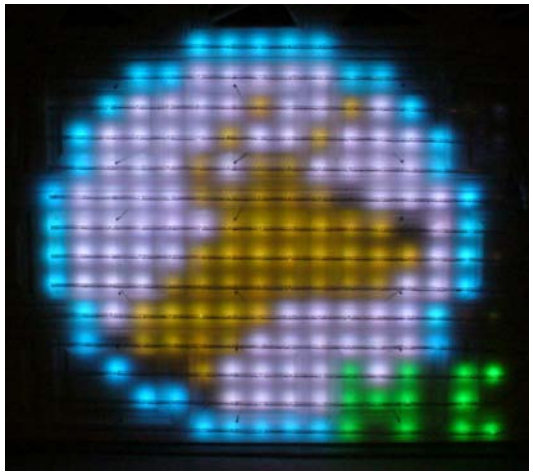
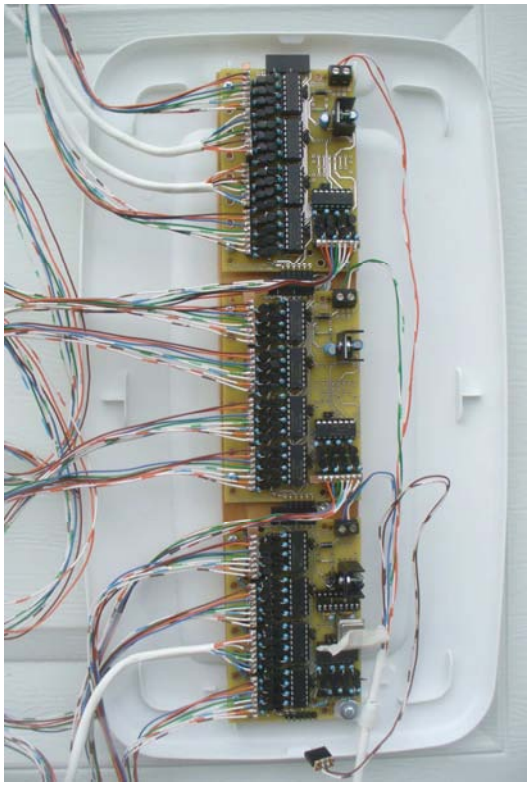
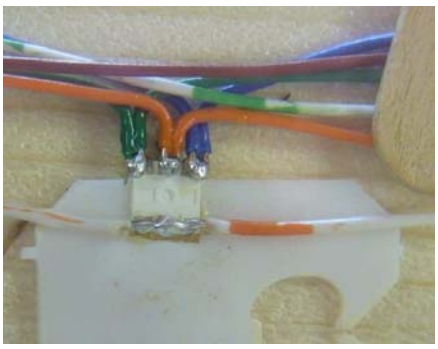






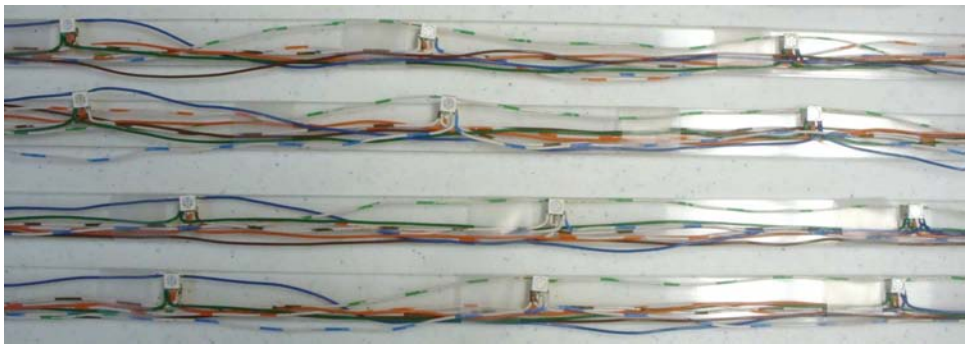
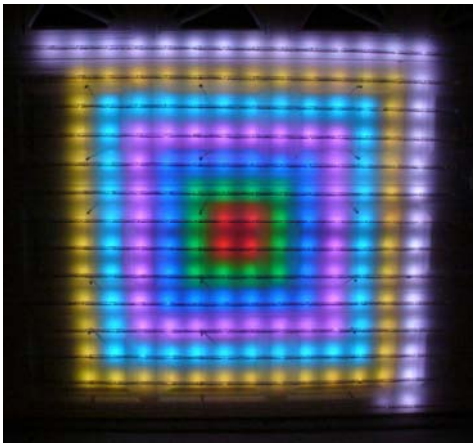


Table of Contents

1. Introduction

This article describes how I built a grid of dumb RGB pixels (charlieplexed) on a sectional garage door, and ran it using an 840 channel Renard-HC controller. This was daisy-chained to a 280 channel Renard-HC running incandescent yard props, all driven by Vixen. The general look and style of the RGB pixel grid is shown below:



Front view

Side view

Works while garage door is moving

Since the overall project or some of the techniques might be of interest to other DIYC members, I will try to describe them in the remainder of this article. However, please be aware that there are many types of display environments and many styles of sequencing, so I cannot guarantee that you will get satisfactory results if you use this info. Also note that I am not an electronics expert, so there are probably details of this project that are non-optimal or flat-out wrong. As always, use your best judgment and validate any info found here against other, more reliable sources. I would be interested to hear of any suggestions for improvements.

Now on to the details... First I'll summarize the motivation and main design goals of this project. Then I'll include a brief summary of the main aspects of LEDs that I found helpful to know when working on this project. After that, I'll describe the details of the RGB grid itself, and then describe the controller and the sequencing briefly. I won't go into much detail on the controller design, since that is already described in a separate article[1], but I will describe construction since there is now a PCB.

My apologies to non-US readers: When I started adding text to show equivalent metric numbers, the text was starting to get a little cluttered up, so I just went back to using only US measurements. Sorry about that.

2. Motivation/Design Goals

For a few years now I have admired the beautiful RGB Mega-trees featured by several of the DIYC forum members. Inspired by AussiePhil's pioneering construction techniques for DIY RGB strings[2], and facilitated by Fathead45's PLCC6 5050 RGB LED group buy[3], I finally decided to try building an RGB grid myself.

I had several design goals when I started working on the dumb pixel grid:

- A simple, low-cost RGB grid that would give me "just good enough" graphics. I didn't really need high speed or capacity, just a few simultaneous colors. I figured that something like 32 x 16 RGB pixels, with a palette size of maybe 5 – 10 simultaneous 24-bit colors that could update at least 10x/second would be sufficient for the sequences that I would be running.

