PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO
KATEDRA MATEMATICKÉ INFORMATIKY

DIPLOMOVÁ PRÁCE

Eliminace šumu při renderování hudby

2002

Aleš Keprt

Katedra matematické informatiky PřF UP Olomouc
Předpis pro vypracování diplomové práce

I.

Při zpracování diplomové práce student usiluje o co nejvýstižnější a nejpříznivější obraz o svých schopnostech, úrovni svých znalostí a osvědčuje, jak si osvojil nezbytné návyky odborného a technického způsobu vyjadřování, jaké má znalosti odborné literatury a jak ji umí používat.

Diplomová práce se v tomto smyslu hodnotí jako celek.

II.

Student se musí ve své práci bez újmy na úplnosti vyjadřovat stručně, odborně, slohově i gramaticky správně. Text (včetně popisků příloh) diplomové práce musí být před odevzdáním pečlivě prohlédnut. Písařské chyby, chyby v tisku apod., musí být opraveny.

Nedostatky diplomové práce v tomto směru snižují klasifikaci i práce jinak obsahově dobré.

III.

Text diplomové práce musí být zpracován programem LaTeX s použitím makra katedry matematické informatiky a vytištěn po jedné straně papíru formátu A4. Diplomová práce se odevzdává takto:

- 1x originál v knihařské vazbě
- 1x kopie spojená v polotuhých deskách
- 1x záznam textu diplomové práce pořízený textovým editorem na disketě, kterou si student před odevzdáním vyzvedne na sekretariátě katedry.

IV.

V originále se na stránku s místopřísežným prohlášením připojí vlastnoruční podpis.

Došlo-li v průběhu zpracování diplomové práce k významným odchylkám od zadání diplomového úkolu, které na základě žádosti studenta schválil vedoucí

katedry, připojí se další list, na němž budou tyto skutečnosti uvedeny. Způsob uvedení se stanoví individuálně a v originále bude k záznamu připojen podpis vedoucího katedry. Nevýznamné odchylky uvádí student v anotaci.

V.

Uspořádání odevzdaných vyhotovení diplomové práce se předepisuje následovně:

- předsádkový list
- zadání diplomové práce
- předpis pro vypracování diplomové práce
- místopřísežné prohlášení o samostatném vypracování diplomové práce
- (list s vyjádřením k odchylkám od zadání, pokud byly schváleny)
- anotace diplomové práce
- obsah diplomové práce s odkazy na odpovídající stránky nebo čísla příloh, seznam tabulek a obrázků
- rozvedení diplomového úkolu podle obsahu s dílčími závěry na konci každé kapitoly
- celkový závěr diplomové práce
- cizojazyčné resumé (cca 15 řádků)
- seznam použité literatury v abecedním autorském uspořádání – tabulky, nákresy, přílohy (doklady, prospekty, elaboráty apod.)

Přední deska originálu bude potištěna stejně jako předsádkový list, je možno vypustit znak univerzity.

Diplomová práce, která nebude vyhovovat tomuto uspořádání, nemůže být přijata.

VI.

Použitá literatura, předlohy apod., použité při zpracování musí být na příslušných místech v diplomové práci označeny odkazem na průběžné číslo ze seznamu použité literatury, příslušná stránka se uvede v hranaté závorce. Jde-li o citát, uvedou se čísla prvního a posledního řádku citátu.

Součástí řešení diplomového úkolu je zdrojový text programu. Způsobilost provozu produktu osvědčuje student při obhajobě diplomové práce.

VII.

Všechny propočty nebo výpočty musí být podrobně a přehledně uspořádány tak, aby každý odborník byl schopen jejich správnost přezkoušet. U použitých vzorců, součinitelů nebo hodnot z praxe musí být uveden původ. Jsou-li uváděny údaje, které mohou tvořit předmět hospodářského nebo státního tajemství, je třeba tuto okolnost uvést při odevzdání diplomové práce. V takovém případě se pro přístup k diplomové práci předepíše zvláštní režim (závazný i pro oponenty).

V Olomouci dne 13. března 1996

Doc. RNDr. Josef Hos
vedoucí katedry
matematické informatiky

Za správnost: Zd. Nesvadbová

Místopřísežně prohlašuji, že jsem celou práci vypracoval samostatně s použitím vypsané literatury.

28. března 2002 Aleš Keprt

Anotace

Cílem této diplomové práce je prozkoumat problematiku vzniku šumu při renderování hudby a možnosti jeho potlačení nebo úplného odstranění. Jádrem práce je velmi praktická aplikace různých poznatků z numerické matematiky, univerzální algebry a počítačové geometrie na zvuková a hudební data. Kromě samotné studie eliminace šumu práce přinesla několik zajímavých vedlejších produktů, jako je samotné renderování hudby, jeho algebraická formalizace, analýza situace na poli hudebních souborových formátů nebo podrobná studie interpolačních metod zvuku a dalších parametrů, které mají vliv na kvalitu počítačem renderované hudby. Při tom jsem také objevil velmi jednoduchý a praktický algoritmus pro softwarové zesilování basů ve zvuku. Výsledkem práce jsou také formulace a důkazy celé řady teoretických tvrzení a vět týkajících se dané problematiky.

Děkuji všem, kteří k vypracování této diplomové práce přispěli podnětnými připomínkami, radami a literaturou. Byli to zejména Mgr. Filip Rachůnek, Bc. Martina Chlupová, Mgr. Tomáš Skopal a vedoucí práce doc. RNDr. Václav Snášel, CSc.

Obsah

Seznam obrázků

1 Úvodní seznámení s problematikou

1.1 Použité termíny

Sloveso **renderovat**, ačkoliv se často používá v počítačové grafice, má svůj původní význam právě v hudbě. Renderování hudby je proces vytváření hudby na základě definovaných nástrojů, not a dalších parametrů. Smysl renderování je totožný s tím, když někdo hraje na hudební nástroj. Význam renderování hudby je v našem případě tedy v jakémsi nahrazení skutečných hudebníků počítačem. Přitom se nejedná jen o čistě matematickou záležitost, protože skuteční hudebníci nejsou stroje a jejich projev nelze simulovat jednoduchými vzorci. Přesněji řečeno to lze, ovšem pouze s nevalným výsledkem.

Tracker a sekvencer je klíčový článek při renderování hudby. Slouží k převádění vstupních dat, která jsou notové povahy (tedy vysoce abstraktní), do příkazů pro syntetizér. Jak už sám název napovídá, jde o jakési siamské dvojče.

Tracker je nejvyšší článek v hierarchii. Z jeho pohledu je hudba vysoce strukturovaná veličina, obsahuje paralelismy a opakování dílčích částí.

Z pohledu informatiky je tracker interpret jistého jazyka, který popisuje hudbu za pomocí not a dalších značek. Tento jazyk tedy není nepodobný klasickému zápisu not v notové osnově. Stejně jako interpretaci not z notové osnovy můžeme chápat jako určité provádění programu, podobně i tracker pracuje na bázi postupného zpracovávání příkazů ze vstupního programu. Problémem jsou pouze cykly a paralelizmy.

Sekvencer je nižší článek, který chápe hudbu na úrovni lineárního seznamu not a dalších značek. Je tedy schopen popsat všechny aspekty klasické notové osnovy, kromě opakování. Sekvencer a „sekvencovaná hudba“ v podobě MIDI souborů je nejčastěji používaným způsobem záznamu hudební tvorby na počítači[1].

Tracker a sekvencer tvoří jeden celek, který nepracuje se zvukem, ale pouze s hudbou. Je algoritmicky velmi rozsáhlý[2], i když jeho činnost není náročná na rychlost mikroprocesoru. Výsledkem sekvenceru je numerická reprezentace funkcí, které jsou na vstupu trackeru popsány programem nebo jiným předpisem.

Poznámka: Vzhledem k vysoké komplikovanosti celé problematiky a také k tomu, že tato otázka není hlavním tématem mé práce, budu se snažit brát tracker a sekvencer jako jeden celek.

Syntetizér je čistě technická záležitost, netýká se hudby, ale pouze zvuku. Syntetizér pracuje na velmi nízké úrovni (low level) a slouží pro samotné vytváření slyšitelného výsledku, tj. jeho vstupem jsou přesně časovaná data sekvenceru a výstupem je obyčejný zvuk (PCM wave).

[1] Zde je třeba rozlišovat mezi záznamem zvuku (MP3) a hudby (MIDI).

[2] To je způsobeno především nutností naučit tracker linearizovat složité struktury a převádět je na primitiva vhodná pro sekvencer.

Obyčejný syntetizér je z teoretického hlediska algoritmicky jednoduchý, ovšem počítačová implementace tohoto algoritmu je velmi obtížná, jelikož jde o práci se spojitými veličinami. Problém je především v rychlosti (musí počítat několik miliónů operací za sekundu). Nepřesností algoritmů, jejich realizací a reprezentací čísel v počítači navíc vznikají někdy docela výrazné výpočetní chyby. Ty se projevují jako šum ve zvuku.

1.2 Složitost syntetizéru

1.2.1 Obyčejný syntetizér

Abych demonstroval rozdíl mezi složitostí obecného algoritmu a jeho počítačové implementace, vysvětlím funkci syntetizéru. Jeho práce spočívá ve výpočtu výsledného zvuku, který chápeme jako funkci času $out(t)$, v závislosti na znalosti množiny vstupních zvuků $in_i(s_i(t))$ a jejich hlasitostí $v_i(t)$. Předpis pro výpočet $out(t)$ je pak *na papíře* velmi snadný:

$$out(t) = \sum_i v_i(t) \cdot in_i(s_i(t))$$

Při implementaci narážíme mimo jiné na tyto problémy:

- Všechny funkce jsou reprezentovány numericky, tj. pomocí seznamu funkčních hodnot v určitých uzlech.
- Funkce in je závislá na další funkci $s(t)$, nejedná se tedy o jednoduché zpracování funkčních hodnot v ekvidistantních uzlech.
- Z historických důvodů (chceme mít program kompatibilní s existujícími standardy) je třeba pracovat výhradně celočíselně. Při sčítání tedy neustále musíme provádět normalizaci, aby hodnoty mezivýsledků nepřerostly meze dané technickými prostředky.
- Hodnoty funkcí $v(t)$ a $s(t)$ dostáváme ze sekvenceru prakticky v reálném čase. Ačkoliv dávkování času je možné řídit podle potřeb syntetizéru, v daný okamžik t známe vždy pouze jednu funkční hodnotu funkcí $v(t)$ a $s(t)$.

Vzhledem k tomu, že syntetizér umí jen sečíst násobky dvojic funkcí, všechny ostatní výpočty dělá tracker a sekvencer. To se týká např. výpočtu zde uvedené funkce $v(t)$, která je ve skutečnosti součinem jiných komplikovaných funkcí.

1.2.2 Syntetizér s vícekanálovým výstupem

Pokud požadujeme stereofonní nebo obecně vícekanálový výstup, máme buď několik nezávislých syntetizérů, anebo speciální syntetizér, který vytváří všechny výstupy současně.

Funkční předpis pro vícekanálový syntetizér vypadá takto (j označuje výstupní kanály, n je jejich počet):

$$\forall j \in \langle 1; n \rangle : out_j(t) = \sum_i v_{i,j}(t) \cdot in_i(s_i(t))$$

Jelikož výpočet hodnot $in_i(s_i(t))$ je pro všechny výstupní kanály stejný, je zmíněný *speciální* syntetizér méně náročný na výkon počítače. Hlasitost $v_{i,j}(t)$ pak chápeme jako vektor $(v_{i,1}(t), \ldots, v_{i,n}(t))$, kde n je počet výstupních kanálů.

Uvedený předpis lze samozřejmě použít i pro vytváření prostorového zvuku, tj. pro vyšší počet kanálů než dva.

1.3 Renderování počítačové hudby

Hudební modul je soubor z některého existujícího hudebního editoru, který obsahuje hudební skladbu. Taková skladba se skládá ze vzorků hudebních nástrojů (aby hudba zněla na všech počítačích stejně), not, zvukových efektů a dalších souvisejících informací. Efekty jsou velmi důležitým článkem, nejpoužívanějšími jsou vibrato, tremolo, portamento, glissando a arpeggio (tyto názvy jsou z hudební teorie, pravděpodobně v italštině).

Hudební moduly se používají v mnoha variantách, mezi kterými často nelze bezztrátově konvertovat. V praxi se to řeší obvykle tak, že trackerů-sekvencerů je několik, anebo je tracker-sekvencer jeden a v některých bodech se kód liší podle typu souboru.

Činnost rendereru, tedy programu, který renderuje hudbu, se dá přirovnat k počítačové grafice a kreslení 3D scény s různými grafickými efekty. Výsledkem renderování je jedno nebo dvoukanálový zvuk (mono nebo stereo), který lze přehrát na zvukové kartě.

Poznámka ke složitosti: Systém Midas, což je momentálně asi jediný volně šiřitelný software realizující tuto problematiku na dobré úrovni, má 60000 řádků v jazyku C a 30000 řádků v Assembleru. Kód je přenositelný na různé operační systémy, kód v assembleru je volitelná součást pro optimalizaci běhu na procesorech Intel.

1.4 Eliminace šumu při renderování hudby

Jelikož při renderování hudby počítáme řádově milióny operací za sekundu, vznikají při výpočtu poměrně velké numerické chyby. Tyto chyby se projevují šumem ve výsledném zvuku. (Obecně platí, že jakékoliv chyby při zpracování zvuku se projevují šumem ve zvuku.)

Zde je tedy velký prostor k bádání a experimentování. Šum lze odstranit násilně **filtrací**, tedy pomocí metod matematické analýzy (na základě spojitosti

zvuku), to se dělá nejlépe **hardwarově**, ale lze to dělat i **softwarově**[3]. Toto není časově tolik náročné (řádově statisíce operací za sekundu). Algoritmů pro filtraci lze vymyslet hodně, ovšem žádný z nich není ideální.

Druhou možností je nedopustit, aby šum vznikl, a zaměřit se na chyby vzniklé reprezentací čísel v počítači a při operacích syntetizéru (řádově milióny operací za sekundu). Zde jde o aplikaci numerické matematiky, především **interpolace**.

Zajímavé jistě je podívat se na to, jakým způsobem může použití jednotlivých interpolačních a filtračních algoritmů ovlivnit kvalitu výsledného zvuku, stejně jako na jejich rychlost[4]. Hodnoty chyb lze také zobrazovat grafem: $x = čas$, $y = odchylka$. V tomto případě by samozřejmě šlo o porovnávání určité metody s jinou, tzv. referenční, metodou.

Jelikož eliminace šumu se týká pouze realizace syntetizéru, pomocným článkem a vlastně nutným základem této diplomové práce je vytvoření trackeru a sekvenceru. Takto tedy bude zajištěno, že se bude pracovat s reálnými daty. Těžko se můžeme zabývat experimentováním okolo šumu, aniž bychom pracovali v reálném prostředí.

[3]Softwarový algoritmus pochopitelně pouze simuluje chování hardwarového analogového obvodu.

[4]Otázka rychlosti zpracování zvuku je na dnešních počítačích již velmi okrajová.

2 Formáty souborů s trackovanou hudbou

2.1 Proč nás zajímají formáty

Chci pracovat v reálném prostředí, proto vycházím z existujících hudebních skladeb. Bohužel neexistuje žádný všeobecně uznávaný standard a různé hudební editory používají lehce odlišný způsob záznamu hudby, proto musím na úvod pečlivě zvážit, na kterých souborových formátech budu stavět.

Jak již bylo řečeno v úvodní kapitole, tracker funguje na bázi provádění programu v určitém jazyku. Tento jazyk je bohužel v každém typu souboru jiný, čili součástí mé práce je najít určité řešení této problematické situace. V mém programu jsem postupoval tak, že jsem navrhl vlastní jazyk a ke každém souborovému formátu jsem navrhl a napsal konvertor. Tyto konvertory fungují podobně jako např. program P2C, který konvertuje Pascal do jazyka C. Tento příklad je docela přesný, protože jazyky různých trackerů jsou sice odlišné, ale jsou si navzájem velmi podobné.

Při správném návrhu vlastního jazyka trackeru je napsání několika konvertorů určitě jednodušší, než napsání interpretů pro každý z těch původních jazyků. Můj jazyk tedy nemá žádnou přímou souvislost s obsahem souborů.

2.2 Historický vývoj

Existuje mnoho formátů, do kterých se ukládá trackovaná počítačová hudba. Na rozdíl od obyčejného MIDI souboru [24, 25], který obsahuje jen lineární seznam událostí s časovými razítky, tyto soubory obsahují strukturovaná data, vhodná k dalšímu editování ve vhodném hudebním editoru.

Za prapředky dnešních editorů jsou považovány programy pro počítač Commodore Amiga, který se roku 1987 stal prvním masově rozšířeným počítačem s možností využití v této oblasti. Neměl sice až tak rychlý mikroprocesor, ale obsahoval čtyřkanálový hardwarový syntetizér[5]. Původní formát **MOD** [7, 8, 9, 5, 6] byl postupně doplňován o nové příkazy na úrovni trackeru a později vznikla také jeho druhá varianta[6], která tu původní naprosto zastínila.

Další varianty **MOD**u, které často ani nejsou nijak označeny, jsou smutným důsledkem železného pravidla „co editor, to nový formát“.

V následujících letech došlo k masovému rozšíření této technologie a postupně vznikaly nové souborové formáty, které si kladly za cíl rozšířit omezené možnosti formátu **MOD** a také eliminovat jeho redundanci, tj.zkrátit ukládaná data na minimální velikost tím, že se nebudou ukládat přebytečné „nuly“.

Teprve v roce 1994 se na PC objevil editor, který naprosto přepsal historii. **Scream Tracker 3** [28]. Tento editor pro PC přinesl s novým formátem **S3M** [13, 14, 5, 6] přímo záplavu nových funkcí, podporoval až 16 kanálů a další kanály

[5] Čtyřkanálový znamená, že umí mixovat čtyři zvuky dohromady.

[6] známá jako M.K.

bylo možno využít pro ovládání AdLib[7]. Podnět pro vznik tohoto oblíbeného editoru zřejmě dala zvuková karta Ultrasound, která byla (také vzhledem k ceně) několik let nepřekonatelná a právě Scream Tracker 3 jako první uměl využít beze zbytku jejích schopností a přitom zachovával nutnou zpětnou kompatibilitu s **MOD**em a Sound Blasterem.

S rozvojem internetu přišel velký boom se sharewarovými a freewarovými hudebními editory, který znamenal jednak záplavu trhu nekvalitním softwarem a také přímo obrovskou záplavu skladeb, které jsou volně a legálně ke stažení na tisících stránkách autorů i speciálních serverech s touto tematikou.

Dodnes jediným editorem, který získal velkou popularitu bez toho, aby se snažil dodržet kompatibilitu s **MOD**em byl **Fast Tracker 2** [26] z roku 1996. Jako první přinesl do digitální hudby systém obálek, které se v profesionální praxi používají už několik desetiletí, ovšem pro trackovanou hudbu byly až do té doby naprosto nezvyklé. Fast Tracker 2 byl i v dalších letech updatován a dodnes je jeho souborový formát **XM** [15, 5, 6] používán mnoha hudebníky. Podle průzkumu serveru *www.modarchive.com* je Fast Tracker 2 v současné době nejpoužívanějším hudebním editorem.

Posledním hitem mezi hudebními editory je **Impulse Tracker 2** [30], který doslova smetl všechny ostatní editory vyjma Fast Trackeru 2 a se svým novým formátem **IT** [17, 6] přinesl spojení všech vlastností předchozích formátů dohromady. Díky technologii NNA navíc téměř zastínil MIDI, jelikož již neexistují téměř žádné limity na počet kanálů[8]. Uživatelské prostředí je identické tomu ze Scream Trackeru 3, v textovém režimu VGA v DOSu, pouze s pomocí speciálních textových fontů. Editor je napsán celý v assembleru a běží v reálném režimu procesoru, paměť nad 1MB tedy využívá pouze pomocí EMS. Na druhou stranu ve Windows 9x funguje velmi dobře a dokonce umí místo zvukové karty použít driver z Windows, takže funguje na každém počítači.

Impulse Tracker 2 je dodnes číslo jedna, jelikož všechny nové hudební editory, které by ho mohly zastínit, jsou prodávány za drahé peníze a nikdy se tedy nemohou tak rozšířit. Dostupné jsou samozřejmě i podobné hudební editory pro jiné operační systémy.

2.3 Koncepce hudebních modulů

2.3.1 Zavedení pojmu modul

Definice 2.1 (Song) *Soubor, který obsahuje kompletní skladbu z hudebního editoru a odkazy na soubory se samply (zvukovými vzorky nástrojů), nazýváme song.*

Definice 2.2 (Modul) *Soubor, který obsahuje kompletní skladbu včetně samplů, nazýváme modul.*

[7]Byl to historicky jediný docela zdařilý pokus skloubit digitální zvuk a AdLib OPL2 chip na zvukových kartách SoundBlaster/Pro.

[8]Slovo téměř je zde velmi důležité.

Uvedené definice pocházejí z počátků trackované hudby. Na začátku historického vývoje bylo možno ukládat samply mimo skladby. To je samozřejmě možné dodnes, ovšem soubory s hudbou již nemohou existovat bez samplů, čili říká se jim moduly.

2.3.2 Struktura modulu

Vnitřní struktura modulu se u různých souborových formátů liší. Tyto jsou sice nazývány podle obvyklé přípony souboru, ovšem toto značení není povinné, takže programy se při nahrávání modulů musí vždy dívat na jejich obsah, aby správně poznaly, o který formát se jedná.

Každý modul obsahuje tyto základní části:

Hlavička Ne vždy hlavička obsahuje nějaký identifikační řetězec, ovšem vždy tu najdeme základní informace o modulu, jako délka, rychlost, tempo, počet samplů apod.

Patterny Zápis not je v trackované hudbě uložen v patternech. Pattern je lineární seznam buněk s notami a příkazy. Buňky jsou v editorech na řádcích pod sebou a každá z nich reprezentuje určitý časový úsek[9]. Aby bylo možno hrát více not současně, patterny mají několik sloupců = tracků = stop. Každý z nich funguje nezávisle na ostatních a až na několik příkazů s globálním vlivem mezi sebou stopy neinterferují. V každé stopě může být zapsána v daném okamžiku jen jedna nota[10].

Editor patternů pak silně připomíná tabulkový editor a vlastně by i bylo možné udělat plugin do Excelu nebo jiného běžného tabulkového editoru, který by toto realizoval[11].

Samply Obvyklým způsobem realizace nástroje na počítači je sampl. Sampl se skládá z těla[12] a dalších nutných informací, které řozšiřují možnosti pouhého těla (jako např. loopy, které umožňují aby i dlouhý opakující se zvuk vystačil s krátkým tělem).

Nástroje Popis nástrojů v souboru se vyskytuje pouze v případech, kdy nevystačíme s obyčejnými samply.

[9] V drtivé většině jedna buňka odpovídá osminové notě v daném tempu, ale pomocí příkazů to lze měnit i během přehrávání.

[10] Kromě jedné výjimky, která bude probrána později.

[11] Klíčovou otázkou by pak byla rychlost provádění skriptů.

[12] Laicky řečeno je to totéž jako soubory WAV, VOC apod.

2.3.3 Tempo skladby

Důležitou vlastností každé skladby je její tempo. Tempo určuje rychlost dané skladby a v notovém zápisu bývá označováno slovně. Na počítači samozřejmě pracujeme s přesnými hodnotami. Tyto jsou uváděny jako BpM.

BpM (beats per minute, česky počet úderů za minutu) je všeobecně uznávaná jednotka pro určení tempa skladby. Je definována jako počet úderů bicích za minutu. Přitom se počítají pouze údery určující tempo skladby. Na počítačích je typicky 125 BpM, což je přímý důsledek frekvence PAL 50 Hz. Dřívější počítače se totiž připojovaly k běžnému televizoru a tudíž měly obnovovací frekvenci obrazu právě 50 Hz. Proto také časovač na těchto počítačích pracuje na stejné frekvenci. Za minutu je to tedy $50 \cdot 60 = 3000$ tiků PAL časovače. Potom pro běžné buňky odpovídající osminové notě připadá $8 \cdot 60 = 480$ buněk na minutu. Vydělením těchto dvou hodnot dostaneme $3000/480 = 6.25$, což je počet tiků PAL časovače připadajících na jednu osminovou notu. Jelikož tato hodnota musí být celočíselná, zaokrouhlíme ji na 6.0. A právě tímto zaokrouhlením dostaneme výslednou hodnotu 125 BpM.

Nové počítače a nové hudební skladby již používají libovolná tempa, obvykle v rozsahu 32-255 BpM. Tempo je možno dokonce měnit během skladby, což samozřejmě způsobuje komplikace numerickým operacím syntetizéru.

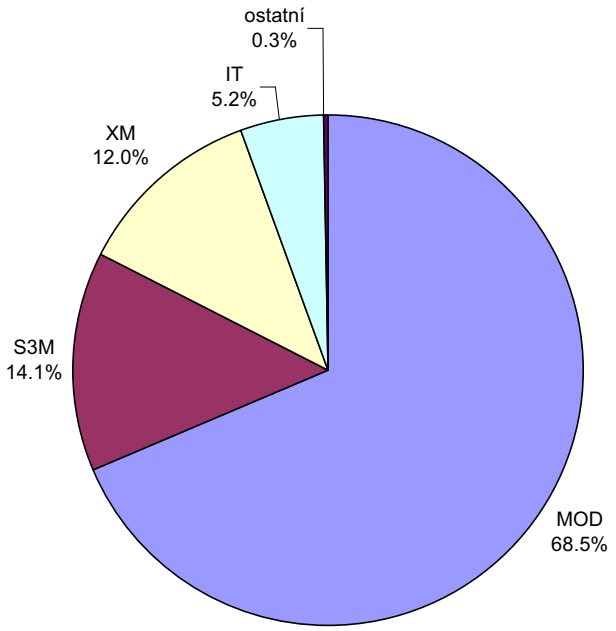
Poznámka: Tempo skladby souvisí především s žánrem. Např. reggae má obvykle tempo kolem 102 BpM, naopak taneční hudba má tempo 134 BpM a výše.

