

HueShift: A NIME That Binds Color and Frequency

Abstract

This paper researches the correlation between the visible light spectrum and musical polyrhythms. It will NOT be an accurate experiment focussed on the physics and precise data, but rather a test to see if using color data is a viable option in a musical context, and as a NIME (New Interface for Musical Expression).

Introduction

The main goal of this research is to create a NIME which can manipulate how visible light gets turned into polyrhythms and controls it via software. To test how our eyes and ears can be tied together ~ an experiment surrounding the human desire to perceive patterns.

Since colors have a wavelength and light also has a constant speed, we can correlate colors to frequencies. This ties into audio and polyrhythms.

Whilst the perception of audio and color are subjective, many studies have been done on why we can all agree when something sounds right, or something looks good. So, this begs the question: will nice looking color combinations lead to nice sounding polyrhythms? I think it will. However, to implement this in a NIME might not be so useful.

Theoretical Framework

There have been many implementations that you can find all over the internet on using audio to get visuals, like music visualisers and almost every concert nowadays drives their visuals using the live audio. However, this research will test the opposite.

To create the NIME, we are using JUCE¹, a plugin framework for audio development in C++.

For the hardware there needs to be a camera to get live color data, and some sort of controller that will send pure data to the JUCE plugin.

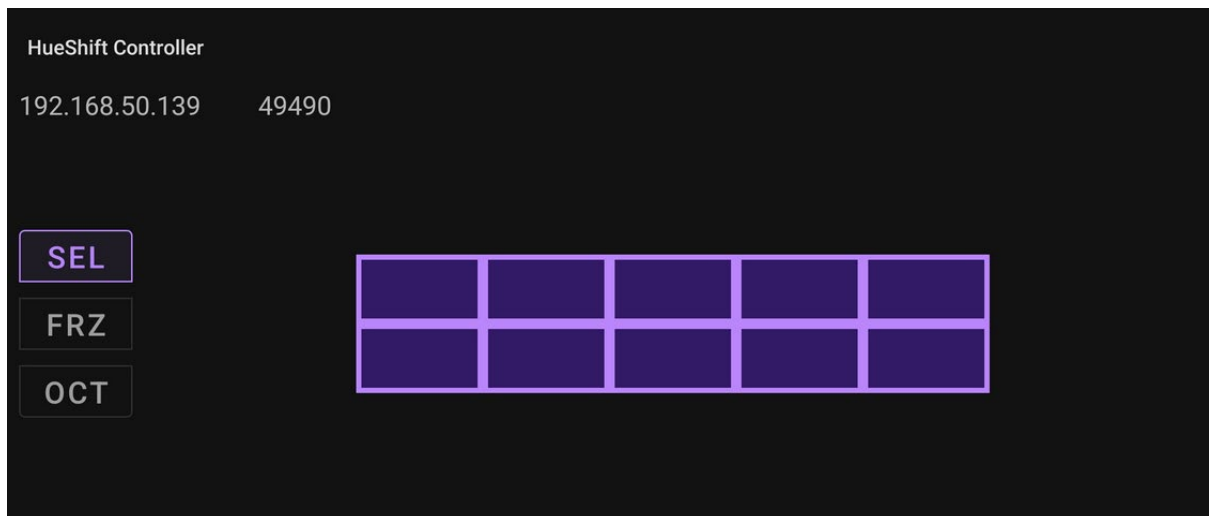
Since both a camera and a separate controller are needed, we will be making a simple app for an Android phone to send all the input data to the plugin.

Build

The plugin software is packaged as a VST3² plugin, which is the standard for audio plugins and can be used in most DAW's³. The controller app is a standard Android app built with Android Studio using Kotlin as the main language.

Input

The controller app contains a grid pattern of buttons and a mode selector for “SELECT”, “FREEZE” and “OCTAVE” modes. There is also the camera which will not be used directly in the app but can be connected using DroidCam⁴.



On the plugin side the user gets to choose a camera device that is connected to the computer, which is easy since JUCE has a cross-platform integration for cameras already.

Data

These inputs send their data via network sockets using the UDP⁵ protocol to a computer where the plugin is running.

There are two main modules for handling the data transfer: “Broadcast/Receive Discovery” to find and connect the controller with the plugin, and the actual input data “MIDI⁶ Handler” which sends the input data directly to the plugin.

We use the UDP protocol since it generally has a faster response time and is also a standard when dealing with network programming. We don't need the stable connection that TCP⁷ would bring, we merely need to send the necessary bytes every now and then, and there could be multiple controllers connected at once.

