Senior Project: Pretty Lights

An LED driven Music Visualization Device

Authors: Nick delMas
Matt Maniaci
Advisor: Prof. Gene Fisher

Spring 2010

Abstract: Digital media players often include a visualization component that allows a user to watch a visualization synchronized to their music or videos. This project uses the visualization plugin API of an existing media playback program (WinAmp) but it displays its visuals using physical LED lights. Instead of outputting visuals to the computer screen, data is sent over USB to a micro controller that runs the LED lights. This project aims to give users a more visceral visual experience than traditional visualizations on the computer screen.

Table of Contents

List of Figures and Tables

Introduction

Beyond the pure delight of music to our ears, humans have sought ways to enhance music through the use of another medium. One of the most prominent forms has been the synchronization of visual art and music. By displaying visuals in combination with music the artists can create a more rich media experience than just one media form alone would allow. The goal in combining the media forms is to provide more depth to the artistic work being presented and give the viewer a greater sense of immersion. When accomplished successfully, this media synchronization can be very powerful and moving to watch. When unsuccessful, the visualization can seem chaotic and unappealing to human senses.

In order to make a sophisticated visualization, high fidelity information about the sounds that are playing must be extracted from the music. This extraction can be very difficult when dealing with analog sound. Recently this extraction has become much simpler as music playback has increasingly been performed by computers, and visual and audio formats have shifted to the digital realm. With the music in a digital form it is possible for a computer to decompose the recording and sample different musical aspects of the sound. In fact, most common programs for media playback already include at least a simple visualization component that displays graphics on the computer screen.

One of the most popular media playback programs is WinAmp, published by NullSoft. WinAmp provides a plugin interface for adding visualization plugins and comes with three plugins pre-installed. The most popular of WinAmp's visualization plugins is MilkDrop. MilkDrop is noted for having some of the best synchronized visualizations among its competitors. It is also noted for being developed by a third party outside of NullSoft. This third party development is possible because WinAmp provides a visualization plugin API that allows anyone to develop their own visualization plugin for WinAmp.

For this project we used the WinAmp visualization plugin API to create Pretty Lights. Pretty Lights is a music visualization device that uses the API to pull data from music playing in WinAmp and then visualize it using physical LED lights.

Pretty lights combines the best of both digital and analog worlds -- the ease of working with data and sampling in the digital realm with the pleasure of perceiving light and sound in the analog world.

The goal of Pretty Lights is to not only look pleasing to the user but also have at least some synchronization with the media that is playing. Generally the more popular visualizations like MilkDrop are the ones that are synchronized best, but this is difficult to achieve. It is also difficult to gauge how good a visualization is because aspects such as color choice are quite subjective.

This paper describes how Pretty Lights was designed to achieve the project goals. A technical description of both the software and hardware components is provided as well as an analysis of the algorithm used to drive the lights. Also included are links to several video demonstrations that showcase the quality and range of Pretty Lights' visualizations.

Project Overview

Pretty Lights consists of two distinct software and hardware components. The first is a software component that communicates with WinAmp via its plugin API. This component analyzes the media playback data provided by WinAmp and then once the data has been processed, it sends instructions across USB to an Arduino microcontroller. The hardware component consists of the microcontroller and LED lights. The microcontroller uses the instructions received from the WinAmp plugin to drive the LED lights creating a visual display synchronized to the media played in WinAmp.

The diagram in Figure 1 gives a high level view of data flow throughout the project. Each piece of the project is discussed in further detail in later sections:

