry. Díky tomu u tohoto zvuku neprobíhá spuštění zvuku

zavoláním metody `PlaySource` na instanci zvuku, ale zavoláním metody `DoEvent`, jež spustí událost na instanci zvuku, zde konkrétně spustí jen jeden zvuk, jež obsahuje instance zvuku.

Zvuk pro úspěšné dokončení herní úrovně budeme chtít spustit až po dokončení herní úrovně, vložíme tedy následující kód do metody `Update` třídy `Level`:

```
public void Update(WindowView gameView, GameTime gameTime)
{
	…
	if (levelData.Count == 0 && enemyObjects.Count == 0)
	{
		finish = true;
		levelEndSound.DoEvent("playWin");
	}
	…
}
```

Následně zvuk pro signalizaci konce hry budeme chtít přehrát, když bude zničena loď hráče a už hráč nebude mít žádný další život:

```
public void Update(WindowView gameView, GameTime gameTime)
{
	…
	if (player.SetDamage(enemies[i].GetDamage()))
	{
		end = true;
		player.Destroy();
		levelEndSound.DoEvent("playGameover");
	}
	…
}
```

Tento zvuk taktéž budeme chtít přehrát, když hráč ukončí hru stiskem klávesy `Escape`, takže vložíme stejný kód do bloku v `Update` metodě třídy `Level`:

```
if (GameKeyboard.IsKeyReleased(System.Windows.Input.Key.Escape))
{
	end = true;
	levelEndSound.DoEvent("playGameover");
}
```

Dále si ukážeme implementaci přehrávání hudby herní úrovně, nástup a zastavení bude podobné jako v případě hudby menu. Navíc od hudby menu zde bude možnost měnit téma hudby za pomoci volání události. Sice by šla hudba této hry naeditovat tak, že přechod by byl automatický po určité době, ale chceme ukázat možnost měnit téma hudby dynamicky za běhu aplikace. Konkrétně budeme ve hře měnit téma hudby herní úrovně ze základní hudby na hudbu souboje s bossem. Toho docílíme zavoláním události s názvem `bossMusic` na instanci zvuku hudby. K tomu ale budeme potřebovat data, kdy máme hudbu přepnout. Předpřipravenou třídu poskytující daná data najdeme v souboru `MusicData.cs` ve složce `Ukazka_hra\Zvukova_data` na DVD. Tento soubor zkopírujeme do projektu do složky `Game\Data` a ve Visual Studiu v `Solution Exploreru` tuto třídu přidáme do projektu. Data v souboru jsou následujícího formátu, kdy je použito

struktury `KeyValuePair`, kde klíčem je počet vteřin od začátku herní úrovně a hodnota je název události, jež se má volat na instanci hudby:

```
KeyValuePair<double, string>(35, "bossMusic")
```

Následně si ve třídě `Level` vytvoříme proměnnou `levelChangeMusicData` typu, jež vrací metoda `GetLevelMusic` třídy `MusicData` a do konstruktoru třídy `Level` vložíme následující kód:

```
…
levelTime = 0;
levelData = GameData.GetLevel(canvas, levelNumber);
levelChangeMusicData = MusicData.GetLevelMusic(levelNumber);
gameStats.SetLevel(levelNumber);
…
```

Pro přehrávání hudby herní úrovně pak vložíme do metody `Update` třídy `Level` následující kód, jež se stará o spuštění hudby, dále o nástup a ztišení hudby, podobně jako tomu bylo u hudby v menu s rozdílem, že hudba se začne ztišovat, když hráč úspěšně dokončí herní úroveň anebo prohraje:

```
public void Update(WindowView gameView, GameTime gameTime)
{
	…

	if (levelTime == 0)
	{
		levelMusic.PlaySource(0);
	}
	levelTime += gameTime.elapsedTime.TotalSeconds;
	if (end || finish)
	{
		if (stateTime < 1)
		{
			levelMusic.SetParametersToEffect(
				levelMusic.GetOutputEffectIndex(),
				(float)(1 - stateTime)
			);
		}
		else
		{
			levelMusic.SetParametersToEffect(
				levelMusic.GetOutputEffectIndex(),
				(float)(0)
			);
		}
		stateTime += gameTime.elapsedTime.TotalSeconds;
	}
	else
	{
		if (levelTime < 2)
		{
			if (levelTime < 1)
			{
				levelMusic.SetParametersToEffect(
					levelMusic.GetOutputEffectIndex(),
```

```
                                (float)levelTime
                        );
                }
                else
                {
                        levelMusic.SetParametersToEffect(
                                levelMusic.GetOutputEffectIndex(),
                                (float)1
                        );
                }
        }

        if (levelData.Count == 0 && enemyObjects.Count == 0)
```

Další částí je zde změna tématu hudby, ta je zajišťována kódem, kde v poli jsou uložena seřazená data o změně tématu, vždy testujeme první data, zda jejich čas odpovídá času herní úrovně. Pokud ano, tak se zavolá událost na instanci hudby herní úrovně s odpovídajícím názvem události uložené v datech.

```
public void Update(WindowView gameView, GameTime gameTime)
{
        …

        if (end || finish)
        {
                …
        }
        else
        {
                if (levelTime < 2)
                {
                        …
                }

                while (
                        levelChangeMusicData.Count > 0 &&
                        levelChangeMusicData[0].Key < levelTime
                )
                {
                        levelMusic.DoEvent(levelChangeMusicData[0].Value);
                        levelChangeMusicData.RemoveAt(0);
                }

                if (levelData.Count == 0 && enemyObjects.Count == 0)
```

Dále budeme chtít na konci herní úrovně hudbu zastavit, to uděláme zavoláním metody `StopSource` na instanci zvuku hudby. Kód umístíme do metody `GetState`, neboť ta vrací status, zda herní úroveň skončila (vrací jiné číslo, než je číslo levelu). Do této metody napíšeme následující kus kódu:

```
if (stateTime > 2)
{
        levelMusic.StopSource();
        App.Current.Dispatcher.Invoke(new Action(() =>
```

Nyní jsme ve stavu, kdy se nám přehrávají základní zvuky herních úrovní. Solution hry v tomto stavu naleznete ve složce `Ukazka_hra\Shooter_game_v4_level_base` na DVD.

Zvuky létajících objektů a hráče v herních úrovních hry

Nyní už máme ozvučen i základ herní úrovně. Dále si ozvučíme i zbytek herní úrovně, neboli ozvučíme si létající objekty a hráče v herních úrovních hry.

Létající objekty jsou ve hře reprezentovány třídou `FlyingObject` ve jmenném prostoru `Shooter_game.Game.FlyingObjects`, od které dědí další třídy reprezentující různé létající objekty hry. Některé létající objekty pak mohou obsahovat různé zbraně, které jsou reprezentované třídou `ObjectWeapon` ve jmenném prostoru `Shooter_game.Game.FlyingObjects.Weapons`. Jak létající objekty, tak zbraně mohou obsahovat zvuky. Proto přidáme do konstruktoru těchto objektů jako parametr instanci audio enginu, jež pak ve třídách `FlyingObject` a `ObjectWeapon` uložíme do proměnné audioEngine, kterou si v těchto třídách vytvoříme a jež bude dostupná i v poděděných třídách (musí být `protected`). Příklad si ukážeme na třídě `PlayerObject` a `FlyingObject`. Třída `PlayerObject` se upraví následujícím způsobem:

```
class PlayerObject : FlyingObject
{
	…

	public PlayerObject(
		Level level,
		UpperLayer audioEngine,
		Canvas canvas,
		GameDataPlayer description
	)
		: base(
			level,
			audioEngine,
			canvas,
			…
		)
	{
		…
```

Tyto třídy jsou ale tvořeny objekty (třídami), jež jsou definované v souborech `GameDataFlyingObject` a `GameDataWeapon`, jež jsou ve složce `Game\Data` projektu hry. Přičemž v souboru `GameDataFlyingObject` jsou definované létající objekty, jež dědí od abstraktní třídy `GameDataFlyingObject`, jež obsahuje abstraktní metodu `CreateObject`, pomocí níž vytváří tyto třídy instance tříd létajících objektů. V souboru `GameDataWeapon` je to obdobně, přičemž abstraktní třída `GameDataWeapon` obsahuje abstraktní třídu `CreateWeapon`. Těmto metodám přidáme jako parametr instanci třídy audio enginu. Ukážeme si to na metodě `CreateObject`, která by nyní měla vypadat následovně:

```
public abstract FlyingObject CreateObject(
	Level level,
	UpperLayer audioEngine,
	Canvas canvas
);
```

Přičemž nesmíme zapomenout předávat instanci audio enginu do konstruktorů tříd létajících objektů a konstruktorů zbraní. Nyní nám ještě zbývá předat instanci audio enginu při volání metod `CreateObject` a `CreateWeapon`, které se vyskytují v následujících třídách:

- `Level` – při vytváření létajících objektů (v metodě `Update`)
- `MeteorBigObject` – při destrukci, pomocí datové třídy `GameDataSmallMeteor` (v metodě `Destroy`)
- `PlayerObject` – při vytváření zbraně hráče (v konstruktoru)
- `EnemyObject` – při vytváření zbraně nepřítele (v konstuktoru)
- `EnemyWithShieldObject` – při vytváření zbraně nepřítele (v konstruktoru)

Posledním krokem této části je předání instance audio enginu při vytváření objektu laseru při výstřelu ze zbraně v metodě `Update` třídy `SimpleLaserWeapon`.

Těmito kroky jsme si zpřístupnili audio engine ve všech létajících objektech.

Pokud si nejste jistí správností vašeho kódu, můžete si porovnat kód se solution hry, který by měl být ve stejném stavu. Solution je uložen ve složce `Ukazka_hra\Shooter_game_v5_flying_objects` na DVD.

Začneme definováním zvuku zbraní. Ve hře máme dvě zbraně: zelený (používá hráč a nepřátelé) a červený laser (používají nepřátelé). Jedná se o jednoduchý laser, který je definován jednou třídou `SimpleLaserWeapon`, jediným zásadním rozdílem je použití jiné grafiky. Toho využijeme, neboli pro zelený laser budeme potřebovat vytvořit instanci zvuku výstřelu zeleného laseru a pro červený laser zvuk výstřelu červeného laseru. Využijeme k tomu metodu `GetSource` na instanci audio enginu, jako už jsme to dělali několikrát. Přičemž abychom mohli instanci zvuku využívat, tak si ji uložíme do proměnné `laserSound` typu `ISource`. Takže do konstruktoru laseru vložíme následující kód:

```
…
GameDataFlyingPath path;
ISource laserSound;

public SimpleLaserWeapon(…)
{
	…
	this.path = path;

	if (laserImage == "laserGreen")
	{
		laserSound = audioEngine.GetSource(
			"greenLaser",
			"stereoListener"
		);
	}
	else
	{
		laserSound = audioEngine.GetSource(
			"redLaser",
			"stereoListener"
		);
	}
}
```

Zvuk výstřelu je pak spouštěn událostí s názvem `fire`, takže tuto událost zavoláme v metodě `Update`, kde zpracováváme výstřel ze zbraně (vytváříme objekt laseru):

```
public override void Update(WindowView gameView, GameTime gameTime)
{
	base.Update(gameView, gameTime);
	if (fire == true && loadingTime > repeatTime)
	{
		App.Current.Dispatcher.Invoke(new Action(() =>
		{
			laserSound.DoEvent("fire");
			level.AddObject(…);
		}));
		…
	}
}
```

Dále si zprovozníme zvuky výbuchu nepřátelských lodí při jejich zničení. Konkrétně budeme upravovat třídy `EnemyWithShieldObject` a `EnemyObject`. Kódy, které budeme přidávat do těchto tříd, jsou stejné, takže si to ukážeme jen na třídě `EnemyObject` (má dva konstruktory z důvodu, že je to třída jak pro nepřátelskou běžnou loď, tak i UFO loď). Do obou konstruktorů třídy vložíme kód pro vytvoření instance zvuku nepřátelské lodi (obsahuje zvuk výbuchu), přičemž si musíme vytvořit proměnnou `enemySound` typu `ISource` pro uložení instance zvuku:

```
public EnemyObject(…)
	: base(…)
{
	…

	enemySound = audioEngine.GetSource("enemySound", "stereoListener");
}
```

Exploze lodi je pak spuštěna událostí `explode`. Kód pro spuštění výbuchu pak umístíme do metody `Destroy`, jež se stará o zničení lodi. Nicméně se stará i o zničení lodi, když projde nezničena až za hráče, tudíž spuštění přehrání zvuku výbuchu musíme obalit výjimkou, že nepřátelská loď je stále od levého okraje obrazovky hry. Jak jsme psali výše, totéž platí i pro třídu `EnemyWithShieldObject`.

```
public override void Destroy()
{
	base.Destroy();

	if (position.X > -objectImage.ActualWidth)
		enemySound.DoEvent("explode");
}
```

Dalšími zvuky jsou zvuky při sbírání bonusů. Týká se to tříd `BonusLifeObject` (objekt pro doplnění života) a `BonusShieldObject` (objekt pro nabití štítu). Kód v těchto třídách bude podobný s rozdílem, že jen načteme jiné zdroje zvuku. Pro `BonusLifeObject` to bude `getBonusLife` a pro třídu `BonusShieldObject` to bude `getBonusShield`. Ukážeme si to na třídě `BonusShieldObject`, kdy si přidáme proměnou pro instanci zvuku, do které přiřadíme instanci zvuku v konstruktoru třídy a v metodě `SetBonus` spustíme přehrávání zvuku, zde metodou `PlaySource` na instanci zvuku:

```
class BonusShieldObject : BonusObject
{
	protected int shieldValue;
	ISource setBonusSound;

	public BonusShieldObject(…)
		: base(…)
	{
		shieldValue = description.shieldValue;
		setBonusSound = audioEngine.GetSource(
			"getBonusShield",
			"stereoListener"
		);
	}

	public override void SetBonus(PlayerObject player, GameStats gameStats)
	{
		base.SetBonus(player, gameStats);
		player.SetShield(shieldValue);
		setBonusSound.PlaySource(0);
	}
}
```

Jako poslední nám zbývá ozvučit loď hráče, jež je reprezentovaná třídou `PlayerObject`. Loď bude vydávat několik zvuků jako: zvuk motoru, zvuk štítu, zvuk poškození, zvuk ztráty života. Zvuk motoru lodi má zvláštní instanci zvuku, neboť na tomto zvuku chceme využívat efektu, že když loď poletí nahoru (z pohledu lodi doleva) tak zvuk bude znít více z levého reproduktoru, když poletí dolu (z pohledu lodi doprava) tak zvuk bude znít více z pravého reproduktoru a když se nebude loď hýbat, tak bude znít z obou reproduktorů stejně. Zbylé zvuky (zvuk štítu, zvuk poškození a zvuk ztráty života) budou přehrávány z instance zvuku, který reprezentuje zvuky lodi.

