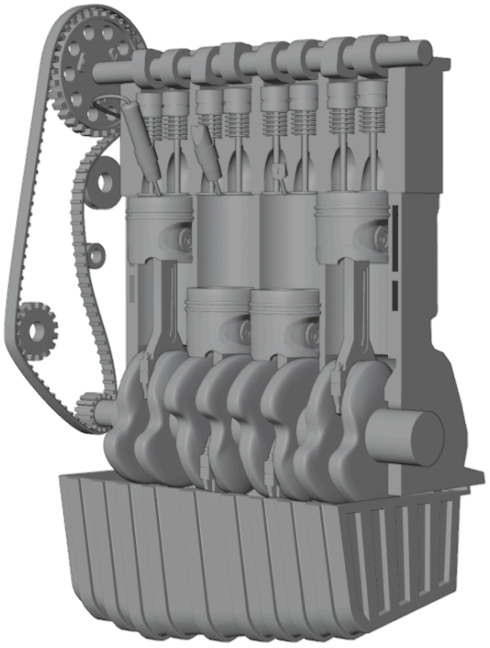
Gymnázium, Praha 6, Arabská 14
předmět Programování, vyučující Ing. Daniel Kahoun



Adess – umělecky dirigovaný syntetizér zvuku motorů

Gymnázium, Praha 6, Arabská 14
tel.: 235 351 708
fax.: 222 262 066
e-mail: ga@gyarab.cz
www.gyarab.cz

Zadání ročníkového projektu

Název: **Adess – umělecky dirigovaný syntetizér zvuků motorů**

Řešitel: **Kubota Leon, 4.E**

Vedoucí práce: **Ing. Daniel Kahoun**

Datum odevzdání: **28. 2. 2026**

Způsob zpracování a kritéria hodnocení:

Zpracování v požadovaném rozsahu se řídí obecně závaznými pokyny zpracování ročníkových projektů. Řešitel elektronicky odevzdáve stanoveném termínu dokumentaci, prezentaci, poster a další vyžádané přílohy (např. zdrojové kódy, ukázková data). Před obhajobou řešitel odevzdá jeden výtisk stejné dokumentace s podepsaným prohlášením o autorství a jeden poster, neurčí-li vedoucí jinak. Hodnotí se odborné zpracování úlohy, použití návrhových vzorů, prezentace při obhajobě a funkcionalita produktu.

Popis (povinná část):

Vytvořte konzolovou aplikaci pro procedurální syntézu zvuků motorů určenou pro film a animaci. Program generuje zvukové vzorky na základě uživatelské konfigurace a klíčových snímků definovaných ve vlastním datovém formátu *DST*.

Upřesnění zadání:

- Implementace parseru konfiguračních souborů *adess*
- Procedurální generování pole vzorků zvuku motoru
- Práce s klíčovými snímky pro změnu zvuku v čase
- Export *audia* do standardního formátu *WAV*
- Ovládání aplikace pomocí příkazové řádky (*CLI*)

Bonus (nepovinná část):

- Podpora pro různé typy motorů (vrtulové, spalovací)
- Přidání zvukových efektů (např. ozvěna, zkreslení)
- Grafické znázornění generované zvukové vlny v *CLI*
- Možnost reálného náhledu (přehrání) před uložením

Platforma:

- C

datum podpisu	podpis řešitele

Anotace

Adess je aplikace přístupná v příkazové řádce, která procedurálně generuje zvuky spalovacích motorů pro využití v animaci a filmu. Generace je plně ovladatelná uživatelem prostřednictvím nastavovacích souborů *adess* ve vlastním „jazyce“ *DST* (datová ukládací věc). V těchto souborech jsou uloženy důležité hodnoty o formátu a charakteristice požadovaného zvuku a klíčové snímky, které určují jeho přeměnu v čase. Hodnoty z těchto souborů jsou načteny do paměti a následovně využity pro procedurální paralelní syntézu zvuků motorů. Výstupem této generace je pole vzorků, které se uloží do souboru *WAV*.

Abstract

Adess is an application that runs in the terminal, it procedurally generates the sound of combustion engines for use in animation and film production. The generation is fully customizable by the user through „adess“ configuration files in a proprietary DST „language“ (data storage thing). These files contain important values about the format and characteristics of the desired sound and keyframes, that determine the sounds change throughout time. The values from these files are loaded into memory and subsequently used for procedural parallel synthetization of engine sounds. The output of this generation is a buffer of samples, which are exported into a *WAV* file.

