Politechnika Śląska w Gliwicach

Wydział Automatyki, Elektroniki i Informatyki

Programowanie Komputerów 3

Edytor muzyczny

autor	Łukasz Ważny
prowadzący	dr inż. Piotr Pecka
rok akademicki	2018/2019
kierunek	Informatyka
rodzaj studiów	SSI
semestr	3
termin laboratorium / ćwiczeń	czwartek nieparzysty, 8:30 – 10:00
grupa	1
sekcja	1
termin oddania sprawozdania	2019-02-10
data oddania sprawozdania	2019-01-29

1 Treść zadania

W oparciu o bibliotekę bass do tworzenia i syntezy dźwięku (zostanie dostarczony prosty działający przykład, który większość tego projektu już realizuje) napisać prosty edytor muzyczny odtwarzający melodie zapisane w pliku tekstowym.

2 Specyfikacja zewnętrzna

Treść zadania zostawia, można powiedzieć, pełną dowolność co do sposobu zapisu melodii w pliku tekstowym. Dlatego też poniżej opiszę w jaki sposób zapisu wymagany jest, aby program działał poprawnie.

Najlepiej wytłumaczyć to na gotowym przykładzie, tak więc poniżej znajduje się przykładowy plik tekstowy z gotową do odtworzenia melodią.

```
2200
2
0.11
8 1P 1F5 1F5
8 1P 2D5 4D5 4E5B 2D5 2E5B
8 4E5B 4E6B 8D6 8C6 8H5B 8A5 2H5B 8P 8P 8F5 8E5B 4D5 4H5B 4H5B 8A5 8G5
8 1P 2H4B 2A4 2H4B 4H4B 4C5
8 1P 4F5 2F6 4F6 1F5
8 1P 1P 2D5 2E5B
8 1P 2P 2F4 2H4B 4H4B 4C5
```

Są to trzy takty z pięknej melodii o tytule „Non Nobis Domine”. Pełny zapis nutowy tej melodii znajduje się w repozytorium na githubie w folderze projekt. Tam również znajduje się plik `melody.txt`, w którym jest przepisane 5 stron tej melodii w formacie niezbędnym do odtworzenia przez program.

Przechodząc do rzeczy - pierwsza linijka tekstu, to prędkość (w milisekundach) odtwarzania jednego taktu. W przypadku powyższego przykładu jeden takt będzie odtwarzał się przez 2,2 s.

Kolejna linijka to liczba kanałów. W powyższym przykładzie będzie to stereo.

Trzecia linijka oznacza głośność. Można tam wpisać dowolną liczbę rzeczywistą z przedziału od 0 do 1.

Każda z kolejnych linijek to osobna linia melodyczna. Można ich wprowadzić nieograniczoną ilość, dzięki czemu uzyskujemy polifonię z nieograniczoną liczbą głosów. Trzeba jednak mieć na uwadzę, iż każda dodatkowa linia melodyczna zwiększa głośność odtwarzania całej melodii, tak więc wraz z dodawaniem nowych lini warto zmniejszyć głośność. W powyższym przykładzie jest 7 głosów.

Pierwsza cyfra w lini melodycznej, to metrum przedstawione w formacie, odpowiadającym na pytanie - ile ósemek znajduje się w jednym takcie. Jak widać każda linia melodyczna może mieć inne metrum, jednak w większości utworów metrum każdej lini jest jednakowe. Tak też jest w powyższym przypadku - metrum każdej lini to cztery czwarte, co oznacza, że w każdym takcie znajduje się osiem ósemek.

Po cyfrze przedstawiającej metrum, znajdują się kolejne nuty lini melodycznej, oddzielone spacjami. Pierwsza cyfra każdej nuty oznacza długość jej trwania. „1" oznacza pełną nutę, „2" - półnutę, „4" - ćwierćnutę, „8" - ósemkę, itd. Można używać dowolnie „małych" nut. Po cyfrze oznaczającej długość znajduje się nieoddzielona żadnym znakiem nazwa nuty, określająca jej wysokość. Stosowana jest tu standardowa notacja przedstawiona na stronie internetowej `http://pages.mtu.edu/~suits/notefreqs.html`, gdzie `A4` oznacza dźwięk o częstotliwości `440 Hz`. Obecnie dostępny jest zakres nut od `C3` do `E6`, jednak program został napisany w ten sposób, aby można było ten zakres łatwo poszerzać. Użycie litery `P` oznacza pauzę o sprecyzowanej poprzednią liczbą długości trwania. Na końcu nuty można dodać znak `B` lub `#`. Który odpowiednio obniża lub podwyża dźwięk o pół tonu.