[1] http://doityourselfchristmas.com/forums/showthread.php?10889 High-channel-count-Renard (chipiplexed)
[2] http://doityourselfchristmas.com/forums/showthread.php?11081 Aussiephil's 2010 journey - RGB Megatree
[3] http://doityourselfchristmas.com/forums/showthread.php?11202 OFFICIAL GROUP BUY: PLCC-6 RGB LED's Ends April …

- A unified data stream. According to the info I read, other types of graphics (true video, LEDTRIKS, etc) run asynchronously to Vixen, which makes it difficult to precisely sync those graphics with the rest of the display. You also need additional, separate cables to run those. Based on the compression ratios I was getting with the Renard-HC from last year's sequences, it looked like there would be enough bandwidth to embed limited graphics directly into the Renard-HC data stream. A shared data stream seemed like a nice way to help cut down on cabling.
- Uniform hardware. Since the Renard-HC controller was able to run charlieplexed AC SSRs, and since the input side of an optocoupler is just an IR LED, it seemed like I could use the same approach to run visible LEDs as well. The only difference would be that the visible LEDs would need to use pseudo-PWM mode to make them bright enough to see, whereas the AC SSRs didn't need that since they are able to latch by themselves (incandescent strings). Using just one controller design for the entire display (RGB grid as well as discrete incandescent strings) seemed like a nice solution, so I decided to make that a goal. A secondary motivation here was to retrofit/upgrade my existing DIYC Renard[4] to become a Renard-HC, so I could reuse all my existing controller hardware and AC SSRs.
- Fill in some missing functionality in Vixen to make sequencing of RGB props easier. I actually did try an eval version of LightShow Pro[5], but it seemed to have some limitations: it did not have a plug-in architecture so I could not use it with my Renard-HC controller, and it didn't seem to allow precision syncing due to its variable time events (although I might have misunderstood that feature). Any way, I figured that any RGB functionality that I was able to add to Vixen might be helpful to others who are using Vixen to control RGB props in their displays.
- Further experimentation with charlieplexing, chipiplexing and the PIC16F688 to find out what the practical limits are. Previously, I thought that I would need to use a PIC with higher power dissipation to drive visible LEDs, but it turns out that the 16F688 could provide enough power for my dumb RGB pixels. This avoided the need to port the Renard-HC firmware to another processor.

I know there are already several excellent solutions available for RGB grids (and even more in progress)[6], and the world doesn't need yet another type of RGB controller, but since this is a DIY forum I wanted to try out some variations at the low end that I felt were overlooked. Most of the RGB grids and trees use "smart pixels" (with a controller dedicated to each pixel), and that is probably the best way to do it for pixels that are spread out, but that tends to cost more and is difficult to construct as a DIY project (I haven't learned how to work with SMD parts yet). So I decided to try using bare RGB LEDs as dumb pixels (no dedicated controllers).

My target application also had some unique form factor requirements that I had not seen addressed by other approaches, so I wanted the additional flexibility that comes with a DIY solution. Specifically:

- I wanted to attach the grid to an operational sectional garage door, so some parts of it moved with respect to other parts and to the power supply. This made for a nice extra little challenge.
- I prefer props that are "barely there". That is, the props have a very small physical foot print so they are barely noticeable in daylight and invisible at night. This reduces the chances of damage from high winds, and allows props behind them to be seen, but makes diffusion of the light more challenging.

My ultimate goal is to find a modular, low-cost display architecture capable of driving about 10K RGB pixels distributed over a large area, so a dumb RGB pixel grid seemed like it might be a useful exploratory step in that direction. I also wanted to be able to intermix RGB pixels with 120 VAC incandescent or LED strings using the same type of controller, which is why I used the Renard-HC for this project. For added amusement, I'm also trying to determine whether I can do this with something like 2K pixels per COM port running at 115K baud and a 50 msec refresh rate (which would require a minimum 12:1 sustained data compression rate).

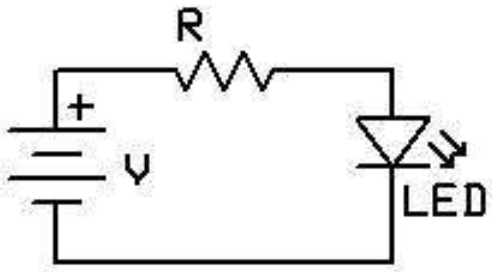
[4] I'm using the term "DIYC Renard" for the standard DIYC Renard circuit, and "Renard-HC" for the chipiplexed version.
[5] I downloaded a trial version 1.7 from http://www.lightshowpro.com
[6] For example, http://doityourselfchristmas.com/forums/showthread.php?13062 RGB LEDs Now Consumer Grade …

3. LED Display Principles

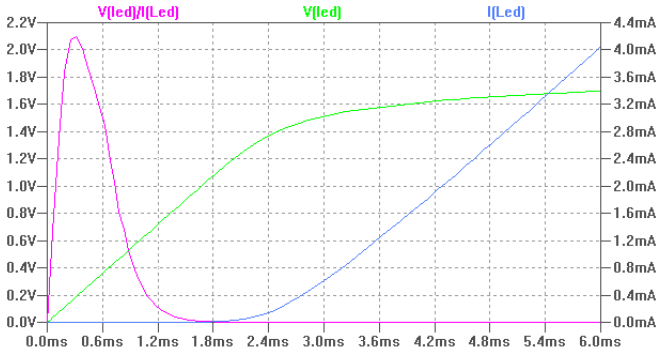
In this chapter I will try to briefly summarize some relevant details about visible LEDs that affected or laid the groundwork for my pixel grid. I believe this info to be accurate, but I am probably over-simplifying it, so please consult more reputable sources on the internet or elsewhere if this doesn't sound correct.

3.1. LED Circuits

Let's start with the basic LED circuit itself. There is tons of info about this elsewhere (for example, Wikipedia has some good explanations[7]), so I won't go into much detail, but here is a brief summary.



Example LED circuit



LED voltage vs. current = effective resistance

An LED, being a diode, is a directional device that only conducts in one direction. It acts like a non-linear resistor, so it has a resistance when current flows through it, but that current is not always proportional to the voltage drop across it (in the chart above, voltage is shown in green, current in blue, and effective resistance in pink). It will not light up until there is enough voltage applied across it (or current flowing through it) to overcome its internal "resistance" (where the blue line starts to climb). Within its operating range, an LED tends to have a nearly fixed voltage drop across it (green line levels off in the chart), so that if you apply at least that voltage, it will light up. However, if you apply more than that voltage, the excess will basically act like a short circuit and result in a large amount of current (pink line stays at 0 after a while; blue line continues to climb even though green line levels off), which will burn out the LED. It is not feasible to apply exactly the correct voltage to an LED, due to manufacturing variations and because the characteristics even vary according to operating temperature. Therefore, additional components are needed to protect the LED from damage due to excess current – this is the purpose of the series resistor shown in the example circuit above.