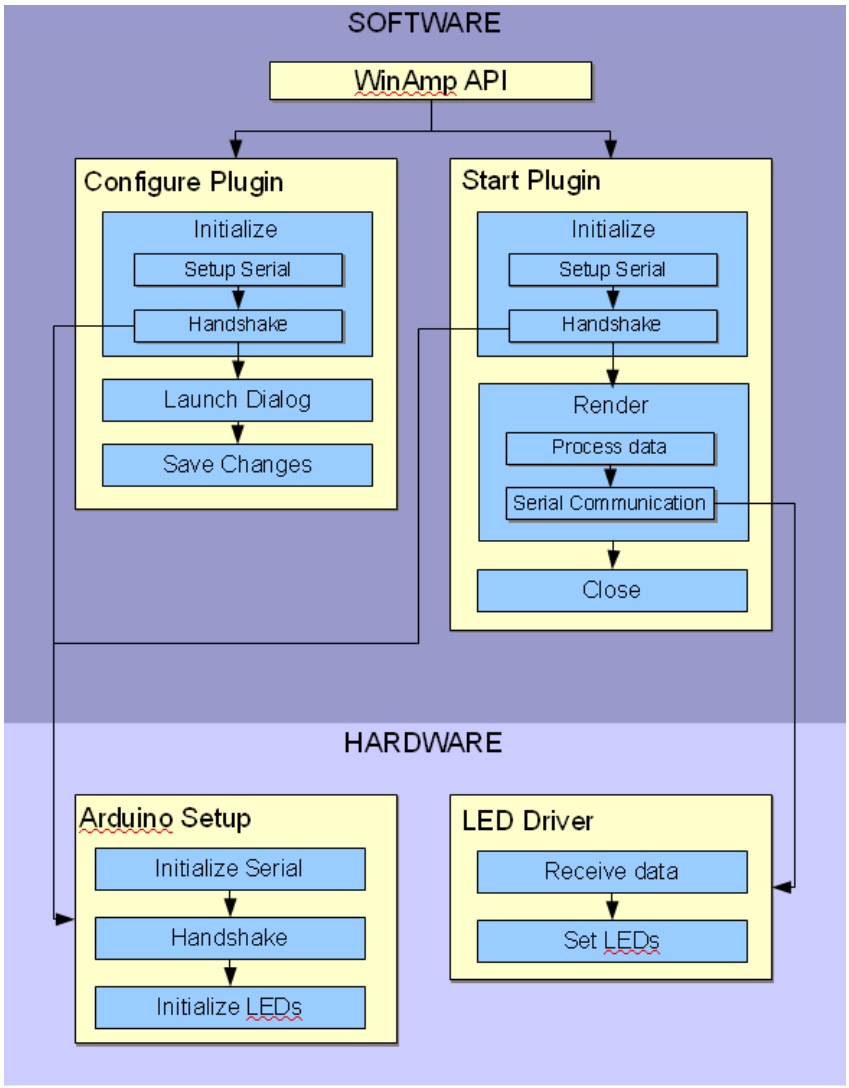

Figure 1: Software and Hardware Components Overview

High Level Data Flow

The WinAmp API is the only source of data in Pretty Lights. The Start Plugin box represents the core functions of Pretty Lights that are used to process audio data and communicate with the Arduino microcontroller. In the Initialize box a handshake is performed with the Arduino Setup part of the hardware in order to confirm that the connection to the Arduino exists. In the Render box audio data is processed and instructions for the LED lights are sent to the LED Driver section in hardware.

The Configure Plugin box represents the functions used by Pretty Lights to configure how the LED lights will perform. Changes to this configuration are initiated from the WinAmp API, then processed by the Conguration Plugin portion of Pretty Lights, and finally sent to the Arduino Setup portion of hardware so that the configuration can be implemented by the Arduino microcontroller.

The Arduino Setup box in the Hardware section represents the functions used on the Arduino microcontroller to receive configuration information from the Pretty Lights WinAmp plugin and initialize the LED's appropriately.

The LED Driver box in the Hardware section represents the simple driver that turns the LED's on and off based upon the instructions from the Pretty Lights plugin.

Hardware

Arduino Duemilanove Microcontroller

The Arduino Duemilanove is a prefabricated board containing an eight bit Amtel ATMEGA328 microcontroller with a clock speed of 20 MHz and 32 Kb of system memory. The serial communication and translation done through an FTDI serial-to-USB chip located on the Arduino board. There are fourteen digital IO pins and 6 analog inputs available for use.

Five of the fourteen digital IO pins are in use on our project. Below is a table showing each pin and what it is used for:

Pin	**Description**
3	PWM Pin 1, Red LEDs
5	PWM Pin 2, Pink LEDs
6	PWM Pin 3, Yellow LEDs
9	PWM Pin 4, Blue LEDs
GND	Common Ground

Table 1: Arduino Pins Used in the Visualization Device

The four pulse-width modulation pins are used to control the brightness of the LEDs and the ground pin completes the circuit. These pins are connected to the device, and therefore the LEDs, via wires soldered onto a PCB board.

Light Emitting Diode (LED) Array

The LEDs are wired directly into the pulse-width modulation pins on the Arduino microcontroller (pins 3, 5, 6, and 9). There are eight LEDs arranged in a square and they are indexed clockwise starting with the bottom, red, LEDs. We used two of the same color LED to increase the brightness. The corresponding lights are wired in parallel,

which means they are controlled by the same signal as indicated by the circuit diagram below.