Čestné prohlášení

Prohlašuji, že jsem jediným autorem tohoto projektu, všechny citace jsou řádně označené a všechna použitá literatura a další zdroje jsou v práci uvedené. Tímto dle zákona 121/2000 Sb. (tzv. Autorský zákon) ve znění pozdějších předpisů uděluji bezúplatně škole Gymnázium, Praha 6, Arabská 14 oprávnění k výkonu práva na rozmnožování díla (§ 13) a práva na sdělování díla veřejnosti (§ 18) na dobu časově neomezenou a bez omezení územního rozsahu.

datum podpisu	podpis řešitele

Obsah

1 Úvod

Tvorba zvuku motorů pro využití ve animaci je velice složitá, konvenčním způsobem je nahrát zvukové stopy skutečného motoru a pomocí komplexních digitálních manipulací získat finálnú zvuk. Takováto tvorba zvuku je velice časově náročná, umožňuje však precizní úpravu zvuku pro tvorbu působivých výsledků. Tento způsob je však zcela nevyhovující menším studiům či jednotlivcům, kteří nedisponují stovky hodin a neovládají tuto tvůrčí disciplínu. [1]

Druhým postupem je zvukovou stopu generovat, toho lze docílit simulací tlakových vln v motoru nebo syntézou, tedy tvorbou zvuku pomocí matematických algoritmů. V této práci využívám pro generaci zvuku spalovacích motorů syntetický přístup.

2 Použité technologie

Celý projekt je napsán v programovacím jazyce *C* (konkrétně *C99*) pro jeho výpočetní účinnost a možnost bližšího kontaktu s *hardwarem*. Tento jazyk jsem zvolil také kvůli svému zájmu o studium letectví a kosmonautiky, kde se hojně využívá pro programování *embedded* systémů.

Pro zkompilování jsem využil *CMake*, který podporuje mnoho *compilerů*. Osobně jsem zvolil běžně používaný *compiler GCC* (*GNU Compiler Collection*). [2, 3]

Již zmíněné technologie jsou vše, co je nutné pro zkompilování a spuštění projektu (uvažuji, že standardní knihovny uživatel má). Při psaní zdrojového kódu jsem však využil několika dalších nástrojů:

- `Sonic Visualiser` – *open-source* nástroj pro analýzu zvuku, umožňuje spektrální a vlnovou analýzu. [4]
- `Spek` – jednodušší, zato rychlejší a uživatelsky přívětivější nástroj pro analýzu zvuku. [5]
- `Valgrind` – nástroj běžící v příkazové řádce pro analýzu paměti programu za běhu, je velice užitečný pro zamezení únikům paměti

3 Zvuk

Abychom dokázaji zvuk tvořit, je potřeba mu alespoň povrchově porozumět. Jako zvuk označujeme vlnění částic vzduchu. Čím rychleji tyto částice kmitají (jejich frekvence), tím vyšší tón slyšíme. Čím větší je amplituda vlnění, tím hlasitější zvuk slyšíme. Jako lidé slyšíme zvuky o frekvencích od 20 až 20000 Hz a hlasitosti alespoň 0 dB. [6]

3.1 Digitální zvuk

Zvuk je ukládán jako pole vzorků, které určují, v jaké poloze se má nacházet oscilátor v reproduktoru. V tomto projektu je využit formát *WAV*.

V jednotlivých vzorcích je tedy uložena amplituda v jednotlivém čase. Datový typ vzorku se liší podle rozlišení vzorků (anglicky *bit depth*), běžně nabývá hodnot 8, 16, 24 a 32. Má aplikace dokáže exportovat v těchto *bit ratech* s využitím datových typů `uint8_t` pro *8-bit* audio, `int16_t` pro *16-bit* audio a `float` pro *32-bit* audio. Pro *24-bit* audio jsem zvolil zabalení do tří `uint8_t` místo jednoho `int32_t` s nulovým vycpáváním.

Sample rate označuje počet vzorků zaznamenaných na každou sekundu. Obvykle se používá hodnota 44100, jelikož s ní dokážeme zaznamenat celé spektrum lidsky slyšitelného zvuku. Při příliš nízkých *sample ratech* dochází k *aliasování*, což je nežádoucí artefakt plynoucí z příliš vysoké frekvence. Tato konkrétní frekvence, při níž dochází k *aliasování*, se nazývá *Nyquistova*. Je rovna polovině *sample ratu*. V mé aplikaci si uživatel může zvolit libovolný *sample rate*.

4 Ovládání aplikace *Adess*

Aplikace *Adess* se ovládá skrze příkazy v příkazové řádce a nastavovací soubory. V Tab. 1 jsou popsány příkazy dostupné v aplikaci *Adess*.

