Oscilloscope vector graphics on a Telequipment D54 using an Arduino MKR Vidor 4000

Teodora Osiac Ioan Dragomir

supervised by
Prof. Dr. Radu Fechete

January 2021

Abstract

Displaying line art on an oscilloscope in XY mode is a common entry level electronics project, which is much beloved by hobbyists worldwide, both for the ease of obtaining minimally functional graphics, as well as the plentiful challenging enhancements it often invites. We aim to display both static vector art, as well as animations at a perceptually smooth refresh rate, while providing a resolution finer than the trace width (cca. 0.5mm), and covering as much of the 10x6cm screen of the Telequipment D54 as possible. We develop a hardware and software stack that enables programming an Arduino MKR Vidor 4000 with static vector art, as well as streaming live animations at 192000 points/second via USB.

Contents

1 Introduction

In the 1960s and 1970s, CRT vector displays were an important advancement in computer graphics technology, a natural follow-up to the oscilloscope displays developed since the early 1900s. As opposed to raster CRT displays (as found in TV sets) which have a fixed beam path and modulate its intensity, vector displays have a mostly fixed electron beam intensity and control its position, tracing arbitrary monochrome paths on the screen. Vector displays were used with early CAD software (Ivan Sutherland's Sketchpad, 1963) and fighter aircraft HUDs, but notably also entered the home market and gained some popularity with the advent of vector arcade games in the 1980s. Games such as Asteroids and the Vectrex console have been since remembered for their distinct graphics style, and so displaying vector graphics on a CRT screen, usually that of an oscilloscope in XY mode, remains a common hobbyist project to this day.

Controlling a vector display consists in coordinating two waveforms – one for the horizontal beam deflection and one for the vertical beam deflection, to position the electron beam arbitrarily on the screen. Due to persistence of vision and the CRT phosphor's slow decay, the fast beam motion looks like streaks of light on the screen. Using this mechanism, various line graphics can be displayed.

We aim to develop an easily programmable system for processing vector graphics on a computer, transmitting the data to an Arduino, and generating the corresponding signals on the device.

