COSC 4P98

Paint to Audio

Image to Audio Synthesis Engine

Dan Chandler

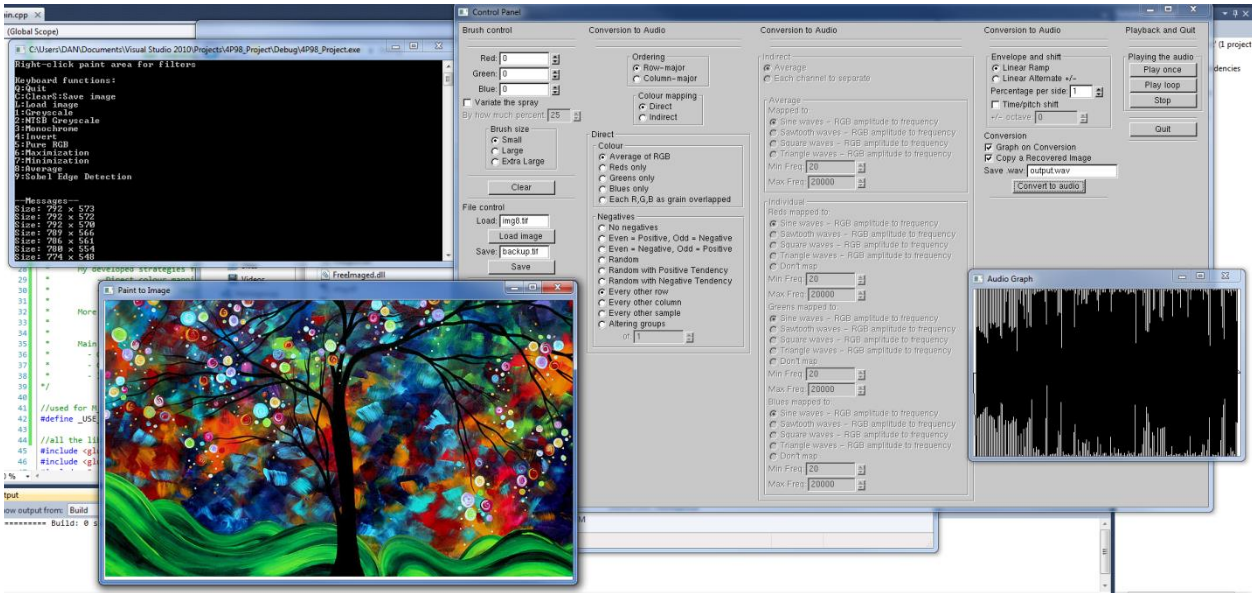
5/9/2014

Contents

Project description

My course project is program that converts images to audio signals. The program allows you to paint on a canvas, load images, and apply various image filters. Once the image is ready, you can convert that image to a .wav file by selecting a number of parameters. There are numerous ways to map images to audio. I've chosen to focus on two of them, which I call: direct colour mapping and indirect colour mapping. More is explained on my interpretations of these techniques later on. The program also graphs audio signals in a separate window on each conversion. In some cases of direct colour mapping, it makes sense to allow for recreation of an image from the converted audio signal, this is done in the program as well.

This idea intrigued me when shown a similar example in class. I decided that with my graphics background, I could reasonably do this and was curious about the results. At the end of the project, I believe I formed some reasonably audible sounds from images and the algorithms used. My program uses OpenGL, GLUT, GLUI, and libsnd to establish all the necessary aspects needed. Here's a screenshot of the program in action:


C:\Users\DAN\Documents\Visual Studio 2010\Projects\4P98_Project\Debug\4P98_Project.exe

```
Right-click paint area for filters

Keyboard functions:
Q:Quit
C:ClearS:Save image
L:Load image
1:Greyscale
2:NTSB Greyscale
3:Monochrome
4:Invert
5:Pure RGB
6:Maximization
7:Minimization
8:Average
9:Sobel Edge Detection


--Messages--
Size: 792 x 573
Size: 792 x 572
Size: 792 x 570
Size: 789 x 566
Size: 786 x 561
Size: 780 x 554
Size: 774 x 548
```

Control Panel

Brush control: Red: 0; Green: 0; Blue: 0; Variate the spray [ ]; By how much percent: 25; Brush size: [✓ Small] Large, Extra Large; Clear. File control: Load: img8.tif; Load image; Save: backup.tif; Save