2.4 Podíl jednotlivých formátů

Z předchozího povídání je tedy patrné, že nejrozšířenějšími souborovými formáty pro ukládání počítačové hudby jsou **MOD**, **S3M**, **XM** a **IT** [7, 8, 9, 10, 13, 14, 15, 17, 5, 6]. Pro upřesnění jsem provedl (snad dostatečně rozsáhlý) průzkum, kdy jsem z internetu stáhl několik tisíc hudebních souborů, ty jsem si poslechl a vybral jsem z nich jen ty kvalitní (částečně subjektivně, částečně podle hodnocení na internetu).

Obrázek 1 ukazuje výsledky průzkumu. Opravdu mě překvapilo, jak je stále rozšířený prastarý **MOD**, který se kvůli svým omezením zdá být dnes už zcela nepoužitelný. Velkou měrou k tomu ovšem přispěl fakt, že tento formát se hodně používá na hudebních soutěžích, kde hudebníci rádi soutěží ve skládání hudby v určitých omezených podmínkách.

Když jsem do výsledků zahrnul pouze skladby za posledních pět let (viz obr.2), výsledky již byly poněkud odlišné. Formát **XM** ukázal největší nárůst a jeho podíl je skoro padesátiprocentní. Stejně tak další moderní formát **IT** ukázal velký vzestup a dostal se na úroveň **MOD**u. Staré formáty **MOD** a **S3M** naopak výrazně oslabily, ovšem i tak je jejich pozice ještě důležitá, zřejmě především kvůli zmíněným soutěžím, ze kterých stále přichází záplava nových skladeb, především ve formátu **MOD**.



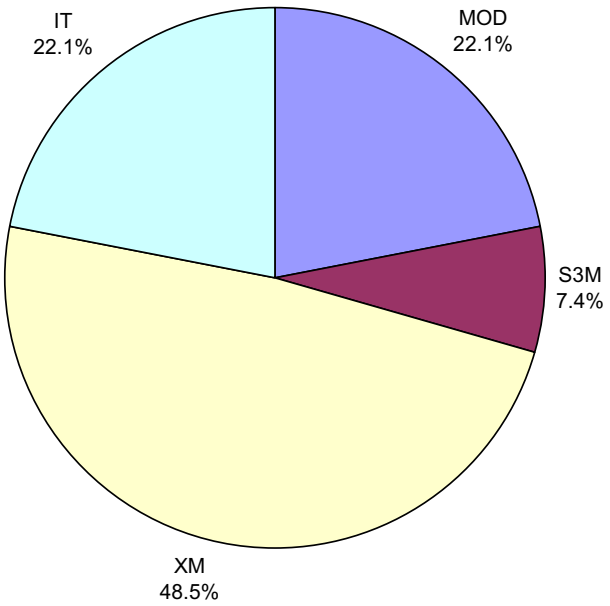
Obrázek 1: Celkový podíl souborových hudebních formátů

Zatímco v prvním grafu se ještě objevilo nepatrné procento „ostatních“ formátů, ve druhém už je zastoupena jen naše známá čtveřice. To nebylo nijak záměrné, výběr byl proveden z celé množiny skladeb dostupných na internetu. Přitom existují desítky dalších formátů, které jsou podporovány v některých editorech a jsou i dobře dokumentovány (viz seznam literatury). Tyto formáty však neposkytují žádnou novou funkčnost.

Porovnáním obou grafů jsem došel k následujícím závěrům: Podíl **XM** a **IT** zhruba čtyřnásobně vzrostl, přičemž jejich vzájemný poměr je zhruba nezměněný - **IT** vykazuje necelou polovinu **XM**.

Pokud sleduji počínání špičkových hudebníků, tak **MOD** ve skutečnosti nepoužívá vůbec nikdo, pokud nemusí, a autoři **S3M** přecházejí na **IT**, což je kvůli jeho stejné koncepci velmi snadné. **XM** je zřejmě mezi těmito lidmi nejrozšířenější, i když **IT** je také na vzestupu. Jelikož se žádný novější a lepší formát než **IT** nevyskytuje, očekávám, že tento trend bude i nadále pokračovat. Rozdělení „trhu“ mezi **XM** a **IT** zřejmě hodně ovlivní dostupnost kvalitního a přitom levného hudebního editoru pro Windows.

Z výsledků tedy vyplývá, že můj program by měl především podporovat formát **MOD** a pokud přidám **S3M**, **XM** a **IT**, dosáhnu stoprocentního řešení. Jelikož **MOD** je podmnožinou **S3M** a **S3M** je podmnožinou **IT**, je výhodné se zaměřit především na formát **IT**. Bohužel formát **IT** je složitější než všechny ostatní formáty dohromady, takže jeho plná podpora není v rámci diplomové práce realizovatelná. Ovšem vedlejším efektem složitosti **IT** je i fakt, že je možno téměř všechny soubory **XM** popsat pomocí prostředků **IT**. Čili když udělám



Obrázek 2: Podíl formátů na souborech z let 1997-2002

vnitřní formu trackeru a sekvenceru na bázi **IT**, měl bych být schopen napsat speciální loadery pro všechny čtyři uvedené formáty. To se mi také podařilo.

2.5 Formát MOD

Formát **MOD** je praotcem všech ostatních formátů. Používá se od roku 1987 a je velmi rozšířený především na Amize, protože slouží především k popisu skladeb pro hardwarový syntetizér na Amize. Z toho také plynou jeho omezení: rozsah pouze 3 oktávy, 12bitová logaritmická tabulka frekvencí, 4 kanály, každý má 6 bitů hlasitosti a pouze 8bitové samply do 64KB[13]. Stereo zvuk se nedá nijak nastavovat, vždy hrají dva kanály vlevo a dva vpravo. Logaritmická tabulka frekvencí je sice přesně podle reality, ovšem algoritmy na zvukové efekty pak dávají jiný výsledek v závislosti na bázové notě. Ve vyšších oktávách je tedy všechno daleko rychlejší. Např. glissando z noty C-1 na C-2 trvá přesně dvakrát déle než glissando z C-2 na C-3.

2.6 Varianty formátu MOD

Kolik editorů, tolik různých mutací formátu **MOD** existuje. Celý problém totiž vychází z toho, že od začátku nebyla oddělena specifikace od implementace.

[13]Ve skutečnosti byla maximální délka samplu 128KB, ale vzhledem k použití na 16bitových počítačích se v praxi dlouhé samply nikdy nevyskytovaly a většina programů je dodnes nepodporuje.

V praxi je tedy třeba držet se pravidla: „Specifikace formátu je určena chováním hudebního editoru.“ Jednotlivé mutace se kromě dále uvedených vlastností liší i množinou podporovaných příkazů trackeru. Každý nový program by tedy měl podporovat celou sadu existujících příkazů, aby nevznikaly komplikace. Kódování příkazů v souboru zaručuje, že jich existuje jen 31 a další již nemohou být přidány. Bohužel to má za následek, že některé z nich mají v různých editorech různý význam.

2.6.1 Původní formát - 15 samplů

Původní formát **MOD** [7, 9] měl omezení na 4 kanály, 15 samplů a 64 patternů. Každý pattern měl vždy 64 řádků ve 4 stopách a časování 125 BpM. Všechny ostatní mutace **MOD**u mají 31 samplů.

2.6.2 Mutace M.K. - 31 samplů, 4 kanály, 64 patternů

ProTrackerovský formát [7, 8, 9, 5, 6] je považován za nejstandardnější **MOD** a všechny ostatní hudební editory na bázi **M.K.** souborů by měly dodržovat kompatibilitu s ProTrackerem[14].

Ve skutečnosti tomu tak není, ovšem já si dovolím trvat na tom, že formát **M.K.** je jen jeden a fakt že na internetu je velká řada neplatných souborů **M.K.** musím připsat na vrub nízké kvalitě někdejších hudebních editorů. U některých z nich, jako např. Fast Trackeru, vedla jejich popularita až k tomu, že některé hudební přehrávače třetích stran umožňují nastavit režim kompatibility. Základním problémem však stále zůstává fakt, že neexistuje ani jeden hudební editor, který by vyloženě kontroloval, zda data zadávaná hudebníkem mají smysl. Zde uvedu jeden příklad sporné situace: Příkaz Axy dělá volume slide, čili zvyšuje hlasitost (když x je nenulové) nebo hlasitost snižuje (když y je nenulové). Stav, kdy jsou obě hodnoty nenulové, by měl být zakázán, ovšem z asi dvaceti hudebních editorů, které jsem zkoušel, ani jeden neprováděl žádnou kontrolu zadávaných hodnot. Každý hudební přehrávač se pak chová jinak.

1. Kolize je rozpoznána a příkaz je ignorován. Toto považuji za nejsprávnější řešení, ovšem v praxi není příliš vhodné, protože to znamená nekompatibilitu s ostatními programy.

2. Slide nahoru má přednost, druhá hodnota je ignorována.

3. Slide dolů má přednost, první hodnota je ignorována.

4. Hodnoty jsou odečteny od sebe navzájem, čili jsou provedeny oba příkazy současně.

[14] ProTracker je slavný editor na Amize a Atari ST. Popis jeho algoritmů je k dispozici ve formě zdrojového kódu v assembleru Motorola 68000.

5. Větší z obou hodnot má přednost, druhá je ignorována.

Ovšem to ještě není všechno. Dalším problémem jsou příkazy typu z00, protože nuly každý editor chápe jinak. Ukažme si možná chování na příkazu 9xx, který nastavuje pozici samplu na xx00 hexa. Při zadání 900 jsou následující možná chování:

1. Příkaz je ignorován.
2. Příkaz je vykonán, čili sampl se vrátí na začátek.
3. Příkaz je akceptován pouze společně s novou notou, čili nemá žádný význam, jelikož nová nota vždy hraje od začátku.
4. Příkaz je akceptován pouze bez nové noty, takže bez uvedení noty se sampl vrátí na začátek, ale při uvedení noty je příkaz ignorován.
5. Místo parametru je vzata jeho minulá platná hodnota. Je-li to první takový příkaz ve skladbě, pak se vezme nějaká naprosto náhodná hodnota, která je zrovna v paměti dané proměnné (tzv. metoda „totální chaos“).

Ovšem ani to ještě není všechno. Většina editorů se chová, zřejmě po vzoru ProTrackeru, tak, že si pamatuje poslední platnou hodnotu každého příkazu a při zadání nul jako parametru se tato minulá hodnota použije. A zde nastávají další problémy.

1. Hodnoty některých podobných příkazů používají společnou paměť, takže se navzájem přepisují.
2. Jiné editory naopak ukládají jiné dvojice nebo trojice příkazů do stejných proměnných, takže to nikdy nefunguje stejně.
3. Toto nahrazování nul je podporováno jen u jisté podmnožiny příkazů, zatímco u ostatních příkazů nikoliv.
4. Každý editor má samozřejmě tuto podmnožinu úplně jinou než ostatní editory. Často se to dokonce liší podle verze editoru.
5. Uživatel může do editoru zadat cokoliv, žádný editor nikdy nic nekontroluje.

Ani toto ještě není konec všem chybám, ovšem dál už pokračovat nebudu. Zřejmě by bylo dobré doplnit do programu validátor patternů, který by procházel celou skladbu a označoval neplatné nebo nejednoznačné konstrukce. Většinou jde právě o zmíněnou nejednoznačnost, která je ovšem typická i pro jiné oblasti informatiky, např. prohlížeče HTML stránek - dnes neexistuje prohlížeč, který by kontroloval a zaručeně spolehlivě vypisoval seznam chyb a nejednoznačností

v HTML stránkách. Scream Tracker sice dělá kontrolu při exportu do **MOD** souboru, ovšem tato kontrola často zcela nepochopitelně propustí vyloženě chybné konstrukce a naopak některé platné konstrukce označí za chybné a smaže je. Takové chování hudebních editorů bych si dovolil označit jako „za trest“.

Všechny uvedené i neuvedené problémy se bohužel týkají všech mutací formátu **MOD**, bez výjimky. Někteří *filutové* navíc záměrně mění identifikační řetězce ve svých souborech, takže tyto pak nejsou ostatními programy správně rozpoznány. Toto je ostatně nejrozšířenější forma „ochrany“ proti softwarovým pirátům.

2.6.3 Mutace M!K! - 31 samplů, 4 kanály, 128 patternů

Mutace **M!K!** [7, 9], je podobná jako **M.K.**, ale nemá omezení na 64 patternů. Počet patternů v **MOD** souboru je de facto omezen na 8 bitů, ovšem tabulka sekvencí má jen 128 buněk, takže používat více než 128 patternů je nesmyslné, protože není možné je zařadit do sekvence, čili nelze je přehrát. V této souvislosti je také třeba připomenout, že **MOD** kvůli svému designu neumí ukládat nepoužité patterny, takže hudebníci často přijdou o kus práce jen kvůli tomu, že soubor uloží v nevhodném okamžiku.

Jelikož řada hudebních editorů není schopna správně rozlišovat mezi **M.K.** a **M!K!**, všechny nové programy by měly akceptovat formát **M!K!** i pro soubory označené jako **M.K.**.

2.6.4 Mutace FLT4 a 4CHN

Mutace **FLT4** [7, 9, 11] a **4CHN** [7] jsou totožné s **M.K.** respektive **M!K!**. Byly používány pouze ve starých editorech, které vznikly před standardizací značení.

2.6.5 Mutace nCHN - 31 samplů, 2-8 kanálů

Mutace **8CHN** [7, 9, 11, 12] vznikla původně jako rozšíření **M!K!** na 8 kanálů. Zobecněním tohoto značení pak vznikla mutace **nCHN**, kde n je číslo 1-9. Většina editorů však podporuje pouze sudé hodnoty (2, 4, 6, 8) nebo dokonce ani nepodporuje nic menšího než 4.

Prokládání kanálů v patternu je intuitivní, čili pro osmikanálovou verzi je 12341234 nahrazeno 1234567812345678 a obdobně pro jiné počty kanálů.

2.6.6 Mutace nnCH - 31 samplů, 10-32 kanálů

Mutace **nnCH** [12, 26] je obdobou předchozí, ovšem pro dvouciferné počty kanálů. Větší hodnoty než 32 nejsou podporovány, ačkoliv tomu pochopitelně žádné technické bariéry nebrání. V mnoha programech jsou opět podporovány pouze sudé hodnoty.

2.6.7 Mutace CD81 a OCTA - 31 samplů, 8 kanálů

Tyto mutace [7, 9, 11] jsou identické s **8CHN**.

2.7 Méně obvyklé varianty formátu MOD

V této sekci uvedu méně obvyklé mutace formátu **MOD**, které drtivá většina programů vůbec nepodporuje, anebo si jen myslí, že je podporuje.

2.7.1 Mutace FLT8 - 31 samplů, 8 kanálů

Jednou z poněkud zvláštních mutací je **FLT8** [11]. Obdoba mutace **8CHN**, tato má však jiné prokládání kanálů v patternu. Každý osmikanálový pattern je rozdělen na dva čtyřkanálové a ty jsou v souboru uloženy za sebou.

2.7.2 Mutace WOW

Mutace **WOW** [10] se liší pouze v příponě souboru (**WOW** místo **MOD**), ale obsah je totožný s **M.K.** respektive **M!K!**. Některé soubory **WOW** však obsahují 8 kanálů bez toho, aby byly nějak označeny!

2.8 Formát S3M

Tento formát [13, 14] je uzpůsoben zvukovým kartám Ultrasound a SoundBlaster a počítačům PC. Z toho vyplývají následující vlastnosti: 8 oktáv, 64KB samply, 16 digitálních kanálů + AdLib, 256 patternů a 99 samplů.

Uvedené hodnoty jsou především omezeny možnostmi originálního editoru **Scream Tracker 3**, zatímco formát **S3M** umí daleko víc. Jelikož samply uložené v souboru obsahují standardní hlavičky S3S[15], samply mohou být libovolně dlouhé, 8 nebo 16bitové. Navíc mohou být kódovány v PCM nebo DP30ADPCM[16]. Originální editor Scream Tracker 3 však podporuje jen 8bitové samply do délky 64KB, stejně tak i mnohé další programy.

Počet kanálů je fyzicky omezen na 32 a počet samplů na 255. Mnohá rozšíření formátu **S3M** používá např. nový editor **Impulse Tracker 2** a pochopitelně také můj program.

Základní nevýhodou formátu **S3M** je, že se mapuje na segmenty paměti DOSu, čili celý soubor musí být dlouhý max. 1MB, přesněji řečeno začátky všech objektů musí být adresovatelné pomocí segmentu, takže za hranici 1MB může technicky vzato jít nějaký dlouhý sampl na konci, ale v praxi to není využíváno. V dnešní době tedy tento formát již neobstojí, navíc mohou teoreticky existovat

[15] Tento formát pochází z DigiPlayeru, programu pro práci se zvukem od stejných autorů jako je Scream Tracker. Označení S3S je neoficiální přípona pro tyto samply, odvozená od přípony modulu **S3M**.

[16] ztrátová komprese DigiPlayeru

soubory **MOD** delší než 1MB[17], které porušují pravidlo, že **MOD** je podmnožinou **S3M**. V praxi jsem ale viděl jediný **MOD**, který by byl delší než 500KB.

2.9 Formát XM

Tento formát [15] je velmi odlišný od všech ostatních. Používá totiž nástroje místo pouhých samplů, čili nabízí nesrovnatelně lepší možnosti než **S3M**. Omezení jsou: 8 oktáv, 32 kanálů, 256 patternů a 128 nástrojů, každý z nich má svých 16 samplů libovolného typu a délky. Samply jsou alokovány pro každý nástroj zvlášť a mapují se na noty podle mapovací tabulky. Je tedy teoreticky možno použít až 128*16=2048 samplů. **XM** je jediný formát, který toto umožňuje, i když v praxi se takové skladby samozřejmě nevyskytují.

Nově zavedený pojem nástroje byl první snahou změnit dříve hodně amatérský přístup k trackované hudbě a zavést určitou abstrakci. Nástroj je popsán lokální tabulkou samplů, obálkami a dalšími efekty. Tabulka samplů přiřazuje každé notě sampl, kterým je realizována, což dovoluje daleko věrnější interpretaci skutečných nástrojů, než umožňují výše uvedené formáty. Obálky jsou dvě funkce času, kterými je modulována hlasitost, resp. panning. Tyto funkce jsou zadány numericky, ovšem poloha jednotlivých uzlů v čase je libovolná (nejsou ekvidistantní) a dají se snadno editovat přímo v hudebním editoru. Definice obálek je omezena na 12 uzlů a čas max. 325 tiků (odpovídá 6.5s při 125 BpM).

2.10 Formát IT

Tento formát [17] vychází z **S3M**, ovšem přidává všechny prvky z **XM** plus mnoho nových vlastností. Největší výhoda **Impulse Trackeru** v podobě **NNA** se paradoxně do souboru projeví jen jako jeden flag-bit. Pokud **XM** neobsahuje více než 99 samplů, lze ho importovat do editoru a poté uložit ve formátu **IT**. Omezení na 99 samplů je opět dáno pouze editorem (2 cifry). Podobný původ má i zdejší omezení na 10 oktáv, což je však stále o dvě více než v ostatních editorech.

Nová volitelná vlastnost NNA přináší další úroveň abstrakce. Poprvé totiž není zvukový kanál pevně vázán na stopu notového zápisu, tj. jeden nástroj může hrát neomezené množství not současně. Má to však stále jeden háček - protože noty sice hrají současně, ale nemohou začít hrát současně, protože do každé buňky lze zapsat pouze jednu notu. To je samozřejmě jen detail, protože lze alokovat více stop pro jeden nástroj, ale je to jednoduše nelogické z programátorského hlediska. Např. MIDI (formát datové komunikace v hudbě) toto omezení nemá a přitom vznikl už na začátku 80.let.

Dalšího rozšíření se v **IT** dočkaly nástroje, protože tabulka samplů je nyní globální - samply tedy mohou být sdíleny mezi nástroji. Tabulka, která v rámci

[17]Maximální délka **MOD**u je při 128 patternech, 32 kanálech a 31 samplech celkem 3MB.

nástroje přiděluje notám samply, nově obsahuje i odkazy na noty. Můžeme si tedy snadno nastavit, že C-5 bude ve skutečnosti hrát A-7 apod. To má velký význam u bicích apod. Příchod NNA si vyžádal mnoho nových parametrů, kterými se definuje chování nástroje při souzvuku not v jedné stopě. A konečně obálky mohou mít až 25 uzlů v čase do 10000 tiků (200 sekund při 125 BpM).

2.11 Další formáty

Jak již bylo řečeno v úvodu, existuje ještě celá řada dalších souborových hudebních formátů. Vzpomeňme alespoň některé z nich, které byly kdysi poměrně populární, anebo přinesly určité zajímavé nové prvky.

2.11.1 Formát MTM

V první polovině 90.let 20.století byl jedním z nejoblíbenějších formátů také MTM [21]. Tento formát byl však pouze multikanálový **MOD**, který kromě podpory 32 kanálů a úsporného způsobu ukládání patternů nepřinesl nic nového. Přestal se používat s příchodem **S3M**.

2.11.2 Formát 669

Formát se zajímavým názvem **669** [22] byl navržen „na tělo“ zvukové kartě Ultrasound firmy Gravis, která byla až do příchodu karty Sound Blaster AWE32 špičkou mezi levnými kartami pro domácí použití.

2.11.3 Formát STM

Tento pohrobek po editoru Scream Tracker 2 [23] byl prvním nesmělým krůčkem k později slavnému formátu **S3M**. Tento je však více podobný původnímu **MOD**u a s příchodem třetí verze editoru se zcela vytratil.

2.11.4 Formát ALM

Velmi jednoduchý formát ALM [19] byl navržen pro použití na 8bitových počítačích Sam Coupé a ZX Spectrum, které mají mnohá hardwarová omezení. Proto je také tento formát hodně omezený, ovšem na druhou stranu jeho celý přehrávač má pouhých 600 bajtů (včetně potřebných frekvenčních tabulek apod.).

2.11.5 Formát ATM

Na rozdíl od ostatních formátů, **ATM** [20] je čistě umělý, tzn. není podložen žádným editorem, ale pouze snahou o zobecnění možností všech klasických formátů a jejich standardizaci do jednoho vhodného a především společného formátu. Tento formát se nikdy neujal a jeho specifikace ani nikdy nedospěla ke konečné verzi.

2.12 Shrnutí

Z této studie vyplývá, že je třeba zaměřit se především na formáty MOD, S3M, XM a IT. Jelikož samotné přehrávání a podpora různých formátů souborů není v žádném případě klíčovým bodem této diplomové práce, ostatním formátům se nebudu věnovat. Snad s výjimkou formátu MP3, který, ačkoliv se nejedná o klasický formát počítačové hudby, může být zdrojem reálných dat pro další zpracování. Podporu MP3 je však třeba chápat pouze jako doplňkovou.

3 Algebraický základ renderování hudby

3.1 Úvod

Existuje mnoho programů, které řeší přehrávání počítačové hudby z hudebních modulů. Tento velmi populární folklór má kořeny v roce 1987, kdy tehdy nový počítač Commodore Amiga jako první domácí počítač přinášel realistickou hudbu bez nutnosti investovat velké peníze do hi-end hardwaru. Později se objevila softwarová řešení, jejichž jediným cílem bylo simulovat hardware Amigy na jiných počítačích, hlavně PC AT, a donést tak kvalitní hudbu i tam. Dnes existují snad desetitisíce skladeb (tzv. modulů), v různých souborových formátech. Formátů existuje opravdu mnoho, každý další editor vždy přinesl nový formát s novými doplňky.

Během posledního desetiletí se renderovací algoritmy velmi zlepšily a dnešní digitální hudba je na úplně jiné úrovni, než byla tehdejší Amiga. Všechno se však točí kolem softwarové implementace renderování. Přestože jsem se opravdu snažil najít nějakou literaturu o teorii renderování, nenašel jsem vůbec nic. Jelikož se domnívám, že pouze prozkoumáním formálních vlastností daného problému lze najít opravdu dobré řešení, rozhodl jsem se sám formalizovat renderování hudby. A ukázalo se, že existuje velmi dobrý nástroj pro tento úkol. Algebra.

Vzhledem k tomu, že veškeré mé poznatky a závěry jsou podloženy pouze prakticky orientovanou literaturou, je možné, že některé mé domněnky a závěry nejsou úplně korektní. Avšak domnívám se, že jako absolutní průkopník mám na jisté omyly právo. Abych se omylů vyvaroval, provádím vždy pokud možno důkazy svých tvrzení.

Definice 3.1 (Hudba) *Hudba je určitý typ zvuku, který člověk chápe zvláštním způsobem.*

Poznámka: Tato definice pouze říká, že při hledání matematického základu se můžeme omezit na zvuk. Hudba je totiž také zvuk. Definice v podstatě říká, že množina všech hudeb je podmnožinou množiny všech zvuků.

Věta 3.1 *Zvuk je spojitá veličina.*

Důkaz: Platnost věty vyplývá z fyzikálních zákonitostí. Zvuk je jak známo mechanické vlnění a vlnění je vždy spojité.

Důsledek: Při realizaci softwarového renderování tedy narážíme na problém dostatečně přesné reprezentace spojitých veličin v diskrétním prostředí. Počítač je totiž zcela jistě výlučně diskrétní prostředí.

Věta 3.2 (Základní věta o složitosti zvuku) *Složitost obecného reálného zvuku je příliš vysoká na to, abychom jej mohli uspokojivě popsat algebraickými rovnicemi.*

Důkaz: Empirický. Zvuk je pseudonáhodná veličina, čili není možno ho algebraicky popsat nějakým jednoduchým vzorcem. Jisté konkrétní zvuky lze popsat poměrně dobře, ovšem nelze to zobecnit.

3.2 Normované funkce

Pojem funkce známe z matematické analýzy. Tato sekce definuje pojem normované funkce, díky kterému pak snadněji odvodíme další vztahy.

Úmluva: Pokud nebude výslovně řečeno jinak, v dalším textu budeme mluvit výhradně o reálných funkcích jedné reálné proměnné, které jsou zdola i shora omezené.

Definice 3.2 (Normovaná funkce) *Normovaná funkce prvního typu je funkce, jejíž obor hodnot je* $\langle -1; +1 \rangle$*. Budeme ji obecně značit NF1. Normovaná funkce druhého typu je funkce, jejíž obor hodnot je* $\langle 0; +1 \rangle$*. Budeme ji obecně značit NF2.*

Věta 3.3 *Každou (omezenou) spojitou funkci lze normovat tak, že ji vynásobíme kladným reálným číslem.*

Důkaz: Při vynásobení funkce kladným reálným číslem se změní její obor hodnot tak, že dolní i horní mez se vynásobí týmž kladným reálným číslem. Je zřejmé, že toto číslo lze zvolit tak, aby výsledné meze byly v souladu s definicí normované funkce. Zřejmě existuje mnoho takových čísel, kterými bychom mohli funkci normovat.

