Sound synthesis rush

Summary: In this rush, you will create a synthesizer able to play a simple music description file format.

Chapter 1

Introduction

Sometimes, programming is just the tool, the implementation detail at the end of a long period of research. In your professional life, you might encounter situations where you will need to apply your programming skills to a field you know none or little of.

In this rush, you will have to learn about how sound is represented on computers, generated, and some rudiments of musical theory, to simulate the kind of music chips that were present on early computers and game consoles. Though your final implementation might be fairly simple and short, it is the path to the required knowledge that will be the challenge.

When memory was limited, both in terms of storage and RAM, music was distributed using descriptive file formats ("play this note with this duration, with that instrument"), and generated on the fly by dedicated hardware. The technical constraints motivated great artistic endeavors and an entire subculture of electronic music. See for instance **https://en.wikipedia.org/wiki/Chiptune** for more context and information.

Chapter 2

Instructions

The objective of this rush is to create a tool called **minisynth**, able to read in a specific music description file format (see next sections), and to play it.

You can use the languages, libraries, frameworks or tools of your choice. You are not required to write the entire project in a single language. You can make multiple tools, each taking care of a specific aspect of the project, as long as it all comes down to an executable called **minisynth** that performs the required function.

The focus of this rush is not error handling. You can assume that your program will be tested with valid input files.

2.1 Command line usage

Your tool should be invoked as follows, at the root of your repository:

```
./minisynth file
```

Where **file** is the path to the music file to play. Your tool should read the file, parse its content, and play the music it represents on the computer's current sound output. Then, it should exit.

Your program may display any kind of information you want while playing.

2.2 Music description format

Your **minisynth** program should accept the following file format (examples come with the extension **.synth**, though this is not a requirement):

- The file format is text based
- Any empty line or line starting with a **#** character is ignored as comment
- The first non-comment line must be in the format **tempo <N>**, where **<N>** is an integer, representing the tempo of the piece in beats per minute
- The next non-comment line must be **tracks** followed by a comma separated list of instruments: **tracks**

<instrument>[,<instrument>,...]. Each entry in this list represent a track, numbered from 1 to the total number of tracks. See further for the list of instruments to support.

- All remaining non-comment lines must be in the following format: **<track>:<notes>**, where **<track>** is the track number, and **<notes>** represents notes to be added to the given track.

2.3 Notes format

The **<notes>** part of each line is parsed for all substrings matching the following pattern: **<pitch>[<alteration>][<octave>][/<duration>]**. Any other character is ignored.

- **<pitch>** is any lowercase letter from **a** to **g**, representing the usual pitch names in Western notation, or the letter **r** to represent a rest (silence)
- **<alteration>** is optional, and can either be **#** or **b**, indicating that the note should be sharp or flat (note that these are the hash symbol and the lowercase letter B, representable in ASCII, not the sharp and flat musical symbols available in Unicode)
- **<octave>** is an optional integer from 0 to 9 representing the octave of the note (using the standard scientific pitch notation, so that middle C is **c4**)
- **<duration>** is optional and preceded with a **/** when present: it is a decimal number, possibly fractional, representing the duration of the note in beats

When the octave is not indicated, the note takes the same octave as the previous note on the line. Similarly, when the duration is not indicated, the note takes the same duration as the previous note on the line. The first note of the line, if not indicated otherwise, is in octave 4, and lasts 1 beat. This resets for each new line.

Each new note successfully parsed in the line is added at the end of the given track. It is assumed that a note begins exactly when the previous one ended (silent notes are used to space out the sounds).

Other characters on the line are ignored. For instance, **|** can be used as a visual separator for bars, as well as spaces, for human readability.

The file may contain any number of music description lines. If several lines begin with the same track number, new notes are simply appended at the end of the track.

2.4 Synthesis

Each track generates notes according to the instrument assigned, as indicated by the **tracks ...** line in the document. Your tool must support the following wave form generation:

- sine waves
- saw waves
- square waves
- triangle waves

The possible values for **<instrument>** are therefore **sine**, **saw**, **square** and **triangle**.

All the tracks in the file start at the same time, and play simultaneously. If a track is shorter than the others, it is assumed to be silent until the end of the piece.

You should use the standard modern tuning of A4 playing at 440 Hz.

Chapter 3

Tips

- Most sound libraries will require you to implement a callback function, which is called whenever the sound card requires more samples to play. Make sure that your generating functions can provide the right number of samples, and track progression through the song.
- You may internally generate your samples in any format you like, though most sound cards will require two channels of 16 bits signed integers at 44100 samples per second. Make sure your program is able to convert to the required format.

Chapter 4

Bonuses

A rudimentary synthesizer like this can be improved in many ways. Here are a few that you can implement if you have time:

- Add a mixer to your synthesizer. Add an additional header line in the file, in the format **`volumes <V>[,<V>,...]`**, where **`<V>`** are numbers from 0 to 100, one for each track, that indicates the relative volumes of each track. For instance, a track with volume 50 should be half as loud as one with volume 100, and a track with volume 0 is entirely silent.
- Add an envelope generator to your synthesizer, to better control the timbre and feel of the notes played. Specifically, add to the syntax of the file, so that each track can have its own settings for envelope. Envelope should be described using the usual four parameters: attack, decay, sustain, release.
- Add a kick drum instrument (**`kick`**). Kicks can be made for instance from a low frequency sine wave with a fast attack and slow decay. It should ignore the note's pitch and duration, using only the start time as indication.
- Add a snare drum instrument (**`snare`**). For instance, white noise with a fast attack and slow decay can work. Same comment as previously for the duration and pitch.
- Add visuals to your player. They should somehow react to the music played in real time (for instance to loudness or pitch).

Chapter 5

Evaluation

As usual, the evaluator will clone your git repository, and should be able to run your project themselves. Since you have the choice of language and tools, make sure that your project can run on the school iMacs, and to include *complete, written instructions* on how to install the requirements.

If you write your project in C, you are not required to follow the Norm. However, regardless of the language, you are still expected to provide source code that compiles and/or runs without errors for valid inputs.

You may use any library or tool for the actual playing of the sound on the computer, but you must write all the sound sample generation yourself.

The music should start playing (almost) immediately when the executable is run.