RUN-TIME MIDI TRANSITION ALGORITHM FOR INTERACTIVE MEDIA AND GAMES

An Interactive Qualifying Project Report:

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC
INSTITUTE

In partial fulfillment of the requirements for
the Degree of Bachelor of Science
By

Yong Piao

Date: January 2nd, 2014

Approved:

_____

Professor Fredrick W. Bianchi, Major Advisor

Table of Contents

Abstract

This project describes the design and development of a MIDI[1] file parsing & playback framework designed for providing users the ability to modify MIDI data in run-time especially suited for interactive media and games.

[1] Acronym for Musical Instrument Digital Interface.

Acknowledgements

I would like to thank Professor Frederick W.Bianchi for his advice and support. His enthusiasm and insight on this subject were crucial to the completion of this work.

List of Figures

Introduction

Video games have been playing an increasingly important role in man's life. "The industry is at around $22 billion for 2008 (conservative estimate) in the US[2] and $30 to $40 billion globally," while "The movie industry is at $9.5 billion (US)[3] and $27 billion globally[4]." Due to the inter-disciplinary nature of video game development, video games have also brought benefits to the concept art industry, 3d modeling industry, music industry, and etc.

Video games are meant to be interactive by its definition, and to achieve so, every component in the video game must also be interactive. Music has always been playing a big role in conveying emotions to audiences, but it has always lacked the flexibility that computer graphics researchers have achieved in the past few years. For example, with the NVidia's latest voxel global illumination technology, it is now possible to simulate highly realistic first-bounce reflective light in real-time. (It is one of the biggest reasons why computer graphics in movies can look more realistic than in video games)

Music in video games in current music production has severe limitations that in the field of computer graphics, have already been solved. Real-time computer graphics

[2] http://arstechnica.com/news.ars/post/20080618-gaming-expected-to-be-a-68-billion-business-by-2012.html
[3] http://www.slyck.com/story1436_MPAA_Reports_Record_Movie_Sales_in_2006
[4] http://www.abc.net.au/news/stories/2008/03/06/2181568.htm

offers the computer the ability to improvise on graphics in real-time. Every moment in the game is dictated by the player's actions and never needs to be intentionally programmed to be displayed at a given time, because every key-frame of the scene is generated in real-time and calculated through algorithms.

NVidia researchers have been focusing on real-time rendering because offline rendering has been very mature for the movie industry, and has less room for discovery. (Due to the architectural design of GPU, real-time GPU rendering can offer as much as 60 times the speed of CPU rendering.[5]) The increasing artistic freedom in real-time graphics allows many in-game cut scenes to be rendered in real-time as well, e.g. Assassin's Creed Unity, Final Fantasy XIII-2, etc.

Music in video games is less interactive than other components of video games, because modern game music uses streamed audio. Streamed audio must be composed, arranged, and mixed by musicians into audio files such as .WAV and .MP3 prior to being played in game. Because the music files are pre-rendered, it is simply impossible to perform notation-level editing or calculation in real-time, because that information has been lost in the rendering process. Therefore it has been difficult to generate sufficient amounts of new materials from the limited track information.

[5] http://www.elnexus.com/articles/GPUComputing.aspx

What will benefit the field of interactive music is the development of a music system with interactive capabilities. It must retain the music notation information and render the music out in real-time, similar to what real-time computer graphics proposes, to achieve notation-level data manipulation in real-time. Only after obtaining algorithmic freedom at this level, will researchers and musicians be able to start developing algorithms that are aesthetically more appealing.

In this project, a MIDI file parsing and playback framework is designed to provide sound engineers and musicians with a small and agile low level framework to dynamically adjust MIDI data during software run-time. It serves as the first necessary knock on the door of an interactive music system.

This framework uses no 3rd party library and is completely written in C++. It differs from most MIDI parsers because instead of using per-stream[6] based MIDI playback and data processing, it uses a per-command[7] based method and thus provides more freedom to handle MIDI data in run-time.

Due to the artistic freedom and complexity of arranging and composition, there is no adequate platform for sound engineers and musicians to experiment with their

[6] Per-stream based MIDI playback refers to sending a large batch of MIDI commands to the MIDI device and have the device handle sequencing and playback.
[7] Per-Command based playback refers to performing timing and sequencing through a computer program loop, and have the program send MIDI commands when the delta time of any commands match the time of the program loop.

music techniques. The preliminary goal of this project is to reduce the low level binary processing complexity with the simplest possible code to allow more artistic freedom.

Currently, music production software such as Logic, Cubase, and Ableton all provide visual editors to access the underlying MIDI data, but the sound libraries required to render the final product requires purchase, and is often very costly. To achieve high quality real-time rendered audio, every client computer must come with a set of sound libraries required by the files. This problem perhaps could be addressed by starting an open source sound library project to provide competitive sound libraires with little to no cost. When such an open source library becomes available to consumers, interactive music will become more approachable to software developers and musicians.

Background

In 1957, the MUSIC-N program allowed an IBM 704 mainframe computer to play a 17-second composition by Max Mathews. Back then computers were ponderous, so synthesis would could take hours.[8] Also, music programs rarely did run in real time, and it was extremely time consuming and expensive for computers to generate just a few minutes of digital audio.

In 1983, MIDI technology was standardized by the MIDI Manufacturers Association (MMA). Because MIDI is sequenced, it can be manipulated on notation-level during software run-time. Since MIDI is designed to store music events instead of the sound waves, it has become the foundation of digital music software products.[9]

For the purpose of this project, sequenced audio is referred to as audio data that are stored in MIDI or a similar format that can be manipulated on notation-level during software run-time. Streamed audio is referred to as pre-rendered audio data such as .WAV, .MP3 that cannot be manipulated on notation-level in software run-time.

[8] Cattermole, Tannith (May 9, 2011). "Farseeing inventor pioneered computer music". Gizmag. Retrieved 28 October 2011 http://www.gizmag.com/computer-music-pioneer-max-mathews/18530/

[9] Swift, Andrew. (May-Jun 1997.), "A brief Introduction to MIDI", SURPRISE (Imperial College of Science Technology and Medicine), retrieved 22 August 2012

MIDI is lightweight, portable, and fast to process. Since the late 1980s, MIDI has played a big role in fitting enough data into memory to provide meaningful musical experiences to the players, because MIDI files use much less data than rendered audio files of the same length.[10]

MIDI files are much smaller than streamed audio files, because MIDI files don't contain sound information. It is up to the hardware to decide how to handle the MIDI information. This often caused inconsistency in sound between different MIDI systems. Because of this, the industry began to look at streamed audio as a solution.