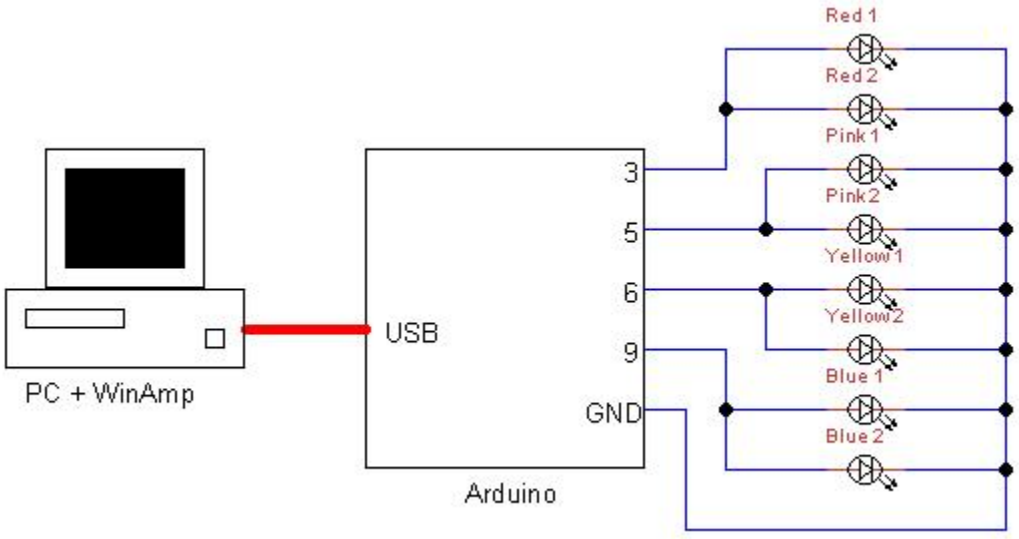

Figure 2: Circuit Diagram of Visualization System Note: Only ports that are in use shown

Software

Software on the Arduino

The Arduino microcontroller runs C code compiled by the avr-gcc compiler through the bundled IDE. The IDE compiles the source code, adds some Arduino-specific functionality to the C language, provides an editor for the code, and also uploads the compiled code to the Arduino board.

Unlike typical C programming, the entry point to Arduino programs is the setup function with the following prototype:

```
void setup();
```

The setup function is where all hardware initialization must be done, like setting up the input and output pins and preparing on-board LEDs. After the initialization is complete, the Arduino will continuously call the loop function with the following prototype:

```
void loop();
```

The loop function will be called repeatedly and contains the controlling logic for the Arduino microcontroller. Below is a simple diagram that illustrates the flow of data for an Arduino program:

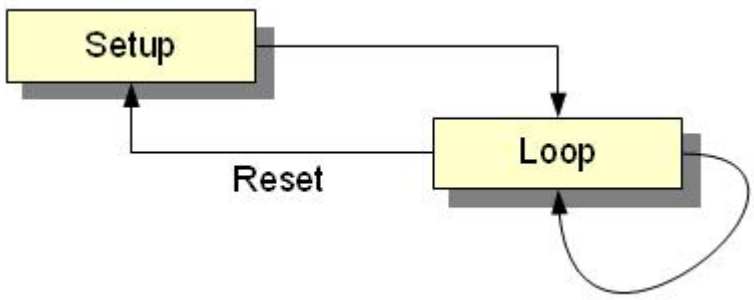

Figure 3: Simple Flowchart for an Arduino Program

Since programs for the microcontroller must be kept fairly simple due to memory and clock speed constraints, we opted to keep the scope of the microcontroller code as

concise as possible. The sole duty of the Arduino is to provide a medium for the WinAmp plugin to communicate to the LED array. The following image shows the logical path that our program takes.

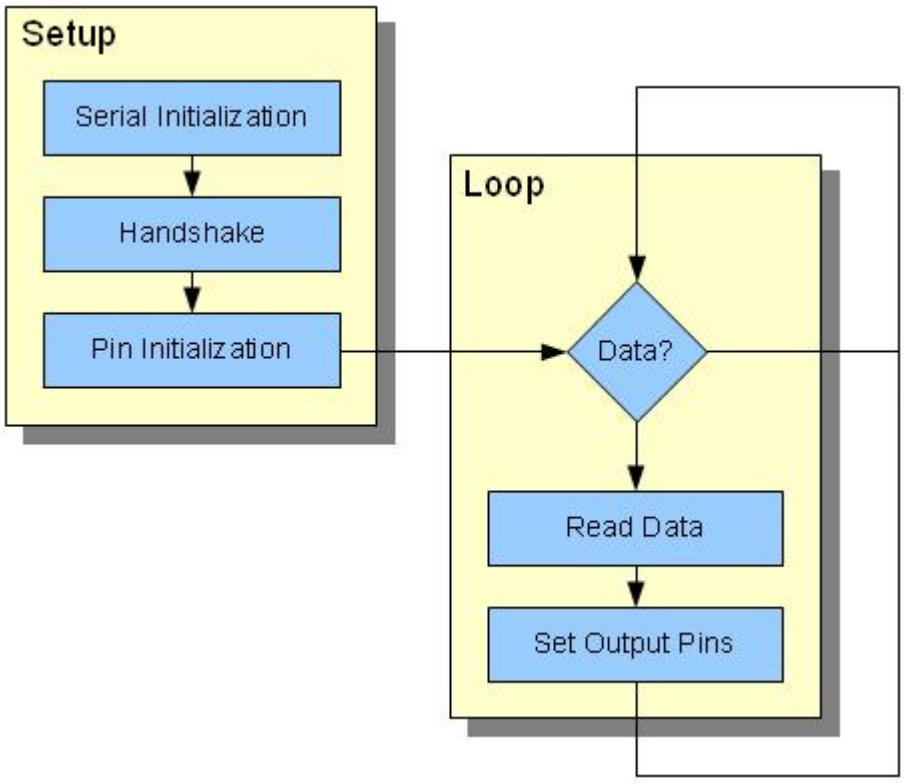

Figure 4: Arduino Control Flow

In the setup portion of the program, the serial communication is set up, a handshake is performed with the PC, and the LED output pins are initialized:

```
void setup()
{
  init_serial();
  handshake();
  init_leds();
}
```