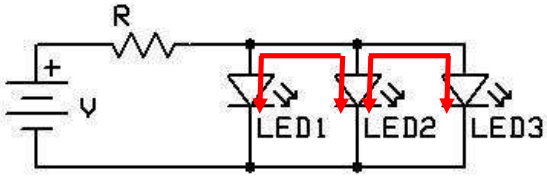
Since the brightness of an LED depends on the amount of current flowing through it, for visible LED applications it is desirable to run it as close to its limit as possible. In more sophisticated LED circuits, a "constant current" circuit is used to constantly monitor the current flowing through the LED, and then adjust the current to keep it within a narrowly defined range. However, this adds complexity and cost to the circuit. There is also a "constant voltage" type of circuit, which tries to control the voltage instead of current.

A cheaper/simpler approach is simply to add a resistor in series with the LED to "absorb" (waste) the extra current. This is not as precise as the constant current circuit, but is safe and reliable if you take into account the tolerance of the resistor, voltage supply, and LED, and then add sufficient padding. An appropriate value for the series resistor can be calculated by using Ohm's Law. For example, if the LED's voltage drop is 2V and you are running it with a 5V supply, and you want to limit the LED supply current to 20 mA, then you would use a (5V – 2V) ÷ 20mA = 150 Ω resistor in series. If the supply voltage is extremely well regulated, and guaranteed to never exceed the LED's voltage drop, then you can skip the series resistor, but this is not typically the case with our logic circuits and power supplies.

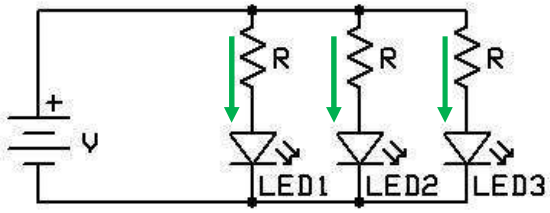
Also note that the voltage drop varies according to the type of LED, since different materials are used to produce the different colors. For example, a GaAs red LED is typically around 2V, while a GaP green LED is

[7] http://en.wikipedia.org/wiki/LED_circuit has a good explanation

around 3V and SiC blue LED is around 3.4V. This needs to be taken into account when calculating series resistor values for RGB LED circuits, since the LED colors are of different types. There is a link to an example LED comparison chart near the end of this article.



Unsafe: possible leakage between branches



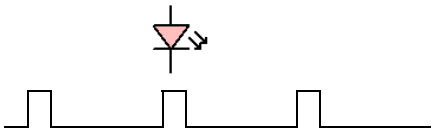
Safe: current controlled individually

In circuits with multiple LEDs, one or both ends are often connected together in order to reduce the wiring. However, because the voltage/current characteristics of the LEDs will vary (no 2 devices are identical), you cannot just connect both ends of the LEDs in parallel – any difference in the effective resistance between the LEDs can damage the LED with the lower resistance or starve the LED with the higher resistance, due to stray leakage current (shown by the red arrows, above). To avoid this, a series resistor needs to be placed in each parallel LED branch, so the current can be individually controlled for each LED (green arrows, above).

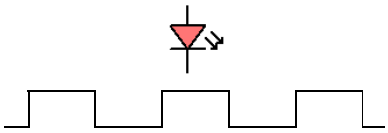
In a "common anode" circuit, the LEDs share the same anode (positive) connection, and in a "common cathode" circuit, they share the same cathode (negative) connection – the example on the right, above, is a common cathode circuit.

3.2. Controlling LED Brightness

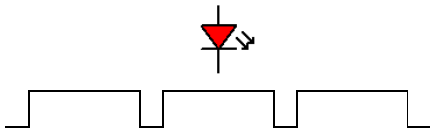
Even though the brightness of an LED depends on the current flowing through it, the curve is not linear so it is difficult to precisely control brightness this way. Instead, Pulse Width Modulation (PWM) is used to turn the LED on and off repeatedly, and then the duty cycle (portion of time on compared to off) is used to control the apparent brightness. If this is done fast enough, the LED will look like it is constantly on (at the intended brightness), but if the refresh rate is too low, the human eye will see flicker. This flicker rate threshold varies slightly from one person to another, but around 60 Hz seems to be the lower limit. The edges of our eyes are also more sensitive to flicker than the centers, which is why you can see flicker in your peripheral vision more than if you look straight at the LEDs.



1/6 duty cycle (dim)



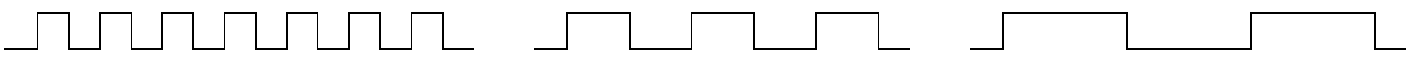
3/6 duty cycle (half intensity)



5/6 duty cycle (bright)

PWM allows precise control of LED brightness, because it allows the circuit designer to select a safe, constant current limit (using a resistor or constant current feedback loop, as described earlier) to maximize LED brightness, and then use simple on/off control to vary the apparent LED brightness. This basically reduces an analog problem of voltage/current to a digital problem of on/off control, which is easier for a microcontroller.

Another thing I found interesting about PWM is that the actual frequency that the LED is turned on/off doesn't matter (as long as it's faster than the eye's "flicker threshold"). For example, the PWM waveforms below will all produce the same apparent brightness for the LED, since they have the same duty cycle (of 50%):



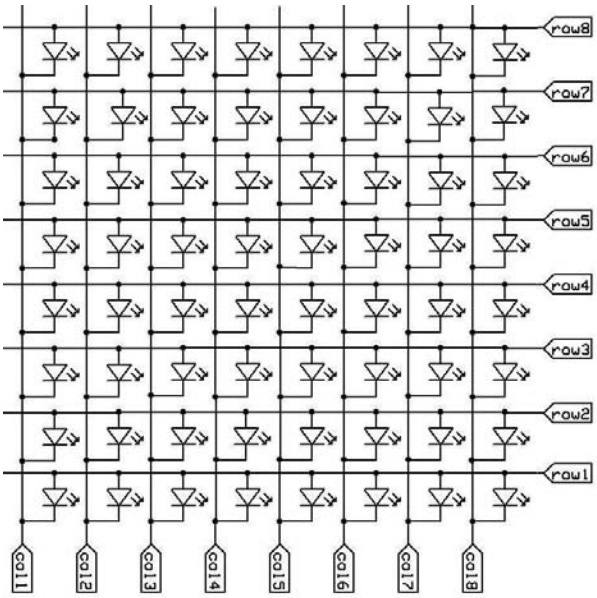
Different PWM frequencies, same duty cycle and brightness