Poznámka: Z uvedeného algoritmu normování také vyplývá, proč jsme definovali dva typy normovaných funkcí. Funkce mající pouze nezáporné hodnoty se normuje na $NF2$, zatímco funkce mající kladné i záporné hodnoty se normuje na $NF1$.

Úmluva: Množinu všech $NF1$ budeme značit $MNF1$. Množinu všech $NF2$ budeme značit $MNF2$.

Věta 3.4 *MNF2 je podmnožinou MNF1.*

Důkaz: Každá $NF2$ je současně $NF1$. Z toho vyplývá platnost uvedené věty.

Definice 3.3 (Skládání) *Na MNF1 a MNF2 je definována n-ární operace skládání takto:*

$$+(f_1(x), \ldots, f_n(x)) = \frac{\sum_{i=1}^{n} f_i(x)}{n}$$

Věta 3.5 *MNF1 s operací skládání je komutativní grupoid.*

Důkaz: Z definice přímo vyplývá, že operace je uzavřená vzhledem k množině. Komutativita vyplývá z komutativity sčítání reálných čísel.

Věta 3.6 $(MNF2, +)$ *je komutativní podgrupoid* $(MNF1, +)$.

Důkaz: Nejprve dokážeme uzavřenost $MNF2$ vůči operaci skládání, tj. že výsledky skládání leží vždy v intervalu $NF2$. Vzhledem k tomu, že vzorec obsahuje pouze sčítání a dělení a vstupují do něj pouze nezáporná čísla, výsledkem jsou také vždy nezáporná čísla. Navíc podle věty 3.4 je $MNF2$ podmnožinou $MNF1$, čímž je uzavřenost vůči operaci skládání dokázána.

Komutativita již byla dokázána v důkazu věty 3.5. $(MNF2, +)$ je tedy komutativní grupoid a je to i podgrupoid $(MNF1, +)$.

Poznámka: Operaci skládání chápeme jako aritmetický průměr hodnot výchozích funkcí a takto ji také definujeme pro libovolný počet výchozích funkcí. Vzhledem k tomu, že skládání není asociativní operace, v praxi se vždy využívá skládání více než dvou funkcí.

Z uzavřenosti operace skládání na $MNF1$ a $MNF2$ také vyplývá důležitá vlastnost zvuku: Dílčí zvuky mohou v průběhu času libovolně vznikat a zanikat, aniž by ovlivnily platnost použitých vzorců. Každý zaniklý zvuk mohu také nahradit zvukem $f(x) = 0$.

Definice 3.4 (Modulace) *Na MNF1 a MNF2 je definována binární operace modulace takto:*

$$\cdot(f(x), g(x)) = f(x) \cdot g(x)$$

Věta 3.7 *MNF1 a MNF2 s operací modulace jsou komutativní pologrupy s neutrálním prvkem.*

Důkaz: Z definice přímo vyplývá komutativita i asociativita operace. Neutrálním prvkem je konstantní funkce $f(x) = 1$.

Věta 3.8 $(MNF2, \cdot)$ *je podpologrupou* $(MNF1, \cdot)$.

Důkaz: Plyne z definic $MNF1$ a $MNF2$ a předchozí věty.

Věta 3.9 $(MNF1, \cdot)$ *ani* $(MNF2, \cdot)$ *nejsou grupy.*

Důkaz: Nejsou to grupy, jelikož se zde nevyskytují inverzní prvky (kromě neutrálního prvku, který je inverzní sám sobě.)

Důsledek: Při mnohonásobné modulaci má funkce tendenci blížit se k neutrálnímu prvku $f(x) = 0$. Zatímco při stonásobném skládání funkce $f(x) = 0,9$ je výsledkem opět funkce $f(x) = 0,9$, při stonásobné modulaci funkce $f(x) = 0,9$ sebou sama dostaneme funkci $f(x) = 0,00002$.

3.3 Zvuk

Definice 3.5 (Zvuk) *Spojitou funkci času, která je NF1 a nikdy nenabývá hodnoty 1, nazýváme zvuk.*

Poznámka: Podmínka, že zvuk nenabývá hodnoty 1, není z matematického hlediska nezbytná, vychází pouze z praktických potřeb souvisejících s reprezentací zvuku v počítači. Podrobněji se této problematice věnuji v kapitole 4.

Definice 3.6 (Hlasitost) *Spojitou funkci času, která je NF2, nazýváme hlasitost.*

Definice 3.7 (Zvuk s hlasitostí) *Funkci vzniklou modulací zvuku hlasitostí nazýváme zvuk s hlasitostí.*

Poznámka: Slova zvuk a hlasitost jsou zde pojmy z předchozích definic.

Věta 3.10 *Zvuk s hlasitostí je zvuk.*

Důkaz: Musíme dokázat, že modulací zvuku hlasitostí vzniká opět zvuk. Hlasitost je $NF2$, čili její hodnoty jsou z intervalu $\langle 0; +1\rangle$. Zvuk je $NF1$, ovšem nikdy nenabývá hodnoty 1, jeho hodnoty jsou tedy z intervalu $\langle -1; +1)$. Když modulujeme zvuk hlasitostí, vzniká tak opět funkce jejíž funkční hodnoty jsou v intervalu $\langle -1; +1)$. Čili je to opět zvuk.

Definice 3.8 (Stereo funkce) *Za kompatibilní funkce považujeme takové, které mají stejný definiční obor i obor hodnot. Dvojici takových funkcí nazýváme stereo funkce a každou funkci z této dvojice nazýváme složkou stereo funkce.*

Poznámka: Tato definice je velmi důležitá, protože obecně definuje stereo zvuk a s tím související věci (stereo hlasitost apod.).

Definice 3.9 *Funkci $h(x)$ modulujeme stereo funkcí $(f(x), g(x))$ tak, že ji modulujeme každou z dvojice složek stereo funkce. Vznikne tak nová stereo funkce $(f(x) \cdot h(x), g(x) \cdot h(x))$.*

Poznámka: Modulace stereo funkcí je tedy totožná s provedením dvou modulací.

Definice 3.10 (Panning) *Spojitou stereo funkci času $(f(t), g(t))$, jejíž složky jsou NF2 a platí pro ně $f(t) = 1 - g(t)$, nazýváme panning.*

Definice 3.11 (Hlasitost s panningem) *Stereo funkci vzniklou modulací hlasitosti panningem nazýváme hlasitost s panningem.*

Věta 3.11 *Složky hlasitosti s panningem jsou hlasitosti.*

Důkaz: Hlasitost i složky panningu jsou $NF2$. Z definice modulace stereo funkcí vyplývá, že složky výsledné hlasitosti s panningem jsou tedy opět $NF2$, čili jsou to hlasitosti.

Poznámka: Podobně jako hlasitost může modulovat zvuk a „ztišovat“ ho tak, panning může modulovat hlasitost. Tentokrát ovšem vzniká dvojice hlasitostí, kterou chápeme jako stereo hlasitost, čili hlasitost levého a pravého kanálu.

Definice 3.12 (Modulace parametru) *Operaci, kdy transformujeme parametr funkce $f(x)$ vynásobením kladným reálným číslem p, nazýváme modulace parametru. Vzniká tak funkce $f(x \cdot p)$. Parametr p nazýváme modulační konstanta.*

Věta 3.12 *Modulací parametru zvuku vzniká zvuk.*

Důkaz: Transformací parametru funkce se vlastnosti jejího oboru hodnot nemění. Čili takto transformovaný zvuk je také zvukem.

Věta 3.13 *Skládáním zvuků vznikne zvuk.*

Důkaz: Platnost této věty plyne z definice zvuku a operace skládání na MNF1.

3.4 Fyzikální vlastnosti zvuku

V této sekci si ukážeme, jak výše uvedené definice a věty souvisejí s reálným zvukem. Přitom se snažíme chápat reálný zvuk jako jeden celek (jednu funkci), nikoli jako souzvuk více zvuků (jak ho slyší člověk). Člověk při poslechu zvuku má schopnost rozkládat ho na jednotlivé frekvence, a proto je schopen rozlišovat více zdrojů zvuku, např. více hudebních nástrojů hrajících společně.

Definice 3.13 *Zvuk je mechanické vlnění, které člověk vnímá sluchem.*

Poznámka: Tato velmi zjednodušená definice vystihuje přesně to, co nás zajímá. To ostatně ukazuje následující věta.

Věta 3.14 *Zvuk má frekvenci, kterou člověk vnímá jako výšku zvuku.*

Důkaz: Každé vlnění, tedy i zvuk, má nějakou frekvenci (hodnota frekvence je obrácenou hodnotou periody). Druhou část věty můžeme považovat za empiricky dokázanou.

Poznámka: Frekvence zvuku je spojitou reálnou funkcí času, ovšem my ji samozřejmě chápeme jako po částech konstantní funkci. To znamená, že hodnotu frekvence určujeme pouze v určitých ekvidistantních uzlech a mezi nimi neprovádíme žádnou interpolaci. Tento přístup je typický pro počítačovou hudbu a nemá žádný hmatatelný vliv na kvalitu hudby.

Jak ukáže následující věta, frekvence úzce souvisí s parametrem zvuku.

Věta 3.15 *Modulací parametru zvuku vzniká zvuk s odlišnou frekvencí. Pokud je modulační konstanta větší než jedna, vzniklý zvuk má vyšší frekvenci. Pokud modulační konstanta je menší než jedna, vzniklý zvuk má nižší frekvenci.*

Důkaz: Platnost věty plyne z faktu, že zvuk je vlnění, a z definice modulace parametru funkce.

Věta 3.16 *Zvuky ze dvou (nebo více) nezávislých zdrojů tvoří v jednom prostoru mechanické vlnění, jehož normovaná funkce je zvuk vzniklý skládáním výchozích zvuků.*

Důkaz: První část věty jistě platí, protože zvuk je prostě zvuk. Druhá část věty ukazuje, že ve skutečnosti tvoří více zdrojů zvuku zvuk, který „obsahuje“ všechny výchozí zvuky, čili jeho amplituda je vyšší než 1. Tedy až po normování takto vzniklé funkce dostaneme funkci zvuku v souladu s naší definicí.

Věta 3.17 *Skládáním zvuků po aplikaci operací modulace na množinu zvuků a jejich parametrů, frekvencí, hlasitostí a panningů lze vytvářet hudbu.*

Důkaz: Věta říká, že dáme-li všechny dosavadní poznatky dohromady, můžeme vytvářet počítačovou hudbu. Důkaz věty je triviální, jelikož hudba je zvuk a jistě existuje taková posloupnost modulací a skládání, jejíž výsledkem je (např. tentýž) zvuk. Např. vezměme za výchozí zvuk nějakou hudbu $h(t)$ (jelikož hudba je zvuk dle definice), hlasitost $f(t) = 1$, panning $g(t) = (1; 1)$ a frekvenci $p(t) = 1$. Potom modulační koeficient $p = p(t) = 1$, čili modulací parametru našeho zvuku vzniká opět původní zvuk. Modulací hlasitosti panningem vzniká stereo funkce $(1; 1)$ a

modulací výchozího zvuku hlasitostí vzniká dvojice zvuků $(h(t); h(t))$. Skládáním těchto dvou funkcí vzniká opět výchozí funkce zvuku $h(t)$, čili je to hudba.

Poznámka: Ačkoliv důkaz této věty je čistě formální, věta samotná není vůbec samoúčelná. Ukazuje, že realizace části zvané „syntetizér" v rendereru počítačové hudby závisí na tom, jakým způsobem jsou reprezentovány funkce zvuku, hlasitosti a panningu. Z jejich reprezentace v počítači také vyplývá způsob implementace operací modulace, skládání a modulace parametru.

3.5 Reprezentace spojitých normovaných funkcí

Přímá reprezentace spojitých normovaných funkcí v počítači není možná. Proto jsou používány přibližné metody numerické matematiky a funkce jsou reprezentovány diskrétními daty.

Zvuk je reprezentován PCM samply, což je řada čísel reprezentující funkční hodnoty zvuku v ekvidistantních uzlech. Vzdálenost mezi uzly je **interval** (hodnota času). Obrácená hodnota intervalu je vzorkovací frekvence, která je nutným atributem každého samplu. Samply jsou buď 8bitové nebo 16bitové, což je však jen záležitostí promítnutí intervalu $\langle -1; +1 \rangle$ do intervalu diskrétních čísel $\langle -128; +128)$ nebo $\langle -32768; +32768)$. Zde je také vidět, proč normovaný zvuk na počítači nikdy nenabývá hodnoty 1.

Hlasitost je obvykle reprezentována funkčním předpisem v podobě imperativního programu. Tyto programy jsou zapsány v patternech a provádí se paralelně pro každou stopu (viz kapitola 2). Výsledkem interpretace patternů je lineární seznam hodnot hlasitosti v ekvidistantních uzlech. Frekvence těchto hodnot je značně nižší než frekvence samplu, obvykle se pohybuje kolem $50Hz$.

Panning je pouze dvojice funkcí hlasitosti (ve smyslu definice), kterými se moduluje hlasitost (ve smyslu předchozího odstavce). Platí pro něj tedy všechno to, co bylo řečeno pro hlasitost.

Modulace se provádí pomocí násobení a normování. Speciální normování je třeba z důvodu reprezentace všech funkcí celými čísly. Práce s celými čísly je rychlejší, ale složitější a používá se pouze pro maximální zpětnou kompatibilitu s historickými systémy.

Transformace parametru vlastně znamená přepočítání numerické reprezentace zvuku na jiné (ekvidistantní) uzly. Aby nedošlo ke vzniku numerických chyb, projevujících se jako šum ve zvuku, provádí se interpolace zvuku. Interpolační metody jsou různé a je to nejpomalejší článek renderování hudby.

Skládání se provádí jako sčítání a normování. Vzhledem k tomu, že je třeba vždy provádět normování, vznikají velké numerické chyby a zvuk je velmi tichý. Proto se i pro 16bitový zvuk používá 32 až 64bitová reprezentace v počítači a na závěr se zvuk ještě násobí určenou konstantou (což má za následek jeho zesílení). Toto zesilování se nazývá **amplifikace** a je potřeba na závěr provést

ještě finální úpravu zvuku, protože při zvýšení hlasitosti mohou dílčí funkční hodnoty překročit meze dané možnostmi osmi nebo šestnáctibitové zvukové karty.

3.6 Bitová aritmetika

Na závěr této kapitoly chci ještě shrnout obecně známé vlastnosti tzv. bitové aritmetiky, čili problematiku „kolik bitů potřebujeme na reprezentaci určitých hodnot“.

Vycházíme z předpokladu, že na vstupu jakéhokoliv algoritmu jsou celá čísla. Ta mohou být buď neznaménková (unsigned) nebo znaménková (signed) v doplňkovém kódu. Cílem je definovat závislosti „počtu bitů“ potřebných pro uložení výsledku sčítání, odčítání a násobení v závislosti na počtu bitů výchozích hodnot.

Definice 3.14 *Neznaménkové číslo o n bitech obecně značíme nU. Znaménkové číslo o n bitech včetně znaménka obecně značíme nS. V obou případech platí $n \in N$.*

Poznámka: Nejčastěji se pracuje s číly $8S$, $16S$, $32S$ nebo $8U$, $16U$ a $32U$.

Věta 3.18 *Pro sčítání a násobení čísel platí:*

$$nU + nU = (n+1)U$$
$$mU \cdot nU = (m+n)U$$
$$mU \cdot nS = (m+n)S$$

Důkaz:

1. Označíme-li největší číslo typu nU jako x, potom největší číslo typu $nU+nU$ je logicky $2x$, což je pochopitelně číslo typu $(n+1)U$.

2. Nejjednodušší důkaz je možno provést s odkazem na klasické „násobení pod sebou“, které se učí už na základní škole. Z algoritmu pro násobení pod sebou ve dvojkové soustavě (kdy jeden bit odpovídá jedné cifře čísla) přímo plyne uvedená rovnost.

3. Důkaz je totožný s důkazem předchozího bodu.

Poznámka: S odkazem na větu 3.17 lze říci, že z hlediska bitové aritmetiky jsou uvedené tři rovnosti právě těmi, které jsou důležité při renderování hudby.

3.7 Shrnutí

Tato kapitola tedy přinesla dva klíčové prvky pro teoretické podchycení renderování počítačové hudby. Prvním z nich je samotný systém normovaných funkcí a na nich definovaných operací a druhým je návod, jak korektně dokázat, že všechny celočíselně reprezentované hodnoty v rendereru budou vždy v mezích své celočíselné reprezentace.

Bohužel jsem se nevyhnul jisté vágnosti v definicích i větách, avšak to považuji za důsledek toho, že byla řeč o pojmech na samotné hranici mezi exaktními vědami a lidským subjektivním chápáním zvuku. Proto nelze očekávat, že bude dosaženo požadovaného výsledku se stoprocentně abstraktní algebrou.

4 Syntetizér

4.1 Úvod

V této kapitole podrobně rozeberu syntetizér. Pokud je mým cílem eliminace šumu při renderování hudby, musím se určitě zaměřit především na syntetizér. Zatímco ostatní části programu slouží především k navození reálného prostředí, syntetizér provádí stěžejní práci se zvukem. Je to vlastně jediný článek procesu renderování hudby, kdy pracujeme přímo se zvukem a pouze zde tedy může vznikat šum.

Věta 4.1 (Základní věta o odstranění šumu) *Problém absolutního odstranění šumu ze zvuku je algoritmicky neřešitelný.*

Důkaz: Tento problém lze považovat za ekvivalentní s nalezením nejlepšího bezztrátového kompresního algoritmu. (Čili nejde o NP-úplný problém, je to zcela neřešitelné.)

Poznámka: Vzhledem k předchozí větě se tedy omezíme na hledání algoritmu, který šum dostatečně eliminuje, tj. sníží ho na tak nízkou úroveň[18], kdy už bude zanedbatelný.

Je samozřejmě otázkou, jak určit úroveň šumu a která úroveň je již zanedbatelná.

4.2 Určení úrovně zvuku a šumu

O reprezentaci zvuku již byla řeč v kapitole 3.5. Při této reprezentaci zvuku je jeho úroveň rovna amplitudě, z čehož plyne i způsob jejího určení.

Definice 4.1 *Úroveň zvuku $f(t)$ je funkce času $u(t)$, která je rovna jeho amplitudě, formálně píšeme $u(t) = |f(t)|$.*

Věta 4.2 *Úroveň zvuku je NF2.*

Důkaz: Plyne přímo z definice NF1 a NF2, jelikož absolutní hodnota intervalu $\langle -1; +1 \rangle$ je právě interval $\langle 0; +1 \rangle$.

Poznámka: Důležité je uvědomit si relativní povahu zvuku. Zvuk je mechanické vlnění, které je plně relativní v tom smyslu, že jsme schopni ho vnímat pouze v určitém prostředí, které chápeme jako **referenční**. Stejné mechanické vlnění vytváří v jiném prostředí zcela odlišný slyšitelný vjem.

Zatímco určení úrovně zvuku spočívá v našem značení jednoduchém spočítání absolutní hodnoty, k určení úrovně šumu je třeba mít k dispozici referenční zvuk

[18] neboli amplitudu

a ten nechat interferovat s jiným velmi podobným zvukem. Vznikne tak zvuk, jehož funkční hodnoty jsou rozdílem funkčních hodnot obou zvuků.

V praxi je tedy třeba provést vždy dva algoritmy a porovnat jejich výsledky. Pokud jsou oba algoritmy v zásadě správné, tak rozdíl jejich výsledků je právě šum, který vznikl nepřesností při výpočtech.

4.3 Algoritmus práce syntetizéru

Syntetizér má na vstupu numerickou reprezentaci zvuků nástrojů (samply) a data ze sekvenceru. Sekvencer poskytuje v reálném čase syntetizéru následující hodnoty:

- Číslo samplu, které odkazuje do veřejné tabulky samplů.
- Hlasitost, která je okamžitou hodnotou funkce hlasitosti (NF2).
- Panning, který je okamžitou hodnotou funkce panningu (NF2).
- Frekvenci, která je okamžitou hodnotou funkce transformace parametru.

Zatímco samply jsou z pohledu sekvenceru pouze „nějaká čísla“, ostatní funkce jsou přímo v trackeru a sekvenceru převáděny na jejich hodnoty. Je to nezbytné z toho důvodu, že zatímco samply jsou reprezentovány lineárním seznamem funkčních hodnot v ekvidistantních uzlech, ostatní funkce jsou popsány složitějšími vzorci na vyšší úrovni. Jejich reprezentace jsou jak úsporné na paměť, tak výhodné pro potřeby editace.

Dále se postupuje přesně podle poznatků z kapitoly o algebrách. Výstupní funkce zvuku je počítána numericky a je tedy opět určena funkčními hodnotami v ekvidistantních uzlech. Následující postup se opakuje pro každou hodnotu výstupní funkce, čili např. při CD kvalitě je to 44100 krát za sekundu, ovšem ještě násobeno počtem kanálů.

1. Provede se transformace parametru a spočítá se funkční hodnota takto vzniklého zvuku v daném časovém okamžiku.
2. Pokud má být výstup stereofonní, provede se modulace hlasitosti panningem a další kroky algoritmu se provedou dvakrát nezávisle na sobě - zvlášť pro levý a pravý kanál.
3. Spočítaná hodnota funkce zvuku je modulována hlasitostí.

Všechny kanály jsou kombinovány pomocí operace skládání zvuku. Výsledek je pak poslán pomocí dalších driverů do zvukové karty nebo může být uložen do souboru.

4.4 Algoritmus transformace parametru

Transformací parametru vzniká z funkce $f(t)$ funkce $f(x \cdot t)$. Jelikož x je libovolné kladné reálné číslo, vzniká problém s tím, že naše ekvidistantní uzly s intervalem $\frac{1}{samfreq}$ se změní na uzly s intervalem $\frac{1}{samfreq \cdot x}$ (kde $samfreq$ je vzorkovací frekvence samplu). Navíc požadujeme hodnotu této nové funkce v úplně jiných uzlech s intervalem $\frac{1}{outfreq}$ závislým na frekvenci výstupního zvuku $outfreq$.

Vzhledem k tomu, že v každém cyklu počítáme hodnoty mnoha uzlů v řadě za sebou, vytvoříme si **krokovací konstantu** *step* podle následujícího vzorce.

$$step = \frac{notefreq \cdot samfreq}{outfreq}$$

V tomto vzorci se pochopitelně objevila ještě nová hodnota $notefreq$, která značí frekvenci noty. Mezi frekvencí noty a samplu je úzká souvislost, jelikož frekvence samplu $samfreq$ je definována jako frekvence bázové noty[19]. Frekvence not jsou tedy převáděny podle speciální tabulky udávající pro každou notu koeficient její frekvence vzhledem k bázové notě.

Za těchto podmínek můžeme realizovat algoritmus velmi jednoduše pomocí obyčejné smyčky (zápis podle konvence C, *nodes* určuje počet uzlů výstupního bufferu):

```
inpos=outpos=0;
for(outpos=0; outpos<nodes; outpos++) {
  output[outpos]=input[inpos];
  inpos+=step;
}
```

Vzhledem k tomu, že sekvencer posílá nová data během našich výpočtů v syntetizéru, hodnoty *inpos* a *outpos* jsou uloženy v pomocných proměnných (*inpos* pro každý kanál, *outpos* globálně). Navíc je třeba v této smyčce zpracovat všechny kanály, takže původní algoritmus rozšíříme takto:

```
for(ch=0; ch<num_of_samples; ch++) {
  inpos=channel[ch].pos;
  outpos=global->outpos;
  for(i=0; i<nodes; i++,outpos++) {
    output[outpos]+=input[inpos];
    inpos+=step;
  }
  channel[ch].pos=inpos;
}
```

[19] Bázová nota je dle konvence C-5, čili C v oktávě č.5, což je právě střední oktáva.

Na začátku je samozřejmě zapořebí vynulovat pole *output*[] a při každé nové notě nulovat *channel.pos*. Vzhledem k tomu, že samply mohou mít definovány loopy[20], uvedený algoritmus je nutno ještě velmi rozsáhle upravovat, než dojdeme k opravdu reálně použitelnému řešení. Hotový algoritmus je možno vidět v souboru **Synthesizer.cpp**, metoda **Synthesizer::mix1ch()**.

4.5 Algoritmus interpolace samplu

Předchozí ukázka kódu má jednu velkou slabinu: nelze jednoduše spočítat *input*[*inpos*], protože *inpos* je desetinné číslo, ale my máme pouze numerickou podobu funkce *input*. Zde tedy přicházejí na řadu interpolační metody reálné funkce. Tato problematika je sama o sobě tak rozsáhlá a důležitá, že jsem jí věnoval samostatnou (následující) kapitolu.

4.6 Monofonní a stereofonní výstup

Dosud studované algoritmy pracovaly pouze s monofonním zvukem. Renderování stereofonního zvuku je založeno na principu modulace hlasitosti panningem. Vzniká tak stereo funkce hlasitosti, která se musí zpracovávat po složkách. Neexistuje žádná obecná metoda, jak tento proces zjednodušit. Ačkoliv zpracování stereo funkce znamená dělat veškeré výpočty syntetizéru dvakrát a, i když se jedná pokaždé pouze o jinou hlasitost téhož zvuku, kvůli nutnosti skládat více kanálů dohromady je nutno zpracovávat levý a pravý kanál prakticky zcela odděleně.

Jelikož hodnoty *step* i *inpos* jsou u obou složek stereo funkce totožné, algoritmus renderuje oba výstupní kanály současně. Nejvnitřnější smyčka potom vypadá takto:

```
for(i=0;i<nodes;i++) {
  output[outpos+0]+=input[inpos]*volume[0];
  output[outpos+1]+=input[inpos]*volume[1];
  inpos+=step;
  outpos+=2;
}
```

Tento algoritmus, doplněný o dříve nastíněné řešení transformace parametru, loopů, interpolaci a především o vhodnou celočíselnou reprezentaci všech reálných čísel, je již konečným algoritmem, použitým v mém syntetizéru. Otázka

[20]Loop je úsek samplu, který se neustále opakuje. Loop je definován jako zprava otevřený interval na funkci zvuku, tj. pro interval $\langle begin; end)$ platí $f(t) = f(begin + (t - begin)\ /\ (end - begin))$. Například máme-li loop definován jako interval $\langle 100; 200)$, potom $f(202) = f(102)$ apod. Navíc existují tzv. ping-pong loopy, které jsou dokonce obousměrné, takže jejich implementace už vůbec není jednoduchá.

reprezentace čísel v počítači je velmi důležitá a budu se jí věnovat v následujícím textu.