The `init_serial` function initializes the serial communication with a bit rate of 9600 bits per second, and then flushes the buffers on the FTDI chip. The `handshake` function sets up a conversation with the PC using a 2-way handshake. The Arduino will continuously poll the serial line for a connection byte of 0x2A (42) . Once the byte is

received, the program will send the same byte back to the PC to indicate that it is ready to receive image data.

To signal to the user that the Arduino has set up communication to the PC and is waiting to render a visualization the LEDs will each briefly light up in succession as they are initialized. The digital pins 3, 5, 6, and 9 are set as output and flashed in the `init_leds` function. Once the connection to the PC is set up, the Arduino will fall into the main loop.

```
void loop()
{
  if (Serial.available() >= 4)
  {
    parse_and_light();
  }
}
```

The `Serial.available()` function polls the serial port and returns the number of bytes available for reading. If there are four bytes (the size of the packet) or more, the bytes are read and sent to the LEDs using the pulse-width modulation (PWM) capabilities of the digital ports inside of the `parse_and_light` function.

PWM is a method of controlling a device using pulses of voltage at specific and varying frequencies. For an LED, PWM means that it is being repeatedly turned on and off, but fast enough that the human eye cannot perceive the flicker. What we do see is varying degrees of intensity in the emitted light. A longer “on-cycle” will result in a brighter LED, and a longer “off-cycle” will result in a dimmer LED. To set an LEDs brightness, the `analogWrite(int val)` function is called with a value equal to the corresponding byte received from the PC. A value of 0 sent to the `analogWrite` function would turn the LED off all of the time. A value of 128 would turn the LED on for 50% of the time, and we would see half of the maximum brightness on the LED. And finally, a value of 255 results in full brightness.

Image Specification

The LEDs used in the visualization are treated much like pixels on a monitor screen. An “image” can be displayed on the LED array in the same way an image is displayed on a typical computer screen. The WinAmp plugin sends an image of the current state of the visualization to the microcontroller, and then the microcontroller will parse and display the image. The data for images in our visualization is a series of four bytes, three bytes representing the strength of each primary color and the fourth byte is the overall intensity.

Figure 5: Data Packet for Visualization

An example of how the image specification is used can be easily outlined by a simple, 4-LED situation. Let’s assume we want the first three LEDs to be red, green, and blue, respectively, and the fourth to be off. The image to achieve this would be:

LED	Red	Green	Blue	Intensity	Bytecode
1	0xFF	0x00	0x00	0xFF	0xFF0000FF
2	0x00	0xFF	0x00	0xFF	0x00FF00FF
3	0x00	0x00	0xFF	0xFF	0x0000FFFF
4	0x00	0x00	0x00	0x00	0x00000000

Table 2: Example Data Sent to the Arduino for a Simple Image

The first LED would be red due to its red value of 0xFF (255), the second LED would be green due to its green value of 255, and so forth. An intensity of 0 corresponds to the LED being turned off, as shown on the fourth row of the table. In the case of LEDs, the color black would also result in the light being turned off.

Any color in the rainbow can be made by mixing the three primary colors in certain degrees of intensity. The RGB LEDs that we planned on using for the project would be able to reproduce a wide range of colors, but due to time constraints and shipping times we were not able to order the LEDs and opted to use four single-color LEDs instead. The WinAmp plugin is programmed to handle the full four bytes of color specification.

WinAmp Visualization Plugin API

The WinAmp visualization API is used by Pretty Lights to retrieve the sample data from a song that is needed to run the LED lights. This section describes the technical features of the API (such as function calls and data structures) that are used by the Pretty Lights plugin.

The visualization plugin API provides four main functions for communicating with WinAmp. What these functions do is left entirely up to the developer, it is only specified when these functions are called by WinAmp as follows:

`Int Init(WinAmpVisModule *module)` – This function is called when the visualization plugin is started by WinAmp. This function is used to set up resources.

`Void Config(WinAmpVisModule *module)` – This function is called when a user selects the visualization plugin and pushes a configuration button in WinAmp. This function is used to set configuration options for the plugin, but is not even functional in some plugins such as AVS. In Pretty Lights the configuration dialog allows the user to set thresholds for the LED lights (this is covered in more detail other sections).

`Void Quit(WinAmpVisModule *module)` – This function is called when WinAmp stops running the plugin. This function is used to clean up resources before the plugin exits.

`Int Render(WinAmpVisModule *module)` – This function is called on a regular interval specified by the plugin and is used to receive and process the audio data provided by WinAmp.