UDP is supported in JUCE as well as in Android, this makes it handier for this project than the OSC⁸ protocol for example. It also gives the user wireless freedom that is absent when using the USB⁹ protocol, which we need, since it can still be of value when the user wants to move their camera around.

The Midi Handler data from the app gets translated to a custom format made for the plugin. It uses 12 bytes per message in the UTF-8¹⁰ format and looks something like this: f-0000000012; which internally is more like this: “<mode><prefix><number><postfix>”.

This way we can specify the mode which is “f” for “freeze”, “s” for “select” and “o” for “octave”. Then there is a prefix symbol that is used to tell where the mode ends, and the number starts.

The number has 9 digits it can work with, so precision up to 100 million, which we don’t need since we use don’t send the camera data but it’s still nice to have some headroom for future implementations. We can specify smaller numbers by leading it with zeroes.

It then ends with a postfix to find where the number stops.

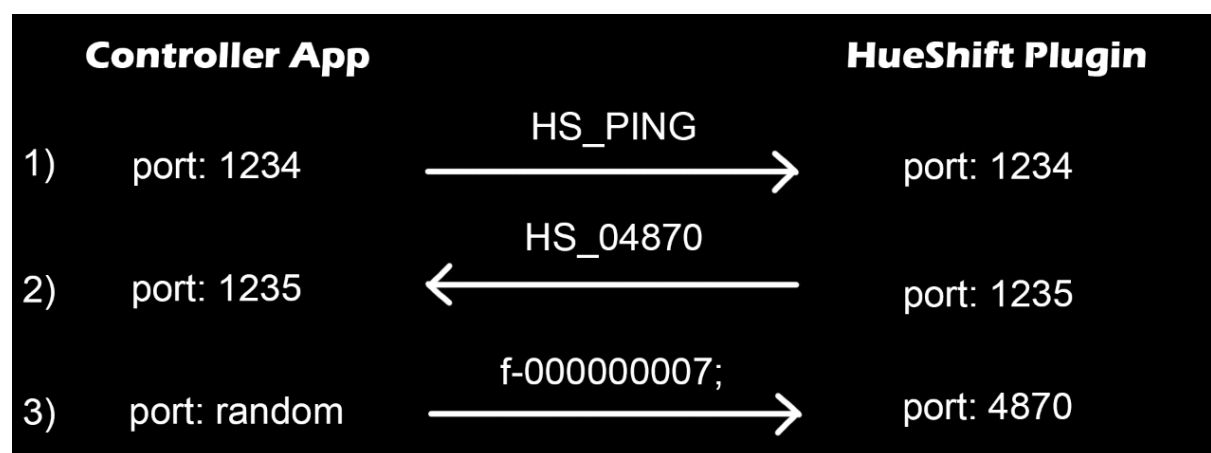
Feature Extraction

Before we can send and receive the inputs, the controller needs to connect with the plugin.

Upon starting the controller app and the plugin, they will periodically send out and listen for a message respectively. The broadcast message is sent to a specific port, but to all local devices.

The message HS_PING (“HS” for “HueShift”) is sent to that port, and the receiver in the plugin will respond with HS_07856 for example. The number is the port of the plugin that will listen for input data, again with a leading zero if needed, so we have a constant number of bytes to listen for. We let the host computer decide which port is used for the MIDI Handler data, but we pre-define the ports for the discovery broadcast and responses.

To turn this data into numbers that we can work with, we just need to do the inverse of creating it. The message “s-000000004;” will be deserialized into “mode: SELECT, grid_index: 4” but then in a fancier C++ class that we can use in the audio processor.



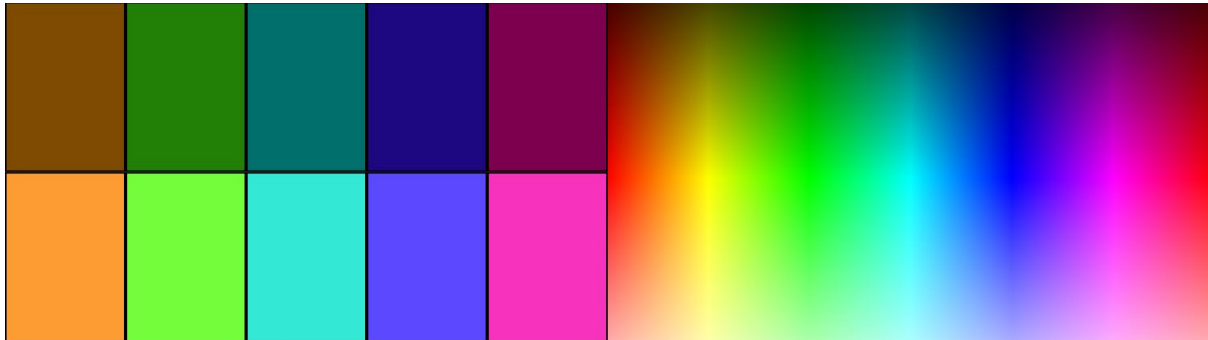
The camera is handled internally by JUCE, so we don’t have to worry about it in this step.