4.7 Celočíselná reprezentace čísel v počítači

Podle teorie z kapitoly 3 již víme, že všechny hodnoty pro syntetizér, tj. sampl, hlasitost a panning, jsou NF1 nebo NF2. Ve skutečnosti ovšem narážíme na problém vzniklý nutností dodržovat kompatibilitu s historickými specifikacemi, které pracují výhradně na celočíselné bázi.

4.7.1 Samply

Hodnoty samplů jsou závislé na tom, jakým způsobem byly zaznamenány do počítače. Jelikož se pro záznam používají výhradně osmi a šestnáctibitové AD převodníky (dále jen ADC = analog-digital converter), máme k dispozici 8 a 16bitové znaménkové hodnoty. Pro jednoduchost se můžeme omezit na studium pouze 16bitových samplů, protože 8bitové samply jsou v zásadě identické, až na nepřítomnost osmi dolních bitů.

Sampl vzniká hardwarově, tj. převodem zvuku do numerické podoby pomocí zmíněného ADC. Výsledkem je hodnota v intervalu $\langle -32768; +32767 \rangle$. Tento interval má typický problém, tzn. buď ho chápeme jako zprava otevřený $\langle -32768; +32768)$, anebo mu chybí nejvyšší kladná hodnota. Tento problém je způsoben tím, že číslo je uloženo v doplňkovém kódu, který má vždy o jednu kladnou hodnotu méně.

Uvedený problém sám o sobě není nijak podstatný. Sampl totiž pochází z ADC a ten pracuje na principu ořezání reálného zvuku z jeho reálného neomezeného intervalu $(-\infty; +\infty)$ do $\langle -1; +1 \rangle$. Chyba ADC je přitom daleko větší než chyba vzniklá zanedbáním celého problému použití doplňkového kódu.

Bitový paradox

Dokonce i algoritmus konverze 8bitového samplu na 16bitový, kdy dolních 8bitů vyplňujeme nulami a vzniká nám tak interval $\langle -32768; +32511 \rangle$, který je zprava již dosti zkrácený, je korektní. Viděl jsem i pokusy posunovat při této konverzi hodnoty ve všech uzlech o 128 jednotek výše. To je z pohledu čistě matematického správný přístup, ovšem vzhledem k popsanému způsobu práce ADC bychom si podobné úpravy měli rozhodně odpustit.

Zmíněný paradox je pouze důsledkem výskytu numerické chyby. U numerických chyb se zastavme podrobněji.

4.7.2 Numerická chyba v samplu

Věta 4.3 (O numerické chybě v samplu) *Při reprezentaci samplu pomocí určitého počtu bitů je nejmenší hodnotou, která je větší než všechny chyby právě hodnota nejnižšího platného bitu.*

Poznámka: Stále kalkulujeme s tím, že hardwarové ADC čipy místo zaokrouhlování používají ořezávání, čili všechny chyby jsou podle této věty ve zprava otevřeném intervalu intervalu $\langle 0; max)$.

Důkaz: Označme b_x hodnotu bitu x, kde $x \in \{0, 1, 2, \ldots\}$. Potom $b_x = 2^{-1-x}$. Hodnoty bitů pochopitelně tvoří klasickou geometrickou posloupnost, kde $q = \frac{1}{2}$. Tato posloupnost tedy konverguje k nule, tj. $\lim_{x\to\infty} q^x = q^\infty = 0$ a její součet $s = b_0 \cdot \frac{1-q^\infty}{1-q} = b_0 \cdot \frac{1}{\frac{1}{2}} = 2 \cdot b_0$. Součet takové řady je tedy roven dvojnásobku hodnoty nejvyššího bitu. Hodnota nejvyššího bitu odřezané části čísla je pochopitelně rovna polovině hodnoty nejnižšího platného bitu, čili její dvojnásobek je roven právě hodnotě nejnižšího bitu.

Poznámka: Mohli bychom se spokojit také s jednodušším důkazem. Nejprve předpokládejme, že věta neplatí, tj. chyba je větší (nebo rovna) hodnotě nejnižšího bitu. Pak nám ovšem nic nebrání v tom, abychom hodnotu samplu podle potřeby zvýšili nebo snížili tak, aby chyba byla minimální, tedy menší než hodnota nejnižšího bitu.

4.7.3 Hlasitost a panning

Hodnoty hlasitosti a panningu přebírá syntetizér v celočíselné podobě ze sekvenceru. Panningem se pouze moduluje hlasitost podle definice, s výslednou stereo funkcí hlasitosti se potom moduluje zvuk.

Věta 4.4 *Za předpokladu, že zvukové karty jsou 16bitové a mixujeme celkem* 2^n *kanálů, stačí nám hlasitost s přesností maximálně* $16 + n$ *bitů k tomu, aby se ořezání neprojevilo do výsledného zvuku.*

Důkaz: Spočítáme maximální hodnotu numerické chyby. Chyba v 16bitové hodnotě hlasitosti je dle věty 4.3 menší než hodnota 16.bitu. Pokud by jeden kanál měl chybu rovnou 1, tak 2^n kanálů má chybu 2^n, což je přesně n bitů. Dohromady je tedy chyba 2^n kanálů menší než hodnota $(16+n)$tého bitu. To je také tvrzením dokazované věty.

4.8 Shrnutí

Za pomocí poznatků z předchozí kapitoly jsem zde nastínil několik konkrétních způsobů implementace syntetizéru zvuku, přičemž důraz byl kladen zejména na kvalitu zvuku, tj. minimalizaci výskytu numerických chyb.

Klíčovým prvkem v této otázce je samozřejmě interpolace numericky reprezentované funkce zvuku. A právě tomu je věnována další kapitola.

5 Interpolace zvuku

Jak již bylo uvedeno v předchozí kapitole, klíčovým prvkem při práci se zvukem je syntetizér a jeho kvalita po zvukové stránce závisí právě na použité interpolační metodě. V této kapitole bude tedy řeč o možných metodách interpolace funkce (ve smyslu dřívější definice zvuku, tj. reálné spojité funkce jedné reálné proměnné).

Zvuk není jen *nějaká funkce*, ale je to velmi specifická veličina, proto jsem se snažil pečlivě vybrat ty nejvhodnější interpolační algoritmy. Přitom jsem samozřejmě sáhl jak ke klasickým numerickým metodám (viz [1]), tak k prostředkům analytické geometrie, která se zabývá spojitostí a interpolací po částech (viz [2]).

5.1 Reprezentace pozice v samplu

Předpokládejme, že máme zvuk reprezentován samplem, tedy seznamem funkčních hodnot v ekvidistantních uzlech (fyzicky je to obyčejné pole). Pokud si zvuk označíme jako $f(t)$, pak známe hodnoty pro $t \in N_0$.

Dále je třeba vycházet z reálné situace, tj. musíme si uvědomit, že číslo na počítači je tvořeno celou a desetinnou částí. Jinými slovy čas t je ve skutečnosti reprezentován dvojicí čísel, kde jedno reprezentuje celou a druhé desetinnou část čísla t. Použití obvyklých čísel v plovoucí řádové čárce není vhodné, protože tato čísla mají vyšší přesnost pro menší hodnoty a naopak nižší přesnost pro vyšší hodnoty. To samozřejmě nechceme. (Je to zcela kritické omezení, které nesmíme připustit!) Pro všechny další výpočty budeme tedy reprezentaci času a samplu chápat takto:

$$\begin{aligned} t &= i + frac \\ f(t) &= input[i + frac] \end{aligned}$$

Toto odpovídá skutečné reprezentaci v počítači, kde *input* je pole hodnot samplu a $i + frac$ je index do tohoto pole. Pokud $frac = 0$, tak je vše v pořádku, v opačném případě to, laicky řečeno, takto jednoduše nefunguje. A právě studiu situací, kdy $frac \neq 0$ je věnována celá tato kapitola.

Poznámka: Číslo $frac$ je samozřejmě reprezentováno celočíselně, obvykle jako 16bitová hodnota ve tvaru $65536 * frac$, tj. hodnota ze zprava otevřeného intervalu $\langle 0, 1)$ je pomocí okénkové transformace rovnoměrně zobrazena do celočíselného intervalu $\langle 0, 65536)$.

5.2 Poznámky k interpolačním metodám

Než přejdeme k detailnímu popisu jednotlivých interpolačních metod, je třeba ještě zmínit několik formálních vlastností a předpokladů, přičemž vycházíme především z toho, že všechny metody jsou založeny na bázi polynomů.

1. Základní vlastností polynomů je spojitost. Zvuk, který vypočítáme jako polynom určitého stupně, je tedy vždy spojitý na celém definičním oboru.

2. Vždy hledáme takovou interpolaci, kde výsledná křivka **nutně** prochází všemi zadanými body. (Tento požadavek vychází z definice problému, jelikož na vstupu máme hodnoty z AD převodníku.)

3. Interpolaci provádíme po částech, tj. pro určitou skupinu sousedících uzlů vypočteme samostatný polynom. Jinak to ani není možné, protože každý sampl má mnoho tisíc uzlů a polynomy tak vysokých stupňů nelze v rozumném čase vypočítat.

4. Snažíme se používat polynomy co nejnižších stupňů, které poskytnou uspokojivé výsledky. Je to totiž výpočetně jednodušší a polynomy vyšších stupňů mají tendenci se hodně *kroutit*, tj. křivka je už pro polynomy stupně 5 a výše velmi nepřirozená.

5. Jelikož interpolaci provádíme po částech, musíme si dávat pozor na spojitost křivky v uzlech, tedy v bodech, kde přecházíme z jednoho úseku do druhého.

Úmluva: Pro zpřehlednění zápisu vzorců budeme psát y_i místo $input[i]$. Tedy y_i je hodnota na indexu i v tabulce samplu. Přitom pochopitelně musí platit, že $i \in N_0$.

5.3 Jednoduchá bodová interpolace

Jednoduchá bodová interpolace je v praxi nejčastěji používanou metodou, protože je velmi rychlá. Spočívá v tom, že vezmeme pouze celou část pozice, tedy rovnou přečteme hodnotu z tabulky:

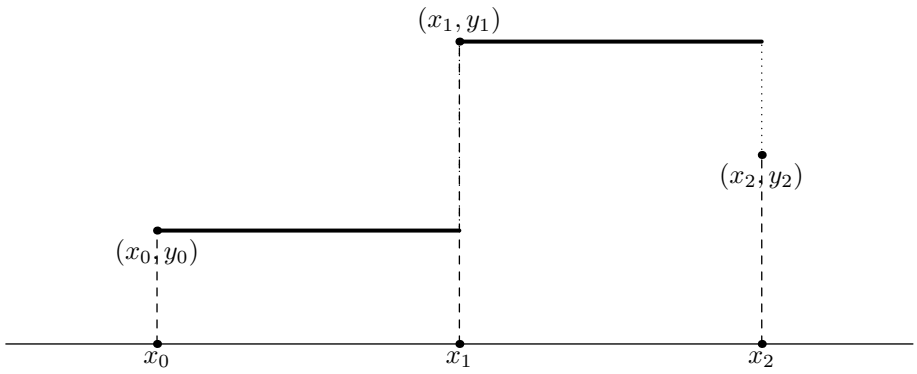
$$f(i + frac) = y_i$$

Problém takové interpolace je v tom, že křivka je v uzlech naprosto nespojitá. Čili jde o interpolaci jen v uvozovkách a výsledek by se tedy raději neměl nazývat křivkou.

5.4 Pravá bodová interpolace

5.4.1 Popis metody

Pravá bodová interpolace je velmi podobná předchozí metodě, tentokrát se však pozice v samplu místo ořezávání zaokrouhluje. Zaokrouhlování na počítači je ekvivalentní se zápisem $(int)(i + 0.5)$, takže je zde o jednu operaci reálného sčítání



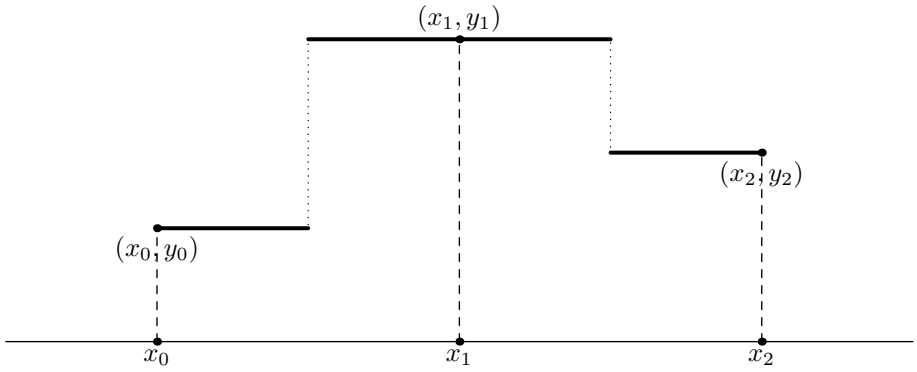
Obrázek 3: Jednoduchá bodová interpolace

navíc. Vzhledem k tomu, že se tento proces provádí mnohomilionkrát za sekundu, jedna taková operace navíc se projeví docela citelně. Zatímco rychlost jako taková pro nás není příliš podstatná, otázku rozdílu této metody oproti jednoduché bodové interpolaci můžeme zkoumat experimentálně.

$$f(i + frac - 0.5) = y_i$$

Při zavedené celočíselné reprezentaci čísla $frac$ můžeme zaokrouhlení provést pomocí bitového posunu a sčítání (stále předpokládáme, že frac je 16bitové číslo.):

$$f(i + frac) = y_{i+(frac>>15)}$$



Obrázek 4: Pravá bodová interpolace

5.4.2 Zhodnocení metody

Vzhledem ke způsobu práce AD převodníku je skutečně nejasné, která ze dvou možných bodových interpolací je lepší. Přestože z čistě matematického hlediska je lepší pravá bodová interpolace, v praxi jsou obě metody zcela rovnocenné. Jediný

praktický rozdíl mezi nimi je ten, že jednoduchá bodová interpolace způsobuje zpoždění zvuku o $\frac{samfreq}{2}$, což je naštěstí daleko pod hranicí vnímání (v řádech desítek mikrosekund).

I u této metody se pochopitelně vyskytuje již zmíněný problém, že vypočtená křivka je v uzlech naprosto nespojitá a výsledek by se opět raději ani neměl nazývat křivkou.

5.5 Lineární interpolace

5.5.1 Popis metody

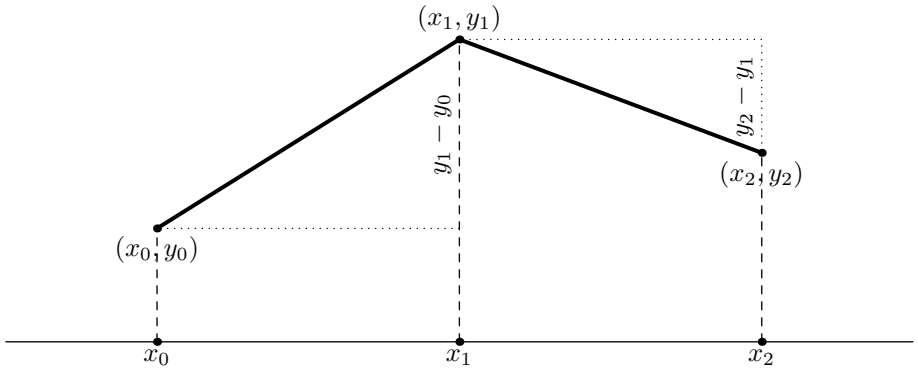
Lineární interpolace nahrazuje neznámý úsek funkce mezi dvěma uzly úsečkou. Hodnotu $f(t)$ pak spočítáme jednoduchým vzorcem:

$$f(i + frac) = y_i + (y_{i+1} - y_i) \cdot frac$$

Tento vzorec se dá přepsat do doplňkového tvaru, kde se každá z hodnot $input[]$ vyskytuje právě jednou:

$$f(i + frac) = (1 - frac) \cdot y_i + frac \cdot y_{i+1}$$

Poznámka: Tento druhý vzorec také odpovídá Bézierově křivce 1.stupně proložené danými dvěma body. Oba vzorce jsou ekvivalentní a určují lineární polynom obecného tvaru $y = k \cdot x + q$, kde $x = frac$, $k = y_{i+1} - y_i$, $q = y_i$.



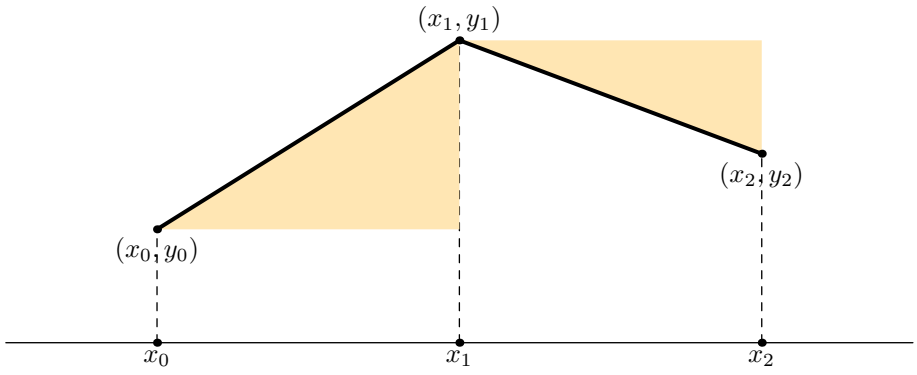
Obrázek 5: Lineární interpolace

5.5.2 Zhodnocení metody

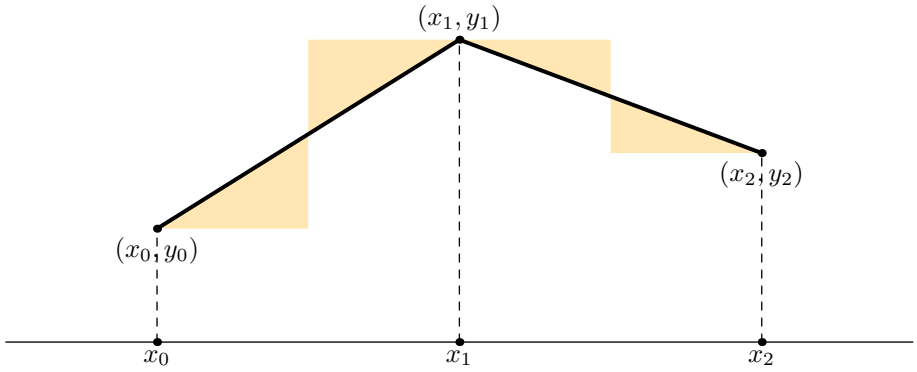
Jak je vidět, lineární interpolace je výpočetně daleko složitější než bodová a navíc vždy potřebujeme znát hodnoty funkce ve dvou sousedních uzlech, což není vždy

triviální problém[21]. Na druhou stranu je třeba si uvědomit, že zvuk je spojitá veličina, takže použití lineární interpolace nám dává nesrovnatelně lepší výsledky než obyčejná bodová interpolace.

Lineární interpolace snižuje nežádoucí šum ve zvuku, jelikož na rozdíl od bodové interpolace zde nevznikají skokové přechody v uzlech. Většina současných programů, stejně jako drtivá většina moderních hardwarových syntetizérů na zvukových kartách, používá právě lineární interpolaci. Zejména její hardwarové řešení je poměrně snadné. Srovnání bodové a lineární interpolace je vidět na obrázcích 6 a 7.



Obrázek 6: Srovnání lineární a jednoduché bodové interpolace



Obrázek 7: Srovnání lineární a pravé bodové interpolace

5.6 Kvadratická interpolace

5.6.1 Popis metody

Po dobrých zkušenostech s lineární interpolací se prakticky sama nabízí kvadratická interpolace. Zatímco rozdíl mezi lineární a bodovou interpolací je tak mar-

[21] Hlavním problémem jsou loopy, neboť je třeba testovat jejich hranice. Nevhodně navržený algoritmus tohoto testování může výrazně zpomalit celý program.

kantní, že je velmi snadno slyšitelný, zde by se mohlo na první pohled zdát, že rozdíl mezi bodovou a kvadratickou interpolací pochopitelně už tak znatelný být nemusí. Praktická zkouška však ukáže, že opak je pravdou.

Problémem při kvadratické interpolaci je zachování spojitosti. Zatímco u lineární interpolace je spojitost zajištěna, zde už to bude horší. Do algoritmu totiž vstupuje lichý počet uzlů (tři), což je poněkud nešťastné. Klasický algoritmus kvadratické interpolace bere za prostřední ten ze tří uzlů, který je nejblíže k bodu, jehož funkční hodnota nás zajímá.

5.6.2 Odvození vzorce

Mějme dány tři body $(-1, y_{i-1})$, $(0, y_i)$, $(+1, y_{i+1})$. Nyní hledáme koeficienty kvadratické rovnice

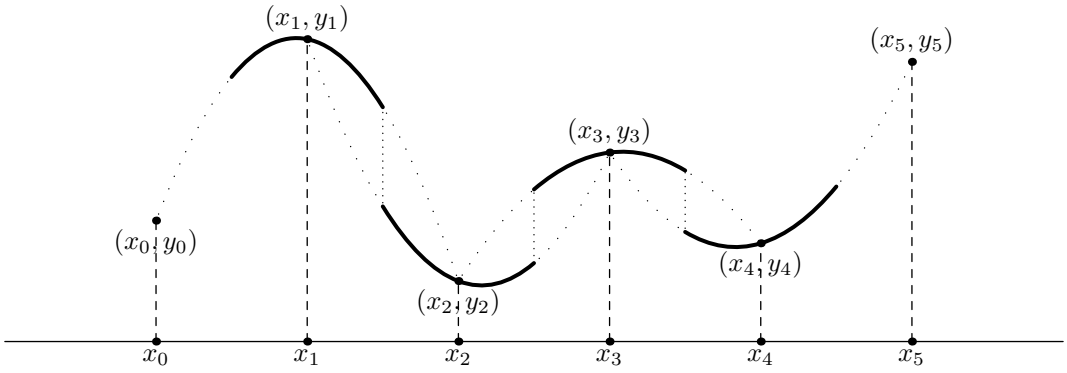
$$y = a_0 + a_1 \cdot x + a_2 \cdot x^2$$

Dosazením souřadnic bodů do této rovnice dostaneme soustavu tří rovnic o třech neznámých:

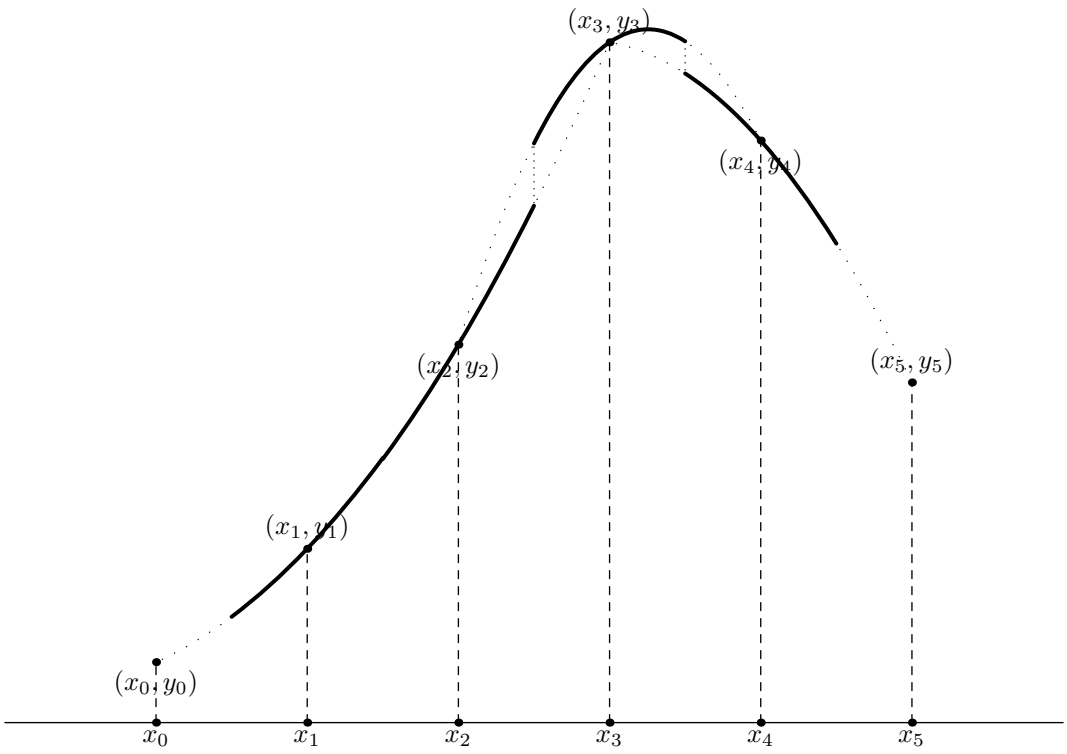
$$\begin{aligned} y_{i-1} &= a_0 - a_1 + a_2 \\ y_i &= a_0 \\ y_{i+1} &= a_0 + a_1 + a_2 \end{aligned}$$

Vyřešením této soustavy dojdeme k hodnotám

$$\begin{aligned} a_0 &= y_i \\ a_1 &= \frac{1}{2} y_{i+1} - \frac{1}{2} y_{i-1} \\ a_2 &= \frac{1}{2} y_{i+1} - y_i + \frac{1}{2} y_{i-1} \end{aligned}$$



Obrázek 8: Kvadratická interpolace



Obrázek 9: Kvadratická interpolace

5.6.3 Zhodnocení metody

Teorie vypadá hezky, ovšem po zhodnocení všech zjištěných poznatků jsem došel k závěru, že interpolace kvadratickými polynomy není možná. Bylo by sice možné sestavit příslušné polynomy, ovšem výsledek by byl kritický. Obecně tedy nemohu doporučit použití žádného polynomu sudého stupně a nikdy jsem ani neviděl, že by se o to někdo v praxi pokoušel.