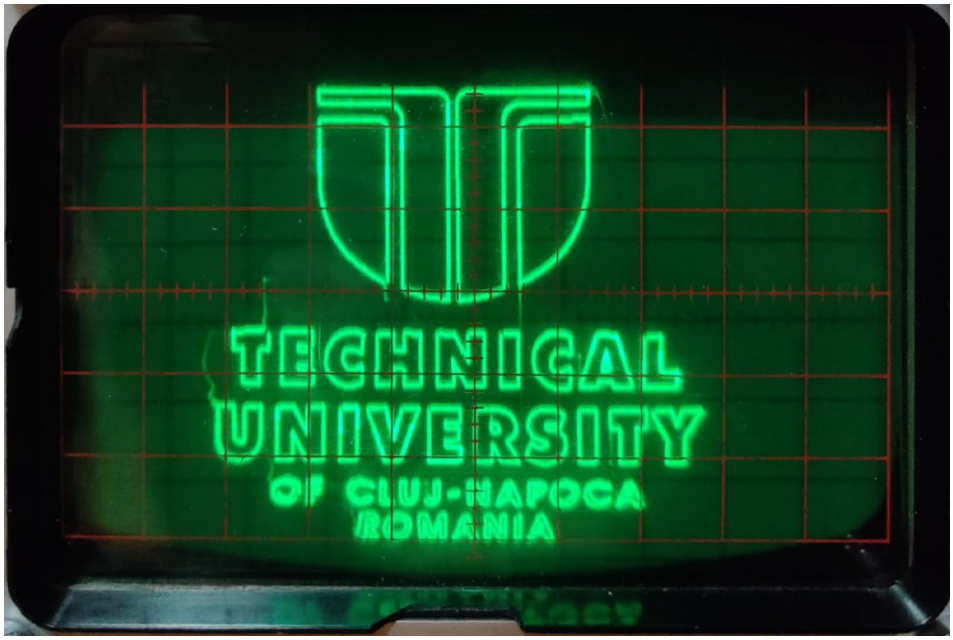

Figure 1: Full display of the TUCN logo

2 Static signal generation

To confirm the feasibility of the project, we start with a rudimentary software stack that enables programming an Arduino with a fixed, two-channel waveform corresponding to the X and Y coordinates of a point walking along a given path. The software consists of some javascript code that converts the first SVG <path> of the current document into a C header file containing a list of points (Appendix A) and some Arduino code that iterates through the generated points and outputs each coordinate pair to the suitable ports (Appendix B). At the hardware level, we require two DACs of sufficient resolution and sample rate, which motivates our choice of the Arduino MKR Vidor 4000, since it has a builtin 10-bit DAC, meaning we only have to implement a single DAC externally.

One of the requirements for a pleasant appearance is that we have control over the position of the beam in increments smaller than half the width of the beam itself. (Figure 3) By some crude measurements, our device's beam is approximately 0.5 in diameter. To span 6cm vertically in increments less than 0.25mm, we require 240 distinct voltage levels, and thus at least 8 bits resolution for the vertical channel. Similarly, the horizontal channel requires at least $\log_2 10\text{cm}/0.25\text{mm} \approx 8.6$ bits. Therefore we chose to use the built-in 10-bit DAC for the horizontal channel and to build an external 8-bit DAC for the vertical channel.

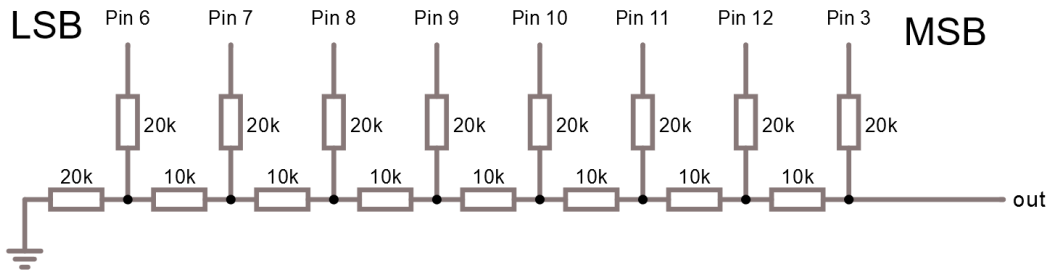

Figure 2: R-2R resistor ladder circuit for generating the vertical deflection signal

DAC design choice is a significant factor in both circuit complexity, cost, and the quality of the resulting graphics. We chose an R-2R ladder design for its relative simplicity, in spite of the expected DAC glitches (Figure 4) that appear when passing between points at a large Hamming distance but small Euclidean distance. (i.e., at coordinates at 1/2, 1/4, 3/4, etc. vertically along the screen)

To counteract such glitches, we ought to set all the output pins to their correct states at the same time, instead of setting each bit of the output sequentially. The Arduino standard library provides direct access to the PORTA register of the SAMD21 board, whose value is a bitmap of the digital values of 32 input or output pins. We use the Arduino pins 6, 7, 8, 9, 10, 11, 12, 3 (LSB to MSB) to control the resistor ladder, and these correspond to bits 20, 21, 16, 17, 19, 8, 9, 11 of the PORTA register. We apply the following bitwise permutation to obtain a value for the PORTA register given a desired 8 bit DAC output "Y", and assign all the bits at once:

```
PORT->Group[PORTA].OUT.reg = ((Y & 0b11) << 20) |      // PA20, PA21
                             ((Y & 0b1100) << 14) |    // PA16, PA17
                             ((Y & 0b10000) << 15) |   // PA19
                             ((Y & 0b1100000) << 3) |  // PA8, PA9
                             ((Y & 0b10000000) << 4);  // PA11
```

By doing so, the delay between the different pins being set to the correct value is far below noticeable, thus mostly solving the issue of DAC glitches. For improved performace, we precompute a lookup table for the 256 possible bit patterns, reducing the time to output an analog value from 23 instructions down to 5, (Appendix C) with a memory cost of 1KB.

Regardless of DAC design, an issue arises whereby signal bandwidth limitations (mostly justified by parasitic capacitance) in both the DAC, the scope probes, and the oscilloscope, will lead to ringing artifacts (Figure 5) at high enough drawing speeds. We will not attempt to prevent these kinds of errors.