Začneme opět vytvořením potřebných instancí zvuků. Vytvoříme si tedy proměnné pro jejich uložení v paměti:

```
class PlayerObject : FlyingObject
{
	ISource shipSound;
	ISource engineSound;

	protected int shield;
```

Dále si do těchto proměnných v konstruktoru třídy uložíme instance požadovaných zvuků:

```
public PlayerObject(…)
	: base(…)
{
	shipSound = audioEngine.GetSource(
		"playerShip",
		"stereoListener"
	);
	engineSound = audioEngine.GetSource(
		"playerShipEngine",
		"stereoListener"
	);

	objectPath = new PlayerPath(this);
	…
}
```

A poté se postaráme o jejich odnačtení pomocí metody `Destroy` v metodě `Unload`, neboť se mohou tyto zvuky přehrávat ve smyčce a potřebujeme je ukončit v momentě odnačtení třídy (u předchozích tříd se také mohli zvuky odnačítat, ale nechali jsme to na `Garbage Collection`):

```
public override void Unload()
{
	…

	shipSound.Destroy();
	engineSound.Destroy();
}
```

Instance zvuků už máme připravené, nyní je začneme přehrávát. Prvním bude zvuk motoru, ten pro jednoduchost spustíme už v konstuktoru třídy hned za načtením instance zvuku motoru zavoláním události `startEngine`:

```
	engineSound = audioEngine.GetSource(
		"playerShipEngine",
		"stereoListener"
	);
	engineSound.DoEvent("startEngine");

	objectPath = new PlayerPath(this);
```

Nyní přijde část, kdy budeme upravovat, že při pohybu lodi se na jednom reproduktoru zvuk zesílí (na úroveň `1.1`) a na druhém zeslabí (na úroveň `0.3`). K tomuto efektu využijeme výstupní efekt instance zvuku a to tak, že mu nastavíme hlasitost na jednotlivých zvukových kanálech zvuku. Vytvoříme si tedy instanci třídy `OutputEffParameters`, který reprezentuje parametry výstupního efektu (hlasitost jednotlivých kanálů). Při vytváření této třídy vložíme jako parametr konstruktoru číslo 2, což je počet kanálů, které chceme editovat (víme to, neboť používáme posluchače `stereoListener`, který směruje výstup na dva reproduktory). Následně

podle pohybu lodi nastavíme na této třídě hlasitost jednotlivých kanálů zvuku. Pak pokud není zvuk motoru odnačten (může se stát, že loď hráče bude zničena – tím i instance zvuku a proběhne ještě jednou update metoda), tak pomocí `SetParametersToEffect` nastavíme parametry výstupnímu efektu instance zvuku motoru lodi, přičemž index výstupního efektu získáme zavoláním metody `GetOutputEffectIndex` na instanci zvuku. Výsledný kód bude umístěn v `Update` metodě a bude vypadat takto:

```
public override void Update(WindowView gameView, GameTime gameTime)
{
	base.Update(gameView, gameTime);

	gameStats.SetLife(life);
	gameStats.SetShield(shield);

	if (moving == Moving.Left)
	{
		OutputEffParameters outputEffParameters =
			new OutputEffParameters(2);
		outputEffParameters.ChannelVolume[0] = 1.1f;
		outputEffParameters.ChannelVolume[1] = 0.3f;
		if (!engineSound.IsDestroyed)
			engineSound.SetParametersToEffect(
				engineSound.GetOutputEffectIndex(),
				outputEffParameters
			);
	}
	else if (moving == Moving.Right)
	{
		OutputEffParameters outputEffParameters =
			new OutputEffParameters(2);
		outputEffParameters.ChannelVolume[0] = 0.3f;
		outputEffParameters.ChannelVolume[1] = 1.1f;
		if (!engineSound.IsDestroyed)
			engineSound.SetParametersToEffect(
				engineSound.GetOutputEffectIndex(),
				outputEffParameters
			);
	}
	else
	{
		OutputEffParameters outputEffParameters =
			new OutputEffParameters(2);
		outputEffParameters.ChannelVolume[0] = 1f;
		outputEffParameters.ChannelVolume[1] = 1f;
		if (!engineSound.IsDestroyed)
			engineSound.SetParametersToEffect(
				engineSound.GetOutputEffectIndex(),
				outputEffParameters
			);
	}

	…
}
```

Dalším krokem bude přehrávání zvuku poškození a ztráty života, oboje budeme spouštět v metodě `SetDamage`, přičemž zvuk ztráty života spustíme až při ztrátě

života. Zvuk poškození spustíme zavoláním události `takeDamage` a zvuk ztráty života zavoláním události `lostLife`:

```
public override bool SetDamage(int damage)
{
	if (immortality > 0)
		return false;

	shipSound.DoEvent("takeDamage");
	if (shield > 0)
	…
                    if (gameStats.RemoveLife())
                    {
                        …
                    }
                    else
                    {
                        life = 100;
                        immortality = 3;
                        shipSound.DoEvent("lostLife");
                        return false;
                    }
	…
}
```

Posledním zvukem lodě hráče je zvuk štítu. Zvuk spustíme při získání štítu, proto budeme spouštět zvuk štítu v metodě `SetShield`, neboť ta se volá při získání (nastavení) štítu lodi. Zvuk následně zastavíme, když loď přijde o štít a to se může stát při poškození lodi, tudíž zastavení zvuku štítu lodi provedeme v metodě `SetDamage` když zjistíme, že už loď nemá štít. Zvuk štítu navíc závisí (bude znít jinak) na hodnotě nabití (celistvosti) štítu. Proto když se bude měnit hodnota štítu (což je v metodě `SetShield` a `SetDamage`), tak nastavíme i parametr `shield` instance zvuku lodi. V metodě `SetShield` tedy spustíme zvuk štítu zavoláním události `startShield` a nastavíme parametr `shield`:

```
public void SetShield(int shieldValue) {
	if (shield < shieldValue)
		shield = shieldValue;
	if (shield > 0)
		shipSound.DoEvent("startShield");

	shipSound.SetParameters("shield", shield);
}
```

V metodě `SetDamage` pak zastavíme zvuk štítu při jeho vybití zavoláním události `takeDamage` a jinak nastavíme parametr `shield`.

```
public override bool SetDamage(int damage)
{
	…
	if (shield > 0)
	{
		…
	}
	if (shield > 0)
		shipSound.SetParameters("shield", shield);
	else
		shipSound.DoEvent("stopShield");
	if (shield <= 0)
	…
}
```

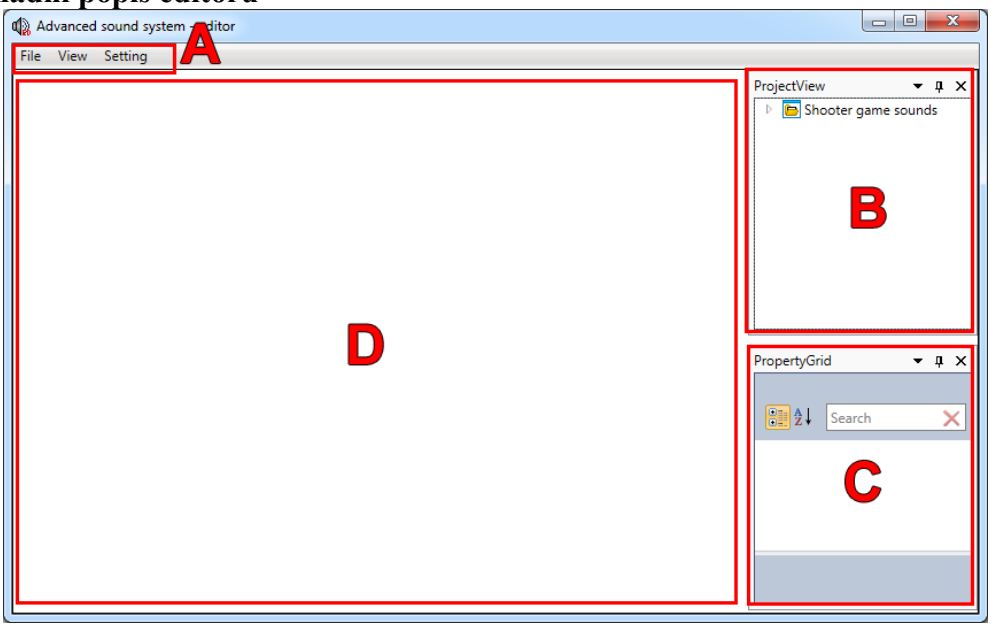
Nyní už máme implementaci zvuku ve hře dokončenou. Pro kontrolu správnosti postupu si můžete otevřít solution hry ze složky `Ukazka_hra\Shooter_game_v6_final` na DVD a porovnat si kód.

Na této hře jsme si ukázali integraci našeho audio enginu do aplikace a následný způsob práce s instancemi jednotlivých zvuků.

7 Uživatelská dokumentace B – editor

V dokumentaci si ukážeme tvorbu jednotlivých zvuků v editoru pro náš audio engine. Konkrétní tvorbu zvuků si ukážeme na projektu hry *„Shooter game“*, do níž jsme implementovali náš audio engine v kapitole *6 Uživatelská dokumentace A – engine*.

Základní popis editoru



Obrázek 7.1 - Okno editoru

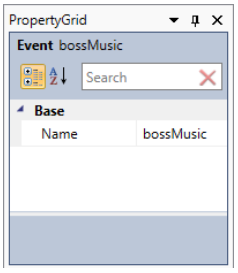
Editor se skládá z hlavních částí, kterými jsou menu (na obrázku *7.1 - oblast A*), postranní panely `ProjectView` (na obrázku *7.1 - oblast B*) a `PropertyGrid` (na obrázku *7.1 - oblast C*) a hlavní část okna (na obrázku *7.1 - oblast D*) pro různé podeditory (dále nazývány jako editory).

Menu

Menu obsahuje položky pro práci s projekty, jako je jejich vytváření a ukládání. Dále umožňuje projekt spustit v testovací aplikaci, jež je popsaná v kapitole *7.2 Testovací aplikace editoru*. Dále menu umožňuje skrývat postraní panely a ovlivnit chování zobrazení editoru grafu.

Panel PropertyGrid

Panel `PropertyGrid` (viz obrázek *7.2*) slouží k editaci položek (atributů) různých objektů, ať už prvků z panelu `ProjectView`, tak prvků z různých editorů. Podobně jako tomu je ve *Visual Studiu*.

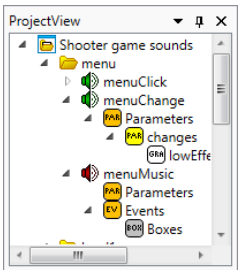


Obrázek 7.2 - PropertyGrid

Panel ProjectView

Panel `ProjectView` (viz obrázek *7.3* na následující straně) slouží pro organizaci struktury projektu. Práce s jednotlivými prvky probíhá tak, že po kliknutí pravým tlačítkem na daný prvek v panelu se zobrazí příslušné kontextové menu, pomocí něhož se přidávají, či mažou jednotlivé prvky projektu. Případně po kliknutí levým tlačítkem na položku se zobrazí informace v panelu `PropertyGrid`.

Interakce s panelem ProjectView bude uvedena v textu následovně:



Obrázek 7.3 - ProjectView

`ProjectView → menuChange → Show SoundEditor` znamená, že klikneme pravým tlačítkem myši na prvek `menuChange` (má unikátní jméno ve stromě projektu) v panelu `ProjectView` a poté klikneme na položku `Show SoundEditor` v kontextovém menu.

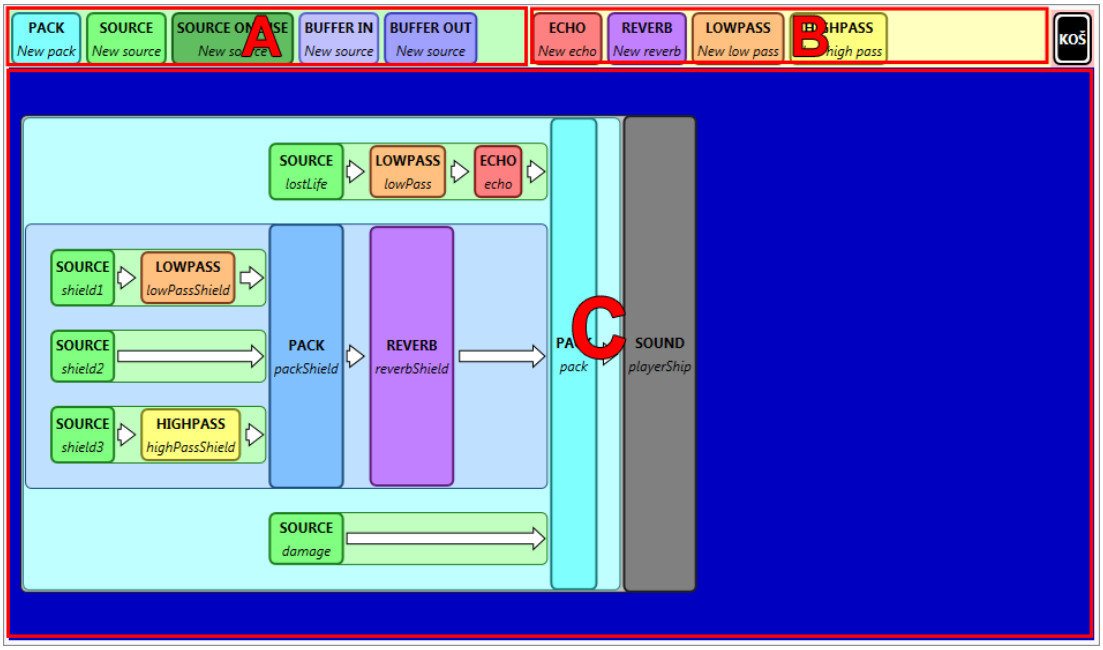
Struktura prvků je daná takto:

- Projekt – symbolizuje projekt, na kterém se pracuje
  - Načítací celek (`LoadingFile`) – seskupuje zdroje zvuků a posluchače do celků, jež slouží k načítání daných zdrojů zvuků a posluchačů do aplikace (hry)
    - Posluchač (`Listener`) – definuje specifický výstup pro zdroje zvuků
      - Výstupní efekt – definuje výstup na reproduktory a možnost dalšího zpracování zvuku
    - Jednoduchý zdroj zvuku (`SimpleSource`) – definuje zdroj zvuku tvořený jen jedním zvukovým souborem
      - Parametry (`Parameters`) – pomocný prvek pro seznam parametrů zdroje zvuku
        - Parametr – definuje parametr zdroje zvuku a jeho možné hodnoty
          - Přiřazený graf – definuje, co ovlivňuje parametr zdroje zvuku
    - Složený zdroj zvuku (`MixedSource`) – definuje zdroj zvuku pro více zvukových souborů, nebo možnosti ovládání pomocí událostí, či přehrávání za pomoci tzv. časové osy
      - Parametry (`Parameters`) – pomocný prvek pro seznam parametrů zdroje zvuku
        - Parametr – definuje parametr zdroje zvuku a jeho možné hodnoty
          - Přiřazený graf – definuje, co ovlivňuje parametr zdroje zvuku
      - Události (`Events`) – pomocný prvek pro seznam událostí zdroje zvuku
        - Seznam zvukových souborů (`Boxes`) – pomocný prvek pro seznamy zvukových souborů, jež se poté dají ovládat pomocí událostí
          - `simple` – obsahující jen jeden zvukový soubor
          - `multi` – obsahující více zvukových souborů (ovládaných zároveň)
          - `random` – obsahující více zvukových souborů (ovládá se vždy náhodně jen jeden, pokaždé jiný)
        - Událost – definuje událost, její název a obsahuje akce
          - Akce – spustí přehrávání definovaného seznamu zvukových souborů

- Akce – zastaví přehrávání definovaného seznamu zvukových souborů
- Akce – nastaví cue bod, že jím lze projít
- Akce – přehrávání skočí na daný čas dané časové osy
- Akce – přehrávání skočí v závistlosti na aktuálním čase časové osy
- Akce – zapne nekonečné opakování daného skoku (cyklu)
- Akce – vypne opakování daného skoku (cyklu)
- Akce – nastaví počet opakování daného skoku (cyklu)
- Akce – nastaví určitý stav časové ose

Editor SoundEditor

Dále zde máme editor, který slouží k sestavení zdroje zvuku z různých zvukových souborů a efektů, které sestavíme tak, jak se mají aplikovat na zvuková data.