Plik tekstowy z taką zawartością można odtworzyć w programie uruchamiając go z lini poleceń, przekazując do programu nazwę tegoż pliku tekstowego. Przykładowe uruchomienie programu:

```
program.exe melody.txt
```

3 Specyfikacja wewnętrzna

Program został podzielony na następujące pliki:

main.cpp	plik zawierający funkcję główną;
generator.h	plik zawierający deklaracje funkcji generującej falę sinusoidalną;
generator.cpp	plik zawierający definicję funkcji generującej falę sinusoidalną;
note_names.h	plik zawierający deklaracje mapy zawierającej nazwy nut;
note_names.cpp	plik zawierający definicję mapy zawierającej nazwy nut;
score.h	plik zawierający deklarację klasy, której obiekty przechowują dane dotyczące pojedynczej lini melodycznej;
score.cpp	plik zawierający definicje metod klasy score;
melody.h	plik zawierający deklarację klasy, której obiekty przechowują dane dotyczące całej melodii;
melody.cpp	plik zawierający definicje metod klasy melody.

3.1 Ogólna struktura programu - plik `main.cpp`

W funkcji głównej sprawdzane jest najpierw, czy do programu została przekazana odpowiednia liczba argumentów. Następnie tworzony jest obiekt `song` klasy `melody` za pomocą konstruktora, który opisany jest na str. 11. Potem wywoływana jest na obiekcie `song` metoda `void playMelody()`, co powoduje odtworzenie melodii. Na końcu zwalniane są zasoby za pomocą funkcji `bool BASS_Free()`.

3.2 Mapa `note_name` - plik `note_names.h` oraz `note_names.cpp`

W pliku `note_names.h` została zadeklarowana mapa:

```
1 extern map<string, int> note_name;
```

Jej elementy zdefiniowane są w pliku `note_names.cpp`. Kluczem każdego elementu jest nazwa nuty w odpowiedniej notacji (patrz str. 3), natomiast wartością elementu jest odpowiednia liczba, potrzebna do obliczenia częstotliwości dźwięku (patrz str. 5).

3.3 Generator fali sinusoidalnej - plik generator.h oraz generator.cpp

W pliku generator.h została zadeklarowana funkcja:

```
short * sinwave(DWORD length, string * notes, int n,
    int metre, DWORD deviceFrequency, float vol, int
    channels);
```

Zdefiniowana jest ona w pliku generator.cpp. Zwraca bufor wypełniony odpowiednimi wartościami fal sinusoidalnych o częstotliwościach odpowiadających dźwiękom, które mają być odtworzone. Przyjmuje parametry:

length	rozmiar zwracanego bufora;
notes	tablica nazw nut, które mają być odtworzone;
n	rozmiar tablicy notes;
metre	metrum lini melodycznej skąd pochodzą nuty do odtworzenia;
deviceFrequency	częstotliwość próbkowania urządzenia, na którym odtwarzana jest melodia;
vol	głośność utworu;
channels	liczba kanałów.

Funkcja ta najpierw alokuje odpowiednią ilość pamięci:

```
int bufferSize = length / sizeof(short);
short * buffer = new short[bufferSize];
```

Następnie, dla każdego elementu tablicy notes wykonywane są następujące kroki.

Obliczenie części bufora, która musi być zapełniona odpowiednimi wartościami, na podstawie długości trwania nuty, metrum oraz ilości kanałów:

```
int d = bufferSize / sizeof(short) / metre *
(1 / (float)(stoi(notes[i].
substr(0, notes[i].
find_first_not_of("0123456789"))))) *
8 * channels;
```

Obliczenie częstotliwości dźwięku:

```
float frequency = pow(2.0, (note_name[notes[i].
substr(1)]) / 12.0) * 440;
```

Wypełnienie bufora odpowiednimi wartościami:

```
int s = sin(pos) * vol;
```

Wartość `s` wpisywana jest do bufora odpowiednią ilość razy (na podstawie zmiennej `d`) w pętli for. Zmienna `pos` modyfikowana jest każdorazowo po wpisaniu `s` do bufora, na podstawie zmiennej `frequency` tak, aby w buforze znajdowały się wartości sinusoidy o odpowiedniej częstotliwości.

Następnie zwracany jest `buffer`.

3.4 Pliki score.h oraz score.cpp

3.4.1 Klasa score

W pliku `score.h` został zdefiniowany następujący pomocniczy typ, używany w klasie `score`:

```
struct bar {
    string * notes;
    int n;
};
```

Typ ten reprezentuje jeden takt lini melodycznej i zawiera w sobie tablicę dynamiczną nazw nut, znajdujących się w tym takcie - `notes` o rozmiarze `n`.

