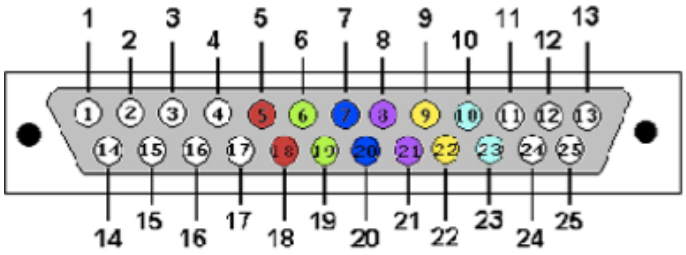
Hamill Industries: Generative laser through sound

Workshop documentation

1. ILDA Standard
2. Arduino to ILDA input
3. Simple laser drawing
4. Audio reactions from Processing to Arduino
5. Alternatives

1 – ILDA Standard



1 – X+, **2 – Y+**, 3 – Intensity+, 4 – Interlock A, **5 – R+**, **6 – G+**, **7 – B+**, 8 – Deep Blue+, 9 – Yellow+, 10 – Cyan+, 11 – Z+, 12 – N/C, **13 – Shutter**.
14 – X-, 15 – Y-, 16 – Intensity-, 17 – Interlock B, 18 – R-, 19 – G-, 20 – B-, 21 – Deep Blue-, 22 – Yellow-, 23 – Cyan-, 24 – Z-, **25 – Ground**.

The standard defines many connections that we won't use. We only use X+ (Pin 1), Y+ (Pin 2) and R+ (Pin 5),G+ (Pin 6),B+ (Pin 7) using the Ground (Pin 25) as a voltage reference.

This input can be found in many commercially available laser lightings displays (LLDs).

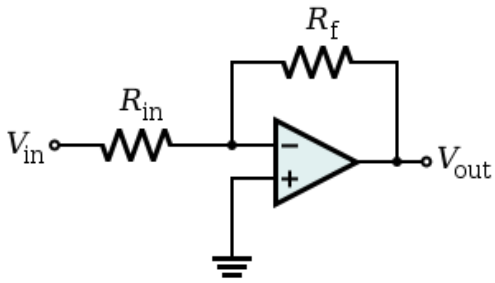
2 – Arduino to ILDA input

The RGB voltage on ILDA is 0V to 5V (0 for the minimum intensity and 5 for the maximum). Since the laser color intensity differences are quite imperceptible, we will use RGB as binary values, so we can work with primary colors (red, green, blue, magenta, cyan and yellow). By using a Digital to Analog Converter (DAC) we could work with a bigger palette (still limited by the perception).

However, the XY voltages go from -10V to 10V, and the Arduino can't give us that. If we use a DAC, we could have a range from 0 to 5V, but that would only let us use one quarter of the right upper quadrant of the ideal projection square. Therefore, a voltage gain and offset must be applied.

The way we're going to adapt the voltage is by: using an operational amplifier with a gain of 2 to obtain an inverted output range of 0V to -10V, then adding 5V from a voltage regulator (e.g. TS7805) to get a 5V to -5V range, and finally applying another gain of 2 to obtain an inverted output range of -10V to 10V.