Obrázek 7.4 - SoundEditor

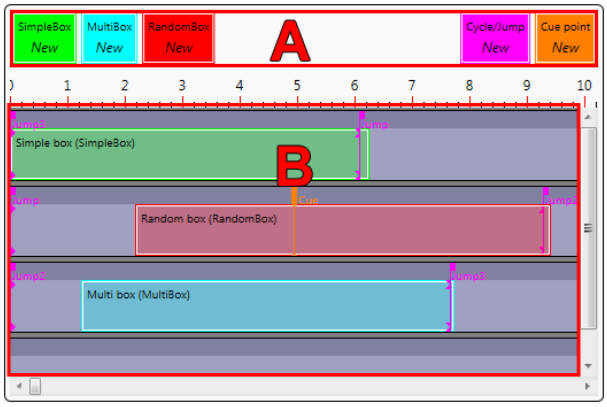
Tento editor se dělí na dvě části, horní panel (na obrázku *7.4 - oblast A a B*), kde jsou prvky, které můžeme přidávat a dolní část (na obrázku *7.4 - oblast C*), v níž sestavujeme výsledný zdroj zvuku. Přičemž si můžeme představit, že zvuková data plynou zleva doprava, tudíž že zvuková data z levého prvku putují do pravého prvku, až dorazí do konečného prvku (znázorňující výstup zvuku).

V levé části horního panelu (na obrázku *7.4 - oblast A*) jsou umístěny zdrojové prvky, které dodávají zvuková data, ať už ze zvukových souborů, tak z bufferu (další zpracování zvuku). Navíc obsahuje speciální prvek `PACK`, jež umožňuje spojovat více zdrojových prvků dohromady. Tyto prvky se vždy vkládají (přetažením) do levé části obdélníků.

V pravé části horního panelu (na obrázku *7.4 - oblast B*) jsou pak umístěny efekty, jež můžeme aplikovat na zvuková data. Tyto prvky se vždy vkládají (přetažením) do pravé části obdélníků.

Editor TimeLine editor

`TimeLine editor` neboli editor časových os je editor pro nastavování přehrávání složeného zdroje zvuku. Tyto zdroje zvuku se většinou skládají z více zvukových souborů a tak potřebujeme určit, kdy se mají spouštět dané zvukové soubory od začátku přehrávaní zdroje zvuku.



Obrázek 7.5 - TimeLine editor

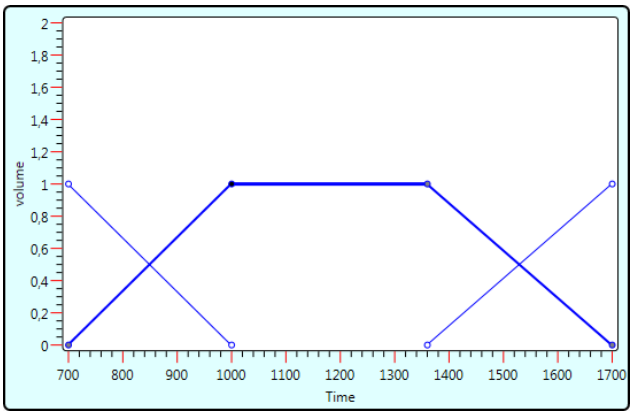
Editor obsahuje dvě části, horní panel (na obrázku *7.5 - oblast A*), jež obsahuje prvky, jež se dají vkládat (přetažením) do časových os a dolní část (na obrázku *7.5 - oblast B*), jež reprezentuje časové osy.

Zvukové soubory se přiřazují na časovou osu pomocí tzv. boxů neboli seznamu zvukových souborů. Jsou celkem tři typy: `SimpleBox` (obsahuje jeden zvukový soubor), `MultiBox` (obsahuje více zvukových souborů – spouští se všechny najednou) a `RandomBox` (obsahuje více zvukových souborů – spouští se náhodně jeden z nich).

Umožňuje navíc definovat několik časových os, přičemž dovoluje mezi nimi přecházet pomocí speciálních prvků `Cycle/Jump` anebo pomocí událostí. Dalším speciálním prvkem je `Cue point`, který zastaví čas časové osy, přičemž umožní přehrávat daný zvukový soubor (umístěný v čase cue bodu) ve smyčce. `Cue point` se pak uvolňuje pomocí události.

Editor grafu

Další editor je editor grafu, ten slouží pro nastavení různých závislostí hodnot. Slouží buď pro nastavování hodnot parametrů různých zvukových prvků (zvukových souborů, efektů) podle parametru zvuku. Dále slouží pro nastavování hlasitosti na časové ose, přičemž zde umožňuje zobrazit více grafů v jednom grafu a tím lépe nastavovat překrývající se části na časové ose (viz obrázek *7.6*).



Obrázek 7.6 - Editor grafu - nastavení hlasitosti na časové ose

Editor grafu se ovládá následovně. Nový bod grafu tvoříme tak, že uděláme dvojklik na modrou čáru mezi dvěma body. Naopak bod odstraníme dvojklikem pravého tlačítka na bod grafu (mimo krajních bodů, ty odstranit nelze). S bodem pak můžeme pohybovat tak, že na bodu stiskneme levé tlačítko a táhneme. Případně klikneme levým tlačítkem na bod, či čáru po pravé straně bodu a upravíme hodnoty v panelu `PropertyGrid`.

7.1 Ukázka tvorby projektu do hry „Shooter game“

Pro ukázku ovládání editoru jsme si připravili postup vytvoření zvuku do projektu hry *„Shooter game“*. Postup se nebude zaměřovat na jednotlivé prvky, ale bude popsán, jako při reálném vývoji zvuků. Tudíž před začátkem si projdeme popis jednotlivých zvuků hry, abychom získali přehled, jaké zvuky budou obsahovat určité prvky (z jakých částí budou složeny).

Popis zvuků do hry „Shooter game“

Hra se dělí do několika celků: *menu* a *jednotlivých herních úrovní*. Přičemž v herních úrovních budou zvuky podobné. Dále nesmíme zapomenout *posluchače*, neboli prvek, přes který budeme přehrávat všechny zvuky, nazveme ho `stereoListener` a bude nám směrovat výstup všech zvuků na přední dva reproduktory.

V menu budeme chtít tyto zvuky:

- `menuChange` – změna položky menu, zvuk se bude při další změně mírně lišit, použijeme parametr `change` s hodnotami od `0` do `10`, který bude ovlivňovat efekt dolní propusti (low-pass filter), který na zvuk aplikujeme [*ukázka SoundEditoru*]
- `menuClick` – potvrzení volby menu [*ukázka tvorby parametru*]
- `menuMusic` – hudba na pozadí [*ukázka tvorby časové osy*]

Ve hře budeme chtít tyto zvuky:

- `levelEnd` – zvuk na konci herní úrovně a zvuk konce hry [*ukázka seznamu zvukových souborů, událostí a akce pro spuštění seznamu zvukových souborů*]
- `playerShip` – zvuk obsahující zvuk ztráty života, zvuk zásahu nepřítelem (poškození lodi), a zvuk štítu, jež se mění podle hodnoty parametru `shield` (hodnoty štítu lodi) [*ukázka akce pro zastavení seznamu zvukových souborů*]
- `playerShipEngine` – zvuk vesmírné lodě hráče
- `getBonusShield` – zvuk při sebrání bonusu štítu
- `getBonusLife` – zvuk při sebrání bonusu života
- `greenLaser` – zvuk výstřelu zeleného laseru
- `redLaser` – zvuk výstřelu červeného laseru
- `enemySound` – zvuk výbuchu nepřátelské lodi
- `level1Music`, `level2Music`, `level3Music` – hudba jednotlivých levelů, skládají se z hudby pro samotný level a hudby pro boj s bossem [*ukázka práce s více časovými osami*]

Značení textu

Jednotlivé části textu si budeme vyznačovat ikonkami, které budou umístěně na pravé straně textu. Každá ikonka označuje, čím se následující odstavce textu zabývají. K označování textu budeme používat následující ikonky:

 Popis tvorby posluchače

 Popis tvorby jednoduchého zdroje zvuku

 Popis tvorby složeného zdroje zvuku

 Popis práce se SoundEditorem

 Popis práce s editorem časových os

 Popis práce s editorem grafů

Popis tvorby seznamu zvukových souborů

Popis tvorby parametrů zvuku

Popis tvorby událostí zvuku

Potřebné složky a soubory

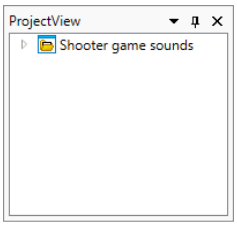
Na DVD jsou všechny soubory, jež jsou potřebné pro tuto ukázku:

- Složka `Ukazka_editor\Shooter_game_v1` – obsahuje základní složkovou strukturu s potřebnými zvukovými soubory
- Složka `Ukazka_editor\Shooter_game_v2_loadings` – obsahuje založený projekt s vytvořenými načítacími celky (zatím prázdnými)
- Složka `Ukazka_editor\Shooter_game_v3_menu` – obsahuje posluchače a všechny zvuky (zdroje zvuku) menu
- Složka `Ukazka_editor\Shooter_game_v4_levelCommon` – obsahuje navíc zvuky společné pro všechny herní úrovně
- Složka `Ukazka_editor\Shooter_game_v5_level1` – obsahuje navíc zvuky první herní úrovně
- Složka `Ukazka_editor\Shooter_game_v6_final` – obsahuje všechny zvuky pro hru *„Shooter game“*

7.1.1 Založení projektu

Začneme přípravami zvuku pro hru *„Shooter game“*. Ve složce `Ukazka_editor\Shooter_game_v1` máme připravenou složkovou strukturu společně s potřebnými zvukovými soubory. Tuto složku si tedy zkopírujeme třeba do složky `C:\Shooter_game_sounds`.

Vytvoříme si tedy projekt pro hru *„Shooter game“*. Klikneme na položku `File → New project` v menu. Otevře se nám okno pro uložení souboru projektu, uložíme si projekt pod názvem `sounds.xml` do složky `C:\Shooter_game_sounds`. Tímto jsme si založili nový projekt, který doporučujeme v průběhu editace ukládat.



Obrázek 7.7 - ProjectView - projekt

Nyní se nám v panelu `ProjectView` zobrazila položka `New project` (viz obrázek *7.7*), klikneme na ní (dále jako `ProjectView → New Project`). Tím se nám změní panel `PropertyGrid` a můžeme editovat položky projektu:

`New project`

`ProjectName ← Shooter game sounds`	Pojmenování projektu, abychom, když příště otevřeme tento projekt, věděli, o který projekt se jedná

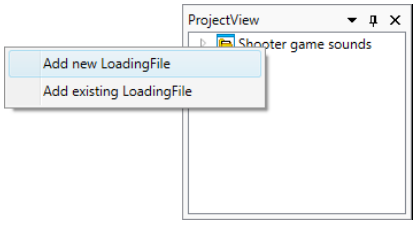
Tím se nám změní název položky v panelu `ProjectView` z `New project` na `Shooter game sounds`, neboli změna názvu položek v panelu `PropertyGrid`

nám změní název položky v původním panelu. Dále pokud v tabulce nebudou uvedeny některé hodnoty, tak to znamená, že necháme nastaveny výchozí hodnoty.

7.1.2 Rozdělení na načítací části

Důležitou částí je rozdělení načítání jednotlivých zvuků na různé celky, neboť nepotřebujeme vždy pracovat se všemi zvuky najednou. Stačí nám mít v paměti jen ty zvuky, se kterými potřebujeme pracovat. Musíme se tedy rozhodnout, jaké celky budou zvořit.

Základním bude načtení prvku `stereoListener`, jež budeme potřebovat po celou dobu běhu hry, proto ho umístíme do načítacího celku, jež se načítá automaticky na začátku. Dalším celkem bude menu, a následně celky pro jednotlivé herní úrovně, přičemž si vytvoříme ještě jeden celek, který bude společný pro všechny herní úrovně.



Obrázek 7.8 – Kontextové menu

Tyto načítací celky se vytvářejí tak, že při kliknutí pravým tlačítkem na ikonu projektu v panelu `ProjectView` vybereme `Add new LoadingFile` (dále `ProjectView → Shooter game sounds → Add new LoadingFile`) jako na obrázku *7.8*. Tím se nám otevře dialogové okno, které nám umožní uložit definici jednoho celku do souboru. Při vytvoření celku se poté v panelu `ProjectView` objeví odpovídající položka s ikonou složky. Když na tuto složku klikneme, tak se nám přizpůsobí panel `PropertyGrid`, jež umožňuje u každého načítacího celku nastavit `LoadingName` (název, kterým načítáme celky do hry) a `SubName` (který nám slouží jen pro popis v editoru. Vytvoříme tedy následující načítací celky (jejichž soubory uložíme do složky `Sound` projektu hry) s následujícími hodnotami (nezapomeňte poté uložit):

`Soubor common.xml`

`LoadingName ← „“`	Prázdný řetězec, díky tomu se načítací celek načítá automaticky, pro pojmenování se použije `SubName`
`SubName ← at startup`	

`Soubor menu.xml`

`LoadingName ← menu`

`Soubor levelCommon.xml`

`LoadingName ← levelCommon`

`Soubor level1.xml`

`LoadingName ← level1`

`Soubor level2.xml`

`LoadingName ← level2`

Soubor level3.xml

```
LoadingName  ← level3
```

7.1.3 Tvorba zdrojů zvuků a posluchačů

V této části si ukážeme vytváření zdrojů zvuků a posluchačů. Budeme popisovat vytváření jednotlivých částí těchto prvků a nastavení hodnot jednotlivých položek. Po tomto kroku budou zvuky připraveny pro přehrávání pomocí audio enginu ve hře.

Posluchač „stereoListener“

Nyní jak jsme si řekli výše, budeme potřebovat používat pro všechny zvuky posluchače. Toho si vytvoříme v načítacím celku `(at startup)`. Postup je podobný jako u načítacích celků, s rozdílem, že tentokrát klikneme na `ProjectView → (at startup) → Add new Listener`. Zobrazí se nám dialogové okno, kam chceme uložit soubor s popisem posluchače, v podstatě nám je umístění souboru jedno, ale pro lepší přehlednost soubor uložíme takto:

`Název souboru`	`← stereoListener.xml`	
`Složka pro uložení`	`← Sounds\common`	Jedná se o relativní cestu od kořene složky projektu

Po této akci se nám v panelu `ProjectView` objeví nová položka s ikonkou ucha. Klikneme na ni a nastavíme následující hodnoty:

```
Name          ← stereoListener
```

Položku `Volume` (hlasitost) necháme na `1` (tedy normální hlasitost). Nyní máme posluchače pojmenovaného, ještě je potřeba ho nastavit. Nejdříve se podíváme do panelu `ProjectView`, kde si rozklikneme posluchače a zobrazí se nám výstupní efekt (ikona čtyř reproduktorů), klikneme na něj a opět začmeme v panelu `PropertyGrid` upravovat položky:

`OutputType`	`← Stereo1`	Chceme, aby posluchač směroval výstup na přední dva reproduktory
`BackBuffer`	`← -1`	Udává, že zvuk dál nechceme zpracovávat, jinak bychom nastavili nějaké kladné číslo
`Name`	`← Stereo`	Slouží jen pro lepší orientaci v editoru

Zvuky menu

Posluchače máme vytvořeného, nyní přejdeme k vytváření zdrojů zvuků. Začneme zvuky menu. Jak máme v seznamu na začátku kapitoly *7.1 Ukázka tvorby projektu do hry „Shooter game“*, zvuky menu se budou sestávat ze zvuku změny volby menu, potvrzení volby menu a hudby na pozadí.