Conversion to Audio: Ordering: [✓ Row-major] Column-major; Colour mapping: [✓ Direct] Indirect. Direct – Colour: [✓ Average of RGB] Reds only, Greens only, Blues only, Each R,G,B as grain overlapped. Negatives: No negatives, Even = Positive, Odd = Negative, Even = Negative, Odd = Positive, Random, Random with Positive Tendency, Random with Negative Tendency, [✓ Every other row] Every other column, Every other sample, Altering groups of: 1

Conversion to Audio: Indirect: [✓ Average] Each channel to separate. Average – Mapped to: [✓ Sine waves – RGB amplitude to frequency] Sawtooth waves – RGB amplitude to frequency, Square waves – RGB amplitude to frequency, Triangle waves – RGB amplitude to frequency; Min Freq: 20; Max Freq: 20000. Individual – Reds mapped to: [✓ Sine waves – RGB amplitude to frequency] Sawtooth waves – RGB amplitude to frequency, Square waves – RGB amplitude to frequency, Triangle waves – RGB amplitude to frequency, Don't map; Min Freq: 20; Max Freq: 20000. Greens mapped to: [✓ Sine waves – RGB amplitude to frequency] Sawtooth waves – RGB amplitude to frequency, Square waves – RGB amplitude to frequency, Triangle waves – RGB amplitude to frequency, Don't map; Min Freq: 20; Max Freq: 20000. Blues mapped to: [✓ Sine waves – RGB amplitude to frequency] Sawtooth waves – RGB amplitude to frequency, Square waves – RGB amplitude to frequency, Triangle waves – RGB amplitude to frequency, Don't map; Min Freq: 20; Max Freq: 20000

Conversion to Audio: Envelope and shift: [✓ Linear Ramp] Linear Alternate +/−; Percentage per side: 1; Time/pitch shift [ ]; +/− octave: 0. Conversion: [✓ Graph on Conversion] [✓ Copy a Recovered Image]; Save .wav: output.wav; Convert to audio

Playback and Quit: Playing the audio: Play once, Play loop, Stop; Quit

Image to Audio

Since image data is simply RGB integer values in some range and audio signals are simply discrete samples in some range, mapping these two is clearly possible. However, as I've learned through using this program, some interpretations create noise, whereas others create audible signals that seem distinct to the image.

Audio has two distinct things we need mapped: time and amplitude. Images have pixel XY location and RGB values. To me, it makes the most sense to map pixel XY location to time – location in the image, location in the sound. Using simply X or Y alone won't give very long sounds using a 44,100 Hz sample rate, when full screen at 1980 pixels wide is like 44 milliseconds. So X*Y will give us an appropriate amount of samples. Since width * height implies an ordering of how it is read, we have two options for reading: row-major and column-major.

With the mapping of time and pixel position established, we need to look at how to map amplitude at each sample. There are definitely countless interpretations of mapping RGB values to amplitude once you consider types of indirect mapping, which I've started to explore in this program. But first, let's take a look at the most obvious way to map – direct mapping.

Direct colour mapping

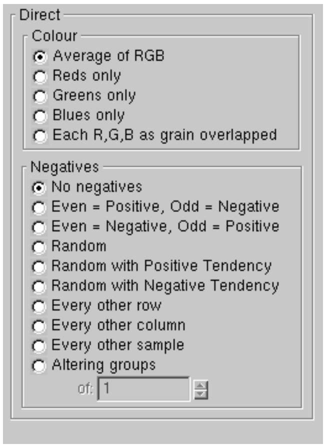
By direct mapping, I mean let the RGB values go straight to sample amplitude. As long as the resolution is the same, there aren't any major problems. In my program, I used 16-bit resolution for both image and audio. So if we're mapping RGB values, which ones? Well, I've got 5 main methods:

- Average of RGB
- Red only
- Green only
- Blue only
- Each RGB as its own signal which is added together

But this leads to another question with direct mapping, what about negative values? Audio signals range from positive to negative amplitude levels, whereas images only range from zero to the maximum bits per pixel. Well, there is countless approaches to use for this, first of which is to use no negatives at all. Here's a list of methods implemented in my program:

- No negatives
- If value is even, then it's negative. Otherwise, it's positive.
- If value is odd, then it's negative. Otherwise, it's positive.
- Random
- Random with a positive tendency
- Random with a negative tendency
- Every other row
- Every other column
- Every other sample
- Altering groups of x, where x is selected by user