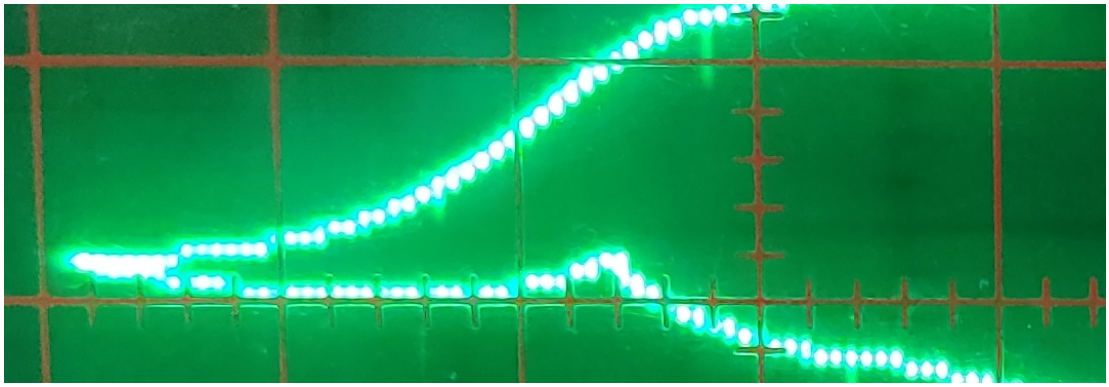

Figure 3: Low resolution leads to noticeably discrete points

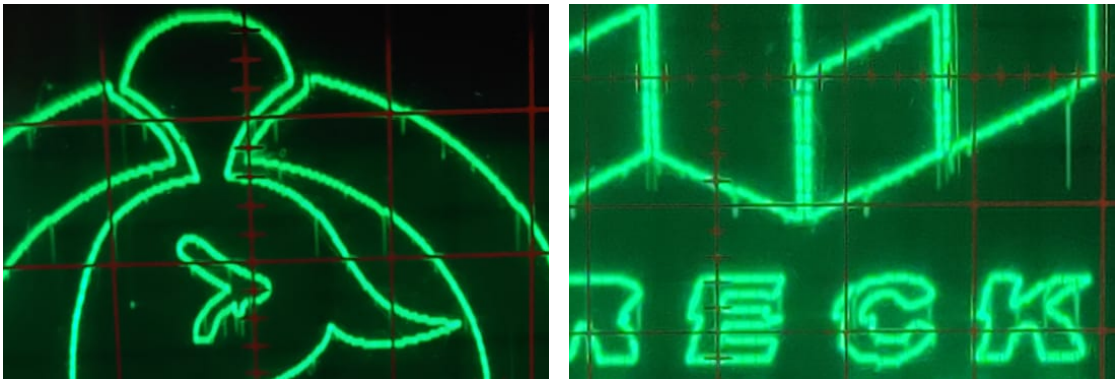

Figure 4: Glitches of the vertical DAC look like spikes at certain Y values

Figure 5: Ringing and parasitic capacitance artifacts, as seen in the final image (unwanted wisps trailing off objects' corners), as well as in a Y coordinate/time graph (slow voltage transitions, unwanted oscillations).

3 Oscilloscope music, animations

Among prior art, so called "Oscilloscope music", especially Jerobeam Fenderson's work, stands out for its exquisite combination of graphics and sound as part of the same medium. Such works of art meticulously coordinate the two waveforms to both draw discernible shapes and animations, as well as to sound like music if interpreted as sound instead of beam coordinates.