W tym samym pliku znajduje się też deklaracja klasy `score`:

```
class score {
    bar * bars;
    int n;
    int metre;
public:
    score(string);

```

```
8     short * getBuffer(int i, DWORD length, DWORD
          deviceFrequency, float vol, int channels);
9
10    bar * getBars();
11    int getLength();
12    int getMetre();
13
14 };
```

Obiekty tej klasy reprezentują pojedynczą linię melodyczną utworu.

Posiada ona następujące pola:

bars	tablica wszystkich taktów, znajdujących się w lini melodycznej
n	rozmiar tablicy `bars`;
metre	metrum lini melodycznej.

3.4.2 Opis implementacji metod klasy score

Definicje metod tej klasy znajdują się w pliku `score.cpp`.

Zdefiniowany został kostruktor:

```
1 score::score(string input)
```

Tworzy on obiekt klasy `score` i przypisuje do jego pól odpowiednie wartości. Jako parametr przyjmuje łańcuch znaków, który pochodzi z jednej z linijek pliku tekstowego, będącą zapisem lini melodycznej w odpowiedniej notacji (patrz str. 3).

Konstruktor ten najpierw liczy liczbę nut w lini melodycznej:

```
1 int i = count(input.cbegin(), input.cend(), ' ');
```

Odczytuje również i zapisuje w polu `metre` metrum:

```
1 metre = stoi(input.substr(0, input.find(' ')));
```

Tworzy też tymczasową tablicę dynamiczną nazw nut o rozmiarze wyliczonym wcześniej (zmienna `i`):

```
1 string * notes = new string[i];
```

Następnym krokiem jest wypełnienie tablicy `notes` odpowiednimi nazwami nut, po czym można, korzystając z tej tablicy oraz odczytanego metrum, obliczyć liczbę taktów w lini melodycznej:

```
1  int j = 0;
2  n = 0;
3  while (j != i) {
4      float sum = 0;
5      while (sum != metre) {
6          sum += (1 / (float)(stoi(notes[i].
7          substr(0, notes[i].
8          find_first_not_of("0123456789"))))) * 8;
9          j++;
10     }
11     n++;
12 }
```

Co pozwala zaalokować pamięć dla tablicy `bars`:

```
1 bars = new bar[n];
```

Następnie dla każdego elementu tablicy `bars` tworzona jest tablica nazw nut `bars[k].notes` o odpowiednim rozmiarze równym ilości nut w k-tym takcie, wypełniona tymi nazwami nut, które w tymże takcie się znajdują.

Na końcu zwalniane są zasoby tymczasowej tablicy `notes`:

```
1 delete[] notes;
```

Kolejna metoda klasy `score`, to:

```
short * score::getBuffer(int i, DWORD length, DWORD
    deviceFrequency, float vol, int channels)
```

Metoda ta zwraca buffor wypełniony takimi wartościami fali sinusoidalnej, aby mógł być odtworzony i-ty takt lini melodycznej.

Przyjmuje parametry:

i	numer taktu, który ma być odtworzony;
length	rozmiar zwracanego bufora;
deviceFrequency	częstotliwość próbkowania urządzenia, na którym odtwarzana jest melodia;
vol	głośność utworu;
channels	liczba kanałów.

W ciele metody wywoływana jest z odpowiednimi parametrami funkcja generująca fale sinusoidalną (patrz str. 5), której wynik przypisywany jest do bufora `buffer`:

```
short * buffer = sinwave(length, bars[i].notes,
 bars[i].n, metre, deviceFrequency, vol, channels);
```

Następnie zwracany jest `buffer`.

Ostatnie metody klasy `score`, to:

```
bar * score::getBars();
int score::getLength();
int score::getMetre();
```

Każda z nich po prostu zwraca odpowiednio wartości pól: `bars`, `n`, `metre`.

3.5 Pliki melody.h oraz melody.cpp

3.5.1 Klasa melody

Deklaracja tej klasy znajduje się w pliku `melody.h`:

```
1  class melody {
2     DWORD deviceFrequency;
3     DWORD speed;
4     int channels;
5     float vol;
6     vector<score> scores;
7     int n;
8     int bars;
9  public:
10    melody(string);
11
12    void playBar(int);
13    void playMelody();
14
15 };
```

Obiekt tej klasy reprezentuje całą melodię.

