Abstract

Realtime music visualization has become very interesting, since modern technology enables us to examine and dividing audio signals very fast. Nearly every software media player available features music visualization, whether it is a simple graph or a complex three-dimensional animation. However, realtime audio signal processing has its limits. In this paper, we present another approach to realtime music visualization. By using MIDI data instead of audio signals, we can concentrate on every single note that is played.

1 Motivation

The main problem of realtime audio signal processing is the nature of its results. They only represent the signal as a whole, or maybe divided into its main frequency spectrum. If an orchestral piece of music is played, traditional technologies would not be able to extract e.g. the notes of the first violin from the rest of the orchestra in a 100% clean way. There exist some projects aiming to extract notes out of audio files, but the more instruments or voices are playing, the higher the error rate grows.

We want an application that focuses on every single note played in a piece of music. By using MIDI data, which is generated by nearly every digital musical instrument, we are enabled to visualize every tone, since MIDI data focuses on the note, not on the sound. A more detailed description of the MIDI protocol follows. Of course we are restricted to MIDI enabled instruments only, but we can achieve very good results and reach a broad application spectrum with our application.

The main project goals are:

1. Demonstration of the possibilities of MIDI visualization
2. Platform independence (Windows, OS X, Linux)
3. Reusability of the source code
4. Ease of use

In the next chapter we make some definitions, followed by a description of the software architecture. After presenting some results, we finally want to give a look into the future.

2 Architecture

2.1 Definitions

Before we get into detail, we want to make certain definitions of the terms we use in this paper.

2.1.1 Plugin

Our application gives the user the possibility to choose between two visualizations, which we call "plugins". A plugin, as we mean it, is not a plugin in the real sense, because it is actually hard-coded into the application and cannot be loaded dynamically into the system. A decent plugin system is one of the future goals, which will be discussed later.

2.1.2 Preset

Every plugin features a settings dialog which enables the user to change the way the plugin looks like and acts to MIDI messages. These settings can be stored into so-called "presets", which show up in the preset list in the main application window, along with the plugins.

2.2 Technologies

We like to give a short overview of the technologies used in our project. MIDI is the most important one, since we want to visualize it. To extend the visualization from the screen to other media, we use the DMX protocol to control lights according to the notes played by the user.

2.2.1 MIDI

The Musical Instrument Digital Interface (MIDI) is a standard which was introduced in the 1980’s to make it possible for musical instruments of different kind and manufacturer to communicate with each other. In contrast to normal audio signal processing, where the real sound of an instrument is transmitted, the MIDI
signal tells the receiver which note is played or released, with certain parameters. These parameters describe the nature of the tone, such as velocity and vibrato. Additionally, system messages can be transmitted. For instance, the "program change" message tells the receiver which instrument to choose for the note messages. The General MIDI standard provides a list of instruments, so that every MIDI file uses the same instruments on different MIDI tone generators.

MIDI.live receives MIDI signals from an attached keyboard or other MIDI devices to visualize the notes played by the musician.

2.2.2 DMX

DMX, or Digital MultipleX, is a technology used to control light fixtures. Modern stage lighting involves complex lighting devices with lots of functions, which have to be controlled in a standardized way. In terms of signal transmission technology, DMX is very similar to MIDI, since both are based on serial communication. We do not want to go too much into detail, as it is not necessary to explain the technical details of these technologies.

MIDI.live features an experimental module which generates DMX signals according to the notes played by the user.

2.3 Application Structure

![Figure 1: Basic application structure.](image)

The basic structure of the application is visualized in Figure 1. It consists of three main parts: The MIDI module manages all signals coming from the machine’s MIDI input device, which receives data from the MIDI device the user is playing on. The output part consists of the rendering interface, which is implemented by the plugins. Both input and output parts are controlled by the system core.

2.3.1 MIDI Module

During application startup, Java’s integrated MIDI libraries are used to detect input devices. The MIDI inputs are then listed in a combobox at the top of the main window, so that the user can choose the appropriate one. On Linux and Windows operating systems nearly all MIDI devices are recognized, on OS X a special library has to be included to access the operating system’s MIDI core. An input device can be a virtual one, such as an on-screen keyboard, or a real one, such as a stage piano or a MIDI drum set. A typical scenario can be seen in Figure 2.