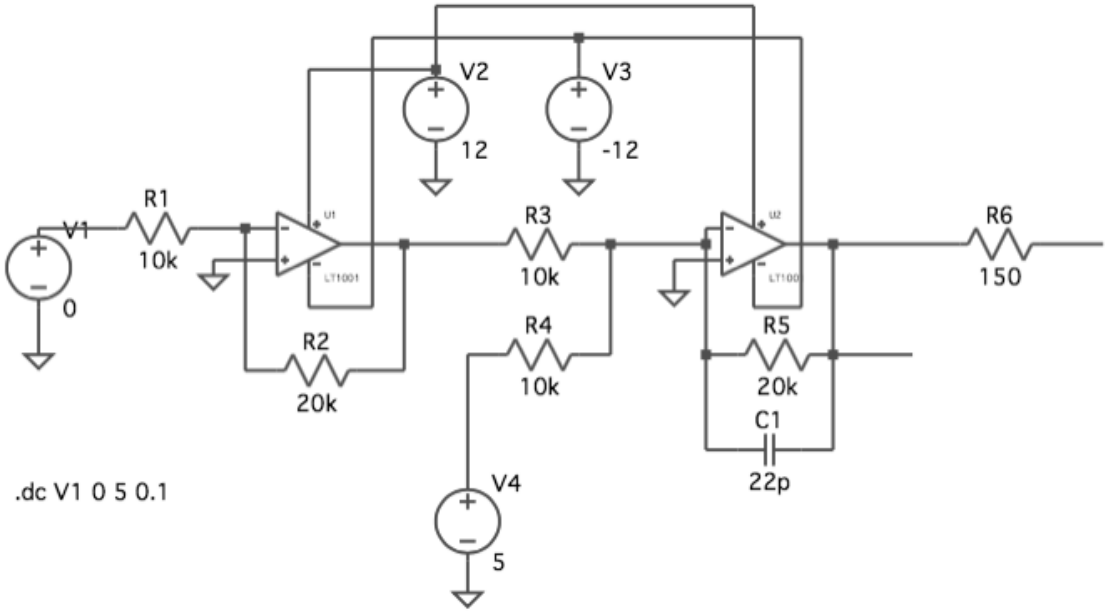
A simple inverting operational amplifier, has the following circuit:



The gain is determined by the ratio Rf/Rin. Since we want to have a gain of 2, we can use a Rf = 20kΩ and a Rin = 10kΩ.

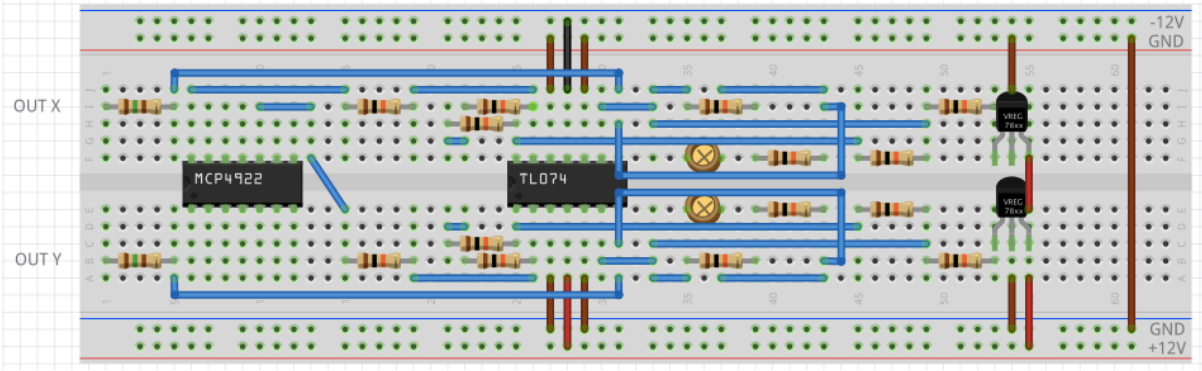
We'll be using a quad op amp (TL074) with a power supply of -12V and 12V (using two independent rails with shared ground).

The circuit for each output has the following scheme:

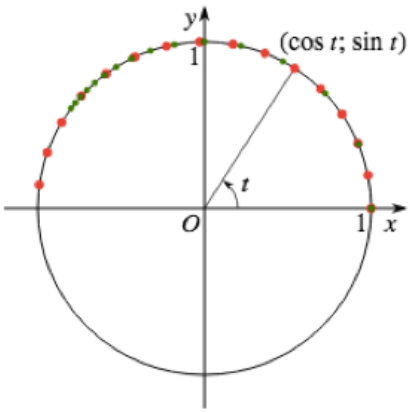


Considerations: to add 5V, we use another 10k resistor to connect with the second op amp's Rin output; to avoid output feedback in the second op amp, we add a small capacitor (around 15 to 25 pF) in parallel with the Rf; finally, for the output of the circuit, we add a resistor of 150Ω to adapt impedances with the ILDA input.

Our breadboard looks like this:



3 – Simple laser drawing



In order to draw with our ILDA controlling DACs, we have to send X,Y points to the DACs using the MCP4922 Arduino library, while selecting the color we'll use to draw the line between them.