As you can see, there are a number of options for negation and most are straightforward. Although they are simple, after lots of testing, their results vary greatly. Even/odd, the random negations, and every other sample usually generate noise. Whereas every other row works well with row-major ordering, and every other column works well with column-major ordering. Altering groups of x varies depending on the image size used and the group size selected because you can vary your results between the best to worst seen. And finally, no negative value actually works fairly well, but with what I'd describe as a less rich sound. This covers the techniques of negation I've implemented. Here's a screen of the direct mapping panel:



Direct

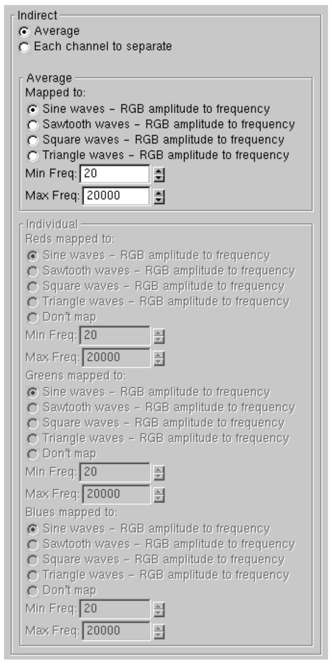
Colour
- [✓ Average of RGB]
- Reds only
- Greens only
- Blues only
- Each R,G,B as grain overlapped

Negatives
- [✓ No negatives]
- Even = Positive, Odd = Negative
- Even = Negative, Odd = Positive
- Random
- Random with Positive Tendency
- Random with Negative Tendency
- Every other row
- Every other column
- Every other sample
- Altering groups

of: 1

Indirect colour mapping

By indirect mapping, I mean taking the RGB values at each pixel and mapping the value in some indirect manner to audio signals. One way I came up with is to map the RGB value to a ratio which is applied to frequency oscillation changes of existing waves, such as sine waves, square waves, saw tooth waves, and triangle waves. A minimum and maximum frequency is used to maintain the frequency changes within a certain range. This leads to two options: use an RGB average for one existing waveform, or use each RGB with a different existing waveform. The results of this vary wildly depending on the ranges used and the manner used. For the base range, the program uses 20 Hz to 20,000 Hz, which can be changed as desired. Here's a screenshot of the indirect mapping control panel:



Indirect
[✓ Average]
Each channel to separate

Average
Mapped to:
[✓ Sine waves – RGB amplitude to frequency]
Sawtooth waves – RGB amplitude to frequency
Square waves – RGB amplitude to frequency
Triangle waves – RGB amplitude to frequency
Min Freq: 20
Max Freq: 20000

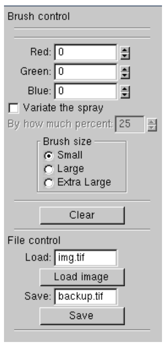
Individual
Reds mapped to:
[✓ Sine waves – RGB amplitude to frequency]
Sawtooth waves – RGB amplitude to frequency
Square waves – RGB amplitude to frequency
Triangle waves – RGB amplitude to frequency
Don't map
Min Freq: 20
Max Freq: 20000
Greens mapped to:
[✓ Sine waves – RGB amplitude to frequency]
Sawtooth waves – RGB amplitude to frequency
Square waves – RGB amplitude to frequency
Triangle waves – RGB amplitude to frequency
Don't map
Min Freq: 20
Max Freq: 20000
Blues mapped to:
[✓ Sine waves – RGB amplitude to frequency]
Sawtooth waves – RGB amplitude to frequency
Square waves – RGB amplitude to frequency
Triangle waves – RGB amplitude to frequency
Don't map
Min Freq: 20
Max Freq: 20000

System Design

The system uses some key libraries for graphics and user interface, as well as audio signal creation and playback. For the paint window, I've used OpenGL and GLUT. For the image reading and writing, I've used FreeImage. For the user interface, I've used GLUI. For writing to .wav file, I've used libsnd. And since this is designed for a Windows OS, I used the .NET framework for playback. All these APIs I used are ports for Windows, so there is no issue with conflict. GLUI uses live variables which are in turn used in the audio algorithms. GLUT and GLUI actually are designed to work together, so this worked out well in terms of the graphics portion with the program's interface needs. Although GLUI's menus aren't the prettiest, they serve their functional purpose. Conversion is done in a callback and based on the parameters set. The program is written in C/C++ on Visual Studio 2010.