A MIDI message can hold various types of information, as you can see in Figure 3. Currently, MIDI/live supports the following types of messages: note, pitchbend, aftertouch, polyphonic aftertouch, program change and control change.

A note event occurs when the player presses or releases a key. The message contains information on the key number and the strength with which the key is pressed.

Pitchbend messages are transmitted when the user performs a pitchbend, usually with a joystick, which means that all current notes are bend up or down.

An aftertouch message is initiated when the user performs pressure on a key which is already held down. Although this effect cannot be used on real pianos, it is a commonly used method to create interesting effects on modern synthesizers. Whereas the normal aftertouch message effect all played notes at a time, polyphonic aftertouch messages affect each note independently. That means that for each polyphonic aftertouch message the associated note is included in the message.

A program change message tells the receiver which instrument to choose. As mentioned in the definition section, there are standardized instrument collections, so that every MIDI song uses the same instruments on different instruments.

The shape of a sound can be altered in many ways. For that purpose, control change messages transmit information on how an instrument should sound, i.e. various types of filters can be applied.

2.3.2 System Core

The system core manages the MIDI input section, processes messages and controls the plugins. The whole functionality can be accessed through the main application window, which provides an input selector, a list of stored plugins and presets, and additional controls for debugging purposes and other features.

2.4 Plugins

To demonstrate the possibilities of realtime MIDI visualization we implemented two plugins. The first one focuses on the nature of the current note played by using bloopy objects which can interact with
each other. The second one is an extended version of the classical bar animation, where each played note is represented by a bar.

2.4.1 Procedural Texture Generation

A Perlin noise generator was implemented to texture the objects. The big advantage of procedural noise textures is, that they have not to be designed or photographed before. This textures are generated when the program starts. The textures are produced by random numbers. Because pure noise images (figure 4 left) looks too undefined, an interpolation between the pixels produces a smoother and more natural appearance of the noise image. (Figure 4).

The perlin noise algorithm simply generates interpolated noise images of different resolutions. (Figure 5).

The weighed sum of the generated images leads to the result image. (Figure 6).

Tri-linear interpolation was used to generate moving, changing textures. Parameter variation (color, noise resolution, coefficients of the weighted sum,) leads to different appearence of the perlin noise textures. Its also possible to execute the perlin noise with a sine texture as base. (Figure 7). The result of this procedure can be seen in Figure 8.

In midilive a program was written which always produces new random generated textures. The so-called Noise Machine has two arrays of texture buffers. In one array four sequences of normal perlin textures are generated. In the second array four sequences of sine-perlin textures are generated. If the i-th sequence is displayed, then the (i-1)-th sequence is generated. If the i-th sequence reaches the end then the texture sequence is interpolated to the (i+1)-th sequence and the (i+1)-th will be displayed.

2.4.2 Blobby Objects

The surface of metaballs is defined by an isovalue of a three-dimensional scalar field. So it is possible to render merging or melting objects if the field of several objects is added together. Blobby objects, metaballs and soft objects have the same underlying principle. They differ only in the computation formula of the scalar field and the spatial domain. We used the following (gravity) formula for each ball:

\[
\text{Energy}(x, y, z, i) = \frac{\text{ball}[i].\text{mass}}{\text{sqDistance}(x, y, z)}
\] (1)
In each grid point \((x,y,z)\) on the voxel grid the gravity formula is computed for each ball \(i\). Each ball has a position and a mass. If the squared distance from the point \((x,y,z)\) to the ball center increases, the energy(field value) decreases. Finally the energy fields of all balls \(i\) are added together which results in the energy field \(e(x,y,z)\). Each ball has a mass which is only a scale factor for the influence of one ball to the resulting energy field. A threshold is used to extract the voxels which are lying on the surface. The energy of all eight grid points of a voxel is compared with the treshold level. If one voxel has grid points which are under as well as over the threshold level, then this voxel is marked. All marked voxels build the surface. The marching cubes algorithm, which will be described here more exactly, was used to build the triangle list for rendering in openGL. The normal vectors were computed by deriving the energy formula. See Figure 9 for a screenshot of the metaballs in MIDI.live.