Posiada ona następujące pola:

deviceFrequency	częstotliwość próbkowania urządzenia, na którym odtwarzana jest melodia;
speed	długość trwania jednego taktu;
channels	ilość kanałów;
vol	głośność melodii;
scores	obiekt klasy `vector` o elementach typu `score`, będący w tym przypadku tablicą wszystkich lini melodycznych, znajdujących się w utworze;
n	rozmiar tablicy `scores`;
bars	ilość taktów każdej lini melodycznej (w poprawnie zapisanym utworze, liczba taktów każdej lini jest taka sama).

3.5.2 Opis implementacji metod klasy melody

Definicje metod tej klasy znajdują się w pliku `melody.cpp`.

Zdefiniowany został kostruktor:

```
melody::melody(string argv)
```

Tworzy on obiekt klasy `melody` i przypisuje do jego pól odpowiednie wartości. Jako parametr przyjmuje łańcuch znaków, który jest nazwą pliku tekstowego, zawierającego dane dotyczące melodii do odtworzenia.

Pierwsze kilka linijek kodu odpowiedzialne jest za konfigurację niezbędną do korzystania z biblioteki BASS. Na końcu konfiguracji przypisywana jest do pola `deviceFrequency` odczytana w jej trakcie częstotliwość.

Następnie otwierany jest plik tekstowy o przekazanej przez argument nazwie:

```
ifstream input(argv);
```

Kolejnymi trzema krokami jest odczytanie oraz przypisanie do odpowiednich pól wartości:

- długość trwania jednego taktu;
- ilość kanałów;
- głośność melodii.

Następnie do tymczasowej zmiennej `temp` odczytywane są kolejne linijki tekstu, będące pojedynczymi liniami melodycznymi utworu oraz za pomocą konstruktora klasy `score` dodawane są do tablicy `scores` obiekty reprezentujące odczytane linie melodyczne:

```
string temp;
while (getline(input, temp)) {
    scores.push_back(score(temp));
}
```

Na końcu do pola `n` przypisywany jest rozmiar tablicy `scores`, do pola `bars` przypisywana jest ilość taktów w pierwszej lini melodycznej (która równa jest ilości taktów we wszystkich pozostałych liniach) oraz zamykany jest strumień obsługujący otwarty plik tekstowy.

Kolejna zdefiniowana w tym pliku metoda, to:

```
void melody::playBar(int i)
```

Po jej wywołaniu odtwarzane są jednocześnie i-te takty ze wszystkich lini. Jedyny parametr jaki przyjmuje, to `i` - numer taktu, który ma zostać odtworzony z każdej lini.

W ciele metody, najpierw tworzony jest, korzystając z funkcji udostępnianej przez bibliotekę BASS, sample o odpowiednich parametrach:

```
HSAMPLE sample = BASS_SampleCreate(speed,
    deviceFrequency, channels, 1, 0);
```

Następnie tworzony jest bufor o odpowiednim rozmiarze, który wypełniony zostaje zerami:

```
int bufferSize = speed / sizeof(short);
short * buffer = new short[bufferSize];
memset(buffer, 0, speed);
```

Kolejny krok, to jeden z kluczowych momentów. Wykorzystany jest tu fakt, iż w przypadku polifoni, na podstawie każdej lini melodycznej, generowana jest osobna fala dźwiękowa. Fale te interferują ze sobą, tworząc jedną falę. Można skorzystać z zasady superpozycji, co w przypadku tego programu oznacza, że trzeba po prostu dodać do siebie odpowiednie wartości z buforów wygenerowanych z każdej lini melodycznej i zapisać do jednego wspólnego bufora, który następnie może zostać odtworzony, uzyskując tym samym efekt polifoni z nieogrniczoną liczbą lini melodycznych.

Dlatego też, dla każdej lini melodycznej tworzony jest tymczasowy bufor `buffer_`, który wypełniany jest wartościami uzyskanymi za pomocą metody `getBuffer`, wywołanej na obiekcie reprezentującym tę linię melodyczną. Następnie odpowiednie wartości z tego bufora dodawane są do odpowiednich wartości bufora `buffer`, w którym na początku znajdują się same zera, lecz po powtórzeniu tej procedury dla każdej lini melodycznej, wartości te stają się sumą wartości z tych tymczasowych buforów. Oczywiście zasoby związane z tymczasowymi buforami są zwalniane.