To be able to display such works, we implement a signal streaming system, whereby the controlling computer sends uncompressed 18-bit-per-sample, 192KHz sample rate audio, and the controlled Arduino receives the data and displays it. While in principle this seems like a rather basic task, in practice it is riddled with various challenges. Ideally, we would send one sample at a time, and while the computer waits for the correct time for the next sample, the beam would stay still on the current sample, but this very inefficiently uses the USB protocol and is very slow, not allowing us to achieve 192KHz. We must group together multiple samples in so-called frames, and have the Arduino cycle through the samples of each frame in the time it takes to receive the next frame. The issue that arises is related to the fact that we cannot control the beam intensity, so when the Arduino is waiting and processing serial data, the beam will stay put for a significant period of time, accumulating to a bright spot on the screen. The longer the frames, the more time it will spend receiving and not moving the beam, leading to a tradeoff between frame lengths and image quality. (Figure 7)

Another quality optimisation we considered was taking advantage of the Arduino's behaviour of interrupting on received data. If we send the frame in equally-spaced sub-frame chunks, the "bright spots" won't only appear at frame boundaries, but also whenever the Arduino is interrupted during the actual drawing. This will have the effect of spreading out the brightness discrepancies (Figure 6), but is limited, because if we split the frames too much we will arrive at the first issue of inefficiently using the USB protocol and bottlenecking the transfer rate. In the end, the results were virtually identical to reducing the frame length, so we decided the code complexity not to be worth it.

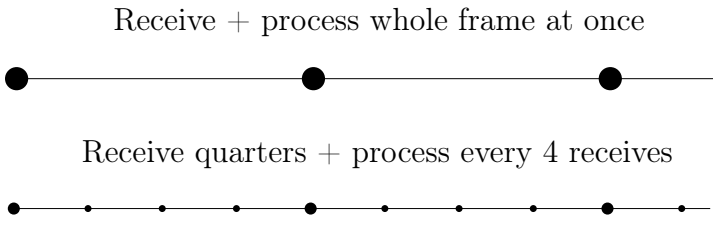

Figure 6: Appearance of normal frames versus sub-frames.

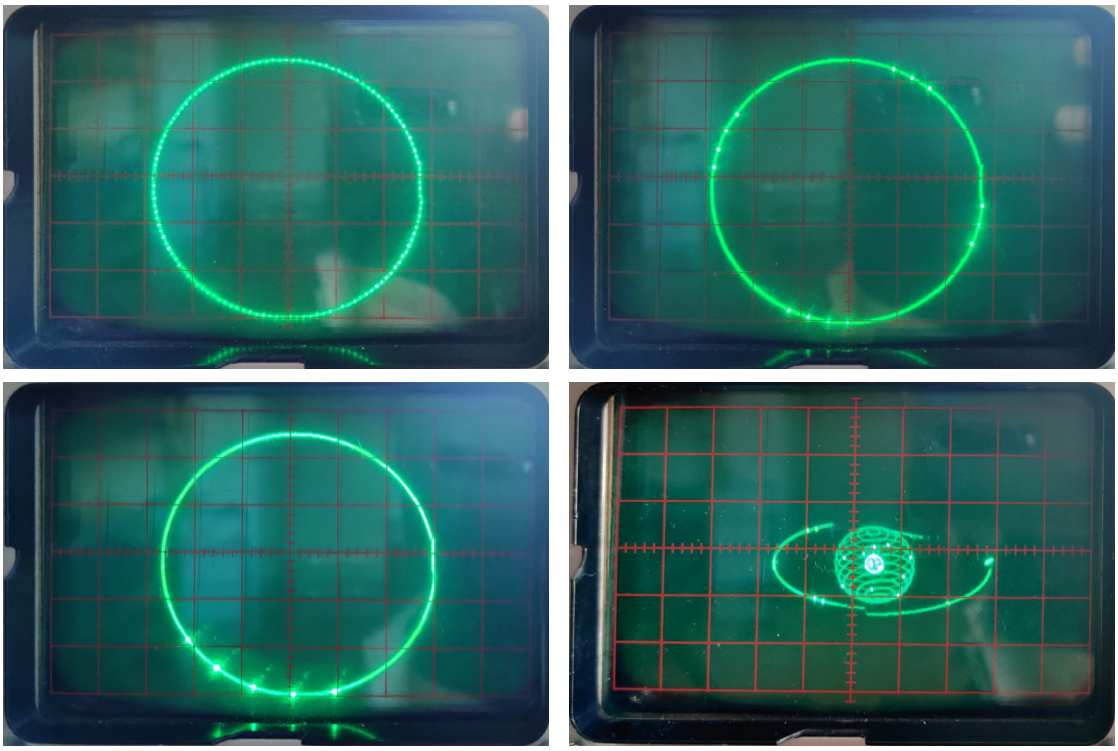

Figure 7: Bright spots appear when the Arduino is idling for more sample data, or processing it. In order, 16 samples/frame, 256 samples/frame, 2048 samples/frame for a circle consisting of 1000 samples, and 192 samples/frame during an arbitrary animation.