In each grid point \((x,y,z)\) on the voxel grid the gravity formula is computed for each ball \(i\). Each ball has a position and a mass. If the squared distance from the point \((x,y,z)\) to the ball center increases, the energy(field value) decreases. Finally the energy fields of all balls \(i\) are added together which results in the energy field \(e(x,y,z)\). Each ball has a mass which is only a scale factor for the influence of one ball to the resulting energy field. A threshold is used to extract the voxels which are lying on the surface. The energy of all eight grid points of a voxel is compared with the threshold level. If one voxel has grid points which are under as well as over the threshold level, then this voxel is marked. All marked voxels build the surface. The marching cubes algorithm, which will be described here more exactly, was used to build the triangle list for rendering in openGL. The normal vectors were computed by deriving the energy formula. See Figure 9 for a screenshot of the metaballs in MIDI.live.

Each ball has an acceleration, a speed and position vector for free movement in space. There are three different modes of movements and MIDI input reactions for the metaballs. In the first mode the acceleration vector of each ball gets redefined by random after a predefined time interval. That effects that the balls always move by random. If a ball hits a spatial border, the speed is set to zero and a new acceleration vector is generated. In this mode the note-on rate is decoded the moving speed of the metaballs. If many notes are played very fast, the metaballs are moving faster. In the second mode the metaballs are also moving randomly if no note is played, like in mode 1. If a note is played the energy field gets recomputed so that all balls get connected through a cylinder. (Figure 10) The energy formula was extended, so that also the normal distance from the line between the balls is considered. A sine oscillation, which was added to the connection-energy between the balls, make the appearance more dynamically.

In the third mode the balls are positioned in a static configuration along a line. Only the start and end point have a mass. If a note is played the mass of the middle balls is increased so that the balls merge together to one object. The balls are re-positioned, so that the resulting object length is indirect proportional to the note value. (Figure 11) Midi control changes like attack time / decrease time/ portamento time is decoded to the speed of merging/smeling/re-positioning.

In this plugin light pads are arranged in a two dimensional grid on the floor. Each light pad is assigned to one note. The midi key range is adjustable with the user interface. The user can decide how many notes effects the visualization. Metaballs are displayed in

Figure 7: Basic sine pattern.

Figure 8: Resulting sine texture.

Figure 9: Metaballs screenshot.

Figure 10: Connected metaballs.

Figure 11: Different tones generate different distances between the metaballs.

2.4.3 Plugin ”Blobby”

In this plugin light pads are arranged in a two dimensional grid on the floor. Each light pad is assigned to one note. The midi key range is adjustable with the user interface. The user can decide how many notes effects the visualization. Metaballs are displayed in
the plugin. The user can also select the metaball mode (explained in the previous chapter) in the user interface. If a note is played the assigned light pad turns on, the metaballs perform they respons to the midi messaga and the magnetic light effects on the position of the metaball. The whole metaballs-object have also a acceleration, speed and position vector. If a note is played the metaballs get attracted by the light. If a light turns on, then the difference vector (scaled with the note velocity) between metaballs position and light position is added to the acceleration of the metaballs. If several notes are played at the same time, then the five notes with the highest velocity are taken for magnetic light-attraction. The user can also select between three different camera modes(Rotating camera/Rotating around metaballs/down-view). The metaballs and the background walls are textured with the perlin noise textures.

2.4.4 Plugin "Tunnel"

The Tunnel plugin is based on the typical representation of music: the score. Instead of printing notes on lines, this plugin visualizes the notes by rendering bars, where the bars’ positions are defined by the height of the tone. To extend the visual impression, the bars are placed on the walls of a three-dimensional tunnel. The tunnel shape can be "morphed" to a scene consisting of a bottom and a top plane. (Figure 12).

![Morphing from tunnel shape to plane shape.](image)

Figure 12: Morphing from tunnel shape to plane shape.

The bars’ appearance can be changed in terms of size, sharpness and color. The keyrange can be altered, so that if only two octaves of the keyboard are used, the 24 resulting bars will expand in distance from the left side of the tunnel to the right side. The symmetry of the bar animation can be set to three modes: Mode 1 displays them in a symmetrical way, so that the lowest note appears on the left side of the tunnel, both on the top and the bottom half. The asymmetrical mode lets the animation render the top half bar on the opposite side. The "single" mode only renders one set of bars on the bottom tunnel half.