Mapping

The plugin is constantly listening for data from the controller app, which drives the state of the plugin. So, let's look at what the plugin is doing by itself now.

Firstly, we divide the camera's output into a grid, then we get the average color of each section by adding sample points into the feed and getting the average of those to gain some CPU performance. This is essentially pixelation or down sampling the output.

The image on the right would be the camera output, and on the left is the down sampled output in grid division.



This can then be translated from color to frequency in a somewhat loose way, as there really is no straightforward way to do this. The sample points are not randomized each iteration since we want a constant output for images.

We convert our RGB input to HSV¹¹, the hue will be approximated to the nearest angle of that color, which is then snapped to a non-complex division closest to the frequencies on the following table.

Range of the visible spectrum			
colour*	wavelength (nm)	frequency (10^{14} Hz)	energy (eV)
red (limit)	700	4.29	1.77
red	650	4.62	1.91
orange	600	5.00	2.06
yellow	580	5.16	2.14
green	550	5.45	2.25
cyan	500	5.99	2.48
blue	450	6.66	2.75
violet (limit)	400	7.50	3.10

With the colors defined as the following adjusted frequencies and normalized ranges correlating to the hue:

```

inline const ColorInfo ColorInfo::red      = ColorInfo(4.5f,      juce::Range<float>(0.000f, 0.042f));
inline const ColorInfo ColorInfo::orange  = ColorInfo(5.f,      juce::Range<float>(0.042f, 0.111f));
inline const ColorInfo ColorInfo::yellow  = ColorInfo(5.166666666f, juce::Range<float>(0.111f, 0.172f));
inline const ColorInfo ColorInfo::green   = ColorInfo(5.5f,      juce::Range<float>(0.172f, 0.431f));
inline const ColorInfo ColorInfo::cyan    = ColorInfo(6.0f,      juce::Range<float>(0.431f, 0.494f));
inline const ColorInfo ColorInfo::blue    = ColorInfo(6.666666666f, juce::Range<float>(0.494f, 0.694f));
inline const ColorInfo ColorInfo::violet  = ColorInfo(7.5f,      juce::Range<float>(0.694f, 0.806f));
inline const ColorInfo ColorInfo::pink    = ColorInfo(8.333333333f, juce::Range<float>(0.806f, 0.944f));

```

This decision was made because we are not focussed on real-life accuracy, but rather on if this can be used as a NIME.

A section has two main states and three sub states. The main states are “enabled / disabled” and “frozen / unfrozen”. The 3 sub states act as a multiplication factor for the output frequency, so ½x, 1x and 2x speed.

When you freeze a section in the app, it will stop updating the color only for that section, which makes the frequency constant. This allows for more control over what you want to hear instead of a random mess. Using the octave mode on that section will then change the rate, and of course the selection mode will either enable or disable the midi output altogether for that section. You can see if a section is selected by the white outline in the plugin’s interface.

Before getting into the MIDI output let’s go over the requirements.

The audio processor should take in audio buffers to comply with the VST3 format, but the size doesn’t really matter in our case.

What does matter is the sample rate, which can be rather low for the timing precision on the midi output. If midi precision was our only factor, then a sample rate of just 1000Hz would give us precision of 0.1ms, which would be enough since studies¹² have shown that 12-15ms is the amount of delay where you can start to feel something is off.

However, we still deal with audio. Having your sample rate that low in the DAW will make the output sound, well, terrible. Therefore, the user should keep it around 48kHz, using the Nyquist rate¹³. Higher would be more precise of course but we don’t need that.

Each camera section has its own timer that activates a stream of MIDI notes for the frequency that we get from the color of that section. We keep this frequency in sync by setting a timestamp when the voices start, which gets determined at the start of the program. Then we deduce that from the number of samples that have been passed to the plugin over its lifetime to find an exact timing per MIDI message.

A MIDI message has 2 parts, the “note on” and “note off” messages. We want to keep the length between these two to a small number of samples, so the generator synth or sampler that will be connected to the plugin can use envelopes to shape their sound, and still allow us to use higher octave multipliers if we want.

Output

Finally, we are left with MIDI output where the notes represent the grid sections, with root - top left - at C1. We can then drive a synth or sampler with that data.