Creative Labs's Sound Blaster series were first introduced in 1989, then three years later Creative Labs released the Sound Blaster 16. They made PCs capable of playing back streamed audio, and thus MIDI was replaced for both hardware inconsistencies and limitations on recording and mixing that made high quality audio difficult to produce. While most game developers adopted these 16-bit playback wavetable-based soundcards in the mid-1990s, disk space was a difficult issue to solve. So the choices had always been either to sacrifice data fidelity, or sacrifice memory.

As an example, a lossless stereo PCM File at 16-bit, 44.1KHz takes up about 10.584MB per minute. With the limited space of a standard CD-ROM (around

[10] http://www.midi.org/aboutmidi/tut_midifiles.php

650-703MB for data, 846MB for pure audio), it can only store computer readable data of the size equivalent to about 66 minutes of lossless stereo PCM. Considering various art assets and application data, it was simply not possible for game developers to deploy both high quality game audio and a satisfying game experience at the same time.

Since 2006, the advent of Blu-ray discs brought many new possibilities in game content creation with a stunning 25GB of data capacity per layer. In 2013, PlayStation 4 has standardized the use of dual-layer Blu-ray discs, meaning each game disc can store up to 50GB of data.

However, in the exponential growth of data storage capabilities of home entertainment systems, one thing has hardly changed: game audio has always been statically pre-rendered and is looped again and again throughout the gameplay.

In fields of digital audio other than video games, there were some attempts that looked to increase the artistic freedom in streamed audio. There had been an attempt that looked to automatically generate music videos based on the change in image signal,[11] but this approach is still based on streamed audio and does not have the benefits of sequenced audio.

[11]http://patft.uspto.gov/netacgi/nph-Parser?Sect1=PTO2&Sect2=HITOFF&p=1&u=%2Fnetahtml%2FPTO%2Fsearch-bool.html&r=14&f=G&l=50&co1=AND&d=PTXT&s1=music.ABTX.&s2=transition.ABTX.&OS=ABST/music+AND+ABST/transition&RS=ABST/music+AND+ABST/transition

"Some games, such as Super Mario Galaxy (2007), had synchronizing sequenced and streaming audio so that additional effects can be added to the streamed music."[12] In the specific case of Super Mario Galaxy, it was possible to mix sequenced and streaming audio because of the specific style of music that had been used in the game. Sequenced audio in Super Mario Galaxy gave satisfactory result because it was specifically designed to convey a MIDI-like style to accompany the orchestral music. There have been other attempts like Super Mario Galaxy, but they were very genre limited and content limited. Many times the sequenced approach is only suitable for classic and cartoon-like games. Overall, there has not been enough attention on further discovering the potential of dynamically generated game audio without such limitations.

This project is dedicated to game audio vanguards who seek to improve game audio from a whole new perspective. It is meant to provide the most basic resource for game developers to get started on revisiting some of the assumptions musicians frequently make about the most common MIDI-based game audio creation pipeline, and thus open up the possibility of using real-time computer generated, aesthetically appealing music in interactive media and games.

[12] http://www.lauraintravia.com/blog/super-mario-galaxy-interview-with-the-sound-team-iwata-asks

Methodology

In order to replicate the inner workings of this MIDI framework, one must first understand the inner workings of Standard MIDI Format[13], and the best way to learn it is to go through the process of reading a MIDI file. The following example assumes the reader is using a little-endian machine with an appropriate C++ compiler installed.

MIDI files are made up of big-endian binary data. It is necessary to be able to translate big-endian data into little-endian for x86 machines, therefore we define:

```
static unsigned short byteSwapShort(unsigned short in)
{
    return ((in << 8) | (in >> 8));
}

static unsigned int byteSwapInt(unsigned int in)
{
    unsigned short *p;
    p = (unsigned short*)&in;

    return ((((unsigned int)byteSwapShort(p[0])) << 16) |
        (unsigned int)byteSwapShort(p[1]));
}
```

Figure 1. Byte-swap functions

Figure 1 converts short and integer into little-endian, because this operation is used frequently, it is important to make sure it is written with performance in mind.

[13] More resources on the Standard MIDI Format:
http://wiki.fourthwoods.com/standard_midi_file_format
http://wiki.fourthwoods.com/midi_file_format

With the big-endian converter code, it is possible to start reading the MIDI file.

Every MIDI file starts with a header that describes the information about the MIDI file:

```
struct MIDIHeaderInfo
{
    unsigned int    id;     // identifier "MThd"
    unsigned int    size;   // always 6 in big-endian format
    unsigned short  format; // big-endian format
    unsigned short  tracks; // number of tracks, big-endian
    unsigned short  ticks;  // number of ticks per quarter note, big-endian
};
```

Figure 2. MIDI header data structure

It is necessary to check the identifier in the MIDI header to make sure the file content is indeed a MIDI file.

1. Size defines the binary file size in bytes, in big endian format.
2. Format defines the track format:
   a) 0 - single-track
   b) 1 - multiple tracks, synchronous
   c) 2 - multiple tracks, asynchronous
3. Tracks defines the number of tracks in the MIDI file.
4. Ticks defines the number of ticks per quarter note, more on this later.

The header chunk is immediately followed by one or more track chunks:

```
struct MIDITrackInfo
{
    unsigned int    id;     // identifier "MTrk"
    unsigned int    length; // track length, big-endian
};
```

Figure 3. MIDI track info data structure

1. Again, the identifier is important to make sure data is correctly read.
2. Length defines the size of the track in bytes, in big-endian format.

Immediately followed by the track info chunk is a sequence of MIDI events for the track. Each MIDI event consists of a command or status byte with associated data, and a time-stamp indicating the number of ticks to wait before sending the event.

Because we plan to perform note manipulation with the MIDI stream, we create a MIDIEvent struct to ensure every event is packed cleanly:

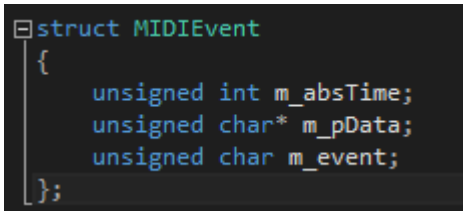

Figure 4. MIDI event data structure

MIDIEvent comes in large quantities and is frequently accessed, so it is important to store a pointer to the data, instead of storing data of the entire event.