For visible LED applications, more brightness is usually desirable. Sometimes circuits will "cheat" by driving LEDs a little more than their maximum rated current, in order to obtain more brightness. Using PWM, you can run excess current through the LED if you ensure that it is not on long enough that heat builds up in the LED

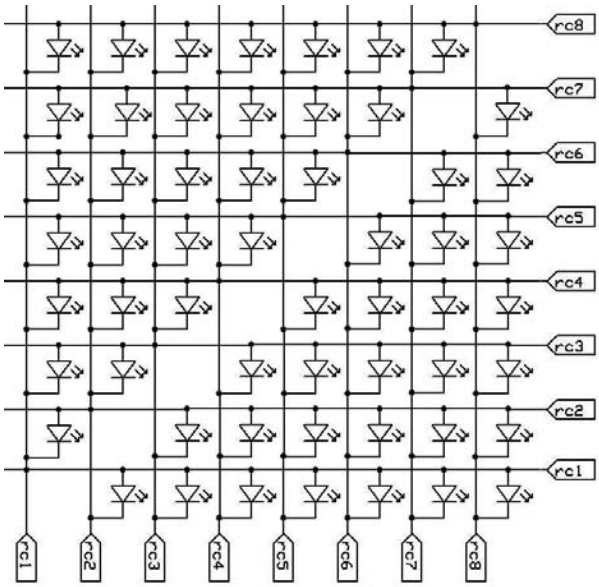
and destroys it. For example, you might be able to run a 20mA LED at 50 mA if you repeatedly turn it on and off so that it is on for only, say, 20% of the time. LED datasheets often quote maximum steady vs. pulsed current for this reason. Depending on how much you cheat, you may or may not shorten the life of the LED.

3.3. Connecting Large Groups of LEDs

RGB pixel grids and arrays, by their very nature, involve larger numbers of LEDs and channels. LEDs are typically connected in a row/column matrix to reduce the number of connections and control signals needed, and this works well if the LEDs are physically close to each other. For example, an 8 x 8 row/column matrix contains 64 LEDs, and only requires 8 + 8 = 16 control signals. In the example below on the left, setting row 8 high and column 1 low would turn on only the upper-left LED:



8 x 8 row/column matrix (16 lines)

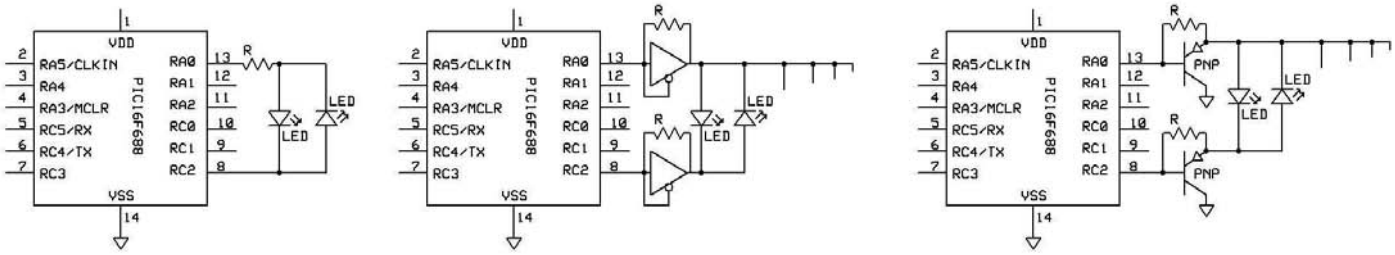


8 x 7 charlieplexing (8 lines)

Charlieplexing is an alternate way to connect a similar but slightly smaller number of LEDs using only half the number of control wires. This technique is documented in Microchip and other literature, so I won't go into a lot of detail, but here is a brief summary. Each signal controls both a row and a column, but the same line cannot perform both functions at the same time, so a diagonal line of LEDs is missing from the circuit. For example, 8 control signals can control 8 x 7 = 56 LEDs. In the example above on the right, setting row/column 8 high and row/column 1 low would turn on the upper left LED. A positive or negative voltage can only be applied to the control lines going to the LEDs that you want to turn on, otherwise additional LEDs would also turn on. Therefore charlieplexing requires the use of tri-state control signals. The reduced cabling and fewer control signals are what initially attracted me to the idea of using charlieplexing, along with the extra little challenge of trying to get it to work with RGB LEDs. ☺

Either approach reduces the number of control lines needed, but the multiplexing nature only allows one row of LEDs to be turned on at a time, which reduces the brightness of the LEDs according to the duty cycle. As the number of rows increases, the duty cycle gets smaller, so a typical practical limit to the number of rows might be 8 to 10 unless the LED current is severely over-driven.

The smaller number of control lines and their tri-state nature make charlieplexing well-suited for use with a microcontroller. Since LEDs do not require much current, they can be driven directly from the microcontroller I/O pins with nothing more than a series resistor, making the circuit very simple. Each LED is connected between a *pair* of microcontroller I/O pins, and the LED is turned on by setting one I/O pin high and the other low. Since LEDs only conduct in one direction, pairs of LEDs can be driven this way. In the examples below, 2 LEDs is a boring example, but the benefit is greater with larger numbers of LEDs.



Charlieplexed LEDs *Tri-state buffers* *Chipiplexed buffers*

Now due to the limited drive capability of the I/O pins, only 1 or a few LEDs can be turned on simultaneously. However, in a charlieplexed matrix of LEDs, we need to be able to turn on an entire row at a time in order to maintain a reasonable duty cycle and brightness. Therefore, a tri-state buffer is needed on the I/O pins to give them more drive capability. This tri-state buffer is actually only needed when the I/O pin is acting as the row address (voltage high if the LEDs are wired as common anode, low if they are common cathode), but it also needs to maintain the high-impedance state of the I/O pin (in order to prevent extra LEDs from being turned on). Then the current-limiting resistor can be connected across the tri-state buffer to complete the circuit when the I/O pin is acting as the column address, as shown in the middle circuit, above.

An EDN article a while back about "chipiplexing"[8] showed a nice little trick for replacing the tri-state buffers with simple bipolar transistors. This reduces the components needed to drive a charlieplexed matrix of LEDs, as shown in the circuit above on the right.

When the pixels are spread out more, the wiring of a row/column or charlieplexed matrix becomes unwieldy, so serial connections are more convenient. However, this convenience brings additional cost; a serial system requires a microcontroller dedicated to each pixel. To get this physically small enough, usually SMD and/or commercial manufacturing are needed, which are not as DIY-friendly. This is why I tried out charlieplexed pixels as an alternate approach. Of course, if commercially-produced smart pixels (with dedicated controllers) can drop low enough in price, then they could also be used in place of a dumb pixel row/column grid.