Results

After using this plugin for a while, the results are pretty much what was expected. It does in fact make some good sounding polyrhythms, but they aren't controllable in a way where the user can express musical intent. What I mean by this is that we don't really know what the output is going to be before you turn on a new MIDI voice for example, which is essential for making music especially live.

Then there's also the issue that generally the user can't quickly change the color. They would have to first unfreeze the section, then hold a color to the camera, and freeze again. Not exactly a swift movement, though the plugin could be expanded to take in videos in which case we remove some of the burden on the user.

So musically, the results are not that great. Only the polyrhythms by themselves are slightly useful, making this more of a tech demo than a NIME.

However, when we look at some basic color theory we do see some slightly interesting results.

Complementary colors tend to give 9/11, 7/9, and 11/15 polyrhythms. Triadic combinations also have rounded polyrhythms with a much higher period, but still these results are not that sporadic, they sound good.

Partly this is because we are approximating the color and mapping the frequency to non-complex divisions, but these are not drastic changes to the inputs.

References

If you want to look at the Github projects, you can find them here:

<https://github.com/Subnite/HueShift>

<https://github.com/Subnite/HueShift-Controller>

Video tutorial on MIDI in JUCE:

- The Audio Programmer. (2017, October 30). *Juce Tutorial 22- Building a Synthesizer Part 2 (Midi Functionality)* [Video]. YouTube.

<https://www.youtube.com/watch?v=ddqwAJilnes>

Paper on RGB to spectrum wavelength:

- Afanasiev, V., Ignatenko, A., & Voloboy, A. (2015, April 3). *The Simple Method of the RGB to Spectrum Conversion for Tasks of Physically Based Rendering*.

<https://sv-journal.org/2015-4/03/en/index.php?lang=ru>

About HSV colors and for the wavelength conversion:

- Körting, S. T. (2015, July 10). *The HSV color model* [Video]. YouTube.

https://www.youtube.com/watch?v=NAw2_NtGNaa

Generally, for mapping wavelength to color:

- Light. (n.d.). Wikipedia. Retrieved 2024 April 21 from

<https://en.wikipedia.org/wiki/Light#Optics>

About inter process communication for reading serial data to use in the JUCE plugin.:

- Axelson, J. (2007). *Serial port complete: COM ports, USB virtual COM ports, and ports for embedded systems*. Lakeview Research.

https://members.tripod.com/data_acq/download/spcch1.pdf

- Schmidt, D. (1993). *Systems Programming with C++ Wrappers*. University of California: Department of Information and Computer Science.

<https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=89f0b3e50cad9e8167086b aa59c2711dac40aaac>

- Programming Electronics Academy. (2021, October 22). *Serial Communication with Arduino - The details!* [Video]. YouTube.

<https://www.youtube.com/watch?v=tpEo5AOSSkg>

General image processing - for other cool results:

- Digital image processing. (n.d.). Wikipedia. Retrieved 2024, 21 April from https://en.wikipedia.org/wiki/Digital_image_processing

Used many times to check my code and ask general questions:

- OpenAI. (2024). *ChatGPT* (June 13 version) [Large language model]. <https://chat.openai.com/chat>

¹ Storer, J. (2011). *JUCE: Jules' Utility Class Extensions (Version 7.0.5)* [Computer Software]. Raw Material Software Ltd.

<https://juce.com/learn/documentation/>(<https://juce.com/learn/documentation/>)

² Steinberg. (n.d.). Virtual Studio Technology Version (3). <https://www.steinberg.net>

³ Digital Audio Workstation

⁴ *DroidCam* [Computer Software]. Dev47Apps. <https://www.dev47apps.com>

⁵ User Datagram Protocol

⁶ Rothstein, J. (1995). *MIDI: A Comprehensive Introduction (2nd ed.)*. A-R Editions. https://books.google.be/books?hl=en&lr=&id=ajDaXh-qgDUC&oi=fnd&pg=PR11&dq=midi&ots=EewWXxZTBF&sig=NGFOL7WtZXlb8wtyohr-2a5cvT4&redir_esc=y#v=onepage&q=midi&f=false

⁷ Transmission Control Protocol

⁸ Open Sound Control protocol, CNMAT.

⁹ Universal Serial Bus

¹⁰ 8-bit Unicode Transformation Format

¹¹ Hue Saturation Value

¹² *Digital Audio latency explained* | PreSonus. (n.d.). PreSonus. <https://legacy.presonus.com/learn/technical-articles/Digital-Audio-Latency-Explained#:~:text=Studies%20have%20shown%20that%20sound,effects%20of%20a%20delayed%20signal.>

¹³ Wikipedia contributors. (2024, July 12). *Nyquist rate*. Wikipedia. https://en.wikipedia.org/wiki/Nyquist_rate