1. absTime indicates the absolute time of the occurrence of the event. We need to calculate the absolute time through incrementally adding the previously processed MIDI events.
2. pData is a pointer that points to the buffer of the supplemental data of the MIDI event. We can use this pointer to feed information to the MIDI device at the time given by absTime.

3. The event attribute is the actual MIDI event. We parse the event from the data buffer according to the MIDI standard table.

Once we have a list of MIDI events for one track, we can pack them into a MIDITrack class for object-oriented design and flexibility later in development:

```
struct MIDITrack
{
    MIDITrackInfo* m_pTrackInfo;
    unsigned char* m_pBuffer;
    unsigned char m_lastEvent;
    unsigned int m_absTime;
};
```

Figure 5 MIDI track data structure

MIDITrack serves as a state machine that stores the playback state of MIDI buffers. These MIDI buffers are represented as pointers in MIDITracks, and these pointers record the memory location of the last played MIDIEvents.

1. pTrackInfo contains the information about the MIDI track, as described in *figure 3.* It is necessary for us to know the length of the track and also double check the data buffer before reading to prevent playing with erroneous data.
2. pBuffer points at the beginning of the track buffer. We use this to locate all MIDI events in the track and convert them into our custom MIDIEvents.
3. The lastEvent attribute is used for the MIDI device to play the same event in running mode. Running mode is used to conserve command data when the same commands are played in sequence. Instead of pairing all data bytes with a status

byte, running mode allows removing the redundant status bytes after the first one, so that it keeps using the “lastEvent” but with different data.

Now the basic parts of parsing a MIDI file are complete. The four major parts are MIDI header info, MIDI track info, MIDI Event, and MIDI Track. Lastly, in order to process the entire file, we shall use a ‘while’ loop to parse small chunks of MIDI data, pack them into MIDI events, and eventually break out of the loop when there are no more events to process:

```
while (TRUE)
{
    // get the nearest next event from all tracks

    // Update track abs time to match the latest processed event
    // update global delta time and current time

    // | 31 - 24 | 23 - 16 | 15 - 8 |    7 - 0    |
    // -------------------------------------------------
    // | Unused  | Data 2  | Data 1 | Status byte |
    // Read midi command into the message
    // Put midi command in midi event and respective midi track
    if (done)
    {
        break;
    }
}
```

Figure 6. MIDI Playback loop

In more detail, the execution flow of the playback ‘while’ loop is:

1. For each track:

    a) If the track is at the end-of-track marker, do nothing.

b) If all tracks are at the end-of-track marker, end playback.

2. Extract the next event closest to the absolute time of the current track:
   a) Advance the track pointer to the next event in the track.
   b) Advance the absolute time for the track by the extracted event's delta-time.
3. The difference between the new absolute time for the track and the absolute time for the score is used as the new delta-time for the event.
4. Continue from step 1.

Every execution of the MIDI playback loop, or tick, is precisely timed by the number of microseconds per tick. In order to calculate this value, we must first obtain the tempo value. In MIDI files, tempo is expressed in microseconds per quarter note. Beats per minute is calculated through use of this value. The default BPM of MIDI files is specified as 120. The equation for calculating microseconds per quarter note is given below:

$$\frac{x\,microseconds\,per\,minute}{y\,beats\,per\,minute} = z\,\frac{microseconds}{quarternote}$$

Figure 7. Microseconds per quarter note formula

In the case of 120 BPM, the result would be:

$$\frac{60\frac{sec}{min}\cdot 1,000,000\frac{microseconds}{sec}}{120\frac{beats}{min}} = 500,000\frac{microseconds}{quarternote}$$

Figure 8. An example of using microseconds per quarter note formula.

Going back to the MIDIHeader struct displayed by *Figure 1*, the ticks attribute specifies the ticks per quarter note used during playback. Tick per quarter note is also called pulses per quarter note. This is the number of CPU clock ticks per quarter note. The PPQN number will affect the precision of playback, and it usually set to 96. Below is an example of 96 pulses at 120BPM:

$$\frac{500,000\frac{microseconds}{quarter\ note}}{96pulses\ /quarter\ note} = 5208microseconds/tick.$$

Figure 9. Pulses per quarter note formula

Results

The sample program is able to play MIDI files independent from existing MIDI libraries. It packs MIDI information into C++ objects and STL containers to make MIDI data much more accessible when it comes to analyzing those data algorithmically.

Sample program instructions:

1. Compile the program with Microsoft Visual Studio, or other IDEs supported by CMake.
2. Run the program, it should then display a window as such:

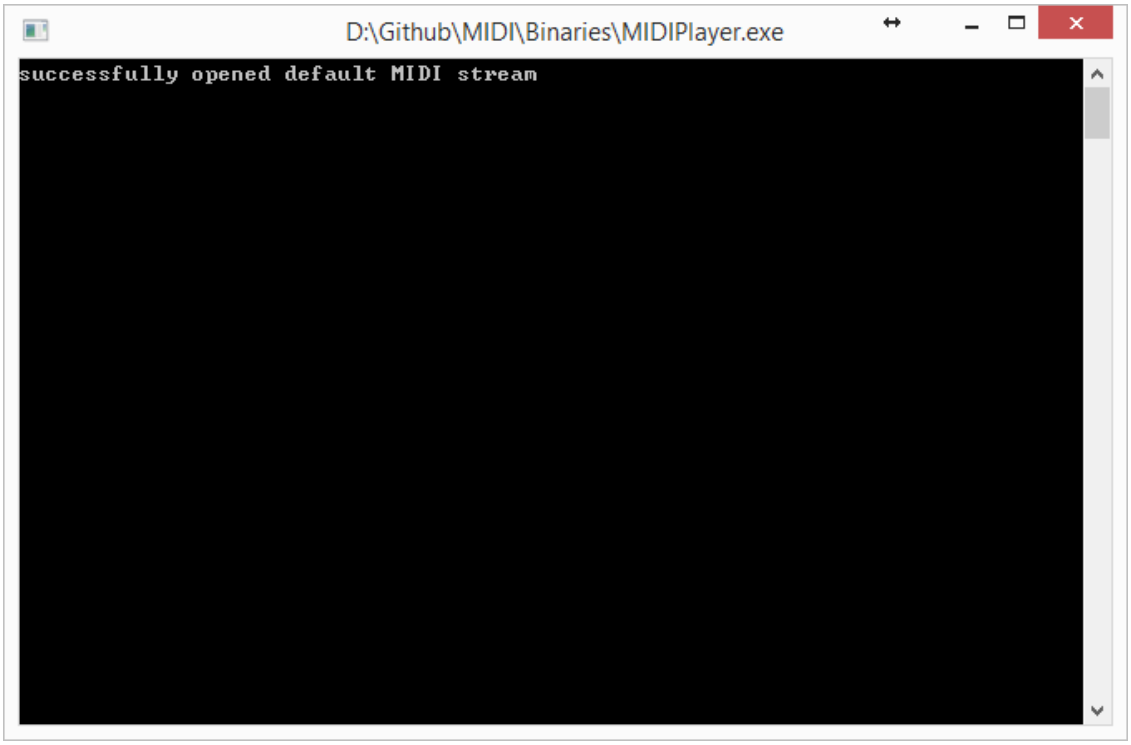

Figure 10. MIDIPlayer console

3. To add a track to the MIDI player, press the "**a**" key, it should then prompt for the MIDI filename:

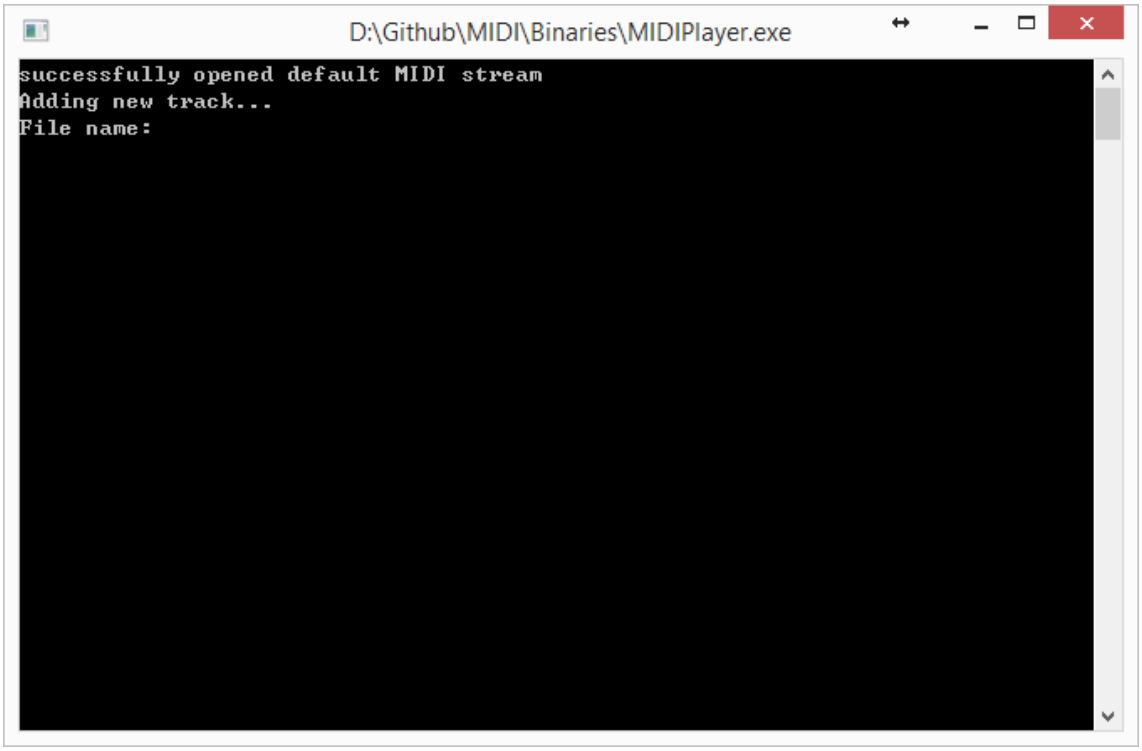

Figure 11. Adding a new track in the MIDIPlayer.

4. After typing the filename, press the **enter** key to submit input. If you would like to play multiple files, repeat step 3 and 4.
5. Press the **space** key to play the MIDI file list. It will the loop through all MIDI files stored in the MIDI player.

To make the music interactive, users shall implement their algorithms, retrieve the MIDI data from the MIDI player, and either change the MIDI data being played, or insert a new MIDI track generated from the algorithms to the playlist.

Conclusions and Recommendations

The original goal of this project was to develop a run-time MIDI transition algorithm. However, existing MIDI libraries were either written in C, or did not provide enough freedom to access the MIDI data. Due to the lack of appropriate MIDI libraries, the decision was made to develop a MIDI playback framework especially designed with run-time MIDI manipulation in mind. We have successfully developed a fully functional MIDI playback framework that meets the requirements of run-time MIDI manipulation.

With this MIDI playback framework, users can look into incorporating their own algorithms without worrying about hexadecimal data in the underlying MIDI files. The industry can also look into adding run-time MIDI manipulation support to existing music production software products. Perhaps a new editor can be developed to define transition behaviors such as, for example, transitioning to a minor version of the track if the camera transitions from a peaceful forest to a fearful dark cave.

Currently, music production software such as Logic, Cubase, and Ableton all provide a visual editor to access the underlying MIDI data, but the sound library required to render the final product requires purchase, and is often very expensive. To achieve high quality, real-time rendered audio, every client computer must come with a

set of sound libraries required by the files. This problem can be addressed by starting an open source sound library project to provide competitive sound libraries with little to no cost.

References

CaldwellDustin.
http://wiki.fourthwoods.com/standard_midi_file_format. 2011 November 02 .

CaronFrank.
http://arstechnica.com/news.ars/post/20080618-gaming-expected-to-be-a-68-billion-business-by-2012.html. 2008 June 18 . arstechnica.

CattermoleTannith. http://www.gizmag.com/computer-music-pioneer-max-mathews/18530/. 2011 May 9 . gizmag.

IntraviaLaura.
http://www.lauraintravia.com/blog/super-mario-galaxy-interview-with-the-sound-team-iwata-asks. 2011 February 5 .

MenneckeThomas.
http://www.slyck.com/story1436_MPAA_Reports_Record_Movie_Sales_in_2006. 2007 March 6 . Slyck.com.

midi.org.
http://www.midi.org/aboutmidi/tut_midifiles.php. MIDI Manufacturers Association.