System Use

The paint controls allow for selecting each RGB channel ranging from 0 to 32767 (16-bit resolution). The brush has 3 sizes: small, large, and extra-large. The small brush is 3 by 3 pixels. The large brush is 10 by 10 pixels. And the extra-large brush is 25 by 25 pixels. You can select to add variation to the brush, and select by what percent. You can clear the paint window using the clear button. Images can be loaded by using the load image button, which loads the image specified by the input text at the last clicked location. The image file must be in the same directory as the program. You can also save whatever is on the paint window to the name specified under save. Here's a screenshot of the brush control:


Brush control

Red: 0
Green: 0
Blue: 0
☐ Variate the spray
By how much percent: 25

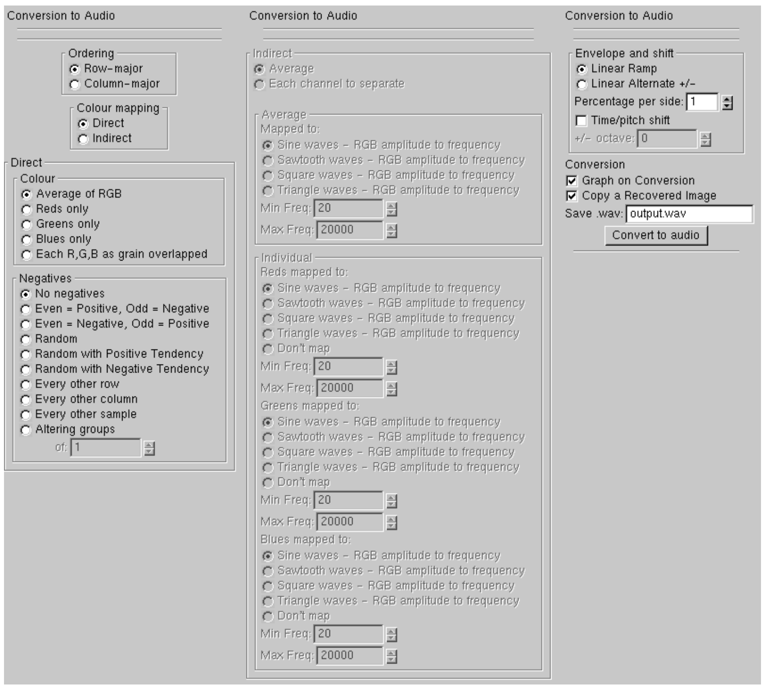
Brush size
[✓ Small]
Large
Extra Large

File control
Load: img.tif
Save: backup.tif

On the paint window, you can right-click and apply image filters to the window. I've added the following filters:

- Greyscale
- NTSB Greyscale
- Monochrome
- Invert
- Pure RGB
- Max filter
- Min filter
- Average filter
- Edge detection

These filters are also available on the keyboard their keys 1 to 9. As I covered earlier, conversion to audio is a major portion of the program. Once you select ordering, you select which mapping method and the necessary parameters. But what I didn't mention before is you can apply an envelope to the created audio signal using a linear ramp or linear ramp with alternating positives and negatives. You can also pitch shift the audio signal by +/- octaves, which adjust the duration of the signal. Here's a screenshot of the main conversion to audio options:



Conversion to Audio

Ordering
- [✓ Row-major]
- Column-major

Colour mapping
- [✓ Direct]
- Indirect

Direct

Colour
- [✓ Average of RGB]
- Reds only
- Greens only
- Blues only
- Each R,G,B as grain overlapped

Negatives
- [✓ No negatives]
- Even = Positive, Odd = Negative
- Even = Negative, Odd = Positive
- Random
- Random with Positive Tendency
- Random with Negative Tendency
- Every other row
- Every other column
- Every other sample
- Altering groups

of: 1

Conversion to Audio

Indirect
- [✓ Average]
- Each channel to separate

Average

Mapped to:
- [✓ Sine waves – RGB amplitude to frequency]
- Sawtooth waves – RGB amplitude to frequency
- Square waves – RGB amplitude to frequency
- Triangle waves – RGB amplitude to frequency

Min Freq: 20

Max Freq: 20000

Individual

Reds mapped to:
- [✓ Sine waves – RGB amplitude to frequency]
- Sawtooth waves – RGB amplitude to frequency
- Square waves – RGB amplitude to frequency
- Triangle waves – RGB amplitude to frequency
- Don't map