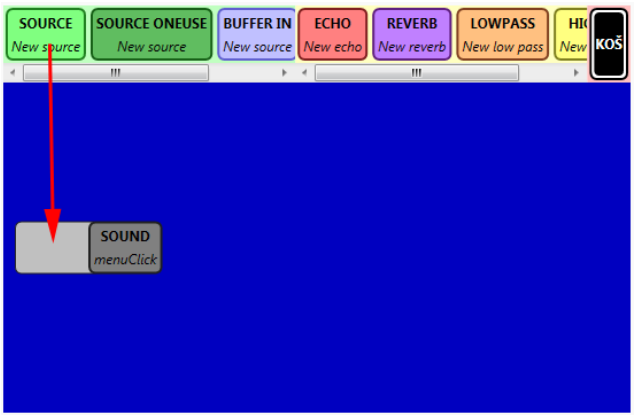
Zdroj zvuku „menuClick“

Začneme zvukem potvrzení volby menu. Ten bude tvořen jedním zvukovým souborem, proto tento zvuk vytvoříme jako jednoduchý zdroj zvuku. Klikneme tedy na `ProjectView → menu → Add new SimpleSource`, vyskočí nám dialogové okno pro uložení souboru s informacemi o jednoduchém zdroji zvuku, tento soubor uložíme takto (viz další stránka):

`Název souboru`	`← menuClick.xml`	
`Složka pro uložení`	`← Sounds\menu`	Jedná se o relativní cestu od kořene složky projektu

Tímto se nám v panelu `ProjectView` zobrazila položka symbolizující jednoduchý zdroj zvuku (ikona zeleného reproduktoru). Klikneme na ní a nastavíme:

`Name`	`← menuClick`



Obrázek 7.9 - SoundEditor - přidávání zvukových souborů

Dále potřebujeme přiřadit zvukový soubor, který budeme přehrávat tímto jednoduchým zdrojem zvuku. K tomu nám slouží panel `SoundEditor`. Tento panel zobrazíme kliknutím na `ProjectView → menuClick → Show SoundEditor`. Tím se nám v levé části editoru zobrazí nové okno. Práce s ním je popsána v kapitole *7 Uživatelská dokumentace B – editor*. Uprostřed máme šedivý obdélník symbolizující výstup zvuku `menuClick`. Neboť chceme jako zdroj zvuku zvukový soubor, tak z levé horní části přetáhneme prvek `SOURCE`, do levé části šedého obdélníka (viz obrázek *7.9*). Tímto krokem jsme zajistili, že zdrojem našeho zvuku bude zvukový soubor. Nyní ho ještě nastavíme. Klikneme na zelený obdélník uprostřed okna, který jsme si vytvořili a nastavíme mu tyto hodnoty:

`Src`	`← Sounds\menu\click.wav`	Cesta k souboru je relativní od složky projektu hry, zadejte absolutní cestu, podle umístění projektu.
`Name`	`← clickSound`	Název slouží pro další prvky, jež chtějí využívat tento prvek
`Volume`	`← 1.5`	Hodnotu víme z autorova testování

Těmito kroky jsme si naeditovali zvuk, jenž se bude přehrávat při potvrzení položky menu. Další zdroje zvuku budou obsahovat stejný počáteční postup, takže popis postupu zestručníme.

Zdroj zvuku „menuChange“

Nyní přejdeme na zvuk, který se bude přehrávat při změně položky. Tento zvuk bude podobný jako zvuk při potvrzení položky s rozdílem, že se bude měnit podle hodnoty parametru `change`. Ke změně zvuku pomocí parametru pak využijeme efekt dolní propusti.

Vytvoříme si tedy v načítacím celku menu jednoduchý zdroj zvuku:

Název souboru	← menuChange.xml	
Složka pro uložení	← Sounds\menu	Jedná se o relativní cestu od kořene složky projektu

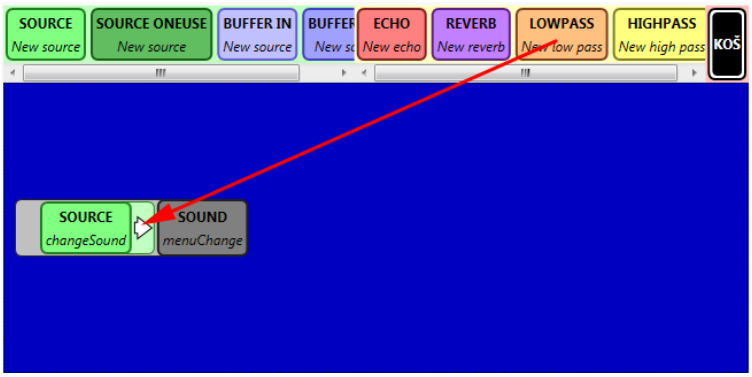
Nově vytvořený zdroj zvuku

Name	← menuMusic	

Další částí bude přiřazení zvukového souboru. Zobrazíme si tedy SoundEditor. V něm podobně jako předtím přetáhneme prvek SOURCE z levé horní části panelu do levé části šedého obdélníku uprostřed panelu a nastavíme mu položky následovně:

Src	← Sounds\menu\change.wav	Zadejte absolutní cestu, podle umístění projektu.
Name	← changeSound	Název slouží pro další prvky, jež chtějí využívat tento prvek

Tímto jsme si vytvořili zvuk pro změnu položky menu, ještě ho ale nemáme dokončený. Chceme, aby se dal upravovat parametrem s názvem change, který může nabývat hodnot 0 až 10, konkrétně efekt dolní propusti (její frekvenci), jež budeme na zvuk ze zvukového souboru aplikovat.



Obrázek 7.10 - SoundEditor - přidávání efektů

Začneme přidáním efektu dolní propusti. Toho docílíme tak, že v panelu SoundEditor jednoduchého zdroje zvuku menuChange, přetáhneme prvek LOWPASS z pravé horní části do pravé části zeleného obdelníku (prvku changeSound, viz obrázek *7.10*). Tím docílíme toho, že se efekt bude aplikovat na data ze zvukového souboru. Klikneme na nově vytvořený prvek a nastavíme ho:

Name	← lowEffect
Q	← 40

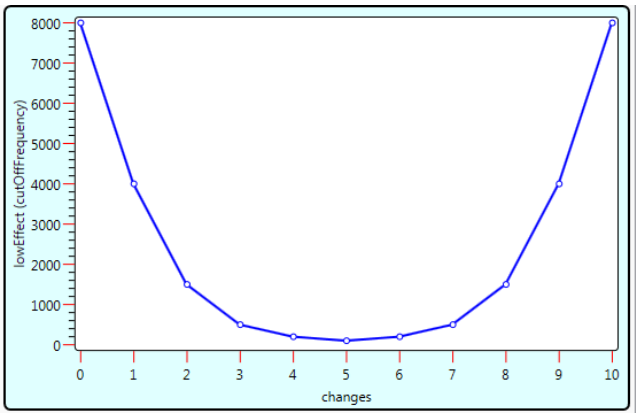
Dalším krokem je přidání parametru change. Klikneme na ProjectView → menuChange → Parameters → Add new parameter. Zobrazí se nám dialogové okno, do kterého zadáme údaje:

Parameter name	← change	Chceme, aby se tak parametr jmenoval
Min value	← 0	
Max value	← 10	
Default value	← 0	
Type of axis X	← Integer	Chceme nastavovat jen celočíselné hodnoty

Údaje potvrdíme stisknutím tlačítka `Add parameter`. Tímto krokem se nám vytvořil parametr `change` u jednoduchého zdroje zvuku `menuChange`. Potřebujeme ale ještě přiřadit, co má parametr ovlivňovat. Toho docílíme tak, že klikneme na `ProjectView → menuChange → Parameters → change → Add new assigned graph`. Zobrazí se dialogové okno s výběrem toho, co můžeme pomocí parametru ovlivňovat. Nastavíme:

`Set object`	`← lowEffect`	Vytvořili jsme ho pomocí `SoundEditoru`
`Set parameter`	`← cutOffFrequency`	Budeme chtít upravovat frekvenci efektu dolní propusti

Volbu potvrdíme kliknutím na tlačítko `Add assigned graph`. V `ProjectView` se zobrazí další položka, reprezentující objekt a parametr objektu, který bude parametr jednoduchého zdroje zvuku ovlivňovat.



Obrázek 7.11 - Editor grafů - nastavení parametru

Nyní potřebujeme nastavit, jak se bude parametr objektu ovlivňovat. K tomu nám bude sloužit editor grafu, vyvoláme ho kliknutím na `ProjectView → menuChange → Parameters → change → lowEffect (cutOffFrequency) → Show graph`. Tímto se na místě panelu `SoundEditor` zobrazí editor grafu. Jeho ovládání je popsáno v kapitole *7 Uživatelská dokumentace B – editor*. Osa `X` nám symbolizuje náš parametr `change`, nabývající hodnot od `0` do `10`. Zatímco osa `Y` symbolizuje `cutOffFrequency` efektu dolní propusti. Díky tomuto editoru grafu nyní můžeme nastavit závislost parametru dolní propusti na parametru zvuku. Zde (díky testování autora) nastavíme graf tak, že bude vypadat jako parabola, což nám na zvuku vytvoří požadovaný efekt. Konkrétně nastavíme tyto body (viz obrázek *7.11*):

```
[0, 8000], [1, 4000], [2, 1500], [3, 500], [4, 200], [5, 100],
[6, 200], [7, 500], [8, 1500], [9, 4000], [10, 8000]
```

Přičemž každému bodu nastavíme:

`LineType`	`← Linear`	Tím se dopočítají hodnoty mezi body
`Continues`	`← √`	Zaškrtneme, tím budou body na sebe navazovat

Tímto jsme si ukázali vytvoření zvuku, jež může být ovlivněn pomocí parametru.

Zdroj zvuku „menuMusic“

Mezi zvuky menu nám ještě chybí hudba. Hudba menu bude obsahovat několik dlouhých zvukových souborů, jež se budou přehrávat postupně za sebou.

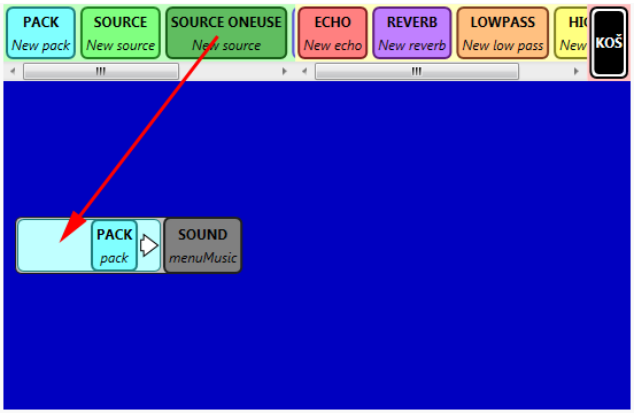
K definici hudby menu použijeme na rozdíl od zvuků potvrzení, či změny položek složený zdroj zvuku. Složený zdroj zvuku zde použijeme, neboť zvuk hudby bude složen z několika zvukových souborů a to nám jednoduchý zdroj zvuku neumožní definovat.

Složený zdroj zvuku vytvoříme kliknutím na `ProjectView → menu → Add new MixedSource`:

`Název souboru`	`← menuMusic.xml`	
`Složka pro uložení`	`← Sounds\menu`	Jedná se o relativní cestu od kořene složky projektu

`Nově vytvořený zdroj zvuku`

`Name`	`← menuMusic`	



Obrázek 7.12 - SoundEditor - přidávání zvukových souborů

Nyní k našemu složenému zdroji zvuku připojíme zvukové soubory. K tomu využijeme jako výše `SoundEditor`. Zobrazíme si tedy `SoundEditor`. V levé části se nám opět zobrazí povědomé okno, kdy uprostřed máme šedý obdélník. Do levé části šedého obdélníka můžeme vložit jen jeden prvek, nicméně chceme přehrávat několik zvukových souborů. K tomu nám u složeného zdroje zvuku přibyl prvek `PACK`, který nám pak umožní přidat více zvukových prvků, které následně slučuje audio engine dohromady. Přetáhneme tedy z levé horní části prvek `PACK` do levé části šedého obdélníka. Nyní se nám tam vytvořil nový světle modrý prvek, ten funguje podobně jako ten šedý s rozdílem, že do levé části tohoto prvku můžeme dát více prvků a do pravé části můžeme vložit efekty, které se mají aplikovat na výsledný sloučený zvuk. Nyní bychom přetáhli seshora prvek `SOURCE`, abychom vložili zvukový soubor. To ale neuděláme, neboť použijeme prvek `SOURCE ONEUSE`. Rozdíl je v tom, že prvek `SOURCE` načítá všechny data zvukového souboru do paměti, zatímco `SOURCE ONEUSE` je načítá postupně a tak šetří paměť, s tím ale souvisí jedno omezení a tím je, že můžeme vytvořit jen jednu instanci tohoto zvuku, jinak bude zvuk zlobit: nemusí se přehrávat, případně skončí chybou! Protože máme celkem čtyři zvukové soubory pro hudbu v menu, tak přetáhneme čtyřikrát prvek `SOURCE ONEUSE` do levé části prvku `PACK` (viz obrázek *7.12*) a nastavíme jim tyto hodnoty:

`1. SOURCE ONEUSE`

`Name`	`← menuSound1`	
`Src`	`← Sounds\music\menu\menuSound1.wav`	Relativní cesta

```
UniqueName    ← menuSound1
```

```
2. SOURCE ONEUSE
```

```
Name          ← menuSound2
Src           ← Sounds\music\menu\menuSound2.wav    Relativní cesta
UniqueName    ← menuSound2
```

```
3. SOURCE ONEUSE
```

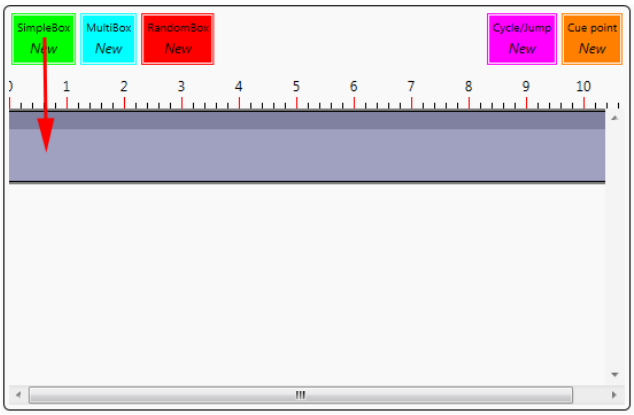
```
Name          ← menuSound3
Src           ← Sounds\music\menu\menuSound4.wav    Relativní cesta
UniqueName    ← menuSound3
```

```
4. SOURCE ONEUSE
```

```
Name          ← menuSound4
Src           ← Sounds\music\menu\menuSound4.wav    Relativní cesta
UniqueName    ← menuSound4
```

Na rozdíl od prvku `SOURCE` tu máme v panelu `PropertyGrid` ještě položku `UniqueName`, ta slouží k tomu, abychom tento zvukový soubor mohli použít ve více zvucích, neboli položka `UniqueName` z daného zvukového souboru vytváří unikátní zdroj, tudíž si dejte pozor na pojmenování `UniqueName`.

Další částí je přehrávání právě vložených zvukových souborů. Od jednoduchého zdroje zvuku to zde lze dvěma způsoby. Buď umístěním na časovou osu a po spuštění zvuku se v určitý čas přehrají zvukové soubory anebo budeme spouštět zvukové soubory pomocí události. Zde si ukážeme přehrávání pomocí časové osy, neboť chceme, aby se jednotlivé zvukové soubory přehrávaly postupně za sebou a po přehrání všech zvukových souborů, chceme tyto soubory přehrávat znovu od začátku. K editování časové osy nám poslouží `TimeLine editor`, jehož popis a práce s ním je popsána v kapitole *7 Uživatelská dokumentace B – editor*. Editor otevřeme kliknutím na `ProjectView → menuMusic → Show TimeLine editor`. Tím se nám v levé části zobrazí editor časových os. Časová osa je zde znázorněna šedým pruhem, nad kterým je osa znázorňující čas.