United States Patent and Trademark Office
http://patft.uspto.gov/netacgi/nph-Parser?Sect1=PTO2&Sect2=HITOFF&p=1&u=%2Fnetahtml%2FPTO%2Fsearch-bool.html&r=14&f=G&l=50&co1=AND&d=PTXT&s1=music.ABTX.&s2=transition.ABTX.&OS=ABST/music+AND+ABST/transition&RS=ABST/music+AND+ABST/transition. 2003 August 3 .

RansonBen. http://www.elnexus.com/articles/GPUComputing.aspx. 2009 02 17 . ElectronicsNexus.

Reuters.
http://www.abc.net.au/news/stories/2008/03/06/2181568.htm. 2008 March 6 .

SwiftAndrew.
http://www.doc.ic.ac.uk/~nd/surprise_97/journal/vol1/aps2/.

Appendix

MIDIPlayer.h

```
#pragma once

#include <vector>
#include <windows.h>
#include "MIDIFile.h"


#define MAX_STREAM_BUFFER_SIZE (512)

class MIDIPlayer
{
public:
    MIDIPlayer();
    ~MIDIPlayer();

    void ProcessInput();

    void AddTrack();
    void Alloc2Slots(unsigned int& _currStreamBufLen);
    void DecodeCurrentA();
    void DecodeCurrentB();
    void USleep(int waitTime);
    static void CALLBACK MIDICallback(HMIDIOUT out,
    UINT msg, DWORD dwInstance, DWORD dwParam1, DWORD dwParam2);

    void Init();
    void Run();
    void Halt();
private:
    HMIDIOUT m_HOutDevice;
    unsigned int m_DeviceID;
    HMIDISTRM m_HMIDIOutStream;

    int m_currFileIndex;

    MIDIEVENT* m_pStreamBuf;
    unsigned int m_PulsesPerQuarterNote;
    int m_streamBufSize;


    bool m_isRequestingExit;
    bool m_isPlaying;
    std::vector<MIDIFile*> m_fileList;
};
```

Utility.h

```
#pragma once
#include "MIDIFile.h"

unsigned char* loadFile(const char* _fileName, int& _fileSize);

static unsigned int read_var_int(unsigned char* buf,
unsigned int* bytesread)
{
    unsigned int var = 0;
    unsigned char c;

    *bytesread = 0;

    do
    {
        c = buf[(*bytesread)++];
        var = (var << 7) + (c & 0x7f);
    } while (c & 0x80); //1000 0000

    return var;
}

static unsigned short byteSwapShort(unsigned short in)
{
    return ((in << 8) | (in >> 8));
}

static unsigned int byteSwapInt(unsigned int in)
{
    unsigned short *p;
    p = (unsigned short*)&in;

    return ((((unsigned int)byteSwapShort(p[0])) << 16) |
        (unsigned int)byteSwapShort(p[1]));
}
```

Utility.h (Continuted)

```
static struct MIDIEvent getNextEvent(const struct MIDITrack* track)
{
    unsigned char* buf;
    struct MIDIEvent e;
    unsigned int bytesread;
    unsigned int dt;

    buf = track->m_pBuffer;

    dt = read_var_int(buf, &bytesread);
    buf += bytesread;

    e.m_absTime = track->m_absTime + dt;
    e.m_pData = buf;
    e.m_event = *e.m_pData;

    return e;
}

static int isTrackEnd(const struct MIDIEvent* e)
{
    if (e->m_event == 0xff) // meta-event?
    if (*(e->m_pData + 1) == 0x2f) // track end?
        return 1;

    return 0;
}
```

MIDIFile.h

```
#pragma once
#include <vector>

#pragma pack(push, 1)

struct MIDIHeaderInfo
{
    unsigned int    id;     // identifier "MThd"
    unsigned int    size;   // always 6 in big-endian format
    unsigned short  format; // big-endian format
    unsigned short  tracks; // number of tracks, big-endian
    unsigned short  ticks;  // number of ticks per quarter note
                            // , big-endian
};

struct MIDITrackInfo
{
    unsigned int    id;     // identifier "MTrk"
    unsigned int    length; // track length, big-endian
};

#pragma pack(pop)

struct MIDITrack
{
    MIDITrackInfo* m_pTrackInfo;
    unsigned char* m_pBuffer;
    unsigned char m_lastEvent;
    unsigned int m_absTime;
};

struct MIDIEvent
{
    unsigned int m_absTime;
    unsigned char* m_pData;
    unsigned char m_event;
};
```

MIDIFile.h (Continued)

```
class MIDIFile
{
public:
    MIDIFile();

    ~MIDIFile();

    int initMIDIFile(const char* _fileName);

public:
    int m_fileSize;
    unsigned char* m_pFileBuf;
    int m_PulsesPerQuarterNote;

    MIDIHeaderInfo* m_pHeader;
    MIDITrackInfo* m_pBody;

    std::vector<MIDITrack> m_tracks;
};
```

MIDIFile.cpp

```cpp
#include <stdio.h>
#include <stdlib.h>
#include "MIDIFile.h"
#include "Utility.h"

MIDIFile::MIDIFile()
{}

MIDIFile::~MIDIFile()
{
    free(m_pFileBuf);
}

int MIDIFile::initMIDIFile(const char* _fileName)
{
    m_pFileBuf = loadFile(_fileName, m_fileSize);
    if (!m_pFileBuf)
    {
        printf("something went wrong during file load.\n");
        return 1;
    }

    //1: process header
    m_pHeader = (MIDIHeaderInfo*)m_pFileBuf;
    int numTracks = byteSwapShort(m_pHeader->tracks);
    m_PulsesPerQuarterNote = byteSwapShort(m_pHeader->ticks);

    //2: process body
    unsigned char* pBody = m_pFileBuf += sizeof(MIDIHeaderInfo);
    auto pCurrentTrack = (char*) pBody;
    auto pCurrentTrackInfo = (MIDITrackInfo*)pCurrentTrack;
    for (int i = 0; i < numTracks; i++)
    {
        if (pCurrentTrackInfo->id != 0x6b72544d)//magic number: 0xkrTM  -> big endian MTrk
        {
            printf("Fatal Error: MTrk identifier not found when parsing file: %s\n", _fileName);
            return 1;
        }

        MIDITrack currentTrack;
        currentTrack.m_pTrackInfo = pCurrentTrackInfo;
        currentTrack.m_pBuffer = (unsigned char*)pCurrentTrack + sizeof(MIDITrackInfo);
        currentTrack.m_absTime = 0;
        currentTrack.m_lastEvent = 0;
        m_tracks.push_back(currentTrack);

        //increment pCurrentTrack to point to the next track
        pCurrentTrack += sizeof(MIDITrackInfo) + byteSwapInt(m_tracks[i].m_pTrackInfo->length);
    }
    return 0;
}
```