To draw a circle, we must set points according to the parametric equation by dividing its domain (i.e. we must divide 2π by the number of sampling points).

$x(\theta) = \sin(\theta), y(\theta) = \cos(\theta)$ or viceversa.

This drawing can be achieved with the following Arduino code.

```
#include <MCP4922.h> //DAC library
#include <SPI.h> //Serial Peripheral Interface library used by the DAC library

MCP4922 DAC(51, 52, 53, 5); // (MOSI,SCK,CS,LDAC) define Connections for MEGA_board,
//MCP4922 DAC(11,13,10,5);    // (MOSI,SCK,CS,LDAC) define Connections for UNO_board,

int sampling_points = 50; //Define how many sampling points are going to draw the circle.
int r = 4, g = 3, b = 2; //Define color pins

int r_v = 1, g_v = 0, b_v = 0;

void setup()
{
  SPI.begin();
  setColors();
}

void loop()
{
  for (int i = 0; i < sampling_points; i++) {
    int x = map(
      sin(i * TWO_PI / sampling_points)*2048, //Compute location of sampling point.
      -2048, 2048, 0, 4095); //Map it to DAC code values.
    int y = map(
      cos(i * TWO_PI / sampling_points)*2048, //Compute location of sampling point.
      -2048, 2048, 0, 4095);  //Map it to DAC code values.
    DAC.Set(x, y); // Send to DAC
  }
}

void setColors() {
  digitalWrite(r, r_v);
  digitalWrite(g, g_v);
  digitalWrite(b, b_v);
}

void resetColors() {
  digitalWrite(r, LOW);
  digitalWrite(g, LOW);
  digitalWrite(b, LOW);
}
```

Observations: the sinusoids have to be mapped to DAC range, since the map function only works with integers, we will multiply the signal by 4096/2=2048 so we get the same resolution the DAC can handle (i.e. 2^12 since it's 12bit).

The number of points we use, and the speed of the components will limit the quality of the drawing, making it edgy or too slow to be perceived as continuous rather than a trace (because of the persistence of vision).

Therefore, the best approach is to use polygonal shapes with the minimum amount of points. With smog, those shapes can be pretty effective if we add a reactive element.

4 – Audio reactions from Processing to Arduino

Arduino doesn't have audio inputs, so one way we can add audio reactions is by analyzing the audio with another software and sending little data to the Arduino via serial.

One nice and kind of intuitive software for audio signal inputs is Processing. With it we can analyze the amplitude of an input signal and its frequency spectrum. If we limit the amount of results from the analysis that get sent through serial, we won't overload the Arduino and we can get a fluid drawing.

The easiest example might be sending a trigger each time the amplitude of the input is higher than a threshold. We can also filter the audio signal, so we only work with bass or treble, or send the peak frequency to modulate whatever we draw. The more parameters extracted from audio we use to modulate the data we send, the more generative those visuals will be.

On Arduino we must define a method that changes a variable whenever there's an input in the serial. We will check the serial in the beginning of each *loop()* iteration. On Processing we must define the method to analyze the incoming audio and send a serial trigger to Arduino. This is a way to do it:

```
import processing.sound.*;
import processing.serial.*;

Serial myPort;
PGraphics pg;
float threshold = 0.5;
float gain = 10;
boolean peak = false;

Amplitude amp;
AudioIn in;

void setup() {
  size(200, 200, P2D);
  background(0);
  pg = createGraphics(200, 200);
```

```
  //Initialize serial
  printArray(Serial.list());
  myPort = new Serial(this, Serial.list()[5], 19200);

  //Initialize sound
  amp = new Amplitude(this);
  in = new AudioIn(this, 0);
  in.start();
  amp.input(in);
}

void draw() {
  background(0);
  float an = gain*amp.analyze(); //Analyze amplitude and apply digital gain

  //Visualize amplitude:
  pg.beginDraw();
  pg.stroke(255);
  pg.line(0, 0+height*(1-an), 0, height);
  pg.stroke(0);
  pg.line(0, 0, 0, height*(1-an));
  pg.copy(0, 0, width, height, 1, 0, width, height);
  pg.endDraw();
  image(pg, 0, 0);

  //Visualize threshold line:
  stroke(0, 0, 225);
  line(0, (1-threshold)*height, width, (1-threshold)*height);

  if (!peak) { //Unless we're in a peak
    if (an>threshold) { //If audio over threshold
      myPort.write('a'); //Send trigger (one byte/char)
      peak = true; //Activate peak
    }
  } else {
    if (an<threshold) peak = false; //Deactivate peak when below threshold.
  }
}

void mousePressed() {
  threshold = 1-(mouseY/float(height)); //Set the threshold according to mouse position
when pressing
}
```