Obrázek 7.13 - TimeLine editor - přidávání prvků

V jeden časový okamžik budeme chtít přehrávat jen jeden zvukový soubor, k tomu nám poslouží `SimpleBox`. Přetáhneme tedy `SimpleBox` z levé horní části do časové osy (viz obrázek *7.13*). Tím se nám na časové ose vytvořil zelený obdélník

znázorňující, že v daný čas (který zabírá zelený obdélník), od spuštění zvuku, se bude přehrávat jeden zvukový soubor. Nyní klikneme na zelený obdélník, který jsme vytvořili na časové ose a nastavíme mu tyto hodnoty:

BoxName	← music1	Box použijeme pro první zvukový soubor
SoundSource	← menuSound1	První zvukový soubor, jeho jméno jsme nastavili v SoundEditoru
StartTime	← 0	Chceme, aby se přehrával zvukový soubor ihned po spuštění zvuku hudby
Length	← 114000	Zvukový soubor má délku 1 minutu a 54 sekund (hodnota je v ms)

Následně do časové osy vložíme další tři prvky SimpleBox pro přehrávání dalších zvukových souborů. Zde si dáme pozor, abychom je stále vkládali do 1. časové osy a ne druhé.

2. SimpleBox

BoxName	← music2	
SoundSource	← menuSound2	
StartTime	← 113750	Chceme, aby se zvuk přehrával až po prvním souboru, začne hrát o 0.25 sekundy před koncem prvního souboru
Length	← 98000	Délka druhého zvukovéhou souboru

3. SimpleBox

BoxName	← music3	
SoundSource	← menuSound3	
StartTime	← 211500	Chceme, aby se zvuk přehrával až po druhém souboru, začne hrát o 0.25 sekundy před koncem druhého souboru
Length	← 118000	Délka třetího zvukovéhou souboru

4. SimpleBox

BoxName	← music4	
SoundSource	← menuSound4	
StartTime	← 329250	Chceme, aby se zvuk přehrával až po třetím souboru, začne hrát o 0.25 sekundy před koncem třetího souboru
Length	← 116000	Délka čtvrtého zvukovéhou souboru

Nyní ještě potřebujeme zajistit, aby se po přehrání všech souborů opakovalo přehrávání opět od začátku. K tomu nám slouží prvek Cycle/Jump. Přetáhneme tedy prvek Cycle/Jump z pravého horního rohu do 1. časové osy. Na časové ose se nám vytvoří takové dvě čáry. Klikneme na jednu z nich, abychom mohli nastavit hodnoty:

JumpTime	← 445000	Udává, z jakého času budeme skákat
TimeTarget	← 0	Udává, kam budeme skákat po časové ose, zde budeme chtít začátek časové osy.

Fadeout	← 250	Udává, jak dlouho se bude přehrávat původní zvuková data (před skokem) zároveň s novými zvukovými daty (po skoku)
EnabledCycles	← -1	Udává, kolikrát se skok bude opakovat
Name	← Jump	

Tímto jsme si nastavili poslední zvuk menu. Takže máme kompletně vytvořeny zvuky menu. Tyto zvuky si můžeme vyzkoušet pomocí testovací aplikace, jež je popsána v kapitole *7.2 Testovací aplikace editoru*. Spouští se z menu `File`, kliknutím na položku `Run project`.

Společné zvuky herních úrovní

Dále se přesuneme na zvuky, jež jsou pro všechny herní úrovně stejné, neboli na načítací celek `levelCommon`. V této části si ukážeme využití událostí složeného zdroje zvuku.

Zdroj zvuku „levelEnd“

Nyní si vytvoříme zvuk, jenž se bude přehrávat při skončení herní úrovně, ať už úspěšně či neúspěšně. V podstatě, když úspěšně skončíme herní úroveň, tak budeme chtít přehrát jeden ze zvukových souborů signalizující úspěšné dokončení herní úrovně, zatímco při neúspěšném dokončení herní úrovně budeme chtít přehrát jeden ze zvukových souborů signalizující nedokončení herní úrovně. Z popisu zvuku už vidíme, že bude tvořen několika zvukovými soubory.

Vytvoříme si tedy složený zdroj zvuku v načítacím celku `levelCommon`:

Název souboru	← levelEnd.xml	
Složka pro uložení	← Sounds\levelCommon	Jedná se o relativní cestu od kořene složky projektu

Nově vytvořený zdroj zvuku

Name	← levelEnd	

Následně si otevřeme `SoundEditor` právě vytvořeného složeného zdroje zvuku. Neboť má složený zdroj zvuku obsahovat více zvukových souborů, budeme postupovat podobně jako u hudby v menu. Přetáhneme tedy nejdřív prvek `PACK`. Nyní na rozdíl od hudby menu se jedná o malé zvukové soubory, tudíž si můžeme dovolit si je načíst do paměti. Takže pro vložení zvukového souboru použijeme prvek `SOURCE`. Přetáhneme celkem tři prvky `SOURCE` do vytvořeného prvku `PACK`, kterými přiřadíme do zvuku potřebné zvukové soubory:

1. SOURCE

Name	← win1	
Src	← Sounds\success\success-1.wav	Relativní cesta
Volume	← 0.5	

2. SOURCE

Name	← win2	
Src	← Sounds\success\success-2.wav	Relativní cesta
Volume	← 0.5	

3. SOURCE

Name	← gameover1	
Src	← Sounds\gameover\game-over.wav	Relativní cesta

Nyní ale chceme umět tyto soubory i spouštět. Na rozdíl od hudby v menu k tomu využijeme události. Abychom ale mohli zvukové soubory spuštět, tak je potřebujeme nejdříve seskupit zvukové soubory do určitých skupin. Slouží nám k tomu tzv. boxy, které bývají spojeny s událostmi zvuku, najdeme je v panelu ProjectView, když rozklikneme složený zdroj zvuku a následně položku Events a poté Boxes. Jako u časových os zde existují zde tři typy boxů: první simple box, jež obsahuje jen jeden zvukový soubor, druhý multi box, jež obsahuje několik zvukových souborů a ovládá je najednou a třetí random box, jež také obsahuje několik zvukových souborů, ale vždy vybere náhodně jeden ze zvukových souborů. Poslední typ je pro náš účel nejvhodnější.

Obrázek 7.14 - ProjectView - skupiny zvukových souborů

Vytvoříme tedy dvě skupiny zvukových souborů, první se zvukovými soubory úspěšného dokončení herní úrovně a druhý se zvukovými soubory nedokončení herní úrovně (viz obrázek *7.14*). Pro vytvoření skupiny zvukových souborů klikneme na ProjectView → Boxes → Add new random list příslušného zdroje zvuku a nastavíme mu hodnoty:

Name	← winSoundList	
SoundSources	← win1, win2	Po kliknutí na tlačítko „…“ se otevře dialogové okno, kde můžeme přiřadit zvukové soubory

Potvrdíme tlačítkem OK. Tím jsme si vytvořili první skupinu. Vytvoříme si ještě druhou obdobným způsobem:

Name	← gameoverSoundList
SoundSources	← gameover1

Tím jsme si vytvořili skupiny zvukových souborů. Teď je potřebujeme ještě umět přehrávat. K tomu nám poslouží události zvuku. Vytvoříme si tedy dvě události, první playWin pro přehrávání zvuku z prvního seznamu a druhou playGameover pro přehrávání zvuku z druhého seznamu.

Událost vytvoříme tak, že klikneme na ProjecView → levelEnd → Events → Add new event. Vytvoří se nám nová událost, žlutá ikonka v panelu ProjectView po rozkliknutí prvku Events. Nastavíme ji:

Name	← playWin	Přes nastavené jméno pak budeme událost pomocí audio enginu volat!

Nyní ještě potřebujeme události přiřadit akci, neboli co má dělat. Po události chceme, aby spustila náš seznam zvukových souborů. Tuto událost přiřadíme tak, že klikneme na `ProjectView → levelEnd → Events → playWin → Add new start action`. Následně po rozkliknutí události `playWin` v panelu `ProjectView` se nám zobrazí nová událost `New start action`. Klikneme na ni a nastavíme ji:

`PlayBoxIndex`	`← 0`	Jedná se o index, který jde od hodnoty 0, odkazuje nás na první seznam zvukových souborů v položce `Boxes` v panelu `ProjectView` (v této verzi editoru nelze přiřazovat podle jména)
`Name`	`← playWinSoundList`	Jen pro lepší orientaci v akcích v editoru

Hodnoty `State` a `TimeLineIndex` necháme nastavené na `-1`, což znamená, že akce se provede nezávisle na stavu složeného zdroje zvuku a aktuální přehrávané časové ose (což u tohoto zvuku nevyužíváme). Obdobným způsobem vytvoříme událost s názvem `playGameover`, které přiřadíme stejný typ akce s hodnotami:

`PlayBoxIndex`	`← 1`	
`Name`	`← playGameoverSoundList`	

Zdroj zvuku „playerShipEngine“

Dalším zvukem, který bude pro všechny herní úrovně stejný, je zvuk motoru lodi. I když bude zvuk motoru lodi tvořit jen jeden zvukový soubor, tak ho vytvoříme, jako kdyby obsahoval více zvukových souborů, kdybychom v budoucnu chtěli tento zvuk rozšířit.

Vytvoříme si tedy nový složený zdroj zvuku v načítacím celku `levelCommon`:

`Název souboru`	`← playerShipEngine.xml`	
`Složka pro uložení`	`← Sounds\levelCommon`	Jedná se o relativní cestu od kořene složky projektu

`Nově vytvořený zdroj zvuku`

`Name`	`← playerShipEngine`	

Následně v editoru `SoundEditor` přiřadíme zvuku prvek `PACK` a do něj vložíme prvek `SOURCE`:

`Name`	`← engine`	
`Src`	`← Sounds\player\ship\engine-loop.wav`	Relativní cesta
`Volume`	`← 0.3`	
`Loop`	`← √`	Zaškrtneme, chceme, aby se zvuk motoru po jeho spuštění přehrával stále dokola (běžel nekonečně dlouho)

Spouštět zvuk motoru lodi budeme pomocí události, jako jsme spouštěli zvuk konce herní úrovně. K tomu si potřebujeme vytvořit seznam zvukových souborů, které budeme chtít ovládat. I když máme jen jeden zvukový soubor, tak musíme seznam vytvořit. Vybereme si `multi box`, neboť kdybychom chtěli zvuk později rozšířit, abychom nemuseli měnit seznam ze `simple box` na `multi box`. Klikneme tedy na `ProjectView → playerShipEngine → Events → Boxes → Add new multi box` a nastavíme mu hodnoty (viz další stránka):

```
Name          ← engineSoundList
SoundSources  ← engine
```

Nyní si vytvoříme dvě události pro ovládání přehrávání zvuku motoru, neboli vytvoříme si jednu událost `startEngine` pro spuštění zvuku motoru lodi a druhou `stopEngine` pro zastavení přehrávání zvuku motoru lodi. Vytvoříme si tedy novou událost:

```
Name          ← startEngine
```

V této události následně vytvoříme akci pro spuštění přehrávání seznamu zvukových souborů kliknutím na položku `ProjectView → playerShipEngine → Events → startEngine → Add new start action` a přiřadíme jí tyto hodnoty:

```
PlayBoxIndex  ← 0
Name          ← startEngineSoundList
```

Dále si vytvoříme druhou událost:

```
Name          ← stopEngine
```

Té ale přiřadíme akci pro zastavení přehrávání seznamu zvukových souborů. Tuto akci přidáme kliknutím na položku `ProjectView → playerShipEngine → Events → stopEngine → Add new stop action`. Akci nastavíme následující hodnoty:

```
StopBoxIndex  ← 0
Name          ← stopEngineSoundList
```

Tím jsme si vytvořili zvuk motoru lodi, který můžeme pomocí událostí spouštět a zastavovat.

Zdroj zvuku „greenLaser“

Dalším zvukem bude zvuk výstřelu zeleného laseru hráče (který také využijeme pro některé nepřátelské lodi). Bude obsahovat jen jeden zvukový soubor, ale připravíme ho tak, že by mohl být složen z více zvukových souborů, přičemž při každém výstřelu by se přehrával náhodně jeden z nich.

V načítacím celku `levelCommon` si tedy vytvoříme složený zdroj zvuku:

`Název souboru`	`← greenLaser.xml`	
`Složka pro uložení`	`← Sounds\levelCommon`	Jedná se o relativní cestu od kořene složky projektu

`Nově vytvořený zdroj zvuku`

```
Name          ← greenLaser
```

Dále si otevřeme editor `SoundEditor`. V tomto editoru následně přiřadíme prvek `PACK`, a do něj vložíme prvek `SOURCE`, který nastavíme následovně:

`Name`	`← laser`	
`Src`	`← Sounds\lasers\greenLaser.wav`	Relativní cesta
`Volume`	`← 0.5`	

Dále pro přehrávání zvuku pomocí události `fire` si vytvoříme seznam zvukových souborů typu `random box`, který nastavíme následovně:


```
Name          ← greenLaserSoundList
SoundSources  ← laser
```

Dále si vytvoříme událost:


```
Name          ← fire
```

Této události pak nastavíme akci pro spuštění přehrávání seznamu zvukových souborů, kterou nastavíme takto:

```
PlayBoxIndex  ← 0
Name          ← playGreenLaserSoundList
```

Tímto jsme si vytvořili zvuk zeleného laseru, který je připraven pro budoucí rozšíření a to, že při každém výstřelu by se přehrával jiný zvukový soubor zvuku laseru.

Zdroj zvuku „playerShip“

Zbývá nám poslední zvuk, který bude pro všechny herní úrovně stejný a to je zvuk lodě hráče. Tento zvuk bude o něco složitější než předchozí, bude se totiž skládat z několika zvuků, které budou nezávisle na sobě přehrávány pomocí událostí. Konkrétně se bude jednat a zvuk při ztrátě života, zvuk poškození lodi a zvuk štítu, jež se bude měnit podle hodnoty štítu (pomocí parametru `shield`).

V načítacím celku `levelCommon` si tedy vytvoříme nový složený zdroj zvuku:

`Název souboru`	`← playerShip.xml`	
`Složka pro uložení`	`← Sounds\levelCommon`	Jedná se o relativní cestu od kořene složky projektu

`Nově vytvořený zdroj zvuku`

```
Name          ← playerShip
```

Poté pomocí editoru `SoundEditor` přiřadíme následně zvukové soubory. Nejdříve přidáme prvek `PACK`. Do tohoto prvku pak budeme přidávat jednotlivé prvky `SOURCE` pro zvukové soubory, přičemž u zvukových souborů štítu nesmíme zapomenout zaškrtnout volbu `Loop`, neboť je chceme přehrávat ve smyčce:

`1. SOURCE`

`Name`	`← lostLife`	
`Src`	`← Sounds\player\life-lost.wav`	Relativní cesta
`Volume`	`← 0.5`	

`2. SOURCE`

`Name`	`← damage`	
`Src`	`← Sounds\player\ship\damage.wav`	Relativní cesta
`Volume`	`← 0.5`	

`3. SOURCE`

`Name`	`← shield1`	
`Src`	`← Sounds\player\ship\shield-sounds-1.wav`	Relativní cesta

Volume	← 1.1	
Loop	← √	Zaškrtneme

4. SOURCE

Name	← shield2	
Src	← Sounds\player\ship\shield-sounds-2.wav	Relativní cesta
Volume	← 1.5	
Loop	← √	Zaškrtneme

5. SOURCE

Name	← shield3	
Src	← Sounds\player\ship\shield-sounds-3.wav	Relativní cesta
Volume	← 2	
Loop	← √	Zaškrtneme

Nyní si vytvoříme seznamy zvukových souborů, abychom je následně mohli spouštět pomocí událostí. První bude typu `simple box`, neboť to bude seznam zvukových souborů pro ztrátu života a ten máme jen jeden, nastavíme ho takto:

Name	← lostLifeSound
SoundSource	← lostLife

Další seznam bude typu `multi box`, který potřebujeme pro ovládání zvukových souborů štítu lodi:

Name	← shieldSoundList
SoundSources	← shield1, shield2, shield3

Poslední seznam bude typu `random box`, jenž bude sloužit k přehrávání jednoho (vybraného náhodně) ze zvukových souborů poškození lodi (příprava pro případné rozšíření těchto zvuků):

Name	← damageSoundList
SoundSources	← damage

Pozor na pořadí vytváření seznamů, je důležité zachovat správné pořadí pro další nastavení zvuku!