The most important of these functions is the Render function. This is where all of the work is done when processing data from WinAmp. As shown above, the Render function takes one parameter, a WinAmpVisModule. The WinAmpVisModule is the main data structure used by WinAmp to store data. To understand how it is used it's best to start with its parent structure the WinAmpVisHeader:

```
typedef struct
{
	int version;
	char *description;
	WinAmpVisModule* (*getModule)(int which);
}
WinAmpVisHeader;
```

The WinAmpVisHeader stores high level information about the plugin such as its version and a general description of the application. The WinAmpVisHeader also contains a function pointer to a function getModule. The getModule function is used to select between any number of modules that can each have entirely separate functionality. However, most visualization plugins, such as MilkDrop, use only one default module and this is also how Pretty Lights works. So the getModule function pointed to in the WinAmpVisHeader simply and ends up looking like:

```
WinAmpVisModule* CPrettyLightsWAPApp::GetModule(int module)
{
	switch (module)
	{
	  case 0:	return &mod1;
	  default:	return NULL;
	}
}
```

Note that the switch statement here is actually essential to the API functionality and without returning NULL by default the entire WinAmp program will hang in an infinite loop when loading the plugin.

WinAmp uses getModule to populate the correct WinAmpVisModule with audio data. The data is stored for access later by the Render function and is laid out as follows:

```
// main structure with plugin information, version, name...
typedef struct WinAmpVisModule
{
  char *description;            // name/title of the plugin
  HWND hwndParent;              // hwnd of the WinAmp client
  HINSTANCE hDllInstance;       // hinstance of this plugin
  int sRate;                  //sample rate
  int nCh;                    //number of channels
  int latencyMS;              //latency
  int delayMS;                //delay, how often render is called

  int spectrumNCh;            //number of spectrum channels
  int waveformNCh;            //number of waveform channels

  //spectrum/waveform usually [2][576]
  unsigned char spectrumData[2][576];
  unsigned char waveformData[2][576];

  void (*Config)(struct WinAmpVisModule *this_mod);
  int (*Init)(struct WinAmpVisModule *this_mod);
  int (*Render)(struct WinAmpVisModule *this_mod);
  void (*Quit)(struct WinAmpVisModule *this_mod);

  void* userData;                              //optional
}
WinAmpVisModule;
```

The data is stored in the two arrays waveformData and spectrumData. The variables nWCh and nSCh specify the number of channels of waveform and spectrum data, which is the number of columns in the waveformData and specturmData arrays. These values are usually set to two. So each array of waveform and spectrum data is separated into two channels of 576 samples. These arrays are the source of the data used by the algorithm to run the LED lights.

In addition to this data the WinAmpVisModule contains several important variables for controlling the plugin. The most important are the delayMS and latencyMS. These variables are used to control the speed at which the visualization will render. Setting these higher will cause the visualization to run slower.

There are several other pointers contained in the struct, including the four pointers to the API functions Config, Init, Render, and Quit. Also, a void pointer is added at the end allowing developers to expand the struct as they see fit. Instead of using this space we decided to move away from the struct paradigm in C and created a class in C++ that is wrapped around the WinAmpVisHeader struct. This class, CPrettyLightsWAPApp, is the final part of the WinAmp API and is used to bridge the gap between the hardware and

software portions of Pretty Lights. It is essentially the core of the project as it contains references to the WinAmpVisHeader and our default WinAmpVisModule as well as all of the WinAmp API functions Init, Config, Quit, and Render. It also has references to the serialization library used to communicate with the Arduino microcontroller and the configuration dialogs used to change settings from WinAmp.

The CPrettyLightsWAPApp class:

```
class CPrettyLightsWAPApp : public CWinApp
{
public:
	CPrettyLightsWAPApp();
	~CPrettyLightsWAPApp();

	// WinAmp Module functions
	virtual BOOL InitInstance();
	static WinAmpVisModule* GetModule(int module);
	static WinAmpVisHeader* GetHeader();
	static void Config(WinAmpVisModule* this_mod);
	static int Initalize(WinAmpVisModule *this_mod);
	static int Render(WinAmpVisModule *this_mod);
	static void Quit(WinAmpVisModule *this_mod);

	// Helper functions
	void Initialize(HWND hwndWinAmp, HINSTANCE hDllInstance,
	bool bShowDialogs);

	void Quit();
	void Config();
	void LoadConfig();
	void WriteConfig();
	void InitializeDialogs(bool bShowDialogs);
	CString GetIniFile();

	CArduinoSerial		m_as;
	HWND			m_hwndWinAmp;
	HINSTANCE		m_hDllInstance;

	// Dialogs
	CDebugConsoleDlg*	m_pDebugDlg;
	CLEDSimulatorDlg*	m_pSimDlg;


	// Configuration
	bool	m_bSimEnabled;
	bool	m_bDbgEnabled;
	int	m_iLowThresh;
	int	m_iMidThresh;
	int	m_iHighThresh;
	int	m_iSimRows;
	int	m_iSimCols;

	DECLARE_MESSAGE_MAP()
};
```