5.6.4 Převod kvadratického polynomu na Bézierovu kubiku

Při snaze nakreslit grafickou podobu kvadratické interpolace narážíme na zásadní problém: Grafické zařízení podporuje obvykle pouze úsečky a Bézierovy kubiky. Převod na úsečky nás samozřejmě nezajímá, takže se pokusím odvodit vzorec na výpočet řídících bodů Bézierovy křivky, která prochází danými třemi body a je totožná s interpolačním polynomem druhého stupně, jenž prochází stejnými body.

Takto získané vrcholy řídícího polynomu lze pak využít při kreslení grafu pomocí zařízení, která podporují klasické Bézierovy křivky. Mezi taková zařízení patří např. Windows GDI, PostScript nebo MetaPost, který jsem také použil pro vytvoření všech obrázků k tomuto textu.

Jsou dány body $A = (x_{i-1}, y_{i-1})$, $P = (x_i, y_i)$, $C = (x_{i+1}, y_{i+1})$. Hledáme

řídící body Bézierovy kubiky, jejíž graf je roven grafu polynomu druhého stupně procházejícímu danými body. Je zřejmé, že tato úloha je řešitelná a má vždy právě jedno řešení.

1. Vypočteme řídící body Bézierovy kvadriky, jejíž graf je shodný s grafem hledaného polynomu. Na obrázku to jsou body A, B, C. Body A a C jsou krajní výchozí body, takže stačí dopočítat bod B. Rozepíšeme si rovnici Bézierovy kvadriky

$$f(u) = (1-u)^2 A + 2u(1-u)B + u^2 C, u \in \langle 0, 1 \rangle$$

   Jelikož pracujeme s grafem reálné funkce jedné reálné proměnné, takže platí $f_x(u) = u$, víme že $f(0.5) = P$. Do rovnice Bézierovy kvadriky tedy dosadíme $u = 0.5$ a upravíme ji do vhodného tvaru.

$$\begin{aligned} P &= \frac{1}{4}A + \frac{1}{2}B + \frac{1}{4}C \\ B &= 2P - \frac{1}{2}A - \frac{1}{2}C \end{aligned}$$

2. Algoritmem pro zvýšení stupně Bézierovy křivky vypočteme body A' a C'. Tím dostaneme řídící body Bézierovy kubiky ve tvaru A, A', C', C.

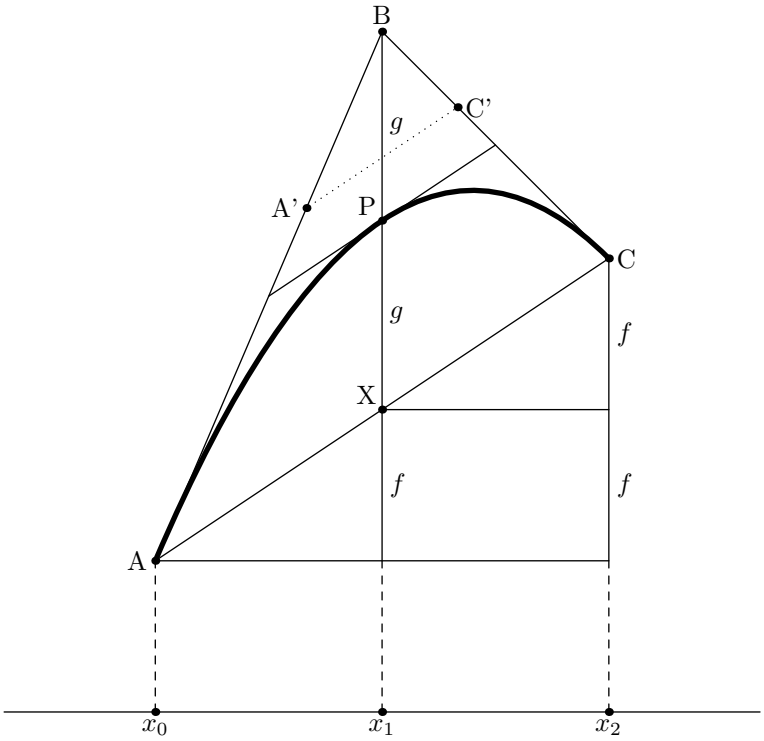
$$\begin{aligned} A' &= \frac{A+2B}{3} = \frac{4P-C}{3} \\ C' &= \frac{C+2B}{3} = \frac{4P-A}{3} \end{aligned}$$

Obrázek 10 ukazuje geometrickou konstrukci. Hledáme body A' a C' takové, aby body A, A', C', C byly řídícími body Bézierovy kubiky. Odvození lze provést pomocí de Casteljau algoritmu. Bod P leží přesně uprostřed spojnice středů úseček AB a BC. Z toho ovšem přímo plyne, že bod P je současně přesně uprostřed úsečky XB, kde X je středem úsečky AC.

Také druhý krok, tedy zvýšení stupně Bézierovy křivky lze provést graficky. Úsečky AB a BC rozdělíme na 3 díly a body A' a C' leží vždy jeden díl od bodu B. Důkaz tohoto geometrického algoritmu lze provést již zmíněným algoritmem de Casteljau, kterým potvrdíme, že pro $u = 0.5$ je bodem Bézierovy křivky určené řídícími body A, A', C', C právě bod P. A to bylo naším cílem.

Poznámka: Jednoznačnost zobrazení $x \to f(x)$, čili neexistence dvou funkčních hodnot pro stejné x, je zaručena vztahem $x(u) = u$.

Odvození tohoto algoritmu si velmi cením, protože jen díky němu se mi podařilo nakreslit obrázky 8, 9, a 10.



Obrázek 10: Výpočet řídících bodů Bézierovy kubiky pro zobrazení kvadratické interpolace

5.7 Kubická interpolace

5.7.1 Popis metody

Kubická interpolace je pouze dalším zvýšením stupně polynomu, tj. do algoritmu už vstupují čtyři uzly. Díky tomu, že počet uzlů je sudý, je počet intervalů lichý, což je výhodné. Přitom je velmi vhodné zahrnout do výpočtu místo čtyř bodů pouze dva body a jejich první derivace. Takto dostaneme rovnici pro jediný úsek křivky v intervalu mezi dvěma vstupními body.

5.7.2 Odvození vzorce

Mějme dány dva body $(0, y_i)$ a $(1, y_{i+1})$ a jejich derivace $(0, y'_i)$ a $(1, y'_{i+1})$. Nyní hledáme koeficienty kubické rovnice

$$y = a_0 + a_1 \cdot x + a_2 \cdot x^2 + a_3 \cdot x^3$$

Pro sestavení hledané soustavy 4 rovnic budeme ještě potřebovat derivaci uvedené kubické rovnice

$$y = a_1 + 2a_2 \cdot x + 3a_3 \cdot x^2$$

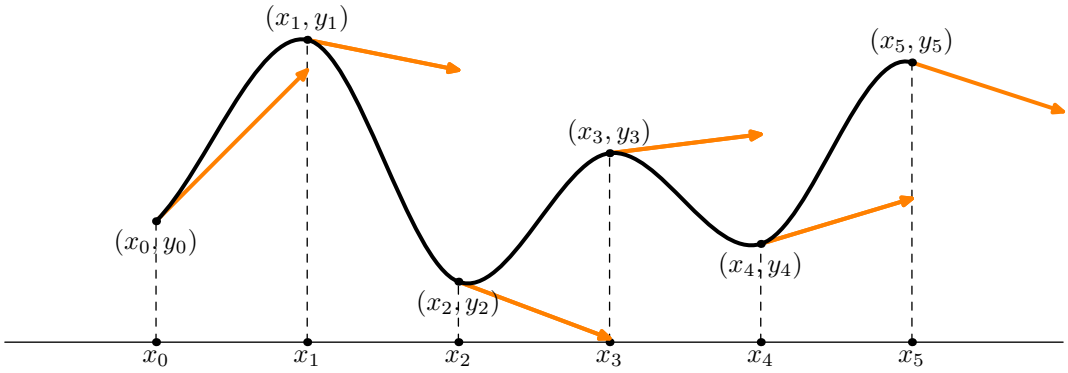
Dosazením souřadnic bodů do těchto rovnic dostaneme následující soustavu čtyř rovnic o čtyřech neznámých:

$$\begin{aligned} y_i &= a_0 \\ y_{i+1} &= a_0 + a_1 + a_2 + a_3 \\ y'_i &= a_1 \\ y'_{i+1} &= a_1 + 2a_2 + 3a_3 \end{aligned}$$

Vyřešením této soustavy získáme hledané koeficienty naší kubické rovnice.

$$\begin{aligned} a_0 &= y_i \\ a_1 &= y'_i \\ a_2 &= -3y_i + 3y_{i+1} - 2y'_i - y'_{i+1} \\ a_3 &= 2y_i - 2y_{i+1} + y'_i + y'_{i+1} \end{aligned}$$

Tyto hodnoty již můžeme použít v praktických výpočtech.



Obrázek 11: Kubická interpolace ($y_{-1} = y_6 = 0$).

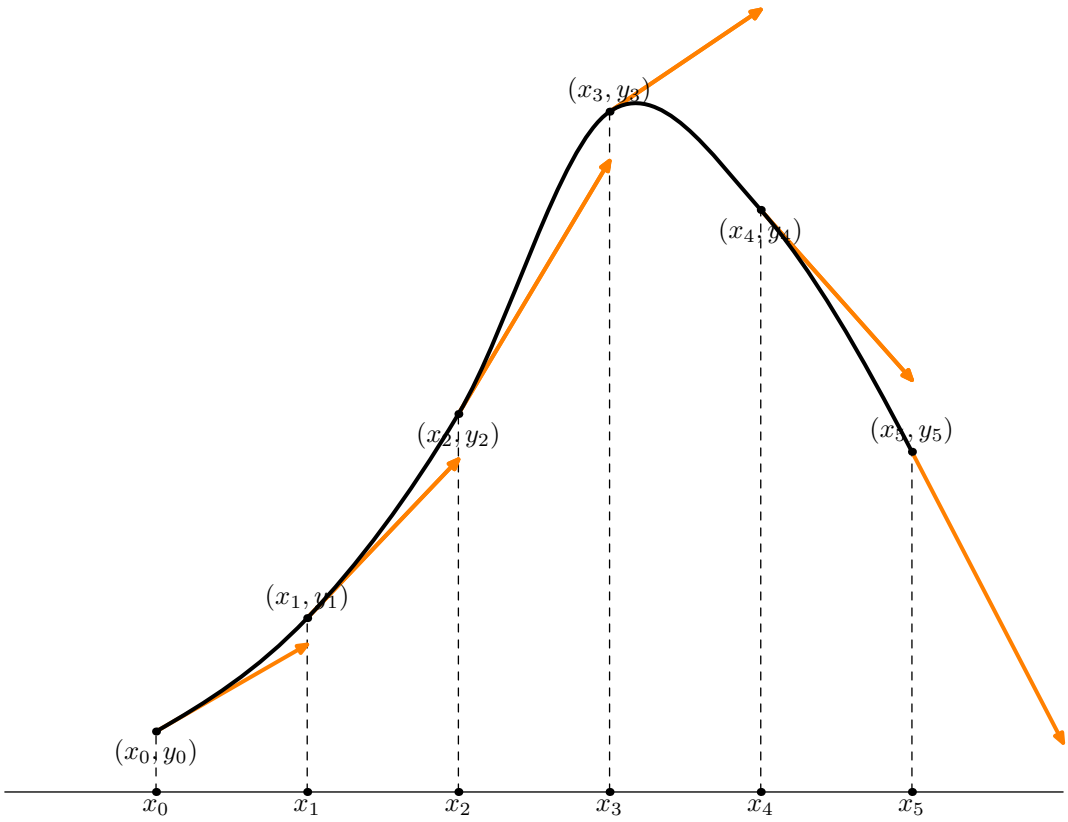
5.7.3 Výpočet derivací

Uvedená metoda naráží na jeden závažný problém: neznáme derivace samplu. V praxi se tento problém řeší dosazením vhodných hodnot. Zde se nabízí výpočet $f'(i)$ tak, že tímto bodem a dvěma okolními body proložíme parabolu (kvadriku) a derivaci položíme rovnu derivaci paraboly v bodě i.

Postup pro výpočet je pochopitelně identický algoritmu uvedenému u kvadratické interpolace. Čili do vzorce pro derivaci kvadriky $y' = a_1 + a_2 \cdot x$ dosadíme $x = 0$ a dostaneme velmi jednoduchou rovnici

$$y' = \frac{y_{i+1} - y_{i-1}}{2}$$

Geometricky tedy jde o polovinu vzdálenosti dvou okolních bodů. Tento algoritmus pro výpočet derivace jsem odvodil z algoritmu Bessel používaném při



Obrázek 12: Kubická interpolace ($y_{-1} = y_6 = 0$).

C^1 kubické interpolaci v analytické geometrii (viz [2]). Algoritmus lze aplikovat i pro proměnnou x. Na obrázcích 11 a 12 je vidět, že $x' = 1$ ve všech uzlech.

5.7.4 Zhodnocení metody

Uvedená metoda na bázi znalosti derivací se jeví jako velmi dobrá, přímo vynikající. Bohužel, kvůli své výpočetní náročnosti není příliš praktická, bývá proto nasazována pouze v systémech pro renderování hudby na špičkové úrovni, ne v reálném čase. Největším plus této metody je zejména dosažení globální C^1 spojitosti, což žádná jiná metoda neposkytuje. Právě proto je také tato metoda dle mého názoru nejlepší. Je to totiž křivka složená z polynomů nejnižšího možného stupně s globální C^1 spojitostí. A to je přesně to, co jsem se snažil najít.

5.7.5 Převod kubického polynomu na Bézierovu kubiku

Stejně jako u kvadratické interpolace (viz kap. 5.6.4), i kubické interpolační polynomy bychom rádi zobrazovali pomocí kubických Bézierových křivek, které nám poskytují výstupní grafická zařízení. Jelikož se jedná o polynom vyššího stupně, odvození hledaného vzorce již bude poněkud obtížnější.

Postup odvození vzorce bude obdobný jako u kvadrik. Mějme dány body $A = (x_i, y_i)$ a $D = (x_{i+1}, y_{i+1})$ a jejich derivace $A' = (x_i, y'_i)$ a $D' = (x_{i+1}, y'_{i+1})$. Hledáme takové řídící body Bézierovy kubiky, aby tato kubika byla identická s polynomem určeným čtveřicí bodů A, A', D, D'. Tato úloha má pochopitelně právě jedno řešení. Označme si tedy hledané řídící body A, B, C, D. Přitom samozřejmě krajní body A a D známe a stačí dopočítat body B a C.

Postup začneme úpravou vzorce pro kubický polynom přesně stejným způsobem, jaký jsme použili pro odvození kubického polynomu v kapitole 5.7.2.

$$\begin{aligned} A &= a_0 \\ A' &= a_0 + a_1 + a_2 + a_3 \\ D &= a_1 \\ D' &= a_1 + 2a_2 + 3a_3 \end{aligned}$$

Postupnou úpravou vzorce pro Bézierovu kubiku dostaneme

$$\begin{aligned} f(u) &= (1-u^3)A + 3u(1-u)^2B + 3u^2(1-u)C + u^3D \\ f(u) &= (1-3u+3u^2-u^3)A + (3u-6u^2+3u^3)B + (3u^2-3u^3)C + u^3D \\ f(u) &= A + (-3A+3B)u + (3A-6B+3C)u^2 + (-A+3B-3C+D)u^3 \end{aligned}$$

Dosazením do druhé a čtvrté z předchozí čtveřice rovnic dostaneme následující rovnice:

$$\begin{aligned} A' &= -3A + 3B \\ D' &= -3A + 3B + 6A - 12B + 6C - 3A + 9B - 9C + 3D \end{aligned}$$

Tyto rovnice vypadají na první pohled děsivě, ovšem po převedení do vhodného tvaru dostaneme tyto vztahy pro výpočet bodů B a C:

$$\begin{aligned} B &= A + \frac{1}{3}A' \\ C &= D - \frac{1}{3}D' \end{aligned}$$

Kontrolu správnosti těchto vztahů lze provést např. geometricky. Jelikož derivace v bodě je rovna směrnici tečny, vzorec ve tvaru $y = y_0 + k \cdot y'_0$ je rovnicí tečny v tomto bodě. To také odpovídá našim vzorcům pro B a C s tím, že ve druhé rovnici se odečítá, protože tečný vektor vede vždy doprava, čili v bodě D je to „ven“ z křivky.

Poznámka: Při pohledu na obrázek se naskýtá otázka, jestli se takto vzniklá kubická křivka nemůže v určitých extrémních případech zkroutit tak, že pro dané x bychom měli dvě funkční hodnoty nad sebou. Odpověď na tuto otázku je však záporná.

Jednoznačnost je samozřejmě dána vztahem $x(u) = u$. Všechny obrázky jsou totiž pouze grafickým znázorněním funkce $y = f(x)$, které má tvar $x(u) = u,\ y(u) = f(u)$ a vzorce Bézierových křivek, které byly použity při odvozování všech vztahů, jsou vektorovými vzorci nezávislými na dimenzi prostoru.

Matematicky lze uvedenou vlastnost dokázat dosazením A, A', D, D' do parametrické rovnice pro x. Podle definice Bézierovy křivky platí $u \in \langle 0, 1\rangle$. Z toho přímo plyne $x(A) = 0$ a $x(D) = 1$. Podle Besselova algoritmu je navíc $x(A') = \frac{1}{3}$ a $x(D') = \frac{2}{3}$. Po dosazení těchto hodnot do rovnice Bézierovy kubiky a následných úpravách dostaneme hledanou rovnost:

$$\begin{aligned} x(u) &= u(1-u)^2 + 2u^2(1-u) + u^3 \\ x(u) &= u - 2u^2 + u^3 + 2u^2 - 2u^3 + u^3 \\ x(u) &= u \end{aligned}$$

5.8 Interpolace polynomy vyšších stupňů

Při snaze dosáhnout ještě lepších výsledků pomocí metody interpolace polynomem vyššího stupně narazíme na dva problémy:

1. Polynomy vyšších stupňů mají tendenci se hodně „kroutit“, což místo očekávaného odstranění šumu generuje jiný šum a celý efekt interpolace se tak ztrácí.

2. Vzhledem k přítomnosti loopů v samplech je dosti obtížně řešitelný přístup k mnoha uzlům najednou, i když se jedná o uzly sousední.

Navíc, jak již bylo řečeno, polynomy sudých stupňů jsou vyloženě nevhodné, takže čtvrtý stupeň odpadá. Další na řadě je tedy až polynom pátého stupně. Algoritmus na jeho výpočet je však už skutečně neprakticky náročný.

5.9 Podmínka jednoznačnosti

Jistě nás právem zajímá, jsou-li polynomy vypočítané některou z uvedených metod jednoznačné a zdali je vůbec lze za každých podmínek ze zadaných hodnot vypočítat. Na tyto otázky odpoví následující věta.

Věta 5.1 *Pro daných $n+1$ bodů existuje vždy právě jeden polynom stupně nejvýše n, jehož graf těmito body prochází. Čili platí, že $f(x_i) = y_i$ pro všech $n+1$ bodů ve tvaru (x_i, y_i). Přitom $x = t$ a $y = f(t)$.*

Důkaz: Existenci a jednoznačnost uvedeného interpolačního polynomu dokážeme prostým využitím podmínek interpolace. Mějme dáno $n+1$ bodů ve tvaru $(x_i, y_i), i \in \langle 0, 1\rangle$. Požadavek $f(x_i) = y_i$ vede k soustavě $n+1$ lineárních rovnic o $n+1$ neznámých $a_0, \ldots, a_n$. Za předpokladu $x_i \neq x_j$ je determinant matice této

soustavy nenulový (Vandermondův determinant, viz [1]) a soustava má právě jedno řešení. Určuje tedy jednoznačně právě jeden polynom stupně nejvýše n.

Z tohoto důkazu také vyplývá, že pro jednoznačné určení polynomu můžeme souřadnice některých bodů nahradit derivacemi tak, aby součet známých hodnot byl vždy $n + 1$. Klíčové je totiž mít možnost sestavit zmíněnou soustavu $n + 1$ rovnic. V extrémním případě můžeme vycházet pouze z jednoho bodu a n derivací (tj. hodnot první až n-té derivace v tomto bodě).

Důsledek: Tato věta nám poskytuje návod, jak dokázat jednoznačnost každé metody ve dvou krocích (**Podmínka jednoznačnosti**):

1. Dokážeme, že vycházíme z $n + 1$ bodů, pro které platí $\forall i \in \langle 0, n + 1\rangle$, $\forall j \in \langle 0, n + 1\rangle$, $i \neq j : x_i \neq x_j$. Uvedenou nerovnost ve skutečnosti nemusíme ani dokazovat, jelikož dva různé body jednoduše nikdy nemohou mít stejnou x-ovou souřadnici.

2. Dokážeme, že (nejvyšší možný) stupeň interpolačního polynomu je právě n.

Všechny mnou uvedené interpolační metody splňují podmínku jednoznačnosti. Důkazy jsem v podstatě provedl již během návrhu a popisu jednotlivých metod, proto je tady už nebudu opakovat.

5.10 Šum

Šum odstraněný použitím správné interpolační metody je ve své podstatě šum složený ze základní frekvence $sampos \cdot step$ a vyšších frekvencí, protože funkční hodnoty v ekvidistantních uzlech o intervalu $sampos \cdot step$ známe. Pouze u funkčních hodnot mezi těmito uzly dochází k chybám, čili výsledný chybový článek (zvuk v podobě šumu) nikdy neobsahuje frekvence nižší než $sampos \cdot step$. Pokud k tomuto poznatku přidáme fakt, že člověk je schopen rozeznávat zvuky pouze v určitém frekvenčním spektru[22], dojdeme k závěru, že kdybychom byli schopni zajistit, aby tato hraniční frekvence byla už v neslyšitelném pásmu, žádný šum by se při renderování hudby neprojevoval.

Bohužel, vzhledem k transformaci parametru však žádná frekvence není dostatečně vysoká. Uvedu příklad: Mějme bázovou frekvenci samplu jen velmi málo odlišnou od frekvence mixovací, tj. např. $step = 0.99999$. Pokud hodnoty sousedních uzlů jsou v řadě $-1, +1, -1, +1$, pak sampl popisuje zvuk na frekvenci $\frac{samfreq}{2}$. Při použití bodové interpolační metody vzniká výsledek, ve kterém se objevuje sekvence $-1, +1, +1, -1, +1, -1$. Tento zvuk má pochopitelně nosnou frekvenci opět $samfreq$ a nikoliv požadovanou $step \cdot samfreq$. Jinými slovy do výsledného zvuku se při hodnotách $step$ velmi blízkých celým číslům projevuje silná interference příslušného celočíselného násobku původní frekvence. Kdybychom

[22] zhruba 20Hz - 20kHz

skutečně chtěli zajistit maximální věrnost zvuku i s použitím transformace parametru, muselo by být $samfreq > 1,3\ GHz$, kde se interference na 16bitových samplech již ve slyšitelném spektru neprojevují[23]. Toto si mnoho odborníků vůbec neuvědomuje, když si naivně myslí, že frekvence 44100 Hz použitá pro CD Audio má nějakou globální platnost, tj. že tato frekvence určuje maximální smysluplnou frekvenci zvuku ve všech aplikacích. Ve skutečnosti tomu tak není.

5.11 Shrnutí

Na závěr tedy ještě shrňme poznatky o probraných metodách interpolace zvuku.

interpolační metoda	výpočetní náročnost	hardwarová realizace	softwarová realizace	kvalita zvuku
jednoduchá bodová	velmi nízká	výjimečně	velmi často	neuspokojivá
pravá bodová	nízká	-	-	neuspokojivá
lineární	vyšší	ano	ano	uspokojivá
kvadratická	vysoká	-	-	neuspokojivá
kubická	velmi vysoká	-	výjimečně	vynikající

Tabulka 1: Základní vlastnosti interpolačních metod

[23]Tato moje domněnka je založena na faktu, že 20 kHz krát 65536 hodnot samplu je zhruba 1,3 GHz. V takovém samplu tedy i ten nejvyšší slyšitelný zvuk má rozdíl dvou sousedních uzlů v řádu nejnižšího bitu, čili jakákoliv interpolační technika dává správné výsledky pro všechny slyšitelné frekvence.

6 Filtrace

6.1 Úvod

Další možností, jak se zbavit nežádoucího šumu při syntéze zvuku, je filtrace. Předlohou pro filtraci zvuku mohou být některé algoritmy pro „rozmazávání“ bitmapových obrázků. Je všeobecně známo, že nežádoucí šum ve zvuku se drží ve vyšších frekvencích, čili vyznačuje se poměrně velkými rozdíly mezi hodnotami sousedních uzlů v samplu. Pokud tedy provedeme něco jako rozmazání sousedních hodnot, můžeme očekávat, že dojde k určitému snížení hladiny šumu.

Otázkou samozřejmě je, jakým způsobem provést ono *rozmazání*. Z hlediska efektivity zřejmě nemá smysl provádět filtraci na vstupních samplech. Na rozdíl od interpolačních metod, zde budeme pracovat pouze s výstupními daty syntetizéru. Právě proto, že filtrace se provádí až v okamžiku, kdy už je výstupní zvuk de facto připraven k přehrávání, patří mezi tzv. **postprocessing**, tedy česky „po zpracování“ nebo „dodatečné zpracování“.

V této souvislosti samozřejmě vyvstává zajímavá vlastnost filtrace: Filtraci tohoto typu lze aplikovat na libovolný zvuk (např. soubory WAV nebo MP3). Můžeme tedy např. vyrenderovanou hudbu uložit do souboru WAV a někdy později (kdykoliv) provést dodatečnou filtraci nebo třeba experimentovat s různými metodami filtrace. Můj program samozřejmě provádí filtraci „on the fly“ (během přehrávání). Pomocí nastavitelných parametrů lze tedy měnit parametry filtrace a ihned poslechem porovnávat jejich vliv na konečnou podobu zvuku. Rozdíl ve zvuku vzniklý filtrací je obvykle docela výrazný, takže je skutečně možno jej sledovat přímo sluchem.

Jak známo, výstupem syntetizéru je zvuk v podobě samplu. U všech filtračních metod je základním požadavkem, aby pracovaly na lokální úrovni, tzn. bez znalosti celého samplu.

6.2 Filtrační metody

6.2.1 Konvence značení

Před tím, než se podíváme na možné metody filtrace, je třeba zavést určité konvence značení. Předně předpokládejme, že filtrační metody jsou parametrizovatelné, tj. lze nastavit úroveň filtrace a případně další parametry. Úroveň filtrace budeme značit $howmuchfilter$ a tato hodnota bude pochopitelně v rozmezí $\langle 0, 1\rangle$. Kromě toho použijeme pomocnou hodnotu $howmuchkeep = 1 - howmuchfilter$, která udává naopak úroveň zvuku, který projde na výstup nezměněn.