Přejdeme k vytváření potřebných událostí pro ovládání přehrávání těchto seznamů zvukových souborů. Jako první si vytvoříme u tohoto zvuku událost pro spuštění zvuku ztráty života:

Name	← lostLife

Této události následně přiřadíme akci pro spuštění přehrávání seznamu zvukových souborů (`start action`):

PlayBoxIndex	← 0
Name	← playLostLifeSound

Vytvoříme další událost pro spuštění zvuku poškození lodi:

Name	← takeDamage

Této události přiřadíme jí stejný typ akce:

```
PlayBoxIndex   ← 2
Name           ← playRandomDamageSounds
```

Následně přidáme událost pro spuštění zvuku štítu:

```
Name           ← startShield
```

Té také přiřadíme akci pro spuštění přehrávání seznamu zvukových souborů:

```
PlayBoxIndex   ← 1
Name           ← playShieldSounds
```

Poté vytvoříme poslední událost pro zastavení zvuku štítu:

```
Name           ← stopShield
```

Té ale nastavíme akci pro zastavení přehrávání seznamu zvukových souborů (`stop action`):

```
StopBoxIndex   ← 1
Name           ← stopShieldSounds
```

V této chvíli už budeme moci přehrávat jednotlivé zvukové soubory zvuku `playerShip`, ale musíme dodělat ještě přehrávání zvukových souborů štítu lodi. Chceme totiž, aby se výsledný zvuk štítu lodi měnil v závislosti na hodnotě štítu. Konkrétně budeme chtít, aby se při hodnotě štítu `<0-33>` přehrával zvukový soubor `shield3`“, dále při hodnotě `[33-67]` se přehrával zvukový soubor `shield2` a při hodnotě `<67-100>`, aby se přehrával zvukový soubor `shield1`. Toho docílíme pomocí parametru zvuku.

Vytvoříme si tedy parametr s hodnotami:

`Parameter name`	`← shield`	Pomocí tohoto jména budeme nastavovat parametr zvuku
`Min value`	`← 0`	
`Max value`	`← 100`	
`Default value`	`← 0`	
`Type of axis X`	`← Double`	Chceme nastavovat všechny hodnoty

K tomuto parametru následně přiřadíme grafy, jež budou nastavovat hlasitost jednotlivých zvukových souborů v závislosti na hodnotě parametru `shield`. Parametru následně přiřadíme, co vše má ovlivňovat, konkrétně přiřadíme tři grafy kliknutím na `ProjectView → playerShip → Parameters → shield → Add new assigned graph`:

`1. graf`

`Set object`	`← shield1`	Vytvořili jsme ho pomocí `SoundEditoru`
`Set parameter`	`← volume`	Budeme chtít upravovat hlasitost zvukového souboru

`2. graf`

`Set object`	`← shield2`	Vytvořili jsme ho pomocí `SoundEditoru`
`Set parameter`	`← volume`	Budeme chtít upravovat hlasitost zvukového souboru

3. graf

Set object	← shield3	Vytvořili jsme ho pomocí SoundEditoru
Set parameter	← volume	Budeme chtít upravovat hlasitost zvukového souboru

Tím se nám v panelu PropertyView objevily nové tři položky reprezentující grafy nastavující hodnotu volume jednotlivých zvukových souborů. Editor pro editaci prvního grafu zobrazíme kliknutím na ProjectView → playerShip → Parameters → shield → shield1 (volume) → Show graph. Graf následně nastavíme (Editace grafů je popsána v kapitole 7) tak, že graf bude mít tyto body:

```
[0,0], [66.999999999, 0], [67, 1], [100, 1]
```

Každý bod grafu přitom bude mít nastavené položky takto:

LineType	← Linear	Tím se dopočítají hodnoty mezi body
Continues	← √	Zaškrtneme, tím budou body na sebe navazovat

Graf shield2 (volume) bude mít následující body v grafu (položky bodů budou nastavené jako u prvního grafu):

```
[0,0], [33, 0], [33,000000001, 1], [66,999999999, 1], [67, 0],
[100, 0]
```

Poslední graf shield3 (volume) bude mít tyto body v grafu (opět bodu budou mít nastavené stejné položky jako u předchozích grafů):

```
[0,1], [33, 1], [33,000000001, 0], [100, 0]
```

Nyní už máme dokončený zvuk playerShip, jež umožní přehrávat zvuk ztráty života, poškození lodi a štítu (v závislosti na jeho hodnotě). Zároveň s tímto zvukem jsme dokončili poslední zvuk z načítacího celku levelCommon.

Načítací celky herních úrovní

Jako další nás budou čekat načítací celky jednotlivých herních úrovní, konkrétně level1, level2 a level3. Tyto načítací celky budou obsahovat zvuky, jež jsou dané pro konkrétní herní úroveň. Ukážeme si zde, že můžeme mít jeden zvuk ve více načítacích celcích, neboli že zvuk, který máme v jednom načítacím celku (zde jedné herní úrovni), můžeme nastavit i jako zvuk dalšího načítacího celku (jiné herní úrovně), neboli si ukážeme, jak se dají jednotlivé zvuky sdílet mezi jednotlivými načítacími celky. Začneme načítacím celkem level1, jež bude obsahovat zvuky 1. herní úrovně.

Zdroj zvuku „getBonusLife“

Prvním zvukem načítacího celku level1 je zvuk, jenž se bude přehrávat při sbírání bonusu doplnění života. Tento zvuk bude tvořen jen jedním zvukovým souborem, proto tento zvuk vytvoříme jako jednoduchý zdroj zvuku.

V načítacím celku `level1` tedy vytvoříme jednoduchý zdroj zvuku:

Název souboru	← getBonusLife.xml	
Složka pro uložení	← Sounds\levelShared	Jedná se o relativní cestu od kořene složky projektu

Nově vytvořený zdroj zvuku

Name	← getBonusLife	

Otevřeme si `SoundEditor` tohoto zdroje zvuku a v editoru zvuku přiřadíme prvek `SOURCE`:

Name	← takeBonusSound	
Src	← Sounds\bonuses\life.wav	Relativní cesta
Volume	← 0.5	

Zdroj zvuku „getBonusShield“

Dalším zvukem bude zvuk sebrání bonusu doplnění štítu. Jedná se o podobný zvuk jako `getBonusLife`. Opět bude zvuk tvořen jen jedním zvukovým souborem a tak bude vytvořen jako jednoduchý zdroj zvuku.

V načítacím celku `level1` tedy vytvoříme jednoduchý zdroj zvuku:

Název souboru	← getBonusShield.xml	
Složka pro uložení	← Sounds\levelShared	Jedná se o relativní cestu od kořene složky projektu

Nově vytvořený zdroj zvuku

Name	← getBonusShield	

V `SoundEditoru` zdroje zvuku pak přiřadíme prvek `SOURCE`:

Name	← takeBonusSound	
Src	← Sounds\bonuses\shield.wav	Relativní cesta
Volume	← 0.5	

Zdroj zvuku „enemySound“

Dále vytvoříme zvuk nepřátelské lodi, konkrétně zvuk výbuchu lodi, jež bude obsahovat více zvukových souborů, přičemž při přehrávání výbuchu lodi se přehraje náhodně jeden z nich. Zvuk pak budeme spouštět událostí `explode`.

V načítacím celku `level1` si proto vytvoříme složený zdroj zvuku:

Název souboru	← enemySound.xml	
Složka pro uložení	← Sounds\levelShared	Jedná se o relativní cestu od kořene složky projektu

Nově vytvořený zdroj zvuku

Name	← enemySound	

Následně si otevřeme editor SoundEditor. V tomto editoru následně přiřadíme zvuku prvek PACK, a do něj přiřadíme prvky SOURCE:

1. SOURCE

Name	← explosion1	
Src	← Sounds\explosions\explosion1.wav	Relativní cesta

2. SOURCE

Name	← explosion2	
Src	← Sounds\explosions\explosion2.wav	Relativní cesta

3. SOURCE

Name	← explosion3	
Src	← Sounds\explosions\explosion3.wav	Relativní cesta

Poté si vytvoříme seznam zvukových souborů typu random box:

Name	← explosionsSoundList
SoundSources	← explosion1, explosion2, explosion3

Následně si vytvoříme novou událost pro spuštění zvuku výbuchu:

Name	← explode

V této události následně vytvoříme akci typu start action pro spuštění přehrávání seznamu zvukových souborů:

PlayBoxIndex	← 0
Name	← playRandomExplosion

Zdroj zvuku „level1Music“

Posledním zvukem načítacího celku level1 bude hudba 1. herní úrovně. Ta se bude skládat ze dvou zvukových souborů s hudbou, konkrétně se zvukovým souborem hudby přehrávaným v průběhu herní úrovně a zvukovým souborem hudby přehrávaným na konci herní úrovně (při boji s bossem), přičemž chceme, aby se vždy přehrával jen jeden z nich ve smyčce. Přepnutí pak provedeme pomocí události bossMusic.

Začneme vytvořením složeného zdroje zvuku v načítacím celku level1:

Název souboru	← level1Music.xml	
Složka pro uložení	← Sounds\level1	Jedná se o relativní cestu od kořene složky projektu

Nově vytvořený zdroj zvuku

Name	← level1Music

Následně si otevřeme SoundEditor a zvuku přiřadíme prvek PACK. Do prvku PACK pak vložíme dva prvky SOURCE ONEUSE (podobně jako u hudby v menu), z důvodu, že nechceme, aby se načítali zvukové soubory celé do paměti:

1. SOURCE ONEUSE

Name	← baseMusic1	
Src	← Sounds\music\level1sound.wav	Relativní cesta

UniqueName	← level1baseMusic1	

2. SOURCE ONEUSE

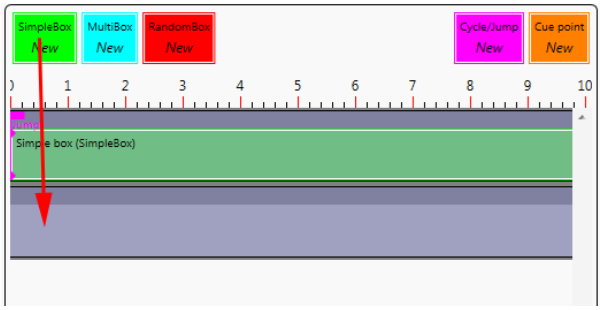
Name	← bossMusic1	
Src	← Sounds\music\bosssound.wav	Relativní cesta
UniqueName	← level1bossMusic1	

Nyní potřebujeme nastavit, jak budeme přehrávat zvukové soubory. Přehrávání zvukových souborů si definujeme pomocí časových os, otevřeme si tedy `TimeLine editor`. Nejdříve se zaměříme na hudbu během herní úrovně. Budeme chtít přehrávat jeden zvukový soubor, a pokud bude herní úroveň delší, tak chceme zvukový soubor přehrávat dokola. Neboť se bude jednat o jeden zvukový soubor, tak do první časové osy (šedý řádek) vložíme prvek `SimpleBox`:

BoxName	← BaseMusic	
SoundSource	← baseMusic1	Zvukový soubor, který jsme nastavili v SoundEditoru
StartTime	← 0	Chceme, aby se přehrával zvukový soubor ihned po spuštění 1. časové osy
Length	← 105000	Délka zvukového souboru (v ms)

Dále vložíme do první časové osy prvek `Cycle/Jump`, abychom pomocí něj zařídili opakování přehrávání zvukového souboru:

JumpTime	← 104750	Udává, z jakého času budeme skákat
TimeTarget	← 0	Udává, kam budeme skákat po časové ose, zde budeme chtít začátek časové osy.
Fadeout	← 250	Dozvuk původního zvuku
EnabledCycles	← -1	Nekonečné opakování
Name	← Jump	



Obrázek 7.15 - TimeLine editor - vkládání do další časové osy

Další bude hudba na konci herní úrovně (při boji s bossem), tu budeme chtít přehrávat zvlášť od hudby během herní úrovně. K tomu využijeme další časovou osu, kde pro přepnutí hudby využijeme událost, jež přepne přehrávání na požadovanou časovou osu. Vložíme tedy opět prvek `SimpleBox` tentokrát do druhé časové osy (šedý řádek pod první časovou osou viz obrázek 7.15):

BoxName	← BossMusic	
SoundSource	← bossMusic1	Zvukový soubor, který jsme nastavili v SoundEditoru

StartTime	← 0	Chceme, aby se přehrával zvukový soubor ihned po spuštění 2. časové osy
Length	← 151000	Délka zvukového souboru (v ms)

Poté vložíme do druhé časové osy prvek `Cycle/Jump`, abychom zajistili opakování přehrávání zvukového souboru:

JumpTime	← 150750	Udává, z jakého času budeme skákat
TimeTarget	← 0	Udává, kam budeme skákat po časové ose, zde budeme chtít začátek časové osy.
Fadeout	← 250	Dozvuk původního zvuku
EnabledCycles	← -1	Nekonečné opakování
Name	← JumpBoss	

Nyní ještě potřebujeme vytvořit událost `bossMusic` pro přepínání hudby z hudby během herní úrovně na hudbu na konci herní úrovně. Vytvoříme si tedy novou událost:

EV

Name	← bossMusic

V události pak vytvoříme akci, jež zařídí skok z první časové osy na druhou časovou osu. Protože ale budeme chtít skok s přechodem mezi hudbami, tak použijeme parametrický skok, neboli `paramJump action`:

Name	← JumpToBossMusic	Název akce
PlayingTimeLineIndex	← 0	Index jde od hodnoty 0. Nastavuje, že se akce provede, jen když se přehrává první časová osa (hudba během herní úrovně)
TargetTimeLine	← 1	Index cílové časové osy (zde druhá časová osa)
FadeoutTime	← 1000	Čas dozvuku původního zvuku (v ms)
MinTime	← 0	Nastaví se automaticky. Udává, že se skok projede, jen když je čas časové osy větší než zadaná hodnota
MaxTime	← 151000	Nastaví se automaticky. Udává, že se skok projede, jen když je čas časové osy menší než zadaná hodnota

Nyní je ještě potřeba nastavit cílový čas skoku. To se u parametrického skoku provádí pomocí grafu, který otevřeme kliknutím na `ProjectView → level1Music → Events → bossMusic → JumpToBossMusic → Show graph`. Osa `X (Time)` grafu je čas, ze kterého se skáče (čas časové osy při skoku) a osa `Y (time)` udává čas, na který se má skočit. Zde budeme chtít skákat na začátek časové osy, takže nastavíme body:

```
[0, 0], [151000, 0]
```

Tímto máme nastavenou událost a dokončený zvuk hudby v první herní úrovni.