MIDIPlayer.cpp

```cpp
#include <stdio.h>
#include <conio.h>
#include <iostream>
#include <string>
#include <windows.h>
#include "MIDIPlayer.h"
#include "MIDIFile.h"
#include "Utility.h"

HANDLE g_event;

MIDIPlayer::MIDIPlayer()
{
    m_isRequestingExit = false;
    m_isPlaying = false;
    m_PulsesPerQuarterNote = 0;
    m_streamBufSize = 0;
    m_currFileIndex = 0;
    m_HMIDIOutStream = 0;
    m_DeviceID = 0;
}

MIDIPlayer::~MIDIPlayer()
{

}

void MIDIPlayer::Init()
{
    m_pStreamBuf = (MIDIEVENT*)malloc(sizeof(MIDIEVENT)* MAX_STREAM_BUFFER_SIZE);
    if (!m_pStreamBuf)
    {
        printf("Fatal Error: Unable to initialize buffer for MIDIPlayer.\n");
    }
    memset(m_pStreamBuf, NULL, sizeof(MIDIEVENT)* MAX_STREAM_BUFFER_SIZE);

    int hr = 0;

    //hr |= midiOutOpen(&m_HOutDevice, 0, 0, 0, CALLBACK_NULL);
    if (hr != MMSYSERR_NOERROR)
    {
        printf("error opening default MIDI stream: %d\n", hr);

    }
    else
    {
        printf("successfully opened default MIDI stream\n");
    }
}
```

MIDIPlayer.cpp (Continued)

```
void MIDIPlayer::Run()
{
    while (!m_isRequestingExit)
    {
        ProcessInput();
        if (m_isPlaying)
        {
            DecodeCurrentB();

            int i = 0;


            int err;
            HMIDIOUT out;
            unsigned int msg;

            err = midiOutOpen(&out, 0, 0, 0, CALLBACK_NULL);
            if (err != MMSYSERR_NOERROR)
                printf("error opening default MIDI device: %d\n", err);
            else
                printf("successfully opened default MIDI device\n");

            i = 0;
            while (i < m_streamBufSize)
            {
                unsigned int time = m_pStreamBuf[i].dwDeltaTime;

                USleep(time * m_PulsesPerQuarterNote);

                //get midi command
                MIDIEVENT event = *(MIDIEVENT*)&m_pStreamBuf[i];
                msg = event.dwEvent;
                if (msg & 0xff000000) // tempo change
                {
                    msg = msg & 0x00ffffff;
                    m_PulsesPerQuarterNote = msg / byteSwapShort(
                    (m_fileList[m_currFileIndex])->m_pHeader->ticks);
                }
                else
                {
                    err = midiOutShortMsg(out, msg);
                    if (err != MMSYSERR_NOERROR)
                        printf("error sending command: %08x error: %d\n", msg, err);
                }
                // Onto next message
                i++;
            }
```

MIDIPlayer.cpp (Continued)

```
            while (i < m_streamBufSize)
            {
                //ProcessInput();
                // get delta time from message
                unsigned int dt = m_pStreamBuf[i].dwDeltaTime;

                //wait before submitting the message
                USleep(dt * m_PulsesPerQuarterNote);

                //get midi command
                MIDIEVENT event = *(MIDIEVENT*)&m_pStreamBuf[i];
                unsigned int msg = event.dwEvent;
                if (msg & 0xff000000) // tempo change
                {
                    msg = msg & 0x00ffffff; // microseconds/quarter note
                    m_PulsesPerQuarterNote = msg / byteSwapShort(
                    (m_fileList[m_currFileIndex])->m_pHeader->ticks);
                }
                else
                {
                    int hr = midiOutShortMsg(m_HOutDevice, msg);
                    if (hr != MMSYSERR_NOERROR)
                    {
                        printf("error sending command: %d\n", hr);
                    }
                }

                // Onto next message
                i++;
            }


            //finished playing
            m_isPlaying = false;
        }

    }
}
```

MIDIPlayer.cpp (Continued)

```
void MIDIPlayer::Halt()
{
    midiOutClose(m_HOutDevice);
    free(m_pStreamBuf);
}

void MIDIPlayer::AddTrack()
{
    int hr;
    std::string trackName;
    //getline(std::cin, trackName);
    trackName = "example8.mid";
    auto newFile = new MIDIFile();
    hr = newFile->initMIDIFile(trackName.c_str());
    if (hr)
    {
        //error checking
        printf("File not found.\n");
        return;
    }
    m_fileList.push_back(newFile);
    printf("Track added: %s\n", trackName.c_str());
}

void MIDIPlayer::DecodeCurrentA()
{
    //reset stream buf size
    m_streamBufSize = 0;
    m_PulsesPerQuarterNote = m_fileList[m_currFileIndex]->m_PulsesPerQuarterNote;
    unsigned char* pFileBuf = m_fileList[m_currFileIndex]->m_pFileBuf;
    pFileBuf += sizeof(MIDITrackInfo);


    while (pFileBuf < m_fileList[m_currFileIndex]->m_pFileBuf +
           m_fileList[m_currFileIndex]->m_fileSize)
    {
        unsigned int msg = 0;

        // |       X - 0      |
        // ------------------------------------------------
        // |  Variable Sized  |
        // Read the delta time variable first
        unsigned int bytesread;
        unsigned int dt = read_var_int(pFileBuf, &bytesread);
        pFileBuf += bytesread;

        // | 31 - 24 | 23 - 16 | 15 - 8 |    7 - 0    |
        // ------------------------------------------------
        // | Unused  | Data 2  | Data 1 | Status byte |
        // Read midi command into the message
        unsigned char cmd = *pFileBuf;
        msg |= cmd;                                     // 0x000000FF
```