Aktuální vstupní hodnotu filtrační metody budeme značit *input* a minulou vstupní hodnotu *lastinput*. Podobně aktuální výstupní hodnotu filtrační metody budeme značit *output* a minulou výstupní hodnotu *lastoutput*.

6.2.2 Žádná filtrace

S ohledem na to, že filtrace patří mezi postprocessing, základní metodou filtrace je tzv. „žádná filtrace“, kdy vstupní data jsou předávána na výstup beze změn. Je pochopitelné, že tato metoda je ze všech nejrychlejší a nemá na zvuk vůbec žádný vliv.

$$output = input$$

6.2.3 Metoda Last Input

Tato jednoduchá metoda průměruje dva sousední body.

$$output = \frac{lastinput + input}{2}$$

Experimentálně lze zavést parametrizaci na bázi $howmuchfilter$, což umožní velmi jemně nastavovat úroveň filtrace.

$$output = lastinput \cdot howmuchfilter + input \cdot howmuchkeep$$
$$howmuchfilter \in \langle 0, \tfrac{1}{2} \rangle$$

Tato metoda je skutečně velmi jemná a přestože evidentně „něco dělá“, na výstupu není její činnost vždy úplně snadno postřehnutelná. Je zřejmé, že je-li na vstupu alternující funkce ve tvaru $-1, +1, -1, +1, \ldots$, na výstupu je konstantní nulová funkce. Tato filtrační metoda tedy filtruje jen nejvyšší frekvence, proto ji označuji za jemnou, a proto také při běžné vzorkovací frekvenci 44kHz není její výsledek příliš slyšitelný.

Jemnost metody vychází zejména z faktu, že průměruje pouze dva sousední uzly numerické funkce zvuku, takže je schopna filtrovat pouze šum, který je tvořen odchylkou v jednom uzlu funkce zvuku. Vyskytne-li se v samplu více výchýlených hodnot v řadě za sebou, tato metoda si s nimi neporadí a šum neodfiltruje. Praktický dopad této vlastnosti je v tom, že při mixovací frekvenci 44kHz dochází k filtraci zvuků blízkých 22kHz, což je mimo slyšitelného spektra. Naopak při použití nižší mixovací frekvence, např. 11kHz, je již filtrace provedená touto metodou slyšitelná velmi zřetelně.

6.2.4 Metoda Last Output

Jakýmsi opakem předchozí metody je metoda zvaná Last Output. Tentokrát je totiž použita určitá zpětná vazba, když hodnota výstupu je současně brána jako vstup v dalším uzlu.

$$output = lastoutput \cdot howmuchfilter + input \cdot howmuchkeep$$
$$howmuchfilter \in \langle 0, 1)$$

Na první pohled by se mohlo zdát, že tato metoda je příliš hrubá a vytváří již poměrně zdeformovaný zvuk. V praktických testech se však jednoznačně ukázala jako velmi dobrá. Zatímco metoda Last Input je vhodná pro low-endové systémy mixující v oblasti slyšitelného spektra, metoda Last Output je vhodná pro hi-endové systémy, protože dokáže filtrovat vyšší frekvence zvuku i v situaci, kdy mixovací frekvence je daleko za hranicí slyšitelného spektra.

Metoda Last Output má ještě jednu důležitou vlastnost. V situaci, kdy se hodnota $howmuchfilter$ blíží jedničce, dochází k téměř maximální filtraci a z původního zvuku zůstává jen basové torzo o velmi nízké hlasitosti. Tuto vlastnost jsem se pokusil dále prozkoumat a využít. Výsledek mých pokusů popisuje následující text.

6.3 Vedlejší efekt filtrace - bass-boost

Jak již bylo nastíněno v předcházejícím textu, metoda Last Output má schopnost provádět velmi silnou filtraci zvuku, jejímž výsledkem je potlačení veškerého zvuku kromě nejnižších basů.

Toho jsem s výhodou využil pro sestavení algoritmu bass-boost, který funguje jako čistě softwarový zesilovač nízkých frekvencí ve zvuku (basů). Tento algoritmus dává překvapivě vynikající výsledky a přitom je založen na velmi jednoduchém principu, nenáročném na výkonu mikroprocesoru.

1. Metodou Last Output s koeficientem $howmuchfilter = 0.995$ se provede silná filtrace. Výsledkem je zvuk o nízké hlasitosti, který obsahuje pouze nejnižší frekvence.

2. Získaný zvuk se 20x zesílí a přimíchá se do původního zvuku, ztišeného na 80%.

Konstanty 0.995, 20 a 80 jsem vybral empirickými pokusy. Zde popsaný algoritmus bass-boost jsem implementoval do diplomové práce v uvedené neparametrické podobě, protože zesilování basů jsem nechtěl komplikovat parametrizací jako filtraci. Bylo by to totiž pro uživatele programu značně nepraktické.

6.4 Implementační poznámka

Na rozdíl od jádra syntetizéru, při implementaci filtrů a vůbec celého postprocessingu není nutné se omezovat na celočíselnou aritmetiku. I v případě, že jsou prováděny všechny zmíněné výpočty dohromady, tedy amplifikace, filtrace a bass-boost, stále se jedná o řádově desetinu výpočtů vzhledem k výpočtům při vlastním mixování. Řádově jde o statisíce matematických operací za sekundu, což běžný počítač zvládne hravě.

Celý postprocessing tedy při výpočtech používá 64bitová FPU čísla (typ double), která jsou nejpřesnějšími čísly na mikroprocesorech Intel (mají 52bitovou

mantisu). Tím je zaručena maximální věrnost přenášené informace (zvuku). K tomuto kroku mě nevedla jen snaha o maximální přesnost výpočtů, ale i potřeba reálného násobení při amplifikaci.

Technicky vzato se během postprocessingu provádí dvojí číslená konverze. Nejprve se vstupní 32bitová celá čísla (typ long) převádějí na 64bitová FPU čísla, po provedení všech výpočtů se výsledek konvertuje na 16bitová celá čísla (typ short) požadovaná na výstupu. Tento postup je běžný i v jiných aplikacích.

6.5 Shrnutí

Ačkoliv reálné hudební renderery se v zásadě dělí na dvě skupiny - ty, které používají jen filtraci pro její jednoduchost, a ty, které filtraci nepoužívají vůbec, protože ji považují za „šarlatánský“ přístup. Já myslím, že vhodná kombinace interpolace samplu a následné filtrace dává při renderování nejlepší výsledek. Kombinace obou metod je vidět např. i v Impulse Trackeru (viz [30]), což je asi nejlepší běžně používaný hudební editor.

Zatímco použitím interpolačních polynomů eliminujeme pouze šum, který by vznikal jako numerická chyba při výpočtu, zde popsaná filtrace jde mnohem dál a ze zvuku násilně odstraňuje všechny vyšší (tedy šumové) frekvence.

Myslím, že tři zde uvedené filtrační metody jsou zcela postačující a pokrývají celou škálu situací, které mohou v praxi nastat. Navíc jde o algoritmy velmi jednoduché (tzv. „za málo peněz hodně muziky“), což také stojí za zmínku. Nalezení a implementaci algoritmu bass-boost považuji také za velmi zdařilý výsledek.

7 Parametrizace syntetizéru

K jakým závěrům tedy lze dojít shrnutím předchozích tří kapitol? Základem studia eliminace šumu je parametrický syntetizér, čili syntetizér postavený tak, aby bylo možno kdykoliv měnit jeho klíčové parametry a přímo sledovat, jak tyto změny ovlivňují výsledný zvuk.

Předchozí tři kapitoly popisovaly syntetizér spíše teoreticky. Ačkoliv i na teoretické úrovni jsem se snažil dojít k nějakým závěrům v otázce eliminace šumu, v této kapitole se pokusím shrnout čistě praktické zkušenosti s implementací parametrického syntetizéru.

7.1 Bitová šířka samplů

Sample bit-width neboli bitová šířka samplů závisí čistě na konkrétním hudebním modulu. Starší moduly mají všechny samply (zvuky nástrojů) 8bitové, zatímco v novějších se příležitostně objevují i 16bitové samply. Soubory MP3 se samozřejmě vždy dekódují do 16bitové podoby.

Syntetizér umožňuje právě pomocí volby **sample bit-width** omezit bitovou šířku na 1-16 bitů. Výsledek takového omezení na hladinu šumu ve zvuku závisí především na hlasitosti jednotlivých zvuků. Jsou-li zvuky nástrojů kvalitní, lze se omezit až na 4 bity a dosáhnout ještě snesitelného výsledku. Při omezení na 2 a méně bitů je zvuk již velmi špatný.

7.2 Bitová šířka mixéru

Tato hodnota určuje bitovou šířku proměnných, do kterých se ukládají mezivýsledky výpočtů v syntetizéru. Ačkoliv většina přehrávačů používá 16bitovou šířku mixéru, protože vstupní i výstupní zvuky jsou nejvýše 16bitové, zkusil jsem použít až 32 bitů pro tyto hodnoty a tím eliminovat numerickou chybu.

Bitová šířka mixéru je hlavním faktorem kvality zvuku. Nejnižší počet bitů, při kterém nedochází ke vzniku šumu v důsledku numerických chyb, přímo závisí na použitých algoritmech, popsaných v předchozích kapitolách. Ideální počet bitů mixéru lze přibližně vyjádřit jako součet logaritmu počtu kanálů s počtem bitů samplu a počtem bitů hlasitosti.

$$mixerbits = log_2(channels) + samplebits + volumebits$$

Pro klasické soubory typu MOD tento vzorec vypadá takto: $log_2(4) + 8 + 6 = 16$, což je jistě zajímavý výsledek a skutečně to odpovídá realitě.

Naopak pro rozsáhlejší hudební díla s mnoha nástroji ve formátu IT může vyjít velmi odlišná hodnota, např. $log_2(128)+16+41 = 64$. Tento výsledek vlastně říká, že bychom měli používat 64bitové proměnné pro mezivýsledky. Praktické pokusy

však ukázaly, že rozdíl např. mezi 30 a 32 bitovou reprezentací je naprosto nulový, čili už při těchto hodnotách je numerická chyba mimo 16 bitů, které jsou výsledně použity zvukovou kartou.

Proč tomu tak je? Důvody vidím hned dva. Jednak je třeba si uvědomit, že do zvukové karty jde vždy jen 16 bitů zvuku, tj. jakékoliv numerické chyby v 17. a dalších bitech jsou nepodstatné. Vyskytne-li se během výpočtu chyba např. ve 20.bitu, je docela pravděpodobné, že dojde během dalších výpočtů, zvláště při násobení, k distribuci této chyby do důležitějších bitů, tedy možná až do bitů zvukové karty. Při výskytu chyby ve 30.bitu je však již tak velká distribuce chyby (přes 14 nebo více bitů) zcela vyloučena.

Druhým důvodem je zřejmě zmíněných 41 bitů hlasitosti. Ačkoliv hodnota hlasitosti má skutečně 41 bitů, v drtivé většině případů se lze omezit na daleko menší přesnost, protože celková hlasitost je násobkem několika dílčích hlasitostí, z nichž mnohé jsou během celé skladby konstantní a lze je tedy zanedbat bez jakéhokoliv následku na kvalitě hudby. Ačkoliv nikdy předem nevíme, které z dílčích hodnot budou konstantní, lze očekávat, že některé ano a tudíž celkovou hlasitost lze reprezentovat menším počtem bitů.

V mé implementaci je počet bitů, které jsou použity pro reprezentaci hlasitosti, určen vždy podle následujícího vzorce:

$$\begin{aligned} volumebits &= mixerbits - samplebits - log_2(channels) \\ volumebits &= 16 - log_2(channels) \end{aligned}$$

Výsledkem je tedy nejvyšší možný počet bitů, který umožní, aby všechny výpočty probíhaly s použitím běžných celočíselných 32bitových proměnných. Podobný přístup je možno vidět i v jiných programech.

7.3 Amplifikace

Amplifikace je dodatečné zesílení hlasitosti. Vzhledem k povaze bitových rovnic v předcházející podkapitole je jasné, že u skladeb s velkým počtem kanálů dochází k nepříjemnému jevu - nejdůležitější informace (tedy zvuk) je přenášena v nižších bitech, zatímco vyšší bity jsou konstantní. Jinými slovy z důvodu modulace zvuku mnohabitovou hlasitostí a skládání velkého počtu zvuků dochází k tomu, že výsledný zvuk je sice přesný, ale má velmi malou hlasitost.

Amplifikace tedy spočívá v jednoduchém vynásobení celého výsledného zvuku nějakou konstantou > 1, čímž dojde ke zvýšení hlasitosti. Má implementace amplifikace je založena na 64bitových FPU[24] proměnných typu double (datový typ jazyka C++), protože zde se již není třeba obávat nějakého zkreslení zvuku při použití (nikdy ne zcela přesných) FPU čísel, jelikož je provedeno vždy právě jedno násobení.

[24]FPU = floating point unit, jednotka mikroprocesoru pro práci s čísly v plovoucí řádové čárce.

Výsledek amplifikace je zaokrouhlen na celé číslo a uložen do 16bitového výstupního bufferu, který již představuje data pro zvukovou kartu (nebo grafickou analýzu). Kromě celočíselného zaokrouhlení dochází k posunu o 16 bitů (ze 32bitové hodnoty bereme horních 16 bitů) a ke kontrole mezí, tj. čísla větší než +32767 resp. menší než −32768 jsou upravena právě na tyto hodnoty.

Jak je vidět, každá jednotlivá hodnota funkce zvuku, na které se provádí amplifikace, tedy vyžaduje *pečlivé zacházení*, ovšem tyto operace jsou zanedbatelné vzhledem k jádru syntetizéru a rychlost renderování nijak výrazně neovlivňují.

Výsledkem amplifikace je to, že klíčová nesená informace (tedy zvuk) je posunuta do 16 horních bitů čísla, čímž je subjektivně potlačen šum. A to docela výrazně, protože každé následné zesílení zvuku (ve zvukové kartě, zesilovačem na reproduktorech, atp.) znamená také zesílení šumu, zatímco amplifikace je provedena pouze na hudbě, ne na šumu, který přidá zvuková karta, zesilovač, reproduktory, apod.

Amplifikace tedy není samospasitelná, protože např. soubory MP3 jí upravovat nelze (mají jen jeden kanál zvuku), ale v situacích, kdy ji použít lze (u ostatních typů souborů) je její přínos velký.

7.4 Výstupní filtry

Problematiku výstupních filtrů jsem již probral v samostatné kapitole. Praktické zkušenosti s filtrováním odpovídají teoretickým předpokladům.

Filtrování odstraňuje šum tím, že doslova ruší vyšší frekvence ve zvuku, čímž ruší i šum, protože ten se vyskytuje právě ve vyšších frekvencích. Na druhou stranu to je za cenu snížení kvality hudby, protože se částečně ztrácí i to, co by mělo zůstat.

Problematika filtrace se především liší od ostatních tím, že zde nejde o minimalizaci numerické chyby při výpočtech, nýbrž o odstranění šumu i tam, kde jsme ho dostali už ve vstupních datech, ve vstupním zvuku.

7.5 Interpolace samplů

Interpolace samplů, tedy metody pro výpočet funkčních hodnot numerických funkcí zvuku, jsou dle mého názoru nejzajímavější částí syntetizéru. Implementoval a vyzkoušel jsem všechny interpolační metody nastíněné v kapitole 5 a došel jsem k závěrům, které zhruba odpovídají očekáváním.

Největším zklamáním je kvadratická interpolace. Její nespojitost už od začátku dávala tušit, že výsledky budou nevalné. V praxi navíc dochází k jevu velmi připomínajícímu nějaký generátor náhodných čísel, protože odchylky $samfreq$ a $outfreq$ jsou často velmi malé, čili skutečný výsledek kvadratické interpolace je ještě daleko horší než ten, který je vidět na obrázcích 8 a 9.

Mezi dvěma variantami bodové interpolace není z hlediska kvality zvuku žádný rozdíl, ale grafická analýza rozdíly ukazuje. To je samozřejmě způsobeno omezenými možnostmi grafické analýzy. Zejména pak porovnávání jednoduché bodové interpolace s jakoukoliv jinou ukazuje nesmyslné výsledky.

Lineární interpolace se ukázala jako velmi dobrá, avšak nejsložitější z interpolačních metod, tedy kubická, je nejlepší a jasně zastínila všechny ostatní. Výsledky poslechu i grafického zobrazení jsou zcela jednoznačné.

7.6 Shrnutí

Tato kapitola tedy přinesla výsledky praktických pokusů a měření. Ačkoliv grafická analýza může ukázat různé zajímavé detaily, nejlepším způsobem, jak vyzkoušet a ověřit jednotlivé parametry syntetizéru, je poslech. I když člověk není zvyklý věřit tomu, co „nevidí na vlastní oči“, v tomto případě je výsledek poslechu na kvalitní zvukové kartě s kvalitním zesilovačem a reproduktory určitě lepší než prohlížení grafů.

8 Struktura aplikace MPI a systém tříd

Program, který implementuje poznatky této diplomové práce, jsem nazval MPI. V této závěrečné kapitole stručně proberu strukturu této aplikace a její objektově orientovaný návrh.

8.1 Základní koncepce

Aplikace MPI se chová jako klasická aplikace pro Win32 a z uživatelského hlediska je její základní funkcí přehrávání hudebních souborů ve formátech MOD, S3M, XM, IT a MP3. Aplikace je celá napsaná ve Visual C++ 6.0 SP5 za použití oknové knihovny MFC 4.2. Z tohoto pohledu tedy jde o naprosto standardní aplikaci. K jejímu spuštění je třeba mít operační systém Windows 9x/NT nebo kompatibilní s podporou DirectX 3.

Jádro MPI je platformě nezávislé (až na přenositelnost C++ kódu) a chová se jako renderer digitální počítačové hudby. Vstupem rendereru je vždy soubor v jednom z podporovaných formátů, který se nahraje celý do paměti, dekóduje, převede do interní formy a připraví tak k renderování. Výsledkem renderování je blok paměti přímo použitelný jako výstup na zvukovou kartu.

Uživatelské rozhraní v MFC umožňuje nejen přehrávat hudbu, ale nabízí také několik grafických analytických nástrojů vhodných pro vizuální porovnávání jednotlivých metod rendereru. Z hlediska oken je uživatelské rozhraní naprosto intuitivní a nijak nevybočuje ze standardu MFC aplikací. (Snad s tou výjimkou, že některá obvykle modální okna jsou zde pro větší pohodlí řešena jako nemodální (modeless).)

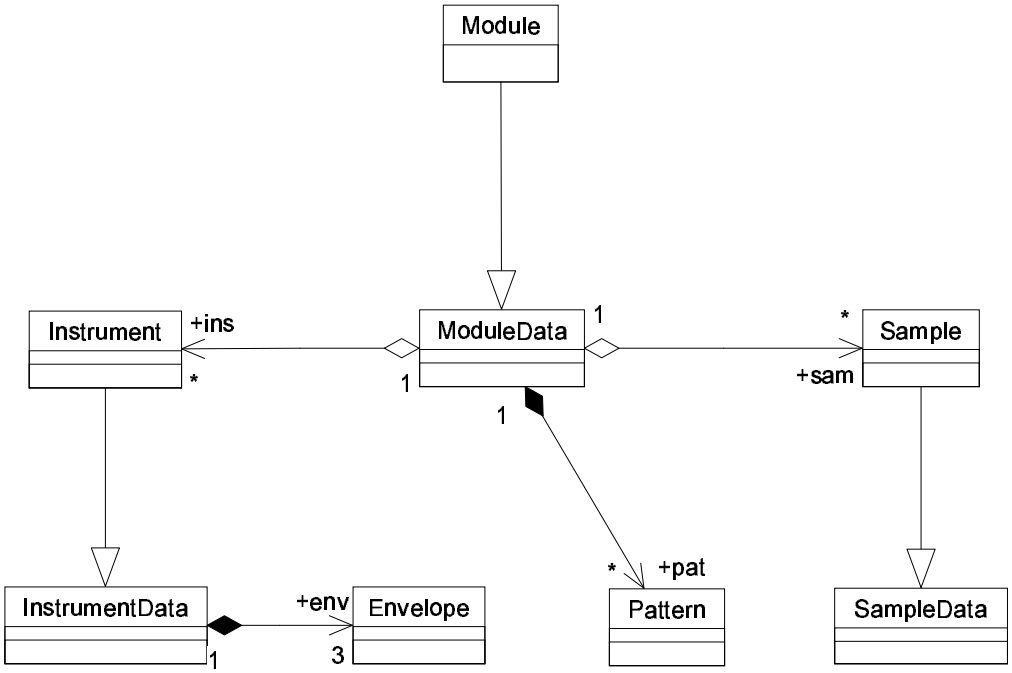
8.2 Struktura aplikace

Třídy, ze kterých se skládá MPI, lze rozdělit do těchto skupin:

1. Module+Loader - třídy zajišťující nahrání hudebních modulů do paměti a jejich datovou reprezentaci v paměti
2. Renderer+Player - třídy zajišťující renderování a přehrávání hudby (kromě platformě závislého kódu zvukové karty)
3. UI - třídy uživatelského rozhraní, tj. jednotlivých oken a dalších věcí závislých na MFC
4. Console - třídy pomocné MPI konzoly, která slouží jako pomůcka pro ladění (realtime zobrazování důležitých dat)
5. Pomocné třídy - OneSound, fstring a další pomocné třídy, doplňující funkčnost CRT (C runtime), STL (C++ standard template library) a MFC knihoven

8.3 Třídy reprezentující hudební modul

Hudební modul (neboli soubor s hudbou) je vysoce strukturovaný soubor obsahující velké množství rozličných dat. Každý modul je nahrán do paměti pomocí speciálního loaderu (v závislosti na typu souboru), formát modulu v paměti je postaven jako systém několika tříd a je zcela nezávislý od formátu souboru na disku.



Obrázek 13: Třídy reprezentující hudební modul

8.3.1 Patterny - třídy PatData, Pattern, PatParser

Třída **PatData** je pětibajtová datová struktura popisující formát jedné buňky patternu (nota s oktávou, nástroj, hlasitost, příkaz, parametr).

Třída **Pattern** uchovává data jednoho patternu v zapakované podobě. Zapakovaný formát je navržen tak, aby mohl být přímo interpretován, odpovídá tedy zápisu programu v jazyku imperativního typu.

Data patternu se inicializují pomocí metody **Pattern::packpat(int tracks, BYTE *src)**, která přijímá nezapakovaný (dekódovaný) pattern z loaderu a vytváří jeho zapakovanou podobu. Nezapakovanou podobu patternu, která slouží jako vstup do této metody, musí vytvořit loader (viz níže).

Třída **PatParser** slouží při renderování jako dodavatel dat ze zapakovaného patternu do sekvenceru. Zatímco Pattern umí data přijmout a uchovávat, PatParser umí data interpretovat, tedy nejen číst, ale i rozeznávat skoky a smyčky.

Jinými slovy PatParser je třída, která zpřehledňuje zápis kódu třídy Pattern a sekvenceru (viz níže) tím, že přebírá část jejich úkolů na sebe[25].

8.3.2 Samply - třídy Sample, SampleData

Samply (vzorky zvuků nástrojů) jsou uloženy ve třídě **Sample**, která dědí ze **SampleData**. Význam dědění je zde čistě implementační a nemá žádnou souvislost s objektově orientovaným návrhem.

Podobně jako Pattern, i třída Sample se sama stará o import dat. Vzhledem k rozsahu zpracovávaných dat je však tentokrát vstupem přímo soubor na disku. Metoda **Sample::load(FILE *f, bool resign, bool delta)** nahraje tělo samplu do paměti, přičemž předpokládá, že parametry samplu jsou již nastaveny. Formát samplu na disku je specifikován dalšími parametry této metody (resign umožňuje konverzi signed⟷unsigned, delta umožňuje provádět import dat v delta formátu namísto klasického PCM).

Třída **SampleData** obsahuje vlastní datové prvky (atributy) samplu. Je to celkem 13 číselných hodnot u každého samplu, pro podrobnější popis viz komentáře v souboru **Sample.h**.

8.3.3 Nástroje - třídy Instrument, InstrumentData, Envelope

Tyto tři třídy popisují nástroje (až na definice samplů). Data obálek jsou vyčleněna do samostatné třídy **Envelope**, protože každý nástroj má tři obálky, které jsou popsány trojicí objektů zmíněného typu.

Podobně jako u samplů, i u nástrojů je vztah mezi **Instrument** a **InstrumentData** na bázi dědičnosti. Pro podrobnější popis všech atributů tříd **InstrumentData** a **Envelope** viz komentáře v souboru **Instrument.h**.

8.3.4 Modul a loadery - třídy Module, ModuleData

Jak již bylo uvedeno, modul je uložen v systému několika tříd (viz obrázek 13). Zastřešením všech těchto tříd a přístupovým bodem do modulu je třída **Module**, která dědí **ModuleData**.

Nahrání obecného souboru zajišťuje metoda **Module::loadmodule(const char *fname)**, která otevře soubor a volá **Module::loadmodule(FILE*)**. Tam je identifikován typ souboru a volána metoda příslušného loaderu loadmod/loads3m/loadxm/loadit/loadmp3. Pro podrobnější informace o loaderech viz soubory **loadmod.cpp**, **loads3m.cpp**, **loadxm.cpp**, **loadit.cpp**, **loadmp3.cpp**.

Všechny loadery pracují rámcově stejným způsobem, proto je k dispozici soubor **loadersup.h**, který obsahuje několik společných pomocných maker pro alo-

[25]Tzv. princip rozděl a panuj.

kaci paměti, ošetření chyb, atp. Využitím těchto maker se výrazně zjednodušily některé obecné operace v loaderech a tím se i zpřehlednil kód.

8.3.5 Nahrávání patternů

V této části bych se rád zastavil u nahrávání patternů. Jak již bylo mnohokrát naznačeno, pattern není nic jiného než určitý časově omezený úsek skladby. Datová reprezentace patternu se v jednotlivých souborových formátech dosti liší, avšak podařilo se mi vysledovat určité společné rysy (viz také kap.8.3.1).