Přidání sdílených zvuků do načítacího celku „level2“

Nyní jsme dokončili všechny zvuky načítacího celku `level1` (první herní úrovně), jako další nás čeká načítací celek `level2` (druhá herní úroveň). Ten bude

obsahovat většinu zvuků z načítacího celku `level1`, konkrétně bude obsahovat `getBonusLife`, `getBonusShield` a `enemySound`.

Sdílení provedeme následujícím způsobem. Po prvním vytvoření sdílených zvuků mezi načítacími celky (což jsme udělali v načítacím celku `level1`) musíme nejdříve projekt uložit, aby se zapsali informace do všech souborů. Po tomto kroku už můžeme kliknout na `ProjectView → level2 → Add existing item`, tím se nám otevře dialogové okno pro výběr souboru. Vybereme tedy soubor `Sounds\levelShared\getBonusLife.xml` (zdroj zvuku `getBonusLife`). Tím se nám přidá do načítacího celku `level2` zdroj zvuku, který máme v načítacím celku `level1`. Stejným způsobem přidáme do načítacího celku `level2` i soubory `Sounds\levelShared\getBonusShield.xml` (zdroj zvuku `getBonusShield`) a `Sounds\levelShared\enemySound.xml` (zdroj zvuku `enemySound`). Tím jsme si přidali sdílené zvuky, ale ještě musíme dovytvořit zvuky, které jsou specifické pro druhou herní úroveň.

Pozor, při změně zdroje zvuku v jednom načítacím celku se mění zdroj zvuku i v druhém načítacím celku, neboť se jedná o stejný zdroj zvuku!

Zdroj zvuku „redLaser“

První ze zvuků druhé herní úrovně bude zvuk červeného laseru. Postup bude stejný jako u zeleného laseru, s rozdílem jiného zvukového souboru.

Vytvoříme si tedy v načítacím celku `level2` složený zdroj zvuku:

`Název souboru`	`← redLaser.xml`	
`Složka pro uložení`	`← Sounds\levelShared`	Jedná se o relativní cestu od kořene složky projektu

`Nově vytvořený zdroj zvuku`

`Name`	`← redLaser`

V editoru `SoundEditor` pak vložíme prvek `PACK`, do nějž vložíme prvek `SOURCE`:

`Name`	`← laser`	
`Src`	`← Sounds\lasers\redLaser.wav`	Relativní cesta
`Volume`	`← 0.5`	

Následně vytvoříme seznam zvukových souborů typu `random box`:

`Name`	`← redLaserSoundList`
`SoundSources`	`← laser`

Poté vytvoříme událost pro spuštění zvuku výstřelu:

`Name`	`← fire`

V této události vytvoříme akci pro spuštění seznamu zvukových souborů `start action`:

`PlayBoxIndex`	`← 0`
`Name`	`← playRedLaserSoundList`

Zdroj zvuku „level2Music“

Druhý a poslední zvuk druhé herní úrovně je hudba druhé herní úrovně. Postup bude stejný jako u první herní úrovně s rozdílem zvukových souborů.

Nejdříve si vytvoříme v načítacím celku `level2` složený zdroj zvuku:

Název souboru	← level2Music.xml	
Složka pro uložení	← Sounds\level2	Jedná se o relativní cestu od kořene složky projektu

Nově vytvořený zdroj zvuku

Name	← level2Music	

V editoru SoundEditor zdroji zvuku přiřadíme prvek PACK. Do tohoto prvku pak vložíme dva prvky SOURCE ONEUSE, jako u hudby první herní úrovně s rozdílem těchto položek:

1. SOURCE ONEUSE

Src	← Sounds\music\level2sound.wav	Relativní cesta
UniqueName	← level2baseMusic1	

2. SOURCE ONEUSE

UniqueName	← level2bossMusic1	

Poté otevřeme editor TimeLine editor a vložíme do něj prvky jako u hudby první herní úrovně. SimpleBox první časové osy se bude lišit touto položkou:

Length	← 100000	Délka zvukového souboru (v ms)

Prvek Cycle/Jump první časové osy se pak bude lišit touto položkou:

JumpTime	← 99750	Udává, z jakého času budeme skákat

Prvky SimpleBox a Cycle/Jump druhé časové osy pak budou nastavené stejně jako u hudby první herní úrovně.

Poté si ještě vytvoříme událost pro přepínání hudby z hudby během herní úrovně na hudbu na konci herní úrovně jako u hudby první herní úrovně, zde budou všechny položky nastaveny stejně.

Tím jsme vytvořili zvuk hudby druhé herní úrovně.

Přidání sdílených zvuků do načítacího celku „level3“

Nyní máme dokončený i načítací celek `level2` a zbývá nám načítací celek `level3`. Ten bude obsahovat většinu zvuků z načítacího celku `level1` a `level2`, konkrétně bude obsahovat `getBonusShield`, `redLaser` a `enemySound`. Dále pak bude obsahovat vlastí zdroj zvuku `level3Music` pro hudbu třetí herní úrovně.

Pro přidání sdílených položek nejdříve uložíme projekt, aby se zapsali informace do všech souborů. Poté přidáme následující zvuky:

Sounds\levelShared\getBonusShield.xml	getBonusshield
Sounds\levelShared\redLaser.xml	redLaser
Sounds\levelShared\enemySound.xml	enemySound

Zdroj zvuku „level3Music“

Nyní nám zbývá vytvořit zdroj zvuku `level3Music`. Ten bude stejný jako v první a druhé herní úrovni s rozdílem pár položek. V načítacím celku `level3` si tedy vytvoříme složený zdroj zvuku:

`Název souboru`	`← level3Music.xml`	
`Složka pro uložení`	`← Sounds\level3`	Jedná se o relativní cestu od kořene složky projektu

`Nově vytvořený zdroj zvuku`

`Name`	`← level3Music`	

Poté si otevřeme editor `SoundEditor` a přidáme prvek `PACK`. Dále přidáme dva prvky `SOURCE ONEUSE`, jako u hudby první a druhé herní úrovně s rozdílem těchto položek:

`1. SOURCE ONEUSE`

`Src`	`← Sounds\music\level3sound.wav`	Relativní cesta
`UniqueName`	`← level3baseMusic1`	

`2. SOURCE ONEUSE`

`UniqueName`	`← level3bossMusic1`	

Následně si otevřeme editor `TimeLine editor` a vložíme do něj prvky jako u hudby první a druhé herní úrovně. `SimpleBox` první časové osy se bude lišit touto položkou:

`Length`	`← 137000`	Délka zvukového souboru (v ms)

Prvek „Cycle/Jump“ první časové osy se pak bude lišit touto položkou:

`JumpTime`	`← 136750`	Udává, z jakého času budeme skákat

Prvky `SimpleBox` a `Cycle/Jump` druhé časové osy pak budou nastavené stejně jako u hudby první a druhé herní úrovně.

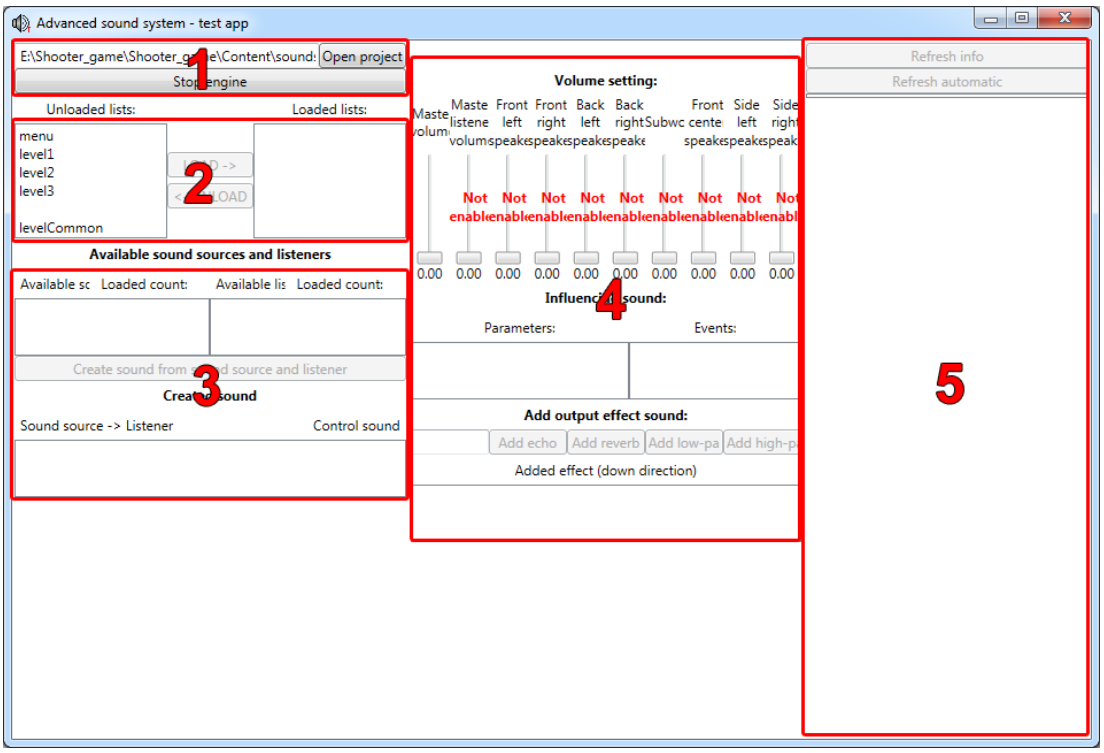
Poté si ještě vytvoříme událost pro přepínání hudby z hudby během herní úrovně na hudbu na konci herní úrovně jako u hudby první a druhé herní úrovně, zde budou všechny položky nastaveny stejně.

Závěr

Tímto jsme vytvořili poslední zvuk našeho projektu pro hru *„Shooter game“*. Projekt uložíme a můžeme si ho vyzkoušet pomocí testovací aplikace, jež je popsána v následující kapitole *7.2 Testovací aplikace editoru*.

7.2 Testovací aplikace editoru

Testovací aplikace editoru slouží k otestování projektu zvuků, který designér vytvoří v editoru zvuků (viz obrázek *7.16*). Aplikace umožňuje designérovi zkontrolovat, zda vše vytvořil správně. Dále si v této části popíšeme, jak si můžeme vyzkoušet přehrávání zvuků z načeho projektru pro hru „*Shooter game*“.



Obrázek 7.16 - Okno testovací aplikace

7.2.1 Rozložení okna

Okno testovací aplikace je rozděleno na pět základních částí (viz obrázek *7.16*):

1. Načítání projektu zvuků
2. Načítání a odnačítání načítacích celků
3. Vytváření, spouštění a destrukce zvuků
4. Nastavení přehrávání vybraného zvuku
5. Informace o přehrávání zvuku

7.2.2 Otevření jiného projektu

Pokud spouštíme testovací aplikaci z editoru zvuků, tak se načte právě editovaný projekt. Pokud jsme ale testovací aplikaci spustili jako samostatnou aplikaci, tak nebude načtený žádný projekt.

K otevření (jiného) projektu slouží tlačítko `Open project`, které je umístění v první části editoru `Načítání projektu zvuků` (viz výše kapitola *7.2.1 Rozložení okna*). Po stisknutí tlačítka se otevře dialogové okno, pomocí nějž otevřeme příslušný soubor s projektem zvuků. Testovací aplikace přitom kontroluje,

zda se jedná o správný typ souboru (soubor s projektem zvuků). Při otvírání nového projektu se zastaví a zničí všechny vytvořené přehrávané zvuky, přičemž dále už bude možno pracovat jen se zvuky právě otevřeného projektu!

Kromě načítání projektu je součástí této části i tlačítko pro spouštění a zastavování celého audio enginu. Jedná se o tlačítko `Start engine` či `Stop engine` (podle stavu audio enginu) hned pod načítáním projektu.

Nyní si otevřeme projekt hry „Shooter game“ v testovací aplikaci, to uděláme kliknutím na `Run project` v menu editoru při editaci projektu hry. Tím se nám spustí testovací aplikace, přičemž se načte projekt hry, takže budeme mít k dispozici několik nenačtených načítacích celků.

7.2.3 Načítání a odnačítání načítacích celků

Další částí aplikace je načítání a odnačítání načítacích celků. K tomu nám slouží druhá část okna (viz výše kapitola *7.2.1 Rozložení okna*). Kde vlevo je seznam načítacích celků, které nejsou načtené, a vpravo jsou načítací celky, které jsou načtené.

Načteme si tedy načítací celek `level1`, to uděláme tak, že načítací celek `level1` označíme myší tak, že klikneme na jeho název a klikneme na tlačítko `LOAD ->`. Tím se nám v testovací aplikaci zobrazí několik zdrojů zvuků a posluchači, které obsahuje načítací celek `level1`.

Odnačítání načítacích celků probíhá obdobně, kdy si myší označíme načítací celek, který chceme odnačíst a klikneme na tlačítko `<- UNLOAD`. Ale pozor, při odnačtení načítacícho celku se zničí (odnačtou) všechny zvuky, jež využívají zdroje zvuku, či posluchače z načítacího celku, který nepatří do jiného načteného načítacího celku!

7.2.4 Vytváření a mazání zvuků

Nyní máme k dispozici zdroje zvuku a posluchače, jež jsou umístěné v třetí části okna (viz výše kapitola *7.2.1 Rozložení okna*). Z těchto dvou prvků nyní můžeme vytvářet zvuky. Například si vytvoříme zvuk štítu lodi, k tomu využijeme zdroj zvuku `playerShip` a posluchače `stereoListener`. Označíme si myší zdroj zvuku `playerShip` a posluchače `stereoListener`, poté klikneme na tlačítko `Create sound from …`, jež nám ze zdroje zvuku a posluchače vytvoří instanci zvuku označenou `playerShip -> stereoListener`.

Pokud chceme některou instanci zvuku zničit (odnačíst), stačí kliknout na tlačítko `Destroy` příslušné instance zvuku, tím se zvuk zastaví (pokud se přehrával) a vymaže z paměti.

7.2.5 Přehrávání zvuků

Když máme vytvořenou instanci zvuku, tak jí můžeme začít přehrávat, to uděláme tak, že klikneme na tlačítko `StartPlaying`, tím se spustí přehrávání instance zvuku. Ne všechny zvuky lze spustit tímto způsobem. Například u některých složených zvuků se přehrávání spouští pomocí událostí (popíšeme níže).

Pro pozastavení přehrávání instance zvuku slouží tlačítko `StopPlaying`, po kliknutí na tlačítko se přehrávaný zvuk pozastaví. Pro opětovné spuštění zvuku (pokračování přehrávání) stačí kliknout na tlačítko `StartPlaying`. Pro úplné zastavení přehrávání zvuku (přetočení na počátek) je nutné dvakrát kliknout na tlačítko `StopPlaying`, v případě pozastaveného zvuku stačí kliknout na tlačítko `StopPlaying` jen jednou.

7.2.6 Nastavování hlasitosti, parametrů a volání událostí zvuku

Při přehrávání instancí zvuků možná bude designér potřebovat upravit jejich hlasitost, parametry, či zavolat jejich události. K tomu slouží čtvrtá část okna (viz výše kapitola *7.2.1 Rozložení okna*).

Pro úpravu hlasitosti slouží horní část se vertikálními slidery. Pomocí sliderů poté nastavíme hlasitost instance zvuků. První slider slouží jako hlavní hlasitost pro všechny zvuky. Druhý slider slouží jako hlasitost posluchače, neboli ovlivní se hlasitost všech zvuků, které využívají daného posluchače. Další slidery pak slouží k nastavení hlasitosti instance zvuku na jednotlivých kanálech. Pokud má instance zvuku stereo výstup, tak můžeme ovlivňovat dva kanály, levý a pravý, zatímco u 7.1 výstupu můžeme ovlivňovat všech osm kanálů.