Příkaz	**Přepínače**	**Popis**
`adess help`	bez přepínačů	Zobrazí uživateli možnosti aplikace *Adess*.
`adess make_project`	`-h`, `-n`, `-d`, `-e`	Vytvoří adresář projektu a naplní jej soubory.
`adess make_scene`	`-h`, `-n`, `-d`, `-e`	Vytvoří soubor scény.
`adess make_engine`	`-h`, `-n`, `-d`, `-e`	Vytvoří soubor motoru.
`adess render`	`-n`, `-a`, `-p`	Vypočítá zvuk a uloží zvukovou stopu.
`adess guide`	bez přepínačů	Zobrazí uživateli podrobnější dokumentaci.

Tab. 1 Přehled příkazů aplikace *Adess*

V Tab. 2 jsou popsány přepínače.

Přepínač	**Název**	**Popis**
`-h`	*help*	Zobrazí možnosti příkazu (alternativa `adess help <příkaz>`).
`-n`	*name*	Nastaví jméno (projektu, scény, motoru či exportovaného souboru).
`-d`	*directory*	Nastaví adresář pro vytvoření (projektu, scény či motoru)
`-e`	*empty*	Vytvoří prázdný projekt, scénu či motor (bez továrních hodnot).
`-a`	*all*	Vytvoří zvukovou stopu všech dostupných scén.
`-p`	*preview*	Přeskočí *post-processing* fázi

Tab. 2 Přehled přepínačů aplikace *Adess*

5 Syntax souborů *Adess*

Syntax souborů *Adess* je velice jednoduchý, jde o pouze o

6 Ovládací soubory

Ovládací soubory aplikace *Adess* jsou rozděleny do tří kategorií: projektový soubor obsahuje parametry výsledného souboru a informace o adresách dalších souborů, je popsán v Tab. 3.

Název	**Typ**	**Popis**
`sample_rate`	`integer`	Vzorkovací frekvence výsledného souboru.
`bit_depth`	`integer`	Rozlišení vzorku (*8*, *16*, *24* či *32 bitů*).
`engine_path`	`string`	Adresa adresáře obsahující soubory motorů.
`scene_path`	`string`	Adresa adresáře obsahující soubory scén.
`output_path`	`string`	Adresa adresáře obsahující výsledné soubory.
`seed`	`integer`	Počáteční hodnota generátoru náhodných čísel.

Tab. 3 Hodnoty v projektovém souboru

Soubory scén obsahují informaci o scéně. Možné hodnoty jsou zapsány v Tab. 4.

Název	**Typ**	**Popis**
`length`	`float`	Délka scény v sekundách.
`engine`	`string`	Název motoru použitého ve scéně.
`keyframes`	`keyframe`	Klíčové snímky obsahující informace o času, okamžitých otáčkách a okamžité zátěži.

Tab. 4 Hodnoty v souboru scén

Nejvíce parametrů obsahuje soubor motoru, jsou popsány v Tab. 5.

Název	**Typ**	**Popis**
`stroke`	`integer`	počet dob v jednom cyklu
`cylinder_count`	`integer`	počet válců
`idle_rpm`	`integer`	otáčky ve volnoběhu
`max_rpm`	`integer`	maximální otáčky
`valvetrain_timing_offset`	`float`	časový posun výfukových ventilů
`low_frequency_noise_frequency`	`float`	frekvence nízkofrekvenčního šumu
`low_frequency_noise_falloff`	`integer`	pokles nízkofrekvenčního šumu
`low_frequency_noise_strength`	`float`	síla nízkofrekvenčního šumu
`harmonics`	`integer`	počet harmonických frekvencí
`base_volume`	`float`	hlasitost základní zvukové stopy
`valvetrain_volume`	`float`	hlasitost zvuku ventilů
`minimum_volume`	`float`	minimální hlasitost
`rpm_volume_multiplier`	`float`	otáčkový násobitel hlasitosti
`load_volume_multiplier`	`float`	zátěžový násobitel hlasitosti
`minimum_noise`	`float`	minimální šum
`load_noise_multiplier`	`float`	zátěžový násobitel šumu

Tab. 5 Hodnoty v souboru scén

Tyto soubory jsou při zavolání příkazu `adess render` načteny do paměti a zpracovány, výsledný soubor bude uložen do adresáře určeného hodnotou `output_path` v projektovém souboru.

7 Postup *renderování* zvuku

Uživatel pomocí příkazu `adess render` se jménem scény jako argument spustí několikafázový proces *renderování*. Ten je podrobně popsán v následujících podkapitolách.

7.1 Čtení vstupních dat

Nejprve se otevře soubor projektu a jeho data se po kontrole syntaxe parsují do struktury `struct Project`. Poté se toto opakuje se scénou a nakonec se souborem motoru.