The camera can be set to three viewing directions. It can look forwards, so that the bars appear in front of the camera and move towards it. Secondly, it can look backwards, so that the bars are disappearing in the background. Camera mode three creates a sidescrolling view, like it is used in Sony’s "SingStar" video game. A particle system has been implemented to extend the appearance of the bars. The particle color can be set to a rainbow mode to make the animation more colorful.

Metaballs can be added to scene and appear in the middle background of the tunnel to bring some movement into the scene.

The bar animation is based on events. The data structure of the bars is an of lists. Every key is represented by a list in that array. If a key is pressed a "note on" message is added to the appropriate list. If a key is released, the corresponding "note off" message is stored. All events start at position z=0.0f in the tunnel. The positions are negatively increased, so that the events move down the negative z-axis. In the rendering process, every event in every list is processed and bars are drawn from event to event. The particle system is very simple. It consists of a list of particles with a certain position, alpha value (=lifetime), color, gravity and dispersion. With every rendering process the particles positions are altered by regarding the main direction, dispersion and gravity. The particle is rendered by calling a small display list containing a quad with a texture of a white fuzzy dot. The particles can be depth-sorted before being rendered, unless depth-testing is disabled.

2.5 User Interface

The graphical user interface consists of the main window, the debug window, the visualization window and the current plugin’s settings dialog. If a projector or second monitor is connected to the computer, the visualization window can be moved to the projector’s (or second monitor’s) part of the extended desktop and can be switched to fullscreen mode. All other windows remain on the main desktop screen and allow continuous control over the visualization without overlapping it. See Figure 13 for a screenshot.

![Application screenshot after startup.](image)

Figure 13: Application screenshot after startup.

The main part of the application window is the list of plugins. Here, the user can load them, make settings and store or delete custom presets of the available plugins. Double-clicking an item of the list will load the appropriate visualization. By clicking the settings icon, the plugin’s corresponding configuration dialog will appear. See Figure 14 for a screenshot. It shows the configuration dialog for the tunnel plugin. This screenshot shows the learning function of MIDLive. A context menu was opened for the "green value" slider, where the user can assign a MIDI controller to the control component, so that he can move the slider by moving a slider on his MIDI device. MIDI controller assignments are also stored in presets.

The debug window is very simple and contains only a textbox with debug output and a button for playback of a test MIDI file, which
can be used to demonstrate the visualizations when no MIDI hardware is available.

Figure 14: A plugin configuration dialog.

3 Results

In this section we want to show a few renderings created by our application. A few images of the blobby plugin have already been shown earlier, so we add just another one, followed by screenshots of the tunnel plugin. (Figures 15-20)

Figure 15: Metaballs with a procedural texture.

Figure 16: Tunnel plugin in sidescrolling view with rainbow colored bars and particles.

The same setup in wireframe mode.

4 Project Future

The main goal of this project was to provide a demonstration of realtime music visualization with MIDI. To ensure that the software is applicable in many scenarios, it must support extensibility through a plugin system. In the future, we want to give the user the possibility to download or write his own visualization and load it in the application without having to recompile the whole MIDI.live source code. Users could upload their creations to a website, which can be accessed by a built-in plugin manager being able to download and install other users’ plugins. The same thing holds for DMX output plugins, which can access a series of controllers from various manufacturers.

The tunnel plugin can be extended to serve for educational purposes. For example, notes which do not go well together are rendered in red color, whereas good chords are rendered green. The plugin can also be implemented like Sony’s “SingStar” for use at music schools. The student has to replay a piece of music, and the plugin shows the player which notes are wrong or played too early or late.

A demo video and all information will be released on the website www.midilive.org for public access. As soon as bugfixing is complete, the application itself will also be accessible.
5 Conclusion

In this paper we demonstrated a way of visualizing music by processing MIDI data. The big advantage of using MIDI data is the fact that no real audio signal is used and therefore does not have to be scanned and divided into its basic frequencies. Nearly all electronic musical instruments have MIDI support, so our application has a broad application scope, whether its live visualization for concert stages, educational purpose or simply home entertainment. Scientific environments can also be an application scenario, where the focus is on examination of the nature of chords, melodies and the tone itself.

6 Resources

6.1 Perlin Noise

"Perlin noise"  http://freespace.virgin.net/hugo.elias/models/m_perlin.htm

6.2 Metaballs

"Polygonising a scalar field" by Paul Bourke http://local.wasp.uwa.edu.au/~pbourke/modelling/polygonise/