MIDIPlayer.cpp (Continued)

```
            // Onto Data1 byte
            pFileBuf++;
            unsigned char data1 = *pFileBuf;

            // if normal command
            if ((cmd & 0xf0) != 0xf0)
            {
                // all normal commands have data1
                msg |= ((unsigned int)(data1 << 8));        // 0x0000FF00

                //Onto Data2 byte
                pFileBuf++;
                unsigned char data2 = *pFileBuf;

                // 0xc0 (Patch change) and 0xd0 (Channel pressure) only take a single byte
                if (!((cmd & 0xf0) == 0xc0 || (cmd & 0xf0) == 0xd0))
                {
                    // Does have data2
                    msg |= ((unsigned int)(data2 << 16));   // 0x00FF0000

                    //Onto the next message
                    pFileBuf++;
                }

                //m_pStreamBuf[m_streamBufSize++] = dt;
                //m_pStreamBuf[m_streamBufSize++] = msg;
            }
            // else if meta-event
            else if (cmd == 0xff)
            {
                cmd = *pFileBuf++; // cmd should be meta-event (0x2f for end of track)
                cmd = *pFileBuf++; // cmd should be meta-event length
                pFileBuf += cmd;
            }
        }
    }
```

MIDIPlayer.cpp (Continued)

```
void MIDIPlayer::Alloc2Slots(unsigned int& _currStreamBufLen)
{
    while (m_streamBufSize + 1 > _currStreamBufLen)
    {
        unsigned int* tmp = NULL;
        _currStreamBufLen *= 2;
        tmp = (unsigned int*)realloc(m_pStreamBuf, sizeof(MIDIEVENT) * _currStreamBufLen);
        if (tmp != NULL)
        {
            //m_pStreamBuf = tmp;
        }
        else
        {
            if (m_pStreamBuf)
            {
                free(m_pStreamBuf);
            }
            return;
        }
    }
}

void MIDIPlayer::DecodeCurrentB()
{
    int hr = 0;
    //Set up midi stream property based on new file
    MIDIPROPTIMEDIV MIDIStreamProperty;
    MIDIStreamProperty.cbStruct = sizeof(MIDIPROPTIMEDIV);
    MIDIStreamProperty.dwTimeDiv = byteSwapShort(
    m_fileList[m_currFileIndex]->m_pHeader->ticks);
    if (hr != MMSYSERR_NOERROR)
    {
        printf("error on setting MIDIStreamProperty: %d\n", hr);

    }
    else
    {
        printf("successfully set MIDIStreamProperty\n");
    }

    //allocate stream buffer
    MIDIHDR MIDIHeader;
    MIDIHeader.lpData = (char*)m_pStreamBuf;
    MIDIHeader.dwBufferLength = MIDIHeader.dwBytesRecorded = MAX_STREAM_BUFFER_SIZE;
    MIDIHeader.dwFlags = 0;
    //hr = midiOutPrepareHeader((HMIDIOUT)m_HMIDIOutStream, &MIDIHeader, sizeof(MIDIHDR));
    if (hr != MMSYSERR_NOERROR)
    {
        printf("error on setting MIDIHeader: %d\n", hr);

    }
    else
    {
        printf("successfully set MIDIHeader\n");
    }

    // Initialize default values for streaming
    m_PulsesPerQuarterNote = 500000 / m_fileList[m_currFileIndex]->m_PulsesPerQuarterNote;
    unsigned int currTime = 0;
    unsigned int currStreamSize = MAX_STREAM_BUFFER_SIZE;
```

MIDIPlayer.cpp (Continued)

```
    // windows format midi event
    MIDIEVENT newEvent;

    static int vivi = 0;

#define TEMPO_EVT 1
    while (TRUE)
    {
        unsigned int dt = (unsigned int)-1;
        unsigned int nearestIndex = -1;
        MIDIEvent MIDIEvent;
        std::vector<MIDITrack>& tracks = m_fileList[m_currFileIndex]->m_tracks;

        // get the nearest next event from all tracks
        for (int i = 0; i < tracks.size(); i++)
        {
            auto tmpEvent = getNextEvent(&tracks[i]);

            if (!(isTrackEnd(&tmpEvent)) && (tmpEvent.m_absTime < dt))
            {
                MIDIEvent = tmpEvent;
                dt = tmpEvent.m_absTime;
                nearestIndex = i;
            }
        }

        // if nearestIndex == -1 then all the tracks have been read up to the end of track mark
        if (nearestIndex == -1)
        {
            break;
        }

        vivi++;
        printf("itr:%d\n", vivi);
        if (tracks.size() > 1)
        {
            printf("wtf\n");
        }

        newEvent.dwStreamID = 0; // always 0
        newEvent.dwParms[0] = 0;

        // Update track abs time to match the latest processed event
        tracks[nearestIndex].m_absTime = MIDIEvent.m_absTime;
        // update global delta time and current time
        newEvent.dwDeltaTime = tracks[nearestIndex].m_absTime - currTime;
        currTime = tracks[nearestIndex].m_absTime;

        // running mode
        if (!(MIDIEvent.m_event & 0x80)) // not above 1000 0000, so not 0x80 or 0x90
        {
            //get last command
            unsigned char lastEvent = tracks[nearestIndex].m_lastEvent;
            newEvent.dwEvent = ((unsigned long)lastEvent);

            // get Data1 byte
            unsigned char data1 = *(MIDIEvent.m_pData);

            // all normal commands have data1
            newEvent.dwEvent |= ((unsigned int)(data1 << 8));       // 0x0000FF00
```