7.2 Předvýpočetní fáze

V této fázi se předvypočítají důležitá pole pro následující fáze. Tyto výpočty probíhají paralelně.

7.2.1 Interpolace klíčových snímků a z nich plynoucích hodnot

První vlákno lineárně interpoluje klíčové snímky pomocí rovnice (1), výstupem je pole frekvencí, pole otáček, pole fází, pole zátěže a pole násobitelů nízkofrekvenčního šumu.

$$h = h_0 + (t - t_0) \cdot \frac{h_1 - h_0}{t_1 - t_0} \tag{1}$$

h je okamžitá hodnota, h_0 a h_1 jsou hodnoty předchozího a následujícího klíčového snímku
t je čas, t_0 a t_1 jsou časy předchozího a následujícího klíčového snímku

Otáčky jsou vypočítány pomocí otáčkové rovnice (2).

$$ot = \frac{f \cdot 60}{n} / v \tag{2}$$

ot jsou otáčky [ot · s^{-1}]
f je frekvence [Hz]
n je inverzní počet pracovních dob za sekundu (2 pro čtyřdobé motory a 1 pro dvoudobé)
v je počet válců

Fázi vypočítáme pomocí rovnice (3).

$$\varphi_n = \sum_{i=0}^{n-1} \tau \cdot f[i] \cdot \Delta t \tag{3}$$

V programu vypočítáme rovnici (3) pomocí kódu uvedeného ve výpisu 1.

```
double phase = 0; // Musí být double, jelikož float neposkytuje dostatečnou
přesnost

while (i < scene->sampleCount) {
  phase += TAU * frequencyBuffer[i] * timeStep;
  phaseBuffer[i] = phase;
  i++;
}
```

Výpis 1 – Výpočet fáze

Násobitel nízkofrekvenčního šumu vypočítáme pomocí rovnice (4).

$$n = s \cdot \left(\frac{-ot^2 - 2 \cdot ot \cdot ot_v}{p} - \frac{ot_v}{p} + 1 \right) \tag{4}$$

n je násobitel nízkofrekvenčního šumu
s je síla šumu

ot a ot_v jsou okamžité otáčky a otáčky ve volnoběhu
p je pokles (vzdálenost od volnoběhu ve které je šum nulový, v otáčkách)

7.2.2 Generace stabilního hnědého šumu

Hnědý šum je šum, který je tvořen Brownovým pohybem, lze získat integrováním bílého šumu. V aplikaci je využit jako obecný zdroj náhodnosti ve většině funkcí, viz výpis 2. [7]

Pro generaci náhodných čísel je využit *32-bitový* `Xorshift`, díky tomuto algoritmu jsou náhodná čísla generována velice rychle a zároveň opakovatelně (dle počáteční hodnoty proměnné `state`). [8]

```
while (i < scene->sampleCount) {
  // Implementace algoritmu Xorshift ve 32-bitové verzi
  *state ^= *state << 13;
  *state ^= *state >> 17;
  *state ^= *state << 5;

  // Přepočítání na hodnoty -1.0 až 1.0
  lastBrown += ((*state / (double) UINT32_MAX) * 2.0f - 1.0f) * 0.02f;
  if (lastBrown > 1.0f) lastBrown = 1.0f;
  if (lastBrown < -1.0f) lastBrown = -1.0f;

  stableBrownNoiseBuffer[i] = lastBrown;

  i++;
}
```

Výpis 2 – Generace hnědého šumu

Hodnoty jsou dále stabilizovány a vyhlazeny pomocí zprůměrování vzorků dle Gaussova rozdělení. Díky tomuto kroku působí šum přirozeněji.

7.2.3 Generace růžového šumu

Tento šum je využit při generaci zvuku klapání ventilů, je velmi podobný bílému šumu, působí však přirozeněji a méně uměle.

Pro generaci růžového šumu jsem zvolil *Voss-McCartneyův*. Pro generaci náhodných čísel byl použit *32-bitový* `Xorshift`, který je však ve výpisu 3 vynechán.

```
while (i < scene->sampleCount) {
  // Implementace algoritmu Xorshift ve 32-bitové verzi (zde vynecháno)
  ...

  // Generace růžového šumu pomocí Voss-McCartenova algoritmu s 32 iteracemi
  sum = 0.0f;
  for (n = 0; n < 31; n++) {
    pinkBuffer[n] = (pinkBuffer[n] + randomFloat) * 0.5f;
    sum += pinkBuffer[n];
  }
  pinkNoiseBuffer[i] = sum * 0.03125f; // Vypočítání průměrné hodnoty

  i++;
}
```