Power distribution also becomes an issue. Even though LEDs draw a fairly small amount of power individually, a large number of them will still add up to a large amount of power. This means that a long string of LEDs will need power "re-injected" into it periodically along the way. Otherwise, much heavier wire would be needed and/or the voltages at the end of the string would be much less than at the beginning, which could lead to variations in brightness.

3.4. IR LEDs vs. Visible LEDs

The input side of the MOC3023 optocouplers used in the DIYC AC SSRs consists of an IR LED optically coupled to a phototransistor on the detector side. Hence, all of the info above also applies to the DIYC AC SSRs.

There were really only 2 differences between visible LEDs and IR LEDs (optocouplers) that seemed to matter for this project:

- High brightness is not important. It is not necessary to run the IR LEDs at maximum brightness, since they are not visible. However, running them brighter does allow the optodetector to turn on faster; in cases where you want shorter turn-on times (when the SSRs are multiplexed), this may be important.
- Visible LEDs running on DC do not need to be synced to the AC cycle (no ZC signal is needed), so the timing of when to turn them on and off is not as critical. Only the relative on/off time matters.

These observations allowed me to use the same Renard-HC circuit and firmware for AC SSRs and dumb pixels.

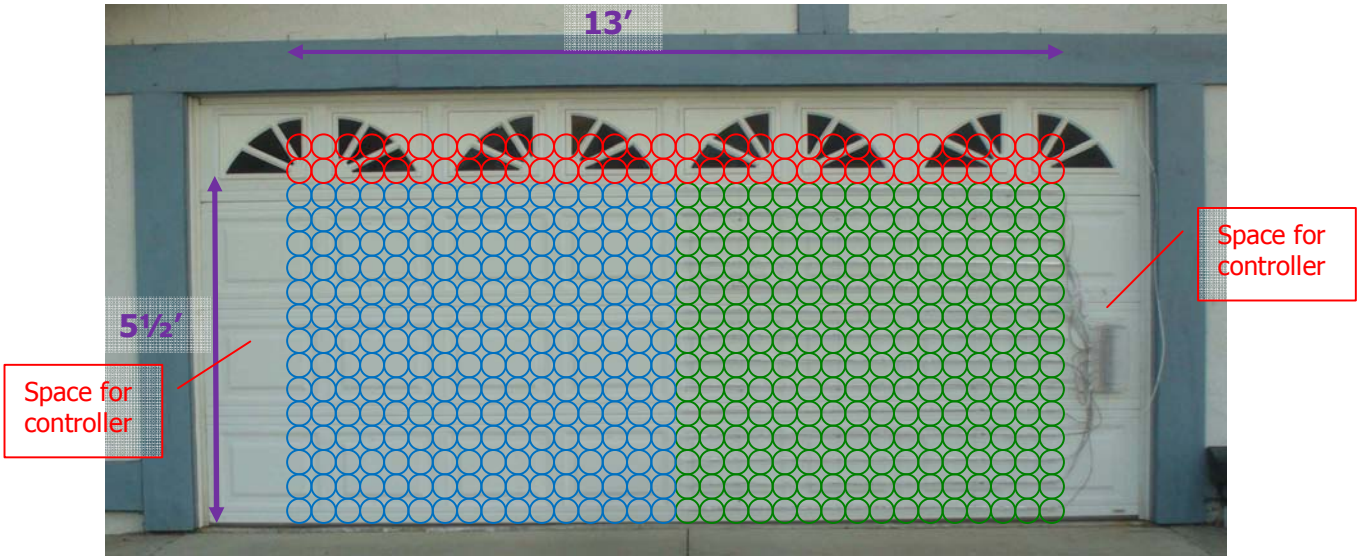
[8] http://www.edn.com/contents/images/6615603.pdf EDN Design Ideas, Nov 2008 article on Chipiplexing

4. Dumb RGB Pixel Grid Details

This chapter will describe the pixel grid itself, including some of the design considerations and construction techniques. The controller and sequencing techniques will be described briefly in following chapters.

4.1. Grid Layout

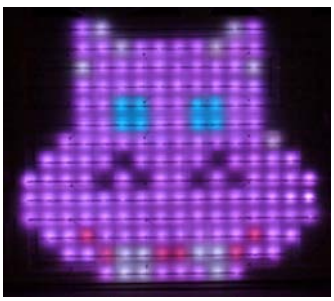
I wanted the grid to have good visibility from the street, but not to distract from the rest of the display. The garage door seemed like a good location, since it was off to the side, and it was a relatively flat and sizable "canvas". The only draw-back was that it moved and wasn't flat all the time – it is a sectional door, consisting of 4 horizontal sections. The overall size is approx 16' W x 7' H, with each section being 21" H. The upper section has windows, which I did not want to cover, so this reduced the usable height to about 5½'.



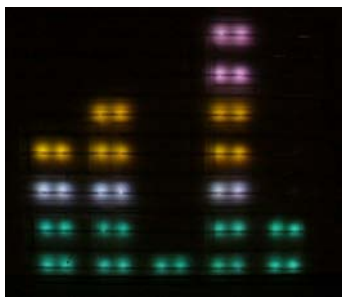
Grid dimensions and pixel layout on door

I had originally planned to use a 32x16 grid, with pixels sized to fill the garage door. I wanted good coverage of the garage door area, but I also wanted a 1:1 aspect ratio for WYSIWYG purposes. By eliminating one row, the grid fit better – this gave me a pixel size of 4¾" in diameter, for a total size of 32 x 4¾" by 15 x 4¾" ≈ 13' W by 6' H. This left about 1½' on either end, enough open space to mount the controllers. However, the height still overlapped onto the windows a little, so I dropped another row and settled on 32x14.

I built the revised intended number of pixels, but ran out of time so I only ended up with half of them installed and running. Although this only gave me a 16x14 grid to work with rather than 32x16, it was still enough to experiment with and display simple icon-like graphics as part of my sequences. Here are some examples:

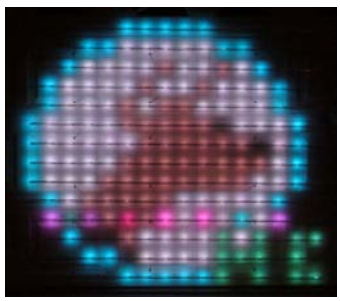


Hippo



Equalizer bar graph

New Years message



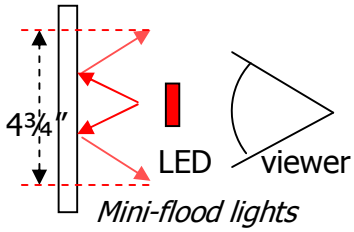
Vixen-HC logo

The grid could also do some limited text messages using small fonts - one example is shown above.