Pro úpravu parametrů zvuku slouží horizontální slidery uprostřed části okna. Slidery jsou nastavené tak, že mají rozsah od minimální do maximální hodnoty parametru, přičemž změna parametru se provádí v průběhu posouvání slideru.

Pak také na zvucích můžeme volat různé události, které jim jsou přiděleny. K tomu slouží tlačítka po pravé straně uprostřed části okna, jež jsou pojmenovaná jmény událostí. Událost vyvoláme kliknutím na příslušné tlačítko.

Na naší vytvořené instanci zvuku `playerShip -> stereoListener` si můžeme vyzkoušet ovládání hlasitosti, pak také spuštení zvuku štítu pomocí události kliknutím na tlačítko `startShield` a poté ovládání nastavování parametrů, konkrétně parametru `shield`. Kdy pro parametry 0-33, 34-66, 67-100 zvuk štítu zní jinak.

7.2.7 Přidávání vlastních (vnějších) efektů

Na instanci zvuku pak také můžeme aplikovat efekty echo, reverb, low-pass filter, high-pass filter. K tomu slouží dolní část čtvrté části okna. Kdy v této části nahoře máme seznam dostupných efektů, v levé části pak přidané efekty na instanci zvuku a v pravé části potom možnost upravovat parametry přidaných efektů. Efekt přidáme tak, že klikneme na jedno tlačítko z horní části. Tím se nám zařadí efekt na konec přidaných efektů. Efekt můžeme odebrat jeho vybráním a kliknutím na tlačítko `Remove effect`. Parametry jednotlivých efektů pak můžeme editovat po jejich označení (kliknutím myší na daný efekt), přičemž každý typ efektu má svoje parametry.

7.2.8 Výpis informací o instanci zvuku

Poslední částí aplikace je výpis informací o intanci zvuku. Tato část je umístěná v páté části okna aplikace (viz výše kapitola *7.2.1 Rozložení okna*). Slouží pro zjištění nastavených hodnot jednotlivým částem instance zvuku.

Výpis informací není stantartně spuštěn, neboť je náročný pro výpis hodnot. K výpisu aktuálních hodnot v určitém okamžiku slouží tlačítko `Refresh info`. Průběžný výpis hodnot pak lze spustit tlačítkem `Refresh automatic`, přičemž stejným tlačítkem se pak průběžný výpis také vypíná.

8 Závěr

V této práci jsme vytvořili audio engine *Advanced Sound System*, jež ulehčuje přehrávání zvuků, ve hrách, případně programech pracujících se zvukem, ve smyslu přehrávání zvukových objektů a ne přehrávání samostatných zvukových souborů. Pro zvukové designéry jsme pak navrhli a vytvořili visuální editor, jenž umožňuje definovat zvuky, které bude následně přehrávat náš audio engine.

Součástí práce je analýza, kdy jsme zkoumali již existující audio enginy a navrhovali strukturu vlastního audio enginu. Následně jsme testovali různé dolní vrstvy, jež bychom mohli v této práci použít. Tato část byla ztížena omezenou dokumentací některých dolních vrstev, tudíž trvalo déle pochopit princip fungování daných dolních vrstev.

Další prací byl vývoj samotného audio enginu a editoru. S touto částí pak vyvstala potřeba vývoje testovacích aplikací. Jedná se o aplikaci pro samotné testování zvuků vytvořených v editoru (je spustitelná z editoru) a jednoduchou hru, jenž testuje samotný audio enginu, a která zároveň slouží jako ukázka použití audio enginu.

Poslední částí práce bylo vytvoření dokumentací a doprovodného textu.

8.1 Zhodnocení (dosažené cíle)

Zjistili jsme, že původní představy dle kapitoly *1.3 Cíle* byly moc nadsazené. V kapitole *3.3 Vybrané cíle* jsme následně zhodnotili, že zúžíme cíle práce a že se budeme pokoušet splnit cíle *C1*, *C3*, *C4* a *C5*. Tyto cíle se nám podařilo splnit následujícím způsobem.

Prvním cílem práce (*C1*) bylo zachování přenositelnosti na jiné platformy. Jako dolní vrstvu této práce jsme sice využili knihovnu XAudio2, jež je dostupná jen pro platformu Windows. Pro zachování přenositelnosti jsme ale navrhli a implementovali mezivrstvu mezi horní (audio engine) a dolní (XAudio2 knihovna) vrstvou, jež zajistí přenositelnost audio enginu, kdy stačí upravit mezivrstvu podle konkrétní dolní vrstvy.

Dalším cílem práce (*C3*) byla práce vhodného načítání dat ze souborů. Pro ušetření paměti při načítání stejných dat jsme využili reference countingu a v případě velkých zvukových souborů jsme (pokud to bylo možné) pro ušetření paměti implementovali postupné načítání zvukových dat do paměti.

Součástí práce také bylo zajistit podporu pro přehrávání složených zvuků, což byl cíl práce (*C4*). Pro přehrávání složených zvuků jsme tedy navrhli a implementovali jednoduché rozhraní, jímž lze složený zvuk ovládat za běhu. Toto rozhraní pak umožňuje nastavovat složenému zvuku definované parametry a volat definované události, jež nám umožní ovlivňovat přehrávání složeného zvuku. Pro přehrávání složeného zvuku jsme dále vytvořili časové osy, jež umožní řízení přehrávání jednotlivých zvukových souborů, z nichž je složený zvuk složen.

Posledním cílem práce (*C5*) bylo vytvoření editoru. Volili jsme mezi různými návrhy (viz kapitola *2.5 Editor zvuků*), základní návrh byl zaměřit se na definování samotných dat pomocí odstupňovaného seznamu prvků a různých formulářů pro zadávání dat. Tento návrh se ale ukázal být komplikovaný pro designéra, neboť by se v tom designér špatně orientoval, vzhledem k závislosti mezi jednotlivými

prvky. Zvolili jsme tedy intuitivnější visuální návrh vzhledem k závislostem mezi jednotlivými prvky. Vytvořili jsme tedy speciální visuální prvky pro práci s časovými osami a pro skládání zvuků z jednotlivých zvukových prvků, kde uživatel může jednotlivé prvky přesouvat myší na správné místo. Díky tomu jsme docílili i lepší přehlednosti jednotlivých prvků v editoru.

Výše zhodnocené cíle tedy považujeme za dostatečně splněné, tak jak jsme si to předsevzali.

8.2 Budoucí práce (zlepšení)

V průběhu práce jsme vzhledem k cílům této práce přicházeli na určitá vylepšení, jež by zlepšila možnosti používání audio enginu, ale nestihli jsme je implementovat. Jedná se o následující vylepšení:

- Rozšíření základních efektů o další efekty, jako band-pass filtr (pásmová propust – propustí signál jen určitých frekvencí), případně můžeme naimplementovat plnohodnotný ekvalizér. Dále kompresor (pro zmenšení dynamického rozsahu zpracovávaného signálu) a případně rozšíření o další efekty (jež ale nemusejí být díky dolní vrstvě vždy k dispozici).
- Přidání podpory pro volání událostí na instancích zvuků s různými parametry, Typ parametrů by se pak mohl odvíjet od toho, jaké akce daná událost bude obsahovat. To by například ulehčilo přehrávání kroku hráče na různých površích, kdy bychom zavolali událost „Přehraj krok“ s parametrem udavajícím typ povrchu, po kterém hráč jde.
- Doprogramovaní experimentální části načítání nekonečných zvukových dat. Konkrétně doprogramování načítání nekonečných zvukových dat v metodě `LoadBuffer` ve třídě `ASS_InterLayer.Source` a doladění tříd `ASS_UpperLayer.BufferIn` a `ASS_UpperLayer.BufferOut`.
- Umožnit programátorům načítání jiných zvukových souborů než souborů typu WAV. Bylo by potřeba navrhnout a implementovat rozhraní na instanci audio enginu, jež by umožnilo registraci tříd implementující rozraní `IReader`, které by si programátoři vytvořili.

Jak jsme uvedli výše na začátku kapitoly *8.1 Zhodnocení (dosažené cíle)*, tak jsme zúžili cíle naší práce na cíle *C1*, *C3*, *C4* a *C5*. Zúžené cíle nás pak vedou k tomu, že by se na tuto práci mohlo navázat celou následující prací, která by náš audio engine posunula o další úroveň dál. Jednalo by se o abstraktnější přehrávání zvuku, kdy by se nespouštěly samotné instance zvuku tvořené zdrojem zvuku a posluchačem, ale spouštěly by se samotné zdroje zvuku definované ve virtuálním prostředí. V závislosti na pozici posluchače ve virtuálním prostředí hry by pak audio engine sám vytvářel a spouštěl potřebné instance zvuku tvořené zdrojem zvuku a posluchačem. V podstatě by se jednalo o splnění vynechaných cílů *C2*, *C6* a *C7* této práce, jež jsou uvedeny v kapitole *1.3 Cíle*.

9 Seznam použitých zdrojů

[1] *Digital game sales growing 33%*. GamesIndustry.biz. [online] 29.3.2013 [cit. 2014-09-11] Dostupné z: *http://www.gamesindustry.biz/articles/2013-03-29-digital-game-sales-growing-33-percent*

[2] *Monkey Island (series) – informace o hře*. Wikipedia, the free encyclopedia. [online] 2009 [cit. 2014-10-11] Dostupné z: *https://en.wikipedia.org/wiki/Monkey_Island_(series)*

[3] *Half-Life (video game) – informace o hře*. Wikipedia, the free encyclopedia. [online] [cit. 2014-10-12] Dostupné z: *https://en.wikipedia.org/wiki/Half-Life_(video_game)*

[4] *FMOD – domovská stránka audio enginu*. FMOD. [online] [cit. 2014-11-20] Dostupné z: *http://www.fmod.org/*

[5] *Wwise – domovská stránka audio enginu*. Audiokinetic. [online] [cit. 2014-11-20] Dostupné z: *https://www.audiokinetic.com/products/wwise/*

[6] *irrKlang – domovská stránka audio enginu*. Ambiera. [online] [cit. 2014-11-20] Dostupné z: *http://www.ambiera.com/irrklang/*

[7] *OpenAL – domovská stránka audio knihovny*. OpenAL: Cross Platform 3D Audio. [online] [cit. 2014-11-20] Dostupné z: *https://www.openal.org/*

[8] *XAudio2 – stránka věnovaná audio knihovny*. Microsoft. [online] [cit. 2014-11-20] Dostupné z: *https://msdn.microsoft.com/en-us/library/windows/desktop/ee415813(v=vs.85).aspx*

[9] *Tablet Marketing: 30 Must-Have Facts [Charts]*. Heidi Cohen. [online] 27.5.2011 [cit. 2015-02-02] Dostupné z: *http://heidicohen.com/tablet-marketing-30-must-have-facts-charts/*

[10] *Windows 8 Market Share Outpaces Vista, but Is Still Far Below Windows 7 and Windows XP. PC Perspective*. [online] 6.7.2013 [cit. 2015-02-06] Dostupné z: *http://www.pcper.com/news/Editorial/Windows-8-Market-Share-Outpaces-Vista-Still-Far-Below-Windows-7-and-Windows-XP*

[11] *SDL – domovská stránka knihovny SDL*. Simple DirectMedia Layer – Homepage. [online] [cit. 2015-04-06] Dostupné z: *https://www.libsdl.org/*

[12] *BASS audio library – stránka věnovaná audio knihovně*. Un4seen Developments. [online] [cit. 2015-04-15] Dostupné z: *http://www.un4seen.com/bass.html*

[13] *NAudio – stránka věnovaná audio knihovně*. CodePlex. [online] [cit. 2015-04-15] Dostupné z: *https://naudio.codeplex.com/*

[14] *OpenAL Soft – stránka věnovaná audio knihovně*. Software 3D Audio. [online] [cit. 2015-04-15] Dostupné z: *http://kcat.strangesoft.net/openal.html*

[15] *Tao – stránka věnovaná wrapperu audio knihovny*. Mono. [online] [cit. 2015-05-10] Dostupné z: *http://www.mono-project.com/archived/tao/*

[16] *OpenTK – domovská stránka wrapperu audio knihovny*. OpenTK. [online] [cit. 2015-05-10] Dostupné z: *http://www.opentk.com/*

[17] *SlimDX – domovská stránka wrapperu audio knihovny*. SlimDX. [online] [cit. 2015-05-25] Dostupné z: *https://slimdx.org/index.php*

[18] *SharpDX – domovská stránka wrapperu audio knihovny*. SharpDX. [online] [cit. 2015-05-25] Dostupné z: *http://sharpdx.org/*

[19] *AvalonDock (docking window control) – stránka věnovaná WPF prvku AvalonDock*. CodePlex. [online] [cit. 2016-03-06] Dostupné z: *https://wpftoolkit.codeplex.com/wikipage?title=AvalonDock&referringTitle=Home*

[20] *PropertyGrid – stránka věnovaná WPF prvku PropertyGrid*. CodePlex. [online] [cit. 2016-03-06] Dostupné z: *https://wpftoolkit.codeplex.com/wikipage?title=PropertyGrid&referringTitle=Home*

10 Příloha A – Struktura přiloženého DVD

Zdrojové kódy a ukázky implementace audio enginu do ukázkové hry jsou určeny pro Visual Studio 2010.

- `/Zdrojove_kody`
  - `/Advanced_Sound_System` – solution audio enginu s editorem
  - `/Shooter_game` – solution s ukázkovou hrou
- `/Dokumentace`
  - `/Advanced Sound System public.chm` – dokumentace generovaná ze zdrojových kódů (jen public)
  - `/Advanced Sound System.chm` – dokumentace generovaná ze zdrojových kódů (včetně private a internal)
- `/Aplikace_a_knihovny`
  - `/Advanced_Sound_System` – složka s přeloženými binárkami a knihovnami audio enginu a editoru
  - `/Shooter_game` – složka s přeloženou binárkou hry a potřebnými soubory
- `/Ukazka_editor` – složka s ukázkou editace audio projektu *Shooter game* pomocí editoru do ukázkové hry
  - `/Shooter_game_v1` – obsahuje základní složkovou strukturu s potřebnými zvukovými soubory
  - `/Shooter_game_v2_loadings` – obsahuje založený projekt s vytvořenými načítacími celky (zatím prázdnými)
  - `/Shooter_game_v3_menu` – obsahuje posluchače a všechny zvuky (zdroje zvuku) menu
  - `/Shooter_game_v4_levelCommon` – obsahuje navíc zvuky společné pro všechny herní úrovně
  - `/Shooter_game_v5_level1` – obsahuje navíc zvuky první herní úrovně
  - `/Shooter_game_v6_final` – obsahuje všechny zvuky pro hru *„Shooter game“*
- `/Ukazka_hra` – složka s ukázkou implementace audio enginu do ukázkové hry
  - `/Shooter_game_v1` – obsahuje solution hry bez implementace zvuku
  - `/Shooter_game_v2_base` – obsahuje solution hry se zapojeným audio enginem (zvuk není implementován)
  - `/Shooter_game_v3_menu` – obsahuje solution hry s implementovanými zvuky menu
  - `/Shooter_game_v4_level_base` – obsahuje solution hry s implementovanými základními zvuky herních úrovní
  - `/Shooter_game_v5_flying_objects` – obsahuje solution hry s předpřipraveným kódem pro implementaci zvuků létajících objektů
  - `/Shooter_game_v6_final` – obsahuje solution hry s hotovou implementací zvuků
  - `/Zvuky` – obsahuje složkovou strukturu se zvukovými soubory a soubory popisujícími jednotlivé zvuky
  - `/Zvukova_data` – obsahuje soubor se třídou MusicData
  - `/Knihovny` – obsahuje knihovny audio enginu
- `/Prace.pdf` – text této práce
- `/README.txt` – základní inforamce, struktura DVD