Základní myšlenkou je tedy pohled na pattern jako na dvojrozměrné pole prvků typu PatData (viz 8.3.1). Každý loader musí umět načíst a dekódovat patterny z jemu odpovídajících souborů a vytvořit toto dvojrozměrné pole (vnitřním rozměrem jsou stopy, vnějším rozměrem je čas). Metoda **Pattern::packpat(int tracks,BYTE *src)** potom parsuje toto dvojrozměrné pole a vytváří tzv. pakovaný tvar patternu, což je de facto kód, který potom lze přímo interpretovat.

Význam některých hodnot v PatData se může u různých souborů lišit. Loadery se snaží v maximální míře vytvářet obecný standardizovaný tvar patternu, ale přesto zůstávají některé dílčí vlastnosti, které je třeba interpretovat specifickým způsobem, podle typu souboru. Proto každý loader nastavuje hodnotu výčtového atributu **Module::volcol** na příslušnou hodnotu. Více viz soubor **Module.h**, definice **enum VolCol** a atributu **volcol**.

8.4 Třídy rendereru a playeru

Renderer a player fungují a existují tak trochu „ruku v ruce“. Zatímco renderer a jeho třídy slouží k výpočtu (renderování) hudby v matematické podobě, player slouží k přehrávání takto vypočítané hudby v reálném čase na zvukové kartě.

8.4.1 Třídy kanálů, stop a zvukových efektů

Nejrozsáhlejším bodem rendereru je zřejmě třída **Track** a její vnitřní třídy. Třída Track reprezentuje zvukovou stopu, což odpovídá stopě v patternu (viz výše). Na druhou stranu Track příliš nesouvisí s konkrétními zvuky, protože jde o třídu na vyšší úrovni než syntetizér, takže s konkrétními zvuky se zde přímo nepočítá.

Vnitřní třídy Trackeru (viz soubor **Tracker.h**) slouží k výpočtu zvukových efektů. Většina zvukových efektů rendereru funguje na bázi dynamické změny některých parametrů jednotlivých nástrojů a zvuků (jako hlasitost, frekvence, apod.). Data řídící funkčnost a parametry efektů jsou zapsána na patřičných místech v patternech a výsledky výpočtů se v reálném čase ukládají do každé stopy (track). Hraje-li ve společné stopě více zvuků současně (což je možné), potom vždy nejvýše jeden z nich může mít přidružený zvukový efekt nebo efekty.

Toto omezení je „by design“ hudebních modulů a nemá nic společného s mou prací.

Zatímco zvuková stopa v podobě Track má přímou návaznost na patterny, další třída nazvaná **Channel** reprezentuje zvukový kanál, tedy jeden konkrétní fyzicky hrající zvuk. Zatímco ve starších hudebních editorech neexistovala možnost nějak odlišit stopu (track) a kanál (channel), novější editory a přehrávače (čili také MPI) je rozlišují.

K jedné stopě tedy může být přidruženo (dle příkazů v patternu) libovolné množství kanálů, ale ne naopak. Každý kanál je pak použit přímo v syntetizéru (viz níže) k renderování konkrétního zvuku. Přitom pomocí některých zvukových efektů je možno parametry kanálu docela rychle měnit a tím vytvářet vibrato, tremolo, portamento, atp.

Pro další informace o třídě Channel viz soubor **Channel.h**.

8.4.2 Třídy sekvenceru

Sekvencer je reprezentován třídou **Sequencer**, přičemž velký díl práce provede již dříve zmíněná třída **PatParser**, která slouží jako odstínění sekvenceru od konkrétní datové reprezentace patternů.

Sekvencer je jakýsi střední bod rendereru, protože na jedné straně do něj přicházejí data z patternů, nástrojů a samplů a na druhé straně z něj vycházejí řídící povely pro syntetizér. Z pohledu datové abstrakce je tedy sekvencer článkem, který funguje jako jakýsi můstek mezi hudbou v notovém zápisu a nízkoúrovňovým streamem (proudem) dat řídících syntetizér.

Sekvencer je v pravém slova smyslu srdcem rendereru. Jelikož je renderování hudby závislé na čase (na renderovacím čase, ne reálném), sekvencer funguje jako srdce, které bije v pravidelném rytmu a posílá pomocí metody **Sequencer::update(bool playing)** impulzy života do dalších dříve zmíněných částí rendereru. Sekvencer dávkuje čas podle přesně nastavených pravidel (která se mohou i během skladby měnit dle příkazů v patternu). Parametr **playing** zmíněné metody umožňuje snadnou implementaci funkce „pause“ do přehrávače, protože jeho nastavením na false dojde k tomu, že vnitřní hodiny stále tikají, ale sekvencer je zmrazen v jednom stavu a syntetizér generuje nulový výstup.

8.4.3 Třídy syntetizéru

Syntetizér je reprezentován třídou **Synthesizer**. Dostává data a časové impulzy ze sekvenceru (viz výše) a podle těchto dat vytváří (renderuje) výstupní zvukový stream (proud PCM dat). Syntetizér je jediným článkem systému, který přímo pracuje se zvukem, proto ho lze považovat za nejnižší článek z hlediska datové abstrakce.

Klíčovým prvkem syntetizéru je metoda **Synthesizer::fillbuf(bool playing, bool stereo)**, která především volá dvě důležité privátní metody. První

z nich je **Synthesizer::mix1ch(Channel *chn, bool stereo)**, která dostane na vstupu ukazatel na kanál a provede přimixování (syntézu) zvuku tohoto kanálu do výstupního bufferu. Tato metoda je tedy volána postupně pro všechny kanály a následně je volána druhá zmíněná metoda **Synthesizer::postprocess()**, která provede tzv. postprocessing, čili filtraci, amplifikaci apod. Více o postprocessingu v kapitole 6.

Další třídou týkající se syntetizéru je **SynthOptions**, třída zajišťující správu profilů syntetizéru. Ačkoliv renderer používá dlouhodobě jen jednu instanci (jeden objekt) syntetizéru, pomocí SynthOptions je možno jednoduše spravovat a přepínat mezi různými profily. Syntetizér reprezentovaný třídou Synthesizer je pochopitelně široce parametrizovatelný a s podporou třídy SynthOptions je možno jednoduše přepínat mezi celými sadami parametrů. SynthOptions navíc zajišťuje ukládání (a následné obnovení) konfigurace profilů v systémovém registru Windows.

8.4.4 Třídy přehrávače

Přehrávač je reprezentován třídou **Player**, která se stará jen o časování rendereru a předávání dat ze syntetizéru do zvukové karty. Výstup na zvukovou kartu zajišťuje pomocná třída **OneSound**, která je vystavěna jako platformě závislá třída zajišťující realtimové přehrávání zvuku pomocí rozhraní DirectSound z balíku DirectX 3. Soubory knihovny DirectX 3 pro C++ jsou v podadresáři **dx3** projektu MPI. Tyto soubory jsou originální verze od Microsoftu, který ovšem podporuje vždy jen nejnovější verzi DirectX. Díky využití DirectX 3 by mělo být možné provozovat MPI i na systému Windows NT 4.0 SP3, který žádnou vyšší verzi DirectX nepodporuje.

Pro správné časování používá třída Player především realtimové informace z třídy OneSound, čili průběžně si zjišťuje stav zvukové karty a podle potřeby doplňuje další data do bufferu (vyrovnávací paměti). Toto „zjišťování“ musí být z technických důvodů prováděno pollingem[26], takže funguje buď na bázi multimedia timeru (časovače) nebo dedikovaného threadu (vlákna). Třída Player obsahuje kód pro časovač i vlákno a ačkoliv se zpočátku zdálo jako více logické využití časovače, nakonec jsem dal přednost vláknu. Hlavní důvod je asi třeba hledat někde v designu operačního systému Windows, vlákna jsou jednoduše lepší a bezpečnější než časovače, zvláště při ladění programu.

8.5 Třídy uživatelského rozhraní

Všechny třídy uživatelského rozhraní na bázi MFC se drží obvyklé konvence názvu **CNěco**, kde „Něco“ je slovo nebo sousloví charakterizující význam třídy.

[26]Polling znamená periodické zjišťování stavu pomocí opakovaných dotazů, obecná zvuková karta sama neumí oznamovat změnu svého stavu pomocí přerušení apod.

Právě kvůli tomu, že uživatelské rozhraní je postaveno na bázi MFC, odpovídá každá jeho třída jednomu oknu. Viz následující seznam:

Analyzer	Grafický analyzér. Kreslí grafy do hlavního okna.
CAboutDlg	Dialog 'About'.
CAnalyzeWizard	Dialog 'Retrieve data' analyzéru.
CChildView	Zajišťuje kreslení vnitřku hlavního okna. Spolupracuje s třídou **Analyzer**.
CMainFrame	Hlavní okno (kromě vnitřku, do kterého se kreslí).
CModProp	Společná třída pro všechny karty dialogu 'Module properties'.
CModPropSheet	Dialog 'Module properties'.
CMpiApp	Třída MFC aplikace.
COptSheet	Dialog 'Options'.
CSoundcard	Karta 'Soundcard' dialogu 'Options'.
CSynth	Karta 'Synthesizer' dialogů 'Options' a 'Retrieve data'.
CTimeCut	Karta 'Cutting' dialogu 'Retrieve data'.

8.6 Systémové prostředky

Jak již bylo naznačeno dříve, MPI používá celou řadu systémových prostředků.

8.6.1 Vlákna

Jelikož mám z praxe velmi dobré zkušenosti s využitím více vláken (threadů), nebál jsem se ani zde použít tolik vláken, aby aplikace nabízela co nejlepší funkčnost.

Primární vlákno procesu se stará o obsluhu fronty zpráv a všechny související záležitosti. Přehrávač používá pro renderování hudby své vlastní vlákno + sadu synchronizačních prostředků. Díky použití samostatného vlákna pro renderování v přehrávači má aplikace velmi dobrou odezvu na povely uživatele, protože ať už zrovna počítá cokoliv, primární vlákno je vždy připraveno reagovat na uživatelské povely, překreslovat okno, kdykoli je to potřeba, atp.

Třetí vlákno je použito v debugové konzole, což je speciální konzolové okno, kde se vypisují různé informace během přehrávání, především s důrazem na rychlost zobrazení. Toto konzolové okno je možno přepnout do celoobrazovkového textového režimu VGA a získat tak nejvyšší možnou rychlost obnovování obrazu.

Poznámka: Celoobrazovkové zobrazení nemusí z technických důvodů správně fungovat na některých počítačích se systémem řady Windows NT. Chování na konkrétním počítači závisí zejména na jeho hardwarové konfiguraci.

Dalších několik vláken používá subsystém MFC a DirectX.

8.6.2 Zvuková karta a DirectX

Ačkoliv podpora zvukových karet je přítomna přímo v původním Win32 API, já jsem použil funkce DirectX, protože nabízejí řadu zajímavých rozšíření, výhodných pro tuto aplikaci.

8.7 Použité knihovny

V závěru kapitoly ještě shrnu knihovny použité v aplikaci MPI.

8.7.1 Obecné knihovny

V prvé řadě jsem se snažil vytěžit maximum ze standardní knihovny C (CRT) a C++ (STL). Pomocí šablon (templates) knihovny STL jsem snadno realizoval veškeré dynamické objekty od malých textových řetězců až po velká dynamická pole.

Rovněž knihovna MFC nabízí řadu zajímavých pomocných tříd. Třídy MFC jsem však nepoužíval v platformě nezávislém jádru rendereru, už proto, že jsem si tam vystačil s STL.

Mezi obecné knihovny lze počítat také DirectX 3, které je součástí Windows a Visual C++.

8.7.2 NekoAmp

Loader souborů MP3 je postaven na dekodéru NekoAmp, který je napsán v objektově orientovaném C++. Soubory NekoAmp verze 1.4 jsou v podadresáři **NekoAmp** a do projektu se linkují jako statická knihovna. V kódu NekoAmp jsem kromě kompilace do DLL neprovedl žádné změny, pouze jsem napsal nadstavbu, která NekoAmp volá. Více viz soubor **loadmp3.cpp**.

8.7.3 AGO

Z (mého vlastního) balíku multimediálních tříd AGO jsem použil třídu **OneSound**, o které je řeč v kapitole 8.4.4.

8.7.4 Další třídy a šablony

Další malé pomocné třídy z mých dřívějších prací jsou **fstring** a **Console**.

Třída **fstring** rozšiřuje třídu **std::string** o schopnost formátování textu a používá se tedy namísto klasického sprintf, protože tato třída pro použití nepotřebuje pomocný buffer jako sprintf.

Třída nebo spíše modul **Console** rozšiřuje modul konzolových funkcí známý jako **conio.h** o podporu barevného textu a další funkce známé z Borland C++ pro DOS. Tyto funkce se používají v debugové konzole.

Kromě toho jsem definoval a použil i několik dalších menších tříd a šablon, které mi zjednodušily práci a zkrátily kód. Viz soubor **const.h** a další.

9 Závěr

V této diplomové práci jsem se zaměřil na dosud nepříliš prozkoumanou problematiku vzniku šumu ve zvuku a hudbě při jejich počítačovém zpracování.

Základem práce je teoretická studie numerické reprezentace a zpracování reálných funkcí jedné reálné proměnné. Na tomto základu jsem potom vystavěl algebraicky založenou teorii systému normovaných funkcí, kterou lze přímo aplikovat při renderování zvuku.

Dále jsem se velmi podrobně zabýval interpolačními metodami vhodnými pro použití při syntéze zvuku a možnostmi vzniku numerických chyb při operacích se zvukem v syntetizéru. Kromě interpolačních metod jsem prozkoumal i otázku filtrace existujícího šumu ve zvuku, bez ohledu na to, jak tento šum vznikl.

Praktická část diplomové práce staví zejména na mých už osmiletých zkušenostech v oblasti renderování digitální počítačové hudby, které jsem využil pro návrh a implementaci trackeru a sekvenceru digitální hudby. Tyto pak posloužily jako zdroj zcela reálných dat pro syntetizér, který realizuje poznatky z teoretické části práce. Ačkoliv se diplomová práce zaměřila především na zvuk, bez detailního nastudování a realizace hudebního rendereru jako celku by nešlo práci realizovat. Proto se nelze divit, že - ačkoliv to z tohoto textu nemusí být zcela patrné - program, který vzešel z této práce, je z velké části zaměřen na renderování hudby jako takové, zatímco problematika eliminace šumu tvoří pouze jeho menší část.

Myslím, že výsledky práce lze považovat za úspěšné, a to jak na teoretické, tak na praktické úrovni. Podařilo se mi snad celkem úspěšně skloubit poznatky z algebry, numerické matematiky, geometrie a také částečně fyziky. Výsledkem je jednak program umožňující velmi kvalitní renderování hudby a také celá řada teoretických poznatků, které lze v budoucnu použít i v jiných oblastech. Zejména závěry související se vztahem numerické matematiky, numerických chyb a počítačové implementace numerických výpočtů jsou široce aplikovatelné i na jiné aplikace a řešení problémů založených na spojitých reálných veličinách.

Program o 12000 řádcích v C++ je zároveň ukázkou, jak lze realizovat renderer hudby a přitom se vejít do slušné velikosti kódu. Během práce jsem zkoušel i různé optimalizace rychlosti v assembleru, ovšem ve finální verzi jsem se s ohledem na to, že jde o diplomovou práci, spolehnul výhradně na C++. Bodem, ve kterém jednoznačně vidím ještě velký prostor pro další práci, je kód trackeru zajišťující výpočet zvukových efektů nutných pro správnou interpretaci všech skladeb. Ten nelze v žádném případě považovat za dokonalý, protože - jak už to bývá - kdybych se snažil dotáhnout tento bod do 100% dokonalosti, rozsah práce by výrazně překročil únosnou mez a nemohl bych se tedy již plně věnovat klíčovým otázkám diplomové práce.

Závěrem lze říci, že tato práce rozhodně přinesla pozitivní hmatatelné výsledky a zároveň vytýčila určitý prostor pro další práci v oblasti.

A Komentovaný popis hudebních modulů XM

2.vydání

Tento dokument vychází z dokumentu Tritonu [16], proto se v případě nějakých nejasností obraťte na originální anglickou dokumentaci.

Tento text je postaven jako komentovaný překlad doplněný o velké množství poznámek překladatele, které vycházejí z praktických zkušeností a měly by čtenáři pomoci lépe pochopit všechny detaily tohoto formátu.

Na začátku každého bloku je čtyřbajtová hodnota určující délku daného bloku. Délka bloku je počítána včetně těchto čtyř bajtů!!!

A.1 Sturktura XM souboru

A.1.1 Začátek souboru

Na začátku souboru je vždy název modulu, který je ukončen terminátorem ($1a ASCII - znak konce textového souboru), takže pro vypsání vnitřního názvu skladby na terminál je možno použít příkaz *type ¡soubor.xm¿*.

offset	délka	typ	popis
0	17	char	ID text: "Extended Module: " (bez uvozovek) Dejte si pozor na velká a malá písmena!
17	20	char	ASCIIZ jméno skladby
37	1	char	$1a (terminátor)
38	20	char	Název trackeru (obvykle "Fast Tracker v2.00")
58	2	word	Číslo verze souboru: hi-byte = celočíselná část lo-byte = desetinná část Současný formát je verze $0104 (verze 1.04)

A.1.2 Hlavička

Je to trochu paradoxní, ale za hlavičku není považován začátek souboru, ale až tato část.

offset	délka	typ	popis
60	4	dword	Délka hlavičky
+4	2	word	Song length (počet záznamů v oderlistu)
+6	2	word	Restart position
+8	2	word	Počet kanálů (2-32, jen sudá čísla)
+10	2	word	Počet patternů v souboru (max.256) [PATTS] (v orderlistu se mohou vyskytnout i vyšší čísla) (neuložené patterny jsou prázdné s délkou 64 řádků)
+12	2	word	Počet nástrojů (max.128) [INSTRUMENTS]
+14	2	word	Flagy: bit 0: 0 = Amiga frequency table (viz níže) 1 = Linear frequency table
+16	2	word	Počáteční rychlost - FT2 značí jako tempo
+18	2	word	Počáteční BpM tempo - FT2 značí jako BPM
+20	256	byte	Orderlist = sequence

A.1.3 Patterny

Za hlavičkou následuje celkem [PATTS] nástrojů. Data každého patternu se skládají z krátké hlavičky a těla patternu, které ihned následuje.

offset	délka	typ	popis
?	4	dword	Délka hlavičky patternu
+4	1	byte	Typ komprese (vždy 0)
+5	2	word	Počet řádků v patternu (1-256)
+7	2	word	Délka zapakovaných dat (může být nula, když je pattern prázdný)
?	?		Vlastní data patternu (viz popis níže)

A.1.4 Nástroje

Po patternech následuje celkem [INSTRUMENTS] nástrojů. Každý nástroj začíná následující krátkou hlavičkou.

offset	délka	typ	popis
?	4	dword	Délka hlavičky nástroje - Tato hodnota je včetně následujícího bloku (do ofs.243). Obvykle je zde číslo 263, protože XM ukládá ještě dalších 20 bajtů na offset 243. Bohužel neznám jejich význam.
+4	22	char	Jméno nástroje (pravděpodobně ASCIIZ)
+26	1	byte	Typ nástroje (vždy 0) (prý se zde někdy objevují i jiná čísla)
+27	2	word	Počet samplů v nástroji

Zbytek popisu nástroje je v souboru uložen pouze u nástrojů, které mají nějaké samply. Novější editory však mohou ukládat nástroje jiným způsobem, proto by každý loader měl kontrolovat údaj o délce hlavičky nástroje a načíst přesně tolik bajtů, kolik udává hlavička. Jen tak bude zaručena bezpečná funkčnost.

offset	délka	typ	popis
+29	4	dword	Délka hlavičky samplu (viz dále, obvykle 40 bajtů)
+33	96	byte	Tabulka not (čísla samplů pro jednotlivé noty)
+129	48	byte	Jednotlivé uzly volume obálky
+177	48	byte	Jednotlivé uzly panning obálky Obě obálky mají stejný formát: 12 uzlů po dvou wordech: tick + value (0-64)
+225	1	byte	Počet použitých uzlů ve volume obálce
+226	1	byte	Počet použitých uzlů v panning obálce
+227	1	byte	Volume sustain point (číslo uzlu, –1=vypnuto)
+228	1	byte	Volume LpBeg - začátek loopu (číslo prvního uzlu)
+229	1	byte	Volume LpEnd - konec loopu (číslo posledního uzlu)
+230	1	byte	Panning sustain point (číslo uzlu, –1=vypnuto)
+231	1	byte	Panning LpBeg - začátek loopu (číslo prvního uzlu)
+232	1	byte	Panning LpEnd - konec loopu (číslo posledního uzlu)
+233	1	byte	Volume type: bit 0=On; bit 1=Sustain; bit 2=Loop
+234	1	byte	Panning type: bit 0=On; bit 1=Sustain; bit 2=Loop
+235	1	byte	Vibrato - typ
+236	1	byte	Vibrato - sweep
+237	1	byte	Vibrato - depth (hloubka)
+238	1	byte	Vibrato - rate (rychlost)
+239	2	word	Volume fadeout (jen při zapnuté volume envelope)
+241	2	word	rezervováno pro další rozšíření

Na offsetu +241 ještě následuje něco „nezdokumentovaného“, obvykle o velikosti 20 bajtů.

Každý nástroj má svou vlastní sadu samplů. Pokud je počet samplů v nástroji > 0, následují v souboru samply jeden za druhým hned za hlavičkou nástroje v následujícím formátu.

A.1.5 Hlavička samplu

offset	délka	typ	popis
?	4	dword	Length - délka samplu
+4	4	dword	LpBeg - začátek loopu
+8	4	dword	LpLen - délka loopu všechny délky jsou v bajtech (pokud je LpLen = 0, není loop)
+12	1	byte	Volume
+13	1	byte	Finetune (signed byte -128..+127)
+14	1	byte	Flagy: bit 0-1: 0 = No loop 1 = Forward loop 2 = Ping-pong loop bit 4: 0 = 8bitový sampl 1 = 16bitový sampl
+15	1	byte	Panning (0-255)
+16	1	byte	Relative note number (signed byte)
+17	1	byte	Reserved
+18	22	char	Název samplu (pravděpodobně ASCIIZ)

A.1.6 Tělo samplu

POZOR! Každý nástroj má uloženy nejprve hlavičky všech svých samplů a teprve potom následují těla těchto samplů. To sice komplikuje konstrukci loaderu, avšak je typické, že autoři formátu XM hleděli především sami na sebe a udělali to tak, jak se jim samým zrovna hodilo. Právě proto jsou také těla samplů uložena v neuvěřitelném delta formátu namísto klasické PCM podoby.

Samply jsou tedy uloženy jako „delta values“, tedy česky „rozdílové hodnoty“. Pro převod na skutečné hodnoty použijte následující postup[27]:

```
for i=1 to length-1 do begin
  sample[i]=sample[i]+sample[i-1];
end;
```

Pro usnadnění ještě uvádím přepis algoritmu do C++:

```
for(int i=1;i<length;i++) {
  sample[i]=sample[i]+sample[i-1];
}
```

[27]Tento algoritmus jsem sám sestrojil, protože originální pětiřádkový kód od autorů XM je neskutečně nepřehledný a jeho efekt je úplně stejný.

A.2 Formát patternu

Buňky jednotlivých kanálů a řádků patternu jsou uloženy stejně jako u MODů, ale každá nota zabírá 5 bajtů:

offset	délka	typ	popis
?	1	byte	Nota (0=nic, 1-96=noty (1=C-0 atd.), 97=key off)
+1	1	byte	Nástroj (0 nebo 1-128)
+2	1	byte	Volume (viz níže)
+3	1	byte	Effect command (příkaz)
+4	1	byte	Effect parameter (parametr)

Navíc je použita jednoduchá kompresní (pakovací) metoda, takže patterny nejsou až tak moc dlouhé. Protože nejvyšší bit prvního bajtu je normálně nevyužit, je použit právě pro kompresi. Je-li tento bit nastaven, ostatní bity mají následující význam:

bit 0 [01] : Následuje nota
bit 1 [02] : Následuje nástroj
bit 2 [04] : Následuje volume
bit 3 [08] : Následuje command
bit 4 [10] : Následuje parametr

Je to velmi jednoduché, ale zdaleka ne dokonalé. Jestli byste chtěli něco lepšího, můžete si kdykoliv patterny přepakovat dle svého gusta, nejlépe při nahrávání z disku. Tak to samozřejmě dělají i mnohé existující přehrávače, včetně MPI.

A.3 Volume & Envelope (hlasitost a obálka)

Výpočet volume:

$$FinalVol = \frac{FadeOutVol}{32768} \cdot \frac{EnvelopeVol}{64} \cdot \frac{GlobalVol}{64} \cdot \frac{Vol}{64} \cdot Scale$$

Poznámka: V oficiální dokumentaci je místo 32768 číslo 65536.

Výpočet panningu:

$$FinalPan = Pan + \frac{(EnvelopePan - 32) \cdot (128 - |Pan - 128|)}{32}$$

A.4 Obálky

Obálky jsou tikány v každém snímku, místo toho, aby se tikaly jen ve snímcích, kdy nezačínají nové noty. Totéž platí pro nástrojové vibrato a fadeout. Cituji autora FT2: „Jelikož jsem tak líný a z FT2 se to dá přímo pochopit, nic víc vám sem už nenapíšu.“

A.5 Periody a frekvence

```
PatternNote = 0..95 (0 = C-0, 95 = B-7)

FineTune = -128..+127 (-128=-1 půltón, +127=+127/128 půltónu)
RelativeTone = -96..95 (0 => C-4 = C-4)

RealNote = PatternNote + RelativeTone; (0..118, 0=C-0, 118=A#9)
```

A.6 Linear frequency table

Zde jsou vzorce pro výpočet lineární frekvenční tabulky. Tyto vzorce nejsou originálními vzorcy autorů FT2, nýbrž mnou zjednodušené verze. Podobně jako u algoritmu nahrávání samplu, i zde autoři FT2 uvádějí zbytečně komplikovaný algoritmus, proto jsem ho zjednodušil.

$$Period = 16 \cdot 4 \cdot (10 \cdot 12 - Note) - \frac{FineTune}{2}$$

$$Frequency = 8363 \cdot 2^{6 - \frac{Period}{12 \cdot 16 \cdot 4}}$$

Tyto vzorce jsou kriticky důležité. Ať už přehrávač pracuje přímo ve formátu XM, anebo (raději) převádí XM hodnoty na skutečné vzorkovací frekvence, je třeba tento algoritmus realizovat na počítači. Můj postup pro výpočet frekvence funguje takto:

1. Výpočet periody je proveden celočíselně, čili dostaneme dvojnásobnou hodnotu.

   $$Period = 128 \cdot (120 - Note) - FineTune$$

   Číslo 120 v tomto vzorci má úzkou souvislost s počtem oktáv, se kterým pracujete. Pokud snad někdo bude mít nepřekonatelné potíže ve výpočtu přesné frekvence, lze postupovat systémem pokus-omyl. Začněte u konstanty 120 a postupně odečítejte 12 jednotek (čili jednu oktávu) tak dlouho, až se trefíte do hledané oktávy (oktávu byste měli být schopni zkontrolovat jednoduše poslechem). Pokud výsledek ještě nebude přesný, může pomoci přičtení nebo odečtení jedničky - v závislosti na reprezentaci nejnižší noty nulou nebo jedničkou.