4.2. Pixel Structure

LEDs are point-sources of light, but I wanted the light to diffuse and fill out the 4¾" area allocated for each pixel. The garage door made an ideal diffusion surface. The PLCC6 LEDs from the DIYC group buy were bright enough to act as miniature flood lights, so I just pointed them back to reflect/diffuse off the door, rather than pointing them forward to face the viewer. The PLCC6 LED body is well-suited to this arrangement, because it is flat and square, and can be aligned easily if fastened to something. The technique I used to mount the LEDs allowed them to be pointed either way, or flipped easily. See below for a comparison:

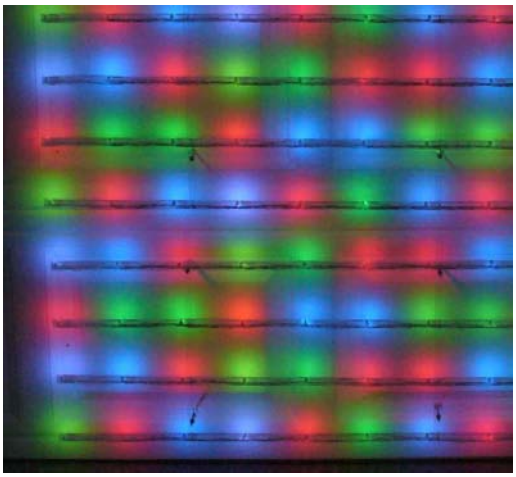
Point source vs. diffused mini-floods



Mini-flood lights

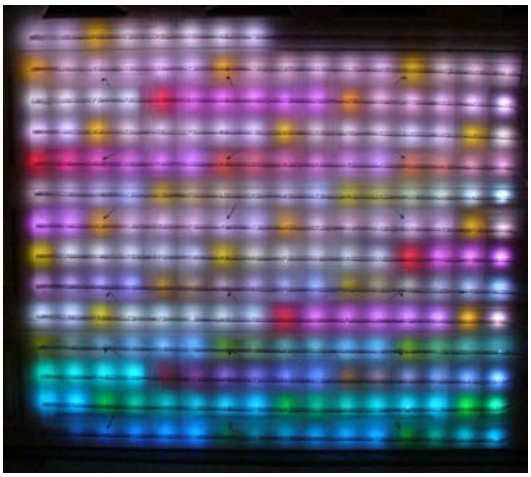
Besides giving the grid a softer, more "filled in" appearance, other advantages to backward-facing pixels were:

- lower cost; no additional materials were needed for a diffuser (just use the garage door)
- construction was simpler (no extra container/diffuser needed), and the pixels were physically much smaller (basically, just the bare LED), which supported my "barely there" goal
- reflected pixel size can be adjusted simply by changing the distance of the LEDs from the garage door



R/G/B test pattern

White mixing



Web-safe colors

The PLCC6 LEDs that I used appeared to have good color mixing - with red, green and blue all on, the LEDs appeared reasonably white, with only a slight tint (as shown in the middle photo above). The red, green and blue diodes seem to be close enough together so that the colors mixed well, even when "magnified" by the reflection/diffusion arrangement that I used. The photo on the right above shows the "web safe colors" (R, G and B multiples of 51), starting with [R,G,B] = [0,0,0] in the bottom left, then [0,0,51] beside it, then [0,0,102], … [0,0,255], then [0,51,0] … [0,51,255], etc. (I added 51 each time, with B being the least significant color, and they are in groups of 6). The colors actually looked a little better in person than in the photo (the camera I used did not have good sensitivity when farther away from the grid), but at least this shows that the pixel grid is able to generate a variety of colors – maybe not 16M separate shades, but certainly enough for the simple graphics I used in my sequences. There is a link to a PLCC6 LED datasheet near the end of this article.

I suppose just about any type of RGB LED would work as long as it is bright enough and the colors mix well (see the note later about 5 mm 4-pin LEDs).

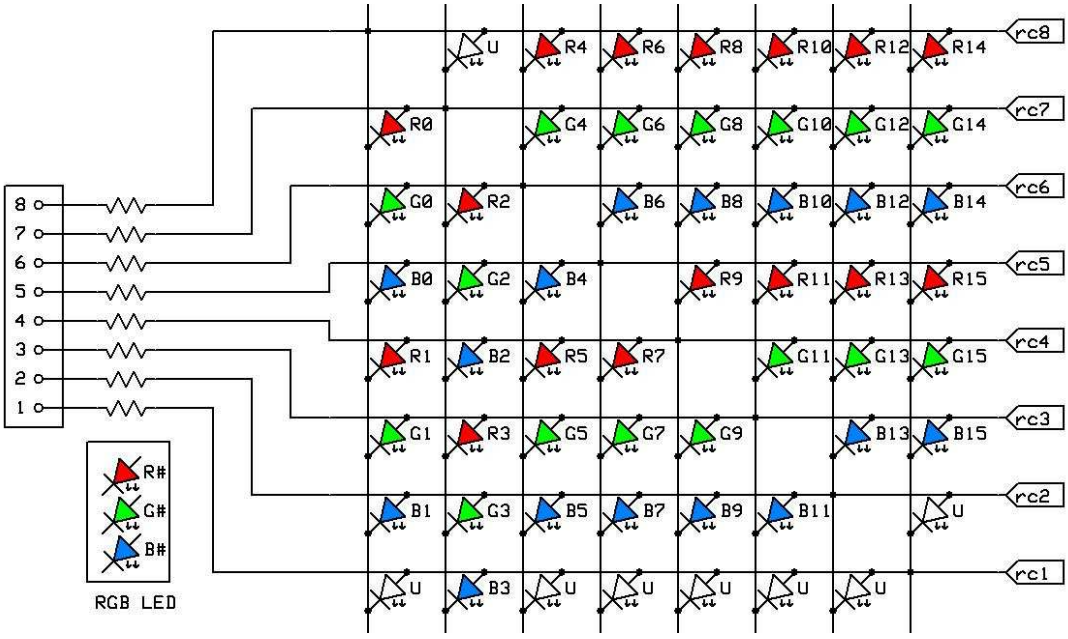
4.3. Circuit Diagram

Since I was using dumb pixels, I could have just wired them in a regular row/column matrix, similar to a LEDTRIKS[9]. However, several factors led me to use strings instead of a matrix:

- The garage door sections moved with respect to each other. I could have allowed a little extra wire at the joints to accommodate this movement, but individual strings of lights (organized in rows) seemed to fit the shape more naturally than a rectangular matrix.
- Using shorter strings rather than a full matrix would allow easier replacement of a failed pixel. One pixel did actually go out after a few weeks, due to a poor solder joint. I was able to replace the string with a spare in about 5 minutes, simply by unplugging the failed string and plugging in another one.
- I wanted the general construction style to be re-usable for lines and other non-rectangular shapes, as well as the rectangular garage door grid.
- When commercial light strings of individually addressable pixels drop to a more affordable price level, I want to be able to switch to them but still use the same general layout technique and Vixen sequences. Since those are now organized as linear strings, I wanted to keep the same shape.
- I wanted to play around with charlieplexed arrangements to find out more about their limitations.