WinAmp API Configuration Dialog

One of the functions provided by the WinAmp API allows the developer to create a configuration dialog when a user selects their plugin from WinAmp and pushes a configuration button. For Pretty Lights, pushing the Configure button pops up a dialog box that looks like this:

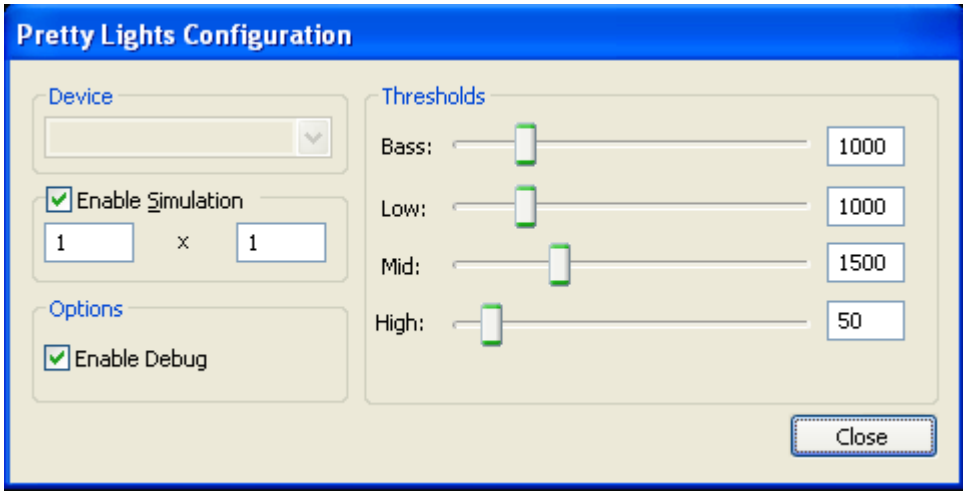

Figure 6: WinAmp Configuration Dialog

This dialog box contains several controls that allow the user to customize Pretty Lights' performance. The device pull down tab allows the user to select the Arduino microcontroller connected to their pc that they wish to connect to. This can be used to tell whether a PC is correctly recognizing the Arduino and connecting to it. The simulation options can be used to enable a simulation dialog box that contains small circles representing the physical LED lights. The number of lights is determined by the numbers in the two text boxes (rows and columns of lights). This can be increased beyond the capacity of the physical LED setup with interesting results. The options section allows the user to enable a debugging console that gives text output about the status of the plugin. The nature of what these debugging messages mean is usually obvious and is not discussed here. The debugging console is most useful for determining whether the Arduino microcontroller is successfully connected to WinAmp.

The final section of the configuration dialog, and most important, is the thresholds section. The sliders and text boxes contained in this section allow the user to directly

control the four threshold parameter used by the Pretty Lights' visualization algorithm. This functionality is essential to the goals of Pretty Lights. Users will invariably have different subjective tastes in both how they want the lights to look and what kind of music they listen to. Allowing the user to change the parameters of the algorithm allows them to customize the algorithm to their personal tastes and hopefully get more enjoyment from Pretty Lights.

Visualization Algorithm

Designing the visualization algorithm was by far the hardest part of creating Pretty Lights. WinAmp simply hands the developer several channels of waveform and spectrum data without any hint as to what to do with it. In order to make the problem approachable we decided to concentrate on two goals. Our main goal for the algorithm was to make the visualization pleasing, the secondary goal was to synchronize the visualization to the audio data provided. The priority of the goals was important because PrettyLights is an entertainment device. Making the device enjoyable to use is more important in this situation than making the device technically impressive. The following sections describe how PrettyLights was designed to meet these goals.

Color Choice

To ensure the LEDs were enjoyable to watch the most important decision when designing the visualization was the color choice. In order to pick appealing colors we consulted a chart found in the paper Real Time Music Visualization: A Study in the Visual Extension of Music by Bain, Matthew N. The chart is taken from the website RhythmicLight.com created by Fred Collopy. Several color schemes mapping colors to certain notes are suggested in a chart reprinted in Figure 7:

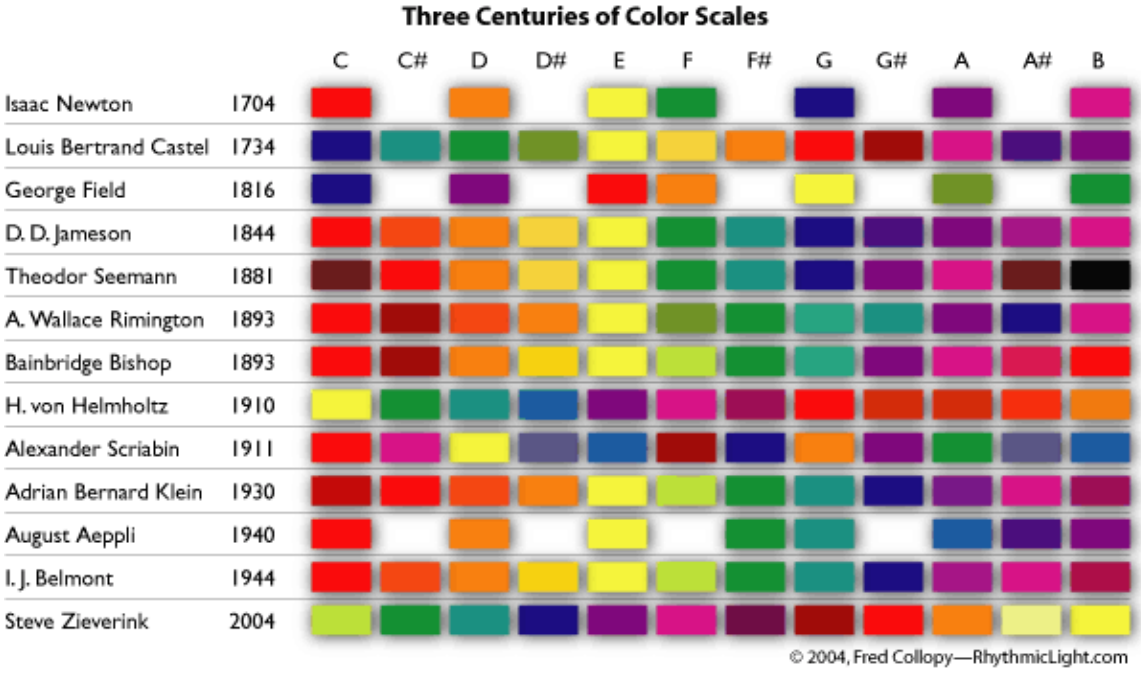

Figure 7: Three Centuries of Color Scales

Figure 7 shows various colors that correspond to specific notes. There is no real justification provided for why the colors were chosen. It seems that these people found that these were the colors they preferred to see with certain notes.

For Pretty Lights we did not match the specific notes played to colors, but used the chart as a guide to choose colors for lower or higher notes. The color scheme closest to what is used in PrettyLights is the one proposed by August Aeppli in 1940. PrettyLights is designed to use four sets of LED's so we chose four colors. Like Aeppli's chart, red was chosen as the main color used to play the lowest notes, green as the color for the middle range, and a teal or blue for the highest range of notes.

Unlike Aeppli's chart Pretty lights uses a pink or magenta color for the low range between red and green. This color was chosen because the brightness of pink LED's are easier for users to see. Changes in the brightness of yellow or orange LED's are fairly hard for the human eye to pick up.

Mining the Spectrum and Waveform Data

WinAmp hands off several arrays containing hundreds of audio samples to PrettyLights every time it renders. These arrays are waveformData and spectrumData and by mining the data from these arrays PrettyLights accomplishes its secondary goal of synchronization. WaveformData is a representation of the audio data as a pure audio waveform that if graphed would look something like a sine wave. Gaps between peaks in this data represent the frequency of the sound being played.

The other array populated by WinAmp, spectrumData, contains volume levels at each frequency across the audible spectrum of sound. WinAmp creates the spectrumData array by performing a Fourier analysis method on the raw waveform data in the waveformData array. Because this operation is performed by the API, PrettyLights only needs to access the frequency data in the spectrumData array.

In order to synchronize the data in spectrumData to the LED lights the spectrumData array must be split up into channels that correspond to each of the lights. SpectrumData is an array of integers of the size [2][576]. The size of this array is merely convention. The first half of this array, index 0, contains volume levels for the lower half of the frequency range, and index 1 of the array contains the upper half of the frequency

range. PrettyLights is designed to use four sets of lights with four colors so the spectrumData across both indices was broken in four channels.

The channels are bass, low, mid, and high. As one might guess, the bass is the channel that covers the lowest frequencies while high covers the highest. The channels are broken up somewhat unevenly in order to give better synchronization performance. The bass channel covers the lowest $1/8^{th}$ of the indices of the first channel of spectrumData (0 – 72) with the low channel covering the remaining indices of the first channel (73-576). The mid and high split the second channel of spectrumData evenly with mid covering the first half of the indices (0-288), and high covering the second half (299-576).

By separating the channels in this way we create the opportunity for the lights to synchronize to specific events in the music. The bass is set very, very low so that it only picks up the bass beat and not any low vocals or instruments. We want the bass to be bass drum and low bass guitar only. On the other hand the low and high portions of the spectrum are usually very sparse so these channels are much larger. This means that the low and high channels will be synchronized to any sound or sounds across a range of frequencies instead of just a specific range.

Creating and Manipulating LED Intensity Thresholds

Thresholds are the final and most important part of the algorithm that runs the lights. In addition to creating specific channels across the spectrumData that correspond to the four sets of lights, the lights need threshold values that indicate when they should blink on and off. PrettyLights is designed so that when no sound is present on a channel, the LED tied to that channel will be completely turned off. Once the volume level on a channel breaks its threshold, the LED will turn on. The further above the threshold that the volume level is, the brighter the LED will shine.