Výpis 3 – Generace růžového šumu

7.2.4 Generace nízkofrekvenčního šumu

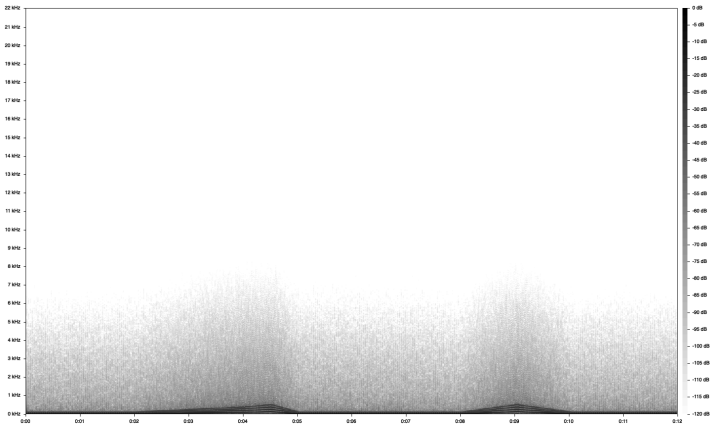
Nízkofrekvenční šum je využit při generaci základního zvuku motoru v otáčkách blízko otáčkám volnoběhu. Pro jeho generaci je také využit *32-bitový* `Xorshift`.

7.3 Výpočetní fáze

V této fázi jsou paralelně vypočteny vzorky jednotlivých zvukových stop. Konkrétně jde o vzorky základní zvukové stopy motoru a o vzorky zvuku klapání ventilů.

7.3.1 Základní zvuková stopa

Základní zvuková stopa odpovídá zvuku spalování v motoru. Vzorky stopy jsou vypočítány pomocí fáze, která již byla vypočítána v předvýpočetní fázi. Zvuková stopa je zachycena ve spektogramu 1.



Obr. 1 Spektogram základní zvukové stopy [Spek]

7.3.2 Zvuk klapání ventilů

Zvuk klapání ventilů je pro celkový zvuk motoru překvapivě důležitý, zejména při nízkých otáčkách. Pro výpočet zvuku ventilů je potřeba znát časy, kdy je vačková hřídel v kontaktu s ventily.

Každý válec má sací a výfukový ventil, každý z nich je otevřen jednu dobu každé dvě otáčky. Frekvenci otevření sacího ventilu lze vypočítat pomocí rovnice (5).

$$f_s = \frac{\frac{ot}{2} \cdot v}{8} \tag{5}$$

f_s je frekvence otevírání sacího ventilu [Hz]
ot jsou otáčky motoru [ot · s^{-1}]
v je počet válců

Abychom z této frekvence získali zvukovou stopu, využijeme modulace. Nejprve je zapotřebí přeměnit frekvenci na pulzy pomocí rovnice (6).

$$n = 10 \cdot |\sin(\varphi_s + 0,1 \cdot p)| - 9 \tag{6}$$

n je momentální hodnota vzorku modulační vlny
φ je fáze frekvence otevírání sacího ventilu
p je momentální hodnota růžového šumu

Tuto pulzovou vlnu posuneme o uživatelem zadanou hodnotu `valvetrain_timing_offset`, čímž vytvoříme pulzovou vlnu výfukových ventilů. Tyto vlny sečteme. Jako nosnou vlnu využijeme pilovou vlnu, ta je generována pomocí kódu ve výpisu 4.

```
valvetrainBuffer[i] = 2.0f * (fmod(phaseBuffer[i] * 2.0f, TAU) / TAU) - 1.0f;
```

Výpis 4 – Generace zvuku ventilů

Po modulaci nosné pilové vlny pulzovou vlnou získáme zvukovou stopu, která mimikuje klapání ventilů v motoru.

7.4 Kombinační fáze

V této fázi jsou zkombinovány stopy vzorků z minulé fáze na základě požadavků uživatele (`base_volume` a `valvetrain_volume`). Tato fáze je velice rychlá, jde pouze o násobení vzorků jejich hlasitostí a následné sečtení. V této fázi neprobíhá normalizace, při překročení maximálnní hlasitosti je uživatel je upozorněn.

7.5 *Post-processing* fáze

Tato část je velice složitá, a tak i časově náročná. Uživatel ji může přeskočit pomocí přepínače `-p` (*preview*).

Cílem této fáze je vyplnit vyšší frekvence smysluplnými daty. Toho docílíme tvorbou harmonických frekvencí kombinovaného pole frekvencí.