In Arduino, we must introduce some changes, in the case of the example; each time we receive a serial input, the circle will increase its size and decay to the normal one. In this example the serial is read, and the circle radius is changed.

```
#include <MCP4922.h> //DAC library from https://github.com/helgenodland/MCP4922-Arduino-
SPI-Library
#include <SPI.h> //Serial Peripheral Interface library used by the DAC library

MCP4922 DAC(51, 52, 53, 5); // (MOSI,SCK,CS,LDAC) define Connections for MEGA_board,
//MCP4922 DAC(11,13,10,5);    // (MOSI,SCK,CS,LDAC) define Connections for UNO_board,

int sampling_points = 50; //Define how many sampling points are going to draw the circle.

boolean new_data; //trigger boolean
int life = 0; //decay of the circle

int r = 4, g = 3, b = 2; //Define color pins
int r_v = 1, g_v = 0, b_v = 0; //Define color values

void setup()
```

```
{
  Serial.begin(19200); //Begin serial to connect with Processing
  SPI.begin();

  //Set color pins as outputs
  pinMode(r, OUTPUT);
  pinMode(g, OUTPUT);
  pinMode(b, OUTPUT);

}

void loop()
{
  setColors(); //Set colors
  recvOneChar(); //Receive trigger

  for (int i = 0; i < sampling_points; i++) {
    int x = map(int(sin(i * TWO_PI / sampling_points) * (500 + 5 * life)), -1000, 1000, 0,
4095); //Map sine to DAC code values.
    int y = map(int(cos(i * TWO_PI / sampling_points) * (500 + 5 * life)),  -1000, 1000, 0,
4095); //Map sine to DAC code values.
    DAC.Set(x, y); // Send to DAC
  }
  if (new_data) life -= 5; //Decay
  if (life <= 0) new_data = false; //Remove trigger when decay is over.

}

void recvOneChar() {
  if (Serial.available() > 0) { //If there's a trigger
    char receivedChar = Serial.read(); //Flush the buffer by reading the char.
    new_data = true; //Set trigger on
    life = 100; //Set life to maximum
  }
}

void setColors() {
  digitalWrite(r, r_v);
  digitalWrite(g, g_v);
  digitalWrite(b, b_v);
}

void resetColors() {
  digitalWrite(r, LOW);
  digitalWrite(g, LOW);
  digitalWrite(b, LOW);
}
```

5 – Alternatives

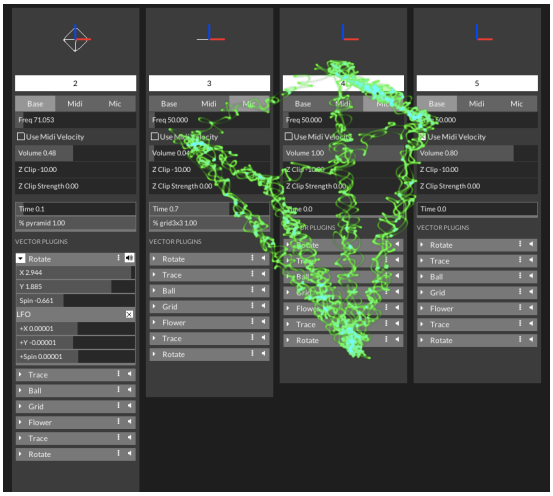
To avoid some of the limitations that can be observed with the Arduino method, we can use other techniques. For instance: using the laser as an oscilloscope with OsciStudio or going the old-fashioned way with a software such as LSX and a commercially available Etherdream DAC.