4 Static and animated demonstration

For the complete proof of concept, we

Appendices

A SVG path to C header conversion

```
const display_width = 728;
const display_height = 255;
const wanted_num_points = 4000;

const path = document.querySelector("path");
const L = path.getTotalLength();

// Get `wanted_num_points` equally spaced points on the path.
const points = Array(wanted_num_points).fill().map((_, i) => {
  const P = path.getPointAtLength(i * L / wanted_num_points);
  return {x: P.x, y: P.y};
});

// Calculate bounding box of path
const minx = points.reduce((a, p) => Math.min(a, p.x-0.5), Infinity);
const maxx = points.reduce((a, p) => Math.max(a, p.x+0.5), -Infinity);
const miny = points.reduce((a, p) => Math.min(a, p.y-0.5), Infinity);
const maxy = points.reduce((a, p) => Math.max(a, p.y+0.5), -Infinity);

// Rescale points so the bounding box maps to the whole display size
const points_remapped = points.map(p => ({
  x: display_width - ~~((p.x-minx)/(maxx-minx)*display_width+0.5),
  y: ~~(display_height - (p.y-miny)/(maxy-miny)*display_height+0.5)
}));

// Create a C header file with the point list
const c_header = `
// Generated from ${window.location}

const struct point points[] = {${
  points_remapped.map(({x,y}) => `{${x},${y}}`).join(',')
}};
const long num_points = ${points_remapped.length};
`;

// And initiate a "download" of the generated file
const c_header_blob = new Blob([c_header], {type: "text/plain"});
window.location.assign(URL.createObjectURL(c_header_blob));
```

B Basic signal generation

```
struct point {
  uint16_t x; uint8_t y;
};

#include "tucn.h"

void setup() {
  pinMode(A0, OUTPUT);
  analogWriteResolution(10);

  pinMode(6, OUTPUT);
  pinMode(7, OUTPUT);
  pinMode(8, OUTPUT);
  pinMode(9, OUTPUT);
  pinMode(10, OUTPUT);
  pinMode(11, OUTPUT);
  pinMode(12, OUTPUT);
  pinMode(3, OUTPUT);
}

void loop() {
  for(int i = 0; i < num_points; i++) {
    analogWrite(A0, points[i].x);

    // write all 8 pins at once
    const long Y = points[i].y;
    PORT->Group[PORTA].OUT.reg = ((Y&0b11)<<20) | ((Y&0b1100)<<14) |
        ((Y&0b10000)<<15) | ((Y&0b1100000)<<3) | ((Y&0b10000000)<<4);

    // slow down ringing
    delayMicroseconds(3);
  }
}
```

C Dissasembly comparison of bitmap calculation versus lookup

Computing bit pattern online:

```
MOVS    R0, #0x80
LDRB    R2, [R7,#2]
LSLS    R0, R0, #0xC
LSLS    R3, Y, #0x14
LSLS    R1, Y, #0xE
ANDS    R1, R5
ANDS    R3, R6
ORRS    R3, R1
LSLS    R1, Y, #0xF
ANDS    R1, R0
MOVS    R0, #0xC0
ORRS    R3, R1
LSLS    R0, R0, #2
LSLS    R1, Y, #3
ANDS    R1, R0
ORRS    R3, R1
MOVS    R1, #0x80
LSLS    Y, Y, #4
LSLS    R1, R1, #4
ANDS    R2, R1
ORRS    R3, R2
LDR     R2, =0x41004400
STR     R3, [R2,#0x10]
```

Using a lookup table of precomputed bit patterns:

```
LDR     R3, =porta_bitmap_lut
LSLS    R5, R5, #2
LDR     R2, [R5,R3]
LDR     R3, =0x41004400
STR     R2, [R3,#0x10]
```