Min Freq: 20

Max Freq: 20000

Greens mapped to:
- [✓ Sine waves – RGB amplitude to frequency]
- Sawtooth waves – RGB amplitude to frequency
- Square waves – RGB amplitude to frequency
- Triangle waves – RGB amplitude to frequency
- Don't map

Min Freq: 20

Max Freq: 20000

Blues mapped to:
- [✓ Sine waves – RGB amplitude to frequency]
- Sawtooth waves – RGB amplitude to frequency
- Square waves – RGB amplitude to frequency
- Triangle waves – RGB amplitude to frequency
- Don't map

Min Freq: 20

Max Freq: 20000

Conversion to Audio

Envelope and shift
- [✓ Linear Ramp]
- Linear Alternate +/−

Percentage per side: 1

- Time/pitch shift

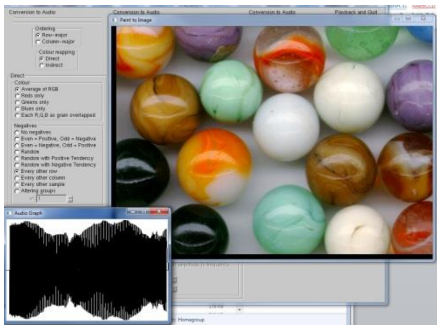
+/− octave: 0

Conversion
- [✓ Graph on Conversion]
- [✓ Copy a Recovered Image]

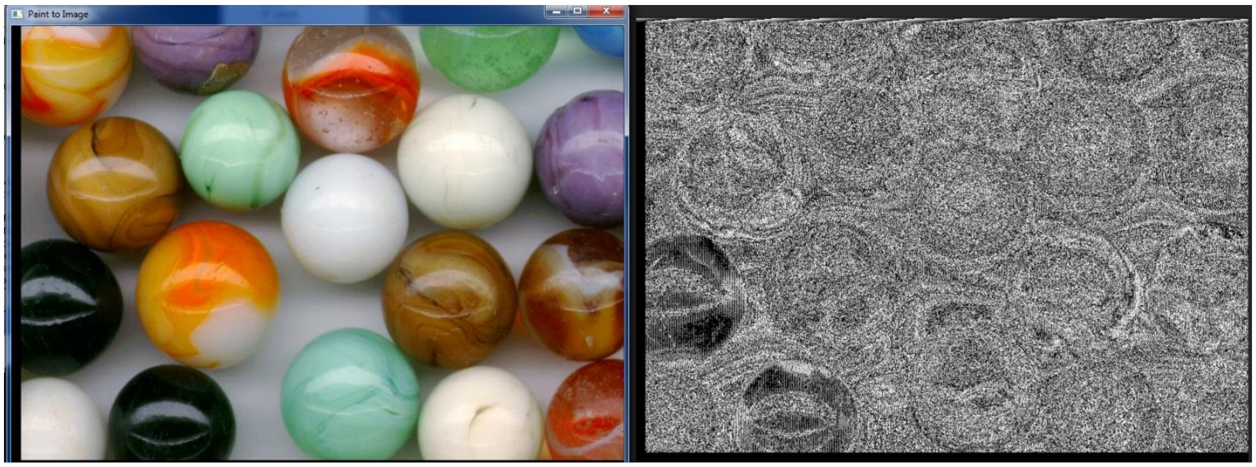
Save .wav: output.wav

Convert to audio

You can also view the audio signal as a waveform in the graph window on conversion, which is an option that is selected on the conversion parameters. Here's a screenshot of an image of marbles and its graphed signal using certain parameters:



You can also somewhat recover the image from the converted audio signal. However, direct mapping would need to be used; otherwise there is no direct correlation to the original image. Pitch shift can't be used since the samples and size is altered, since pitch shift changes the signal, we don't know what width and height to rebuild at. And since we're not separately maintaining the RGB channels and using an average or one channel or overlapped, the rebuilt image will be greyscale and the individual RGB value is lost. Here's a screenshot of the marbles and their recovery:



As we can see, the image is somewhat recovered and you can see what the old image was. The last main feature to know is playback. You can simply press the play once button to play the last converted audio signal. You can also play the signal in a loop and stop any other sounds.

Conclusion

In conclusion, the paint to audio program worked fairly well, as you can see with running the program. It didn't create a symphony, but I wasn't expecting it to. It did prove to make interesting and unique sound from images and showed this concept is very much possible and had interesting results. This was a worthwhile project and I enjoyed working on it. This application could easily be extended to make a real-time audio synthesis engine, and use more image-to-audio mapping techniques.