2. Nyní je potřeba spočítat především reálnou mocninu dvojky.

$$\begin{aligned} Frequency &= 8363 \cdot 2^{(6 \cdot 12 \cdot 128 - Period)/(12 \cdot 128)} \\ \ln(Frequency) &= \ln(8363) + ((6 \cdot 12 \cdot 64 - Period)/(12 \cdot 64)) \cdot \ln(2) \end{aligned}$$

Hodnoty ln(8363) a ln(2) jsou konstanty, takže si je můžeme spočítat dopředu. Označme si tyto hodnoty $L8368$ a $L2$.

$$Frequency = L8363 + ((6 \cdot 12 \cdot 64 - Period)/(12 \cdot 64)) \cdot L2$$

Ovšem, jak už tomu tak bývá, většina ostatních programů používá zcela jinou metodu - tabulku reálných mocnin dvojky. Výsledkem je bohužel to, že můj matematicky přesný algoritmus dává lehce odlišné výsledky než ty, se kterými pracuje FT2 a další programy. Dle definice XM formátu je však můj algoritmus jednoznačně lepší.

A.7 Amiga frequency table

$$Period = PeriodTab\left[8 \cdot (Note \ \mathbf{MOD}\ 12) + \frac{FineTune}{16}\right] \cdot \frac{16}{2^{Note/12}}$$

Perioda je interpolována pro jemnější hodnoty finetune, tzn. pokud $FineTune$ není dělitelné 16, provede se lineární interpolace pro dvě sousední hodnoty z PeriodTab.

$$Frequency = 8363 \cdot \frac{1712}{Period}$$

```
PeriodTab = Array[0..12*8-1] of Word = (
  907,900,894,887,881,875,868,862,856,850,844,838,832,826,820,814,
  808,802,796,791,785,779,774,768,762,757,752,746,741,736,730,725,
  720,715,709,704,699,694,689,684,678,675,670,665,660,655,651,646,
  640,636,632,628,623,619,614,610,604,601,597,592,588,584,580,575,
  570,567,563,559,555,551,547,543,538,535,532,528,524,520,516,513,
  508,505,502,498,494,491,487,484,480,477,474,470,467,463,460,457
);
```

Opět pro jistotu ještě tatáž deklarace v C++:

```
unsigned short PeriodTab[12*8];
```

Tyto komplikované výpočty samozřejmě nejsou nutné, pokud přehrávač používá klasické frekvenční algoritmy z Amigy. Jedná se totiž o totéž, zde pouze s rozšířením pro 8bitové finetune.

A.8 Standardní příkazy

Příkazy XM jsou rozšířením příkazů MODu, tzn. příkazy 0-F by měly být (dle možností) identické s MODem. Kompletní seznam příkazů je v tabulce.

značka	00	popis příkazu
0		nic / Arpeggio
1	⊗	Portamento up
2	⊗	Portamento down
3	⊗	Tone portamento
4	⊗	Vibrato
5	⊗	Tone portamento + Volume slide
6	⊗	Vibrato + Volume slide
7	⊗	Tremolo
8		Nastav panning
9		Nastav sample offset
A	⊗	Volume slide
B		Position jump
C		Nastav volume
D		Pattern break (ukončení patternu)
E1	⊗	Fine portamento up
E2	⊗	Fine portamento down
E3		Nastav glissando control
E4		Nastav vibrato control
E5		Nastav finetune
E6		Nastav začátek/konec loopu
E7		Nastav tremolo control
E9		Retrig note
EA	⊗	Fine volume slide up
EB	⊗	Fine volume slide down
EC		Note cut
ED		Note delay
EE		Pattern delay
F		Nastav speed/tempo (FT2 značí jako tempo/BPM)
G		Nastav global volume (0..64)
H	⊗	Global volume slide (stejné parametry jako volume slide)
K		Key off
L		Nastav envelope pozici
P	⊗	Panning slide (Pxy: x=right speed, y=left speed)
R	⊗	Multi retrig note
T		Tremor
X1	⊗	Extra fine portamento up
X2	⊗	Extra fine portamento down

⊗ = Opakuje 00. To znamená, že když je parametr 00, příkaz opakuje posledně zadanou hodnotu (podobně jako MOD nebo S3M).

A.9 Příkazy ve volume sloupku

Všechny efekty ve volume sloupku pracují stejně jako standardní efekty. Volume sloupek je interpretován před standardními efekty, takže zde uvedené příkazy mají při kolizi nižší váhu.

Hodnota	00	Význam
0		–
$10-$50		Nastaví hlasitost (hodnota-$10)
:		: :
$60-$6f		Volume slide down
$70-$7f		Volume slide up
$80-$8f		Fine volume slide down
$90-$9f		Fine volume slide up
$a0-$af	⊗	Set vibrato speed (sdílí paměť s normálním vibratem)
$b0-$bf	⊗	Vibrato (sdílí paměť s normálním vibratem)
$c0-$cf		Set panning
$d0-$df		Panning slide left
$e0-$ef		Panning slide right
$f0-$ff		Tone portamento

A.10 Další poznámky

1. Fadeout je aktivní jen se zapnutou volume envelope. Pokud je volume envelope vypnutá, po příkazu **key off** se nota ihned vypne. (Rozdíl oproti IT.)

2. Příkaz Pxy má opačné pořadí argumentů než IT (zde je levý parametr „right“ a pravý parametr je „left“ - možná proto to později v IT udělali naopak).

3. Hodnoty v hlavičce nástroje označené jako *Volume type* a *Panning type* mají oproti IT přehozený bit 1 a 2 (loop a sustain loop).

Toto by mělo být zhruba vše. Pro plné pochopení je samozřejmě také nutná znalost formátů MOD [7] a S3M [13].

Aley Keprt, 1996
2.rozšířené vydání v roce 2001
aley@atlas.cz

B Komentovaný popis hudebních modulů S3M

4.přepracované vydání

Tento dokument je z části překladem originálního technického manuálu ke Scream Trackeru 3.20/3.21 a z části vlastním dílem. Dopsané části se nemusí nutně shodovat s názory a postoji autorů originálního technického manuálu. Současně byl upraven design celého balíku. Toto vydání obsahuje také některé doplňující informace, které v původním anglickém manuálu nejsou. Poznámky k poslednímu vydání najdete v samostatné sekci na konci textu.

První tabulka popisuje hlavičku S3M souboru. Všechny ostatní bloky jsou adresovány pomocí **parapointerů** v hlavičce, takže teoreticky mohou být kdekoliv v souboru. V praxi se však používá toto pořadí:

1. Hlavička (hlavní a rozšířená)
2. Hlavičky nástrojů po řadě - nástroje jsou ve smyslu S3M dvou typů:
   a) Hlavičky samplů
   b) AdLib nástroje (celé)
3. Patterny po řadě (celé, pakované)
4. Těla samplů po řadě

Následuje popis hlavičky nástroje a samplu. Každý nástroj a sampl má svou hlavičku, která se vždy ukládá do S3M a je také na začátku každého souboru s nástrojem nebo samplem, který uložíte na disk ze Scream Trackeru. Naprosto stejnou hlavičku používá i Advance Digiplayer (editor zvuku od Future Crew).

Parapointery (tzv. parapy) jsou adresy vztahované k začátku souboru a jsou to vlastně segmentové ukazatele pro MS-DOS. Absolutní adresu tedy zjistíte, když parap vynásobíte šestnácti. Toto řešení je použito proto, aby celá skladba (včetně samplů, případně i bez nich) mohla být nahrána do paměti jako jeden dlouhý blok. Z toho také vyplývá, že téměř každý S3M soubor obsahuje nějaké nevyužité bajty všude tam, kde „něco“ končí a další data pokračují až na hranici dalšího segmentu.

B.1 Hlavička S3M souboru

Hlavička je vždy na samotném začátku souboru. Jsou zde všechna důležitá data a ukazatele (parapy) na další bloky (samply, nástroje, patterny). Současně slouží k identifikaci, že jde právě o S3M soubor. Pro identifikaci načtěte 30h bajtů a testujte text "SCRM" na offsetu 2ch. Podobně se testují i soubory se samply a

nástroji, pouze musíte testovat příslušné identifikační texty na offsetech 4ch (viz níže).

atribut	popis
char name[28]	Jméno skladby (ASCIIZ)
byte _term	ASCII terminátor (znak $1A)
byte typ	Typ souboru (vždy 16)
word _x1	–nevyužito–
word ordnum	Délka orderlistu (vždy sudé číslo)
word samples	Počet samplů a/nebo AdLib nástrojů
word patts	Počet patternů
word flags	Flagy (význam bitů závisí na proměnné ffi): [ffi=1] bit 0: ST2 vibrato [ffi=1] bit 1: ST2 tempo [ffi=1] bit 2: Amiga slides [ffi=2] bit 3: 0vol optimizations (viz níže) [ffi=2] bit 4: Amiga limits (viz níže) [ffi=1] bit 5: Povoluje filter/sfx na SoundBlasteru [ffi=2] bit 6: ST3 volume slide (viz níže) [ffi=2] bit 7: Speciální data v S3M souboru (viz níže)
word cwtv	Název a číslo verze hudebního editoru Horní 4 bity určují editor, dolních 12 bitů je číslo verze. např. Scream Tracker 3.21 má číslo 0x1321 pozn. ST3.00: Volume slide jede na každém snímku.
word ffi	Formát souboru - existují jen dva: 1 = starý formát (ST3.00): znaménkové samply 2 = nový formát (ST3.01+): neznaménkové samply
char sign[4]	text "SCRM" (značka S3M souboru)
byte gv	Global volume
byte is	Počáteční rychlost (initial speed - příkaz A)
byte it	Počáteční tempo (initial tempo - příkaz T)
byte mv	Master volume a mono/stereo flag: bit 0-6: Master volume bit 7: 0=mono, 1=stereo
byte uc	Ultra click removal (viz níže)
byte dp	Nastavení channel pan: 252 = V souboru je uloženo nastavení panningu kanálů jinak se použije klasické rozložení LRLR
byte _x2[8]	–nevyužito–
word special	Pointer na speciální uživatelská data (viz níže)
byte chanset[32]	Počáteční nastavení kanálů (viz níže)

Toto je tedy první část hlavičky. Zbytek hlavičky je nutno nahrávat samostatně,

protože teprve po nahrátí první části hlavičky se dozvíme velikost ostatních částí.

atribut	popis
byte orderlist[X]	Orderlist, X=ordnum (viz níže)
word inspos[X]	Pozice samplů/nástrojů v souboru (parapy), X=insnum
word patpos[X]	Pozice patternů v souboru (parapy), X=patnum
byte chanpan[32]	Počáteční nastavení panningu (viz níže)

0vol optimizations: Automaticky vypne loop u not, jejichž hlasitost je nulová déle než dva řádky. (Šetří čas procesoru při přehrávání.)

Amiga limits: Nepřipustí žádné noty mimo limit standardního formátu MOD (C-3 až B-5). To znamená, že glissando nahoru (příkaz 2) se zastaví na B-5 apod. Tato volba má také vliv na některé další detaily ohledně kompatibility s Amigou.

ST3 volume slide: Volume slide na Amize je standardně prováděno od druhého snímku každého řádku. Pokud je zapnutá tato volba, provádí se volume slide také před prvním snímkem. Tato volba se automaticky nastavuje, když `Cwt/v` = 1300h.

Speciální data: Flag povoluje speciální uživatelská data v S3M souboru. Ukazatel na speciální data je v proměnné `Special`. ST3 speciální data nepoužívá a nepodporuje.

Global volume: Hodnota Global volume přímo dělí hlasitost při hraní. Takže když je nota s hlasitostí 48 a global volume je 32, výsledná hlasitost je 24. Toto zasahuje Sound Blaster i Gravis Ultrasound.

$$FinalVolume = NoteVolume \cdot \frac{GlobalVolume}{64}$$

Pokud ještě stále nemáte jasno, co to je parap, zde je další vysvětlení: Parap (parapointer) k offsetu Y od začátku souboru se spočítá jako (Y-offset hlavičky souboru)/16. Parapy si můžete představit jako relativní segmenty vzhledem k začátku S3M souboru. Jinak řečeno PARAP=OFFSET/16.

Master volume: Master volume se týká jen Sound Blasteru, přesněji řečeno všech softwarově mixovaných karet. Udává úroveň násobení samplu (viz část o mixování S3M). Větší hodnota zesílí výstupní hlasitost a tím zvýší kvalitu. Nicméně, pokud je hodnota příliš vysoká, mixovací program nedokáže nacpat nejhlasitější pasáže do šířky osmi bitů. Základní hodnota pracuje velmi dobře. Pamatujte, že při stereo přehrávání je všechno ještě násobeno 11/8, protože je k dispozici ještě další (devátý) bit pro přehrávání.

Ultra click removal: ST3 používá `uc` kanálů na kartě Gravis Ultrasound, takže `uc`/2 kanálů hraje bez jakéhokoliv „cvakání“. Je-li kanálů více, mohou se někdy „cvakání“ objevit. Číslo, které se zobrazuje v *order page* ve Scream Trackeru je `uc`/2.

Chanset: Každý kanál má v této tabulce jeden bajt. Význam bajtu závisí na tom, zda jde o samplový kanál nebo o AdLib kanál. Tyto dva typy jednoduše rozlišíte podle bitu 4.

Digitální (samplový) kanál:

bit 7:	0=kanál otevřen/používán
	1=kanál nepoužit
bit 6-5:	–nepoužito– (0)
bit 4:	0 (rozeznávací bit, že jde o digitální kanál)
bit 3:	0=vlevo
	1=vpravo (určení stereo zvuku)
bit 2-0:	číslo kanálu 1-8 (zde v podobě 0-7)

AdLib kanál:

bit 7:	0=kanál otevřen/používán
	1=kanál nepoužit
bit 6-5:	–nepoužito– (0)
bit 4:	1 (rozeznávací bit, že jde o AdLib kanál)
bit 3-0:	číslo kanálu AdLib (9 kanálů + 5 bicích)

Orderlist: Orderlist určuje pořadí, ve kterém se mají přehrávat patterny. Je to totéž jako sada sequencí v ProTrackeru. V S3M ovšem může orderlist obsahovat i tyto speciální bajty:
255 (– –) znamená konec hudby
254 (++) rezervováno (přeskakuje se - možno odstranit hned v loaderu)

Chanpan: Tato nepovinná součást hlavičky určuje 4bitový panning kanálů. ST3 podporuje nastavování panningu jen na kartách Gravis Ultrasound, ovšem ostatní programy takové omezení nemají.

bit 7-6:	–nepoužito– (0)
bit 5:	0=použít standardní nastavení
	1=nastavit podle bitů 3-0
bit 3-0:	Nastavení panningu (3=vlevo, 7=uprostřed, 12=vpravo)

B.2 Formát Digiplayer/ST3 samplu

Samply jsou uloženy ve formátu S3S. Tento formát je používán jak pro samostatné soubory se samply pro ST3, tak pro samply uvnitř S3M modulu.

atribut	popis
byte type	Vždy 1 (pro rozpoznání od AdLib nástroje)
char fname[13]	Jméno souboru pro DOS (8.3 ASCIIZ)
word memseg	Parapointer na tělo samplu
dword length	Délka samplu
dword LoopBeg	Začátek loopu
dword LoopEnd	Konec loopu
byte volume	Hlasitost (0-64)
byte _x1_	–nevyužito– (0)
byte pack	Typ kódování samplu: 0=PCM data 1=DP30ADPCM (ST3 nepodporuje)
byte flags	Flagy: bit 2: 1=16bit sampl (ST3 nepodporuje) 0=8bit sampl bit 1: 1=stereo (ST3 nepodporuje) 0=mono bit 0: zapíná loop (opakování)
dword c2spd	Frekvence bázové noty
byte _x2_[12]	–nevyužito– (0)
char name[28]	Jméno samplu (ASCIIZ)
char id[4]	identifikátor "SCRS"

Stereo: Pokud je sampl stereo, po bajtech pro levý kanál následuje stejný počet bajtů pro pravý kanál. Celková délka samplu je potom Length*2 bajtů. Stereo samply však ST3 ani žádný jiný hudební editor nepodporuje.

Poznámka: ST3 vůbec nepodporuje samply 16bitové, stereo nebo snad ADPCM. Podobně Length, LoopBegin, LoopEnd a C2Spd jsou 32bitové parametry, ačkoliv ST3 akceptuje jen samply do 64000 bajtů a 16bitové frekvence. Větší soubory jsou v ST3 ukrojeny právě na tuto velikost při nahrávání z disku. Pamatujte také, že LoopEnd ukazuje jeden bajt ZA KONEC samplu.

Souborové formáty S3S a S3M jako takové však tyto věci podporují a některé další editory a přehrávače také. Velmi doporučuji, aby všechny nové přehrávače podporovaly všechny vlastnosti (snad kromě stereo a ADPCM samplů, které nejsou tak snadno aplikovatelné).

B.3 Formát pakovaných patternů

Zapakovaný formát je ve skutečnosti jistá forma jazyka. V souboru je uložen nejprve word (2 bajty) určující délku zapakovaných dat a potom následují samotná data. Jak tedy postupovat při rozpakovávání patternu:

1. Přečti jeden bajt. Když je nulový, tento řádek je hotový a jde se na další (celkem 64 řádků). Znovu přečti 1 bajt a tak pořád dokola. Pokud není nulový, jdi dál.

2. Dolních 5 bitů (AND 31) je číslo příslušného kanálu.

3. Když je bit 5 nastaven, přečti číslo noty a číslo nástroje/samplu (celkem 2 bajty).

4. Když je bit 6 nastaven, přečti hlasitost (1 bajt).

5. Když je bit 7 nastaven, přečti příkaz a jeho parametr (2 bajty).

6. Zpět k bodu 1, celkem 64 průchodů = 64 řádků v každém patternu.

Informace o příkazech a jejich přehrávání najdete v manuálu ST3.

B.4 C2SPD a výpočet frekvencí v ST3

Jemné ladění (C2SPD) je ve skutečnosti frekvence noty C-4. Název je zkratkou **C-2 Speed** a pochází ze Scream Trackeru 2, který měl jen tři oktávy a tudíž bylo střední C-2. Formát S3M používá 8 oktáv, proto je střední notou C-4. Takže by se C2SPD mělo vlastně jmenovat C4SPD...

Tabulka period použitá ve Scream Trackeru III:

```
nota:     C    C#   D    D#   E    F    F#   G    G#   A    A#  B
perioda: 1712,1616,1524,1440,1356,1280,1208,1140,1076,1016,960,907
```

Střední oktáva je 4. Periody ostatních oktáv se spočítají jednoduchým násobením nebo dělením těchto čísel dvěma. Vyšší oktávy mají kratší periody, nižší oktávy mají delší periody[28]. Zde je několik jednoduchých vzorečků podle originální technické dokumentace k St3.

$$note\_st3period = 8363 \cdot \frac{perioda >> (oktáva - 4)}{C2SPD}$$

$$note\_amigaperiod = \frac{note\_st3period}{4}$$

$$note\_hertz = \frac{14317056}{note\_st3period}$$

$$note\_hertz = 65536 \cdot \frac{218.46}{note\_st3period}$$

[28] $perioda = frekvence^{-1}$

Poznámka: Číslo 218.46 udává frekvenci středního C v Hertzích.

Všimněte si, že ST3 používá periody čtyřikrát jemnější než původní MOD. To dovoluje daleko jemnější glissando (portamento), které je nyní čtyřikrát jemnější než na Amize.

B.5 Poznámka ke 4.vydání

Kromě rozsáhlých obsahových i typografických úprav doznalo toto vydání ještě další důležité změny. Vzhledem k rychlému vývoji na poli hudebního softwaru i hardwaru byly zcela vypuštěny kapitoly, které již v dnešní době nemají žádný význam. Pokud vám však přesto bude něco chybět, hledané informace lze nalézt ve starších vydáních nebo přímo v originální dokumentaci Scream Trackeru. Zde uvádím alespoň stručný přehled vynechaných kapitol:

AdLib nástroje: Kapitola popisující souborový formát nástrojů AdLib. Tyto nástroje nesouvisejí s digitální hudbou a nepodporuje je žádný program kromě originálního ST3. Jejich popis je tedy zcela bezpředmětný.

Stmik300old: Popis mutace S3M formátu pro SoundBlaster, kdy je na disk ukládán balast za každým samplem. Nemá žádný praktický význam.

STImport: Popis formátu STI sloužícího pro snadnější import souborů z jiných formátů. Autoři předpokládali, že bude jednodušší onen „jiný“ formát zkonvertovat do STI než přímo do S3M. Tento formát se však nikdy neujal.

SimplexFormat: Jednoduchý exportní formát obsahující jen surový tok not.

Formát patternů v paměti: Bezvýznamný popis interního formátu patternů v ST3.

Mixování samplů a postprocessing: Přesný, ale velmi komplikovaný popis mixovacího algoritmu ST3, který je založen na předpočítaných tabulkách, a proto je velmi rychlý, ale nekvalitní.

Aley Keprt, 1995,1996,1997,2001
4.přepracované vydání v roce 2001
aley@atlas.cz

Resumé

The main goal of this diploma work is to find origins of unwanted noise in computer-rendered music and to specify methods of its suppression or elimination. The key part of this work is the very practical application of several pieces of knowledge of numerical mathematics, universal algebra, and computer geometry to musical and sound data. Besides the study of noise elimination itself, this work has brought some other interesting secondary products, as the rendering of music itself, its algebraic formalization, analysis of situation in computer music file formats, or detailed study of interpolation methods and other parameters with influence to the quality of computer-rendered music. Besides this, I have also found a simple and practical software-only algorithm of enhancing bass frequencies in sound. The work also brought wide range of theoretical theorems and their proofs.

Literatura

[1] Jiří Kobza: *Numerické metody.* Vydavatelství Univerzity Palackého, Olomouc, 1993

[2] Josef Drdla: *Geometrické modelování křivek a ploch.* Josef Drdla, Olomouc, 2001

[3] Pavel Horák: *Algebra a teoretická aritmetika I.* Masarykova Univerzita, Brno, 1991

[4] Miroslav Mašláň, David Žák: *Analogové obvody.* Vydavatelství Univerzity Palackého, Olomouc, 1995

[5] Jamal Hannah: *The Computer Music File Formats Collection 1.0.* Jamal Hannah, 1994

[6] FH a kolektiv: *Music Format Description Library 2.0.* Contortion, 1996 *http://www.drp.fmph.uniba.sk/~fh*

[7] Kurt Kenett: *Modfil 1.0.*

[8] Firelight: *Module Effects Description.* FireModDoc

[9] Andrew Scott: *Noisetracker/Soundtracker/Protracker Module Format - 3rd revision.*

[10] Stefan Danes: *IBM "WOW" Format.*

[11] Björn Wesen: *Amiga "StarTrekker" MOD format.*

[12] Jamal Hannah: *Take Tracker File Format.* 1994

[13] Aleš Keprt: *Scream Tracker 3.20/3.21 File Formats And Mixing Info.* Aleš Keprt, 1997

[14] Aleš Keprt: *Scream Tracker 3.21 Command List.* Aleš Keprt, 1995

[15] Aleš Keprt: *Popis formátu XM modulů pro XM verze $0104.* Aleš Keprt, 1997

[16] Fredrik Huss: *The XM module format description.* Triton, 1996

[17] Jeffrey Lim: *The technical specs for the IT format.* Jeffrey Lim, 1997

[18] Aleš Keprt: *Formát WAV (Windows Wave) - popis 1.30.* Aleš Keprt, 1998

[19] Aleš Keprt: *ALM File Format Documentation - updated version 1.20.* Wotsit's Format, 1998
http://www.wotsit.org

[20] Aleš Keprt: *ATM - A Tracker Module - version 0.82.* Aleš Keprt, 1999

[21] Daniel Goldstein: *Multitracker Module Editor 1.01β.* Renaissance, 1993

[22] Stefan Danes: *669 And Extended 669 File Format.* Intertia

[23] Sami Tammilehto: *Scream Tracker 2.24 - Technical Information.* Future Crew, 1990

[24] *Standard MIDI-File Format Specification 1.1.* The International MIDI Association

[25] Miloš Tesař: *MIDI - Komunikace v hudbě.* Miloš Tesař, 1993

[26] Magnus Högdahl, Fredrik Huss: *Fast Tracker 2.08.* Triton Productions, 1998
http://www.starbreeze.com

[27] Urban Jonsson: *Fast Tracker 2 Manual.* Triton Productions and Under World Digital Publishing, 1996

[28] Sami Tammilehto: *Scream Tracker 3.21.* Future Crew, 1994

[29] Kalle Kaivola: *Scream Tracker 3.2 User's Manual.* Electromotive Force, 1995

[30] Jeffrey Lim: *Impulse Tracker 2.14.* Jeffrey Lim, 1998
http://www.citenet.net/noise/it

[31] Petteri Kangaslampi, Jarno Paananen: *Midas Digital Audio System 1.1.2.* Housemarque, 1999
http://www.s2.org/midas

[32] Jean-Paul Mikkers a kolektiv: *MikMod 2.10.* Hardcode, 1995

[33] Jean-Paul Mikkers: *MikIT 0.90.* Jean-Paul Mikkers, 1997
http://www.stack.nl/~mikmak

[34] Sami Tammilehto: *Advanced DigiPlayer 3.0.* Future Crew, 1991

[35] Kolektiv autorů: *Reality Adlib Tracker 1.1a.* Reality Productions, 1995

[36] Jean-Paul Mikkers a kolektiv: *Extended MikMod 1.98a.* Hardcode 1996
http://www.dragonfire.net/~TSSF/players