Oscilloscope with OsciStudio

An oscilloscope can draw beautiful and smooth signals when we use two individual audio channels for the X and Y inputs. We, at Hamill Industries, use this a lot when we work with the 80’s video-console/refresh vector display

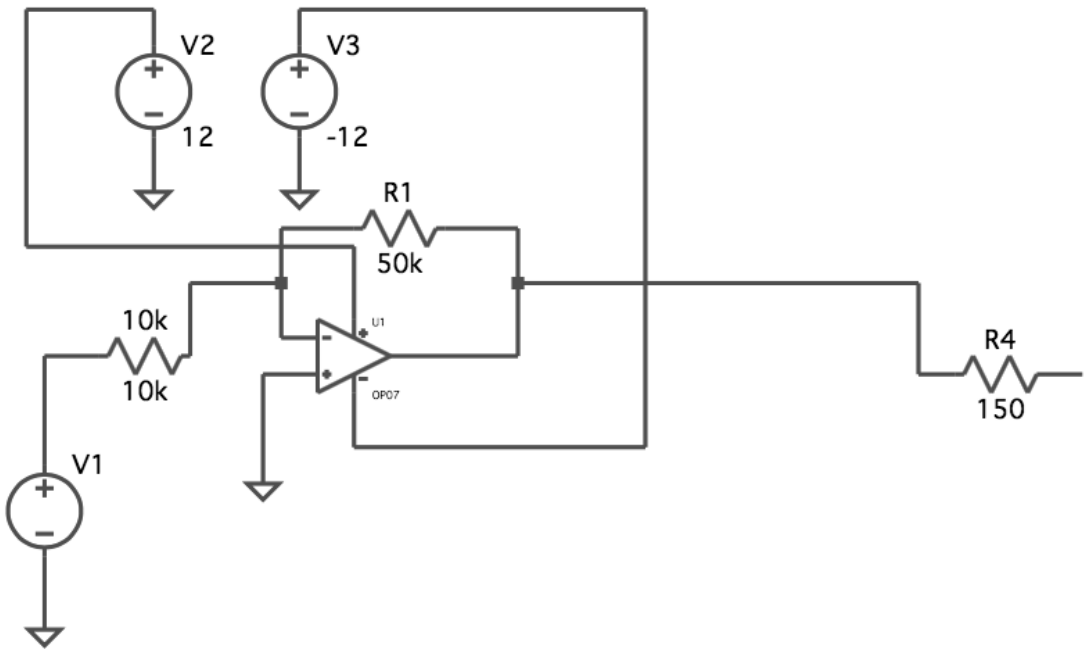
Vectrex. This has been prominently featured in our work with Floating Points and is the main part of the show we're doing this Saturday 20th on the SonarComplex stage at 14:00.

Oscistudio is a software developed by Hansi Raber that allows you to create oscilloscope music and visuals. You can load 2D .(svg) or 3D objects (with .obj files or blender), choose view parameters and modulate or automate them. It's available for 34€ at https://oscilloscopemusic.com/oscistudio.php.



It can also sum audio inputs to make it audio reactive, use scripts and assign MIDI controls to the parameters.
To connect it to the laser we can use a soundcard with a high sampling rate (ideally 192kHz) since we're dealing with video and not with audible-only sounds. The higher the sampling rate, the better it will be at creating complex figures (rather than smoothed ones, because of anti-aliasing filters). Bear in mind that complex figures might still be limited by the galvos of the LLD.

However, soundcards usually work with a -1V to 1V or -2V to 2V range; so, we'll also have to use an amplifier such as the one for the DACs.



In this case, the gain ratio is 5 instead of 2, so we use a 50kΩ resistor instead of a 20kΩ one.

Processing2LSX

LSX is a software used broadly in the LLD industry. It has many features that are worth exploring, one of them is receiving frames via OSC. By using this medium, we can send frames in an ILDA standardized format to display them with the laser. This is the technology we used for a show called Fundamental Noise which was premiered on June 2018.

All the sound processing done in Processing can be applied with less limitations using this library, and in particular with the examples in the *LsxLiveOscOutput* folder: https://github.com/colouredmirrorball/llda

The software (LaserShow Xpress) and the hardware (Etherdream DAC) costs around 300€.