Procedura ta jest widoczna w poniższej częsci kodu:

```
for (int j = 0; j < n; j++) {
    short * buffer_ = scores[j].getBuffer(i, speed,
        deviceFrequency, vol, channels);
    for (int k = 0; k < bufferSize; k++) {
        buffer[k] += buffer_[k];
```

```
5         if (buffer[k] > 32767)          buffer[k] = 32767;
6         else if (buffer[k] < -32768)    buffer[k] = -32768;
7     }
8     delete[] buffer_;
9 }
```

Następnie wartości z `buffer` ładowane są do zadeklarowanego na początku sampla:

```
1 BASS_SampleSetData(sample, buffer);
```

Potem odtwarzana jest melodia zapisana w buforze `buffer`:

```
1 HCHANNEL channel= BASS_SampleGetChannel(sample, FALSE);
2 BASS_ChannelPlay(channel, FALSE);
```

Następnie korzystając ze zmiennej typu bool `sem` oraz pętli `while`, w której ciągle sprawdzane jest, czy odtwarzanie melodii dobiegło końca; program czeka aż odtwarzanie się skończy. Gdy to nastąpi, zwalniane są używane zasoby.

Ostatnią zdefiniowaną metodą klasy `melody` jest:

```
1 void melody::playMelody()
```

Odtwarza ona całą melodię, wywołując za pomocą pętli `for` metodę `playBar` dla każdego taktu.

4 Testowanie

Program został przetestowany na pliku tekstowym o nazwie `melody.txt`, który znajduje się w repozytorium na githubie w folderze z projektem. Znajduje się w tym folderze również oryginalny zapis nutowy tego utworu w formacie `pdf`.

Podczas testowania nie znaleziono żadnych nieprawidłowości w działaniu programu.

Ze względu na dosyć czasochłonny proces przepisywania oryginalnego zapisu nutowego na zapis wymagany do prawidłowego działania programu, jest

to jedyny plik, na którym przetestowano program. Biorąc jednak pod uwagę to, jak bardzo obrzerny jest ten plik oraz ile różnych konfiguracji danych wejściowych zawiera, można stwierdzić, że testowanie programu odbyło się w dosyć szerokim zakresie możliwych danych wejściowych.

Ponadto program uruchamiano z różną wartością długości trwania jednego taktu i za każdym razem działał on poprawnie.

Manipulowano również głośnością, co też nie wykazało żadnych błędów, z wyjątkiem tego, że przy zbyt dużej głośności zakres zmiennej typu `short` był przekraczany, co powodowało znaczne zniekształcenie odtwarzanego dźwięku, co jest naturalną konsekwencją ograniczających nas limitów (ewentualnie można wykluczyć na pewną skalę te zjawisko używając większego typu zmiennej).

5 Wnioski

Program ten był programem dosyć wymagającym.

Pierwszym sprawiającym trudności zadaniem było zrozumienie działania używanej biblioteki `BASS`. W internecie jest niezwykle mało tutoriali dotyczących tej biblioteki, musiałem więc polegać jedynie na dokumentacji, która również nie była zbytnio przyjazna początkującemu programiście. Udało mi się to jednak wykonać, dzięki czemu nauczyłem się nowych niezwykle przydatnych umjejętności programistycznych, polegających na szybkim zaadaptowaniu się do nieznanych wcześniej rozwiązań, które ta biblioteka oferuje oraz wykorzystaniu ich jak najlepiej w wyznaczonych celach.

Kolejnym wymagającym problemem było zaznajomienie się z czysto fizycznym aspektem zjawiska jakim jest dźwięk. Co prawda pewne informacje na ten temat uzyskałem w wyniku mojej edukacji szkolnej. W tym przypadku musiałem jednak bardziej wgłębić się w temat i dowiedzieć się jakimi dokładnymi wzorami opisywana jest idealna fala dźwiękowa, jaką zmienia się częstotliwość dźwięku wraz ze zmianą wysokości dźwięku o zadaną liczbę tonów, jakie zależności zachodzą między falami dźwiękowymi, gdy mamy do czynienia z polifonią. Okazało się jednak, że spędzenie znacznej ilości godzin na analizie dostępnych w internecie źródeł rozwiązało ten problem.

Następnym zagadnieniem, które okazało się przeszkodą do przejścia, była

kwestia implementacji fizycznych zależności, mając do dyspozycji narzędzia takie jak język `C++` oraz dostępne biblioteki, w tym biblioteka `BASS`. Zadanie te okazało się szczególnie trudne ze względu na fakt, iż wykonywałem je bez przykładu, który został wspomniany w treści zadania. Musiałem całkowicie od nowa napisać wiele funkcji i metod, przechodząc po drodze przez niezliczoną ilość błędów; co jednak wykształciło we mnie wiele cennych umjejętności.