MIDIPlayer.cpp (Continued)

```
            //Onto Data2 byte
            (MIDIEvent.m_pData)++;
            unsigned char data2 = *(MIDIEvent.m_pData);

            // 0xc0 (Patch change) and 0xd0 (Channel pressure) only take a single byte
            if (!((lastEvent & 0xf0) == 0xc0 || (lastEvent & 0xf0) == 0xd0))
            {
                // Does have data2
                newEvent.dwEvent |= ((unsigned int)(data2 << 16));  // 0x00FF0000

                //Onto the next message
                (MIDIEvent.m_pData)++;
            }

            // cast stream buffer pointer to windows MIDIEVENT pointer
            Alloc2Slots(currStreamSize);
            *(MIDIEVENT*)(&m_pStreamBuf[m_streamBufSize]) = newEvent;
            m_streamBufSize++;

        }
        else if (MIDIEvent.m_event == 0xff) // meta-event
        {
            MIDIEvent.m_pData++; // skip the event byte
            unsigned char meta = *MIDIEvent.m_pData++; // read the meta-event byte
            unsigned int len;

            switch (meta)
            {
            case 0x51: // set tempo
            {
                    unsigned char a, b, c;
                    len = *MIDIEvent.m_pData++; // get the length byte, should be 3
                    a = *MIDIEvent.m_pData++;
                    b = *MIDIEvent.m_pData++;
                    c = *MIDIEvent.m_pData++;

                    newEvent.dwEvent = ((unsigned long)TEMPO_EVT << 24) |
                        ((unsigned long)a << 16) |
                        ((unsigned long)b << 8) |
                        ((unsigned long)c << 0);

                    // cast stream buffer pointer to windows MIDIEVENT pointer
                    Alloc2Slots(currStreamSize);
                    *(MIDIEVENT*)(&m_pStreamBuf[m_streamBufSize]) = newEvent;
                    m_streamBufSize++;
            }
                break;
            case 0x00:
            case 0x01:
            case 0x02:
            case 0x03:
            case 0x04:
            case 0x05:
            case 0x06:
            case 0x07:
            case 0x21:
            case 0x2f: // end of track
```

MIDIPlayer.cpp (Continued)

```
            case 0x2f: // end of track
            case 0x54:
            case 0x58: // time signature
            case 0x59: // key signature
            case 0x7f: // sequencer specific information
            default:
                len = *MIDIEvent.m_pData++; // ignore event, skip all data
                MIDIEvent.m_pData += len;
                break;
            }
        }
        // if normal command
        else if ((MIDIEvent.m_event & 0xf0) != 0xf0)
        {
            tracks[nearestIndex].m_lastEvent = MIDIEvent.m_event;

            // | 31 - 24 | 23 - 16 | 15 - 8 |    7 - 0    |
            // -------------------------------------------------
            // | Unused  | Data 2  | Data 1 | Status byte |
            // Read midi command into the message
            unsigned char cmd = *(MIDIEvent.m_pData);
            newEvent.dwEvent = cmd;                                          // 0x000000FF

            // Onto Data1 byte
            (MIDIEvent.m_pData)++;
            unsigned char data1 = *(MIDIEvent.m_pData);


            // all normal commands have data1
            newEvent.dwEvent |= ((unsigned int)(data1 << 8));       // 0x0000FF00

            //Onto Data2 byte
            (MIDIEvent.m_pData)++;
            unsigned char data2 = *(MIDIEvent.m_pData);

            // 0xc0 (Patch change) and 0xd0 (Channel pressure) only take a single byte
            if (!((cmd & 0xf0) == 0xc0 || (cmd & 0xf0) == 0xd0))
            {
                // Does have data2
                newEvent.dwEvent |= ((unsigned int)(data2 << 16));  // 0x00FF0000

                //Onto the next message
                (MIDIEvent.m_pData)++;
            }

            // cast stream buffer pointer to windows MIDIEVENT pointer
            Alloc2Slots(currStreamSize);
            *(MIDIEVENT*)(&m_pStreamBuf[m_streamBufSize]) = newEvent;
            m_streamBufSize ++;

        }
        else
        {
            // not handling sysex events yet
            printf("unknown event %2x", MIDIEvent.m_event);
            exit(1);
        }

        //increment pointer
        tracks[nearestIndex].m_pBuffer = MIDIEvent.m_pData;

    }
}
```

MIDIPlayer.cpp (Continued)

```
void MIDIPlayer::ProcessInput()
{
    int key;

    if (kbhit())
    {   // If a key on the computer keyboard has been pressed
        key = _getch();

        if (m_isPlaying)
        {
            switch (key)
            {
            case 'a':
                //blocking
                printf("Adding new track...\n");
                printf("File name: ");
                AddTrack();
                break;
            case ' ':
                m_isPlaying = true;
                break;
            case 'q':
                //non-blocking
                printf("walawala\n");
                m_isRequestingExit = false;
                break;
            default:
                break;
            }
        }
        else
        {
            switch (key)
            {
            case 'a':
                //blocking
                printf("Adding new track...\n");
                printf("File name: ");
                AddTrack();
                break;
            case ' ':
                m_isPlaying = true;
                break;
            case 'q':
                //non-blocking
                printf("walawala\n");
                m_isRequestingExit = false;
                break;
            default:
                break;
            }
        }
    }
}
```

MIDIPlayer.cpp (Continued)

```
void MIDIPlayer::USleep(int waitTime)
{
    LARGE_INTEGER time1, time2, freq;

    if (waitTime == 0)
        return;

    QueryPerformanceCounter(&time1);
    QueryPerformanceFrequency(&freq);

    do
    {
        QueryPerformanceCounter(&time2);
    } while ((time2.QuadPart - time1.QuadPart) * 1000000ll / freq.QuadPart < waitTime);
}

void CALLBACK MIDIPlayer::MIDICallback(HMIDIOUT out,
UINT msg, DWORD dwInstance, DWORD dwParam1, DWORD dwParam2)
{
    switch (msg)
    {
    case MOM_DONE:
        SetEvent(g_event);
        break;
    case MOM_POSITIONCB:
    case MOM_OPEN:
    case MOM_CLOSE:
        break;
    }
}
```

Utility.cpp

```cpp
#include "Utility.h"
#include <iostream>

unsigned char* loadFile(const char* _fileName, int& _fileSize)
{
    unsigned int hr;
    unsigned char* fileBuf;
    int fileSize;
    FILE* fd = fopen((char*)_fileName, "rb");
    if (fd == NULL)
        return nullptr;

    fseek(fd, 0, SEEK_END);
    fileSize = ftell(fd);
    fseek(fd, 0, SEEK_SET);

    fileBuf = (unsigned char*)malloc(fileSize);
    if (fileBuf == 0)
    {
        fclose(fd);
        return nullptr;
    }

    hr = fread(fileBuf, 1, fileSize, fd);
    fclose(fd);

    if (hr != fileSize)
    {
        printf("fread returned wrong file size.\n");
        free(fileBuf);
        return nullptr;
    }

    _fileSize = fileSize;
    return fileBuf;
}
```

Main.cpp

```
#include "MIDIPlayer.h"

///*
int main()
{
    MIDIPlayer gMIDIPlayer;
    gMIDIPlayer.Init();
    gMIDIPlayer.Run();
    gMIDIPlayer.Halt();
}
//*/
```