Having decided to go with "strings" rather than a hard-wired row/column matrix, the next step was to choose the string length. Since the Renard-HC controller can drive 8 x 7 = 56 LEDs per port (using all the wires in a Cat5 cable for data signals), and I was investigating whether the Renard-HC could also drive a pixel grid, that gave a maximum of 56 ÷ 3 = 18 RGB LEDs per Renard-HC port (with 2 channels left over, perhaps for a 2-color status LED). Now the PLCC6 LEDs have 6 pins, so they are actually more like separate R, G and B LEDs and 18 of them can be charlieplexed and driven with a Cat5 cable. However, 4-pin RGB LEDs are more common, in which the anode or cathode is common to all 3 diodes.

Looking at an 8 x 7 matrix of charlieplexed LEDs (shown below), only 16 RGB LEDs of one polarity (common anode or common cathode) can be connected. If I mixed in 2 of the opposite polarity, I could use 18, but I wanted to use all the same type of LEDs within any given string, and 16 is a nice round binary number, so I settled on 16 RGB LEDs per string, leaving 8 unused channels per port. That meant that each PIC I/O pin would be driving an average of 2 RGB LEDs (16 RGB LEDs ÷ 8 I/O pins = 2 per pin), so there is a compression ratio of 6:1 inherent with this scheme, as compared to other types of controllers that dedicate one I/O pin per LED (3 pins for an RGB LED).



With an 8 x 7 charlieplexed matrix, 16 RGB LEDs of same polarity can be connected

[9] http://doityourselfchristmas.com/forums/forumdisplay.php?13-Ledtriks DIYC LEDTRIKS

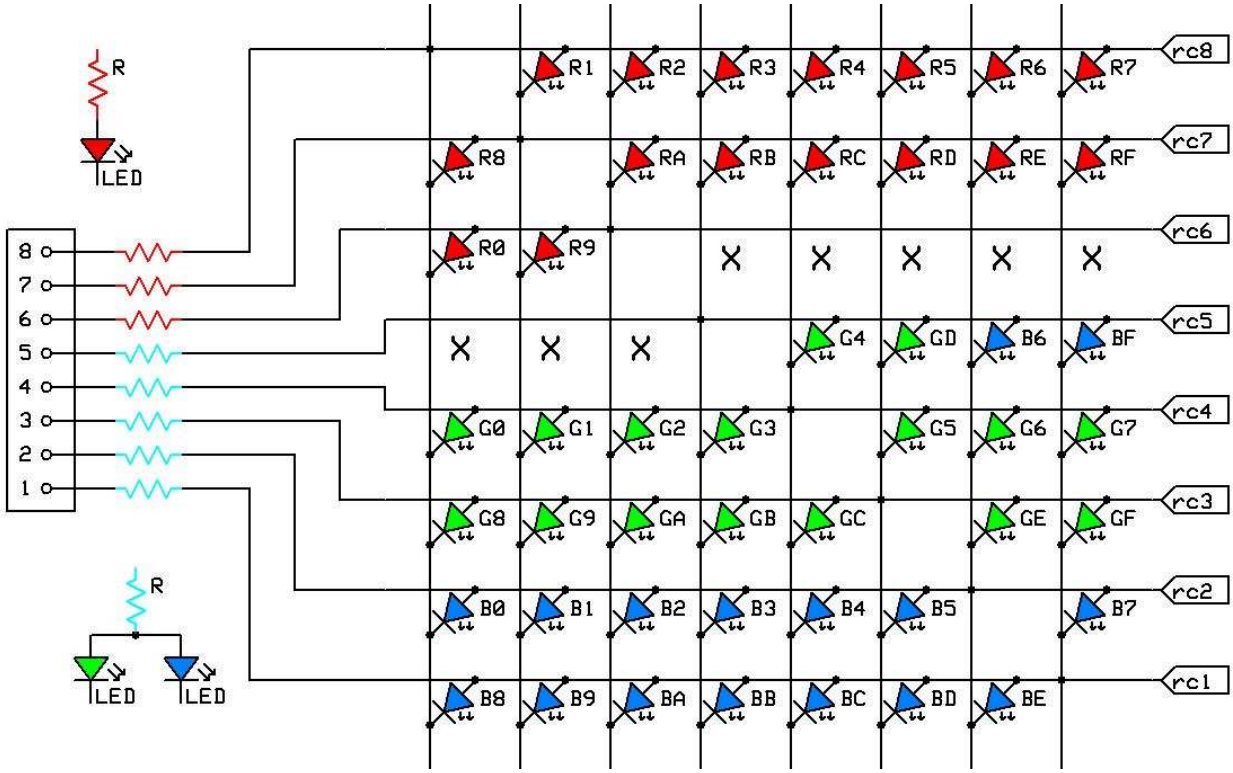
Using this circuit, the controller basically drives the RGB LEDs in pairs at a time (via 2 common cathodes), since there are 8 "rows" of 2 RGB "columns". This gives a 1/8 duty cycle, which is comparable to LEDTRIKS or other row/column multiplexed LED grids. This reduces the brightness, but also the power consumption – if the LEDs are run at 20mA, then each string of 16 RGB pixels can only draw at most 2 x 3 x 20mA = 120 mA when the pixels are white (R, G and B all on), because only 2 R, G, and B LEDs can actually be on at a time, instead of 16 x 3 x 20 mA = 960 mA when all 16 RGB LEDs are on at the same time.

I used a common-cathode circuit, so the R/G/B LEDs in the diagram above and throughout this article are labeled that way, but charlieplexing will also work with common anode LEDs – just remember to reverse the polarity of the driver transistors to match.

4.3.1. Resistor values

In a multiplexed matrix (charlieplexed or row/column), the series resistor is shared between all LEDs in the column. Now there's an interesting little puzzle – red, green, and blue LEDs have different voltage drops, so sharing the same series resistor between them will mean that their drive current will be different, which will cause the colors to vary in brightness. That is, driving one of the colors at its maximum current will prevent the other 2 colors from being driven at their maximum current, given a shared voltage supply and series resistor. However, since 8 x 7 charlieplexing supports 56 LEDs and only 16 x 3 = 48 are needed for 16 RGB LEDs, the 8 "spares" can be rearranged for more uniform LED brightness.