7.5.1 *Fourierova* transformace

Při výpočtu polí byla ztracena informace o frekvenci, potřebujeme ji získat pomocí *Fourierovy transformace* (viz rovnice 7). [9]

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x) \cdot e^{-i2\pi\xi x} \cdot \delta x, \quad \forall \xi \in \mathbb{R} \tag{7}$$

K jejímu výpočtu byl využit *Cooley-Tukeyovský* urychlený algoritmus, který využívá komplexních čísel a rekurze. Má adaptace tohoto algoritmu je ve výpisu 5. [10]

```
void fft(complex float *input, uint64_t n, complex float *temp) {
  if (n > 1) {
    uint64_t k, m;
    complex float w = 0.0f + (0.0f * I);
    complex float z = 0.0f + (0.0f * I);
    complex float *even = temp;
    complex float *odd = temp + (n / 2);

    for (k = 0; k < (n / 2); k++) {
      even[k] = input[2 * k];
      odd[k] = input[2 * k + 1];
    }

    // Rekurze
    fft(even, n / 2, input);
    fft(odd, n / 2, input);

    // Výpočet FFT
    for (m = 0; m < (n / 2); m++) {
      w = cos(TAU * m / (float) n) - (sin(TAU * m / (float) n) * I);
      z = w * odd[m];
      input[m] = even[m] + z;
      input[m + (n / 2)] = even[m] - z;
    }
  }
}
```

Výpis 5 – Adaptace algoritmu rychlého *Fourierova* algoritmu [10]

Podobně funguje i inverzní funke `inverseFastFourierTransform`.

Takto převádět zvukovou stopu do frekvenční dimenze lze pouze v případě, že frekvence zůstane konstantní.

7.5.2 Tónový posun

Pro zvukové stopy s proměnlivou frekvencí je třeba využít složitější algoritmus: pole rozdělíme na malá okna, která se vzájemně překrývají. Každé okno trvá pouze několik milisekund, cílem je získat okamžité frekvence v daném čase. Okna převedeme do frekvenční domény pomocí *FFT* algoritmu (jako velikost oken je proto vhodné zvolit mocninu dvou). Tato okna složíme zpět do zvukové stopy, přičemž roztáhneme mezery mezi nimi (dle faktoru tónového posunu).

7.5.3 Hanningovo okno

Při spojování oken mezi nimi vznikají prudké přechody (vlivem nesynchronyzovaných fází), které se ve zvuku projeví jako bzučení. Tento jev se nazývá *spektrální přeliv*, lze mu částečně předejít použitím *Hanningova okna*, které je definováno rovnicí (8). [11]

$$w_{H(n)} = \begin{cases} 0,5[1 - \cos(\tau \frac{n}{N})]; & 0 \leq n = N - 1 \\ 0 \quad \text{jinak} \end{cases} \tag{8}$$

Tato rovnice je v kódu využita pro předvypočítání *Hanningova okna*, viz výpis 6.

```
while (i < windowSize) {
  hanning[i] = 0.5f * (1.0f - cos((TAU * i) / ((float) windowSize - 1.0f)));
  i++;
}
```

Výpis 6 – Implementace *Hanningova okna*

Přestože bylo využito *Hanningovo okno*, *spektrální přeliv* přetrvává. K jeho eliminaci se využívají složitější algoritmy, jako například ——. V současnosti se pro synchronizaci fází zkoumá využití umělé inteligence.

7.6 Zapisovací fáze

Data z předchozí fáze se zapíší do výsledného souboru `WAV`. Předtím se však vzorky musí převést do správného datového formátu, k tomu využijeme funkci `convert`, který využívá ukazatel typu `void` pro generičnost. Převod z typu `float` na typ `uint8_t` (pro *8 bitové audio*) nebo `int16_t` (pro *16 bitové audio*) je triviální, pro export *32 bitového* audia není převod potřeba.

Nejobtížnějším převodem je převod pro export *24 bitového audia*, jelikož pro jeho zápis jsou využity 3 čísla typu `uint8_t` (součet *bitů* je 24). Toho docílíme pomocí kódu ve výpisu 7.

```
uint8_t *buffer = (uint8_t *) voidBuffer; // 3x delší

uint64_t k = 0;
int32_t currentSample = 0;

while (i < sampleCount) {
  currentSample = (floatBuffer[i] * 8388607); // 8_388_607 = 2^24 / 2

  // Rozdělení 24 bitového čísla do tří 8 bitových čísel
  buffer[k] = (uint8_t) ((currentSample) & 0xFF);
  buffer[k + 1] = (uint8_t) ((currentSample >> 8) & 0xFF);
  buffer[k + 2] = (uint8_t) ((currentSample >> 16) & 0xFF);

  i++;
  k += 3;
}
```

Výpis 7 – Převod pole typu `float` na trojnásobně dlouhé pole typu `uint8_t`

7.6.1 Anatomie souboru `WAV`

Soubor `WAV` se skládá z hlavičky, ve které jsou uloženy metadata, a z pole vzorků. Hodnoty potřebné v hlavičce jsou v tabulce 6 [12].