This allows the LED's to be very dynamic and synchronize well with the music. On songs with heavy beats that cut in and out the lights blink on and off in time with the beat. On songs with heavy guitar strumming or keyboards the lights fade in and out as the rhythm picks up and slows down. When the song is flooded with the classic Specter

‘Wall of Sound’ all of the lights turn on with full brightness and then cut off as the song breaks and they suddenly sink below the threshold.

So thresholds are what really make synchronization work, however setting them correctly is very difficult. Setting the thresholds high for a really loud dance song may work great but then the lights won’t blink at all when using those setting to play a slow and mellow solo acoustic piece. In order to make this easier Pretty Lights contains a configuration dialog (discussed more fully in the software section). This dialog allows a user to set the exact threshold values for each of the four channels in Pretty Lights (bass, low, mid, and high). By giving the user direct control over the thresholds Pretty Lights allows users to customize the plugin to their own subjective tastes.

Demonstration

Links to several video demonstrations of the Pretty Lights WinAmp visualization plugin are provided here. Each video features several seconds of Pretty Lights performing a song meant to show off a certain genre. While Pretty Lights probably best for visualizing electronic dance music, other genres and some slower and quieter songs are also shown in order to show the range of Pretty Lights' features. For each song Pretty Lights has been configured with optimal thresholds for the sequences shown.

Featured song "Let 'em Know it's Time to Go" by Pretty Lights.
Genre: High volume dance music
http://users.csc.calpoly.edu/~mmaniaci/pretty_lights.html

Featured son "Genesis" by Justice.
Genre: High volume dance music
http://users.csc.calpoly.edu/~mmaniaci/justice.html

Featured song "Grim Reaper Blues" by Entrance.
Genre: High volume psychedelic rock/blues
http://users.csc.calpoly.edu/~mmaniaci/entrance.html

Featured song "Run" by Air.
Genre: Low volume electronic/ambient
http://users.csc.calpoly.edu/~mmaniaci/air.html

Conclusion

Has Pretty Lights achieved its goals? Is it fun to watch? Absolutely. Does it synchronize well to the music? Most of the time. Is there room for improvement? Definitely. For the purpose of this project the scope of Pretty Lights was kept at a small scale. Considering this project as a technical demonstration one could easily extrapolate upon the algorithm used in Pretty Lights to create a commercially viable lighting system.

Compatibility with commercial lighting could be accomplished through use of the DMX protocol. DMX is a standardized protocol used to control stage lighting in industry. Replacing our custom protocol with a DMX compatible protocol would allow Pretty Lights to drive most commercially available stage lights.

In addition to compatibility with commercial lighting, it would be nice if Pretty Lights could interface with programs besides WinAmp. Ideally we would like to be able to hook up the Pretty Lights algorithm to any source of audio data output. Use of a more advanced API for sampling (such as MAX/MSP) should allow us to do this.

So, while this specific project is complete, the development of Pretty Lights is far from over. With possibilities to expand in so many ways, it will be interesting to see which direction Pretty Lights' development will take. Whatever direction this may be, we look forward to it with the same enthusiasm that started this project.

Bibliography

Real Time Music Visualization: A Study in the Visual Extension of Music.
Bain, Matthew N. 2008. Published by Ohio State University.

WinAmp.com Forums (www.forums.WinAmp.com)

Milk Drop visualization plugin source code
(Available from www.nullsoft.com/free/milkdrop)

Microsoft Developer Network (www.msdn.microsoft.com)

Analysis of Senior Project Design

Project Title: Pretty Lights

Quarter / Year Submitted: Winter ‘10

Student: Matthew Maniaci

Advisor: Dr. Gene Fisher

Functional Requirements: The Pretty Lights LED Visualizer connects to the WinAmp visualization system to create an entertaining display of LED lights that change in brightness and color with the music played.

Primary Constraints: Combining the abstract world of software with real world hardware is always a challenge that takes time and effort to overcome. The Pretty Lights project encompassed the entire spectrum of software and hardware, dealing with low-level circuitry, microcontrollers and signaling along with integrating the Microsoft Foundations Classes (MFC) with the WinAmp plugin library to provide a graphical interface to the visualization device. Communicating up the ladder of abstraction from hardware to software creates a daunting design problem due to the inherent complexity of combining drastically different systems.

Economic: Original estimated cost: $75, final cost: $80

Arduino	$33.00
LEDs and various wiring components	$47.00

If manufactured on a commercial basis: This product would never be sold commercially. It is unfeasible and its purpose was simply experimental.

Environmental: The Arduino uses a microcontroller and various other electronic devices that must be recycled in proper fashion. The LED peripheral board also uses electronics that are unsuitable for the landfill.

Development: The software uses the WinAmp visualization plugin library and Microsoft Foundation Classes (MFC). The Arduino microcontroller used is programmed in a special flavor of C geared towards hobbyists.