	Název	Délka	Hodnota
RIFF	`ChunkID`	4	Řetězec „RIFF“ v `ASCII` formátu
	`ChunkSize`	4	Velikost souboru bez `ChunkID` a `ChunkSize`
	`Format`	4	Řetězec „WAVE“ v `ASCII` formátu
fmt	`Subchunk1ID`	4	Obsahuje řetězec „fmt " (s mezerou) v `ASCII` formátu
	`SubchunkSize`	4	16 pro `PCM`
	`AudioFormat`	2	1 pro standardní `PCM`, 3 pro `IEEE` (desetinná čísla)
	`NumChannels`	2	Počet kanálů (v *adess* vždy 1)
	`SampleRate`	4	Vzorkovací frekvence, zadáno uživatelem
	`ByteRate`	4	`SampleRate*NumChannels*BitsPerSample/8`
	`BlockAlign`	2	`NumChannels*BitsPerSample/8`
	`BitsPerSample`	2	Rozlišení vzorku (8, 16, 24 nebo 32)
data	`Subchunk2ID`	4	Řetězec „data“ v `ASCII` formátu
	`Subchunk2Size`	4	`NumSamples*NumChannels*BitsPerSample/8`
	`data`	-	Pole vzorků

Tab. 6 Hodnoty v hlavičce souboru `WAV` [12]

Data jsou do binárního souboru `WAV` zapsány pomocí funkce `fwrite`.

8 Možné rozšíření

Tento projekt nabízí mnoho možností v rámci rozšíření. Za nejdůležitější z nich považuji podporu operačního systému *Windows*, která je v současnosti možná pouze s použitím *WSL*. Dalším možným vylepšením je usnadnění použití *Adess* dalšími aplikacemi, které mohou uživateli nabídnout snažší ovládání pomocí grafického uživatelského rozhraní. *Adess* už je možno takto použít, můj bratr vytvořil program, který dokáže ovládat *Adess* pomocí plynu a brzdu, což může být pro uživatele výrazně snažší. Pokud bych se rozhodl pro tento přístup, pravděpodobně bych zvolil tvorbu knihovny namísto vlastní aplikace a soustředil bych se na samotnou generaci kódu (nikoliv *parsování* a práci se soubory).

Druhou cestou, kterou by mohl projekt dále postupovat, je využití simulace. Simulace se pro tvorbu zvuku motorů již využívají, (například *engine-sim* od Angeho Yaghe), tyto simulátory často pracují v reálném čase. Jelikož aplikace *Adess* nevyžaduje práci v reálném čase, bylo by možné využít *CFD* simulace. Takto vytvořený zvuk by byl násobně kvalitnější než současný stav *Adess*. [13]

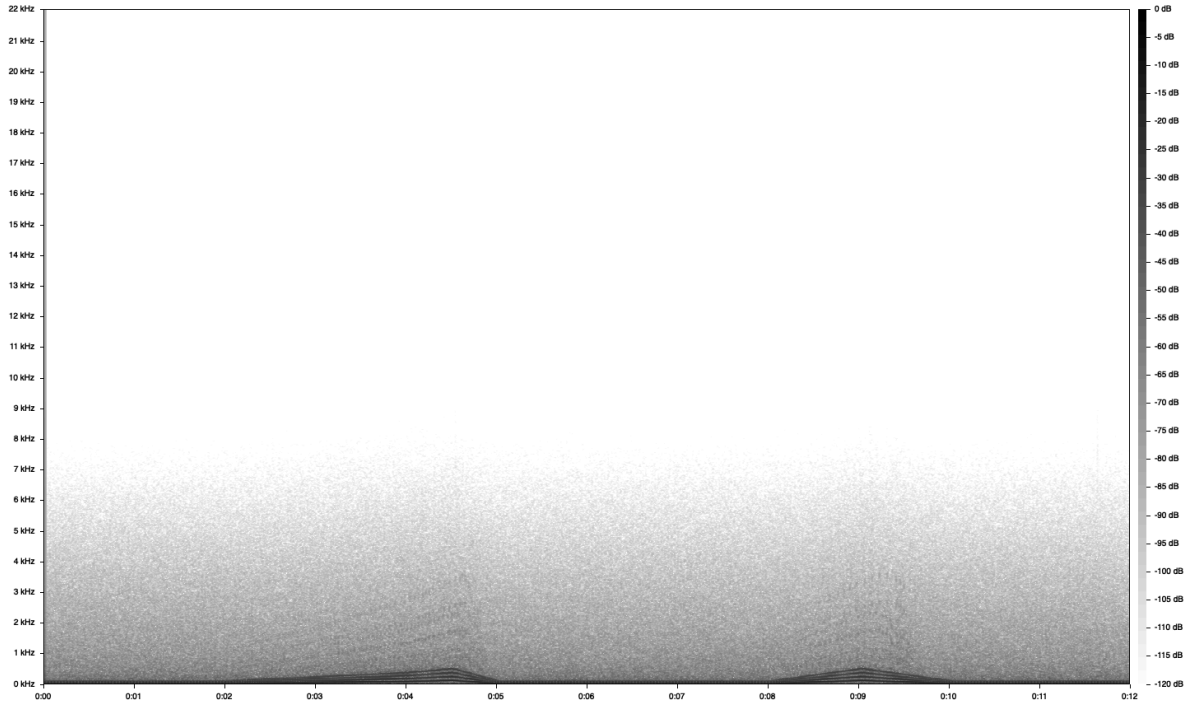
9 Závěr

Syntetická tvorba zvuku je velice obsáhlé a komplexní téma. Je velice obtížné synteticky vytvořit zvuk, který nezní jako byl vytvořen synteticky. Přes značnou komplexitu tohoto témata jsem byl schopen vytvořit produkt, který ukazuje, že v tomto přístupu pro generaci zvuku je potenciál.

Díky tomuto projektu jsem měl možnost zkombinovat několik různých disciplín: fyziku (zvuk), inženýrství (pro znalost spalovacích motorů) a samozřejmě programování. Pro dosažení vysoké rychlosti programu jsem se naučil základy jazyka `C` a paralelního programování.

Příklad hotové zvukové stopy je na Obr. 2, příklady zvukových stop jsou v kapitole 11.

Tento projekt vedl k značnému rozvinutí mých znalostí ohledně programování a prokázal, že syntetická generace zvuku motorů je možná.



Obr. 2 Hotová zvuková stopa [`Spek`]

10 Seznam použitých zdrojů

[1] JACKSON, Blair. Engine FX With Personality in Pixar's Cars. *Mix* [online]. 2006 [vid. 2026-01-14]. Dostupné z: https://www.mixonline.com/sfp/engine-fx-personality-pixars--cars-369139

[2] KITWARE, INC. *CMake* [online]. 2026 [vid. 2026-01-18]. Dostupné z: https://cmake.org/

[3] FREE SOFTWARE FUNDATION. *GNU Compiler Collection (GCC)* [online]. 2026 [vid. 2026-01-18]. Dostupné z: https://gcc.gnu.org/

[4] CANNAM, C., C. LANDONE a M. SANDLER. Sonic Visualiser: An Open Source Application for Viewing, Analysing, and Annotating Music Audio Files. In: *Proceedings of the ACM Multimedia 2010 International Conference*. 2010, s. 1467–1468.

[5] KOJEVNIKOV, Alexander. *Spek – Acoustic spectrum analyzer* [online]. 2025 [vid. 2026-01-18]. Dostupné z: https://www.spek.cc/

[6] DOSITS. [online]. 2025 [vid. 2026-01-18]. Dostupné z: https://dosits.org/science/measurement/what-sounds-can-we-hear/

[7] WIKIPEDIA CONTRIBUTORS. *Brownian noise* [online]. 2026 [vid. 2026-02-06]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Brownian_noise&oldid=1331654429

[8] WIKIPEDIA CONTRIBUTORS. *Xorshift* [online]. 2026 [vid. 2026-02-06]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Xorshift&oldid=1335770712

[9] WIKIPEDIA CONTRIBUTORS. *Fourier transform* [online]. 2025 [vid. 2026-01-26]. Dostupné z: https://en.wikipedia.org/w/index.php?title=Fourier_transform&oldid=1328640468

[10] WICKERHAUSER, Mladen Victor. [online]. [vid. 2026-01-02]. Dostupné z: https://www.math.wustl.edu/~victor/mfmm/fourier/fft.c

[11] BRAUN, S. *WINDOWS* [online]. Oxford: Elsevier, 2001 [vid. 2026-02-04]. ISBN 978-0-12-227085-7. Dostupné z: doi:https://doi.org/10.1006/rwvb.2001.0052

[12] CENTER FOR COMPUTER RESEARCH IN MUSIC AND ACOUSTICS, STANFORD. [online]. [vid. 2025-10-04]. Dostupné z: http://soundfile.sapp.org/doc/WaveFormat/

[13] YAGHI, Ange. [online]. [vid. 2026-01-14]. Dostupné z: https://github.com/ange-yaghi/engine--sim

11 Přílohy