The trick is to group the 2 colors with the closest voltage drop together with one series resistor, and then run the third color by itself with a different series resistor value. For example, from the datasheet for the PLCC6 LEDs that I used: Vred = 2.0V, Vgreen = 3.2V and Vblue = 3.2V. This was nice because the green and blue were the same (although the samples I measured actually differed a little, but close enough). In this case, the green and blue can share the same series resistor without sacrificing brightness, and then the red can be run using a separate resistor to obtain its maximum brightness also. So 3 of the 8 control lines can be used to drive the red LEDs, while the other 5 control lines can drive the green and blue LEDs, as shown in the following circuit diagram:



Charlieplexed 8 x 7 matrix with LEDs and spares arranged for resistor-sharing

When connected to a 5V DC supply, the entire supply voltage lies across the LED + series resistor. However, in a charlieplexed circuit, the supply voltage is only the difference between 2 I/O pins (one high and one low), for an effective voltage supply of something like 5V – 1.3V = 3.7V. This means a smaller series resistor is needed for maximum brightness. Some sample calculations are shown below for a PLCC6 RGB LED:

Diode color	Voltage drop	Series R with 5 V supply	Series R with 3.7V supply
Red	2.0 V typical	(5V-2V)/20mA = 150 Ω	(5V-1.3V-2V)/20mA = 85 Ω
Green	3.2 V typical	(5V-3.2V)/20mA = 90 Ω	(5V-1.3V-3.2V)/20mA = 25 Ω
Blue	3.2 V typical	(5V-3.2V)/20mA = 90 Ω	(5V-1.3V-3.2V)/20mA = 25 Ω

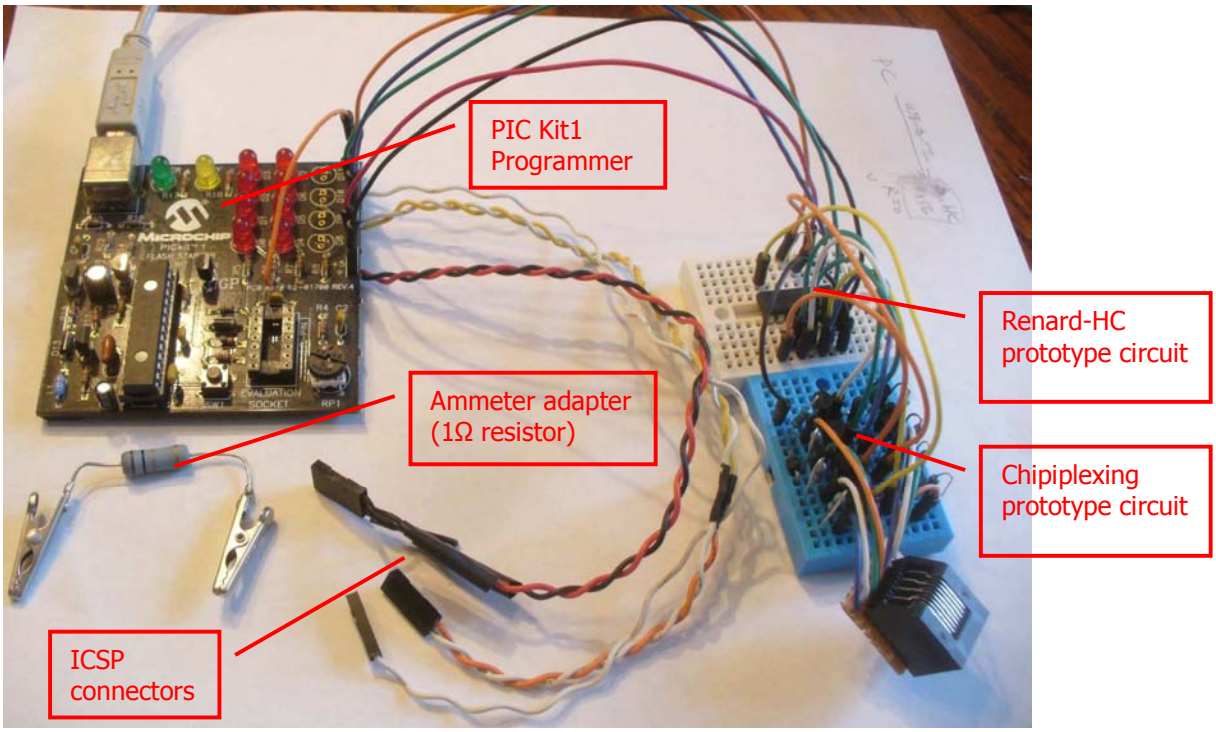
Each PIC16F688 in the Renard-HC has a combined limit of 90 mA[10] for all I/O pins. Originally I intended to run the RGB LEDs at 20mA per color, which would take 6 x 20mA = 120 mA for a pair of white pixels (R, G, and B all on). However, I found that the PLCC6 LEDs seemed bright enough at lower current, so I chose 100 Ω for the red LEDs and 47 Ω for the green and blue LEDs, which would theoretically drive the reds at (5V-1.3V-2V) ÷ 100Ω = 17 mA and the greens and blues at (5V-1.3V-3.2V) ÷ 47Ω ≈ 11 mA. Since 2 x 17 mA + 4 x 11 mA = 78 mA, I was safely within the total I/O limit of 90 mA for the PIC16F688. The greens actually looked a little brighter than the blues, but they needed to share the same resistor so I just went with these values.

Another factor to consider is what happens if there is a short circuit. When the LED is driven with a regular DC voltage supply, if the LED is shorted then the full voltage is applied across the series resistor and this just wastes a little power. However, in a charlieplexed circuit, I/O pins are used as the supply voltage, and they have a much lower current limit and can be damaged easily. For example, using the series resistor values shown above, if a charlieplexed red LED becomes shorted, then (5V-1.3V-0V) ÷ 85Ω ≈ 44 mA will flow through the I/O pins instead of the intended 20 mA, or (5V-1.3V-0V) ÷ 25 Ω = 148mA will flow for a shorted green or blue LED. There is not a good solution for this – I suppose a low enough wattage resistor would act like a fuse if it burns out faster than the microcontroller I/O pins. ☹ The fancier constant current circuits are more reliable in this regard. OTOH, we're only talking about a $1.80 PIC in the Renard-HC, so the cost/ complexity trade-off vs. reliability may still be worth it. I use sockets for the PICs in my Renard-HC anyway, so they can be replaced easily 😉 (I only burned up 1 PIC during my prototyping).

4.3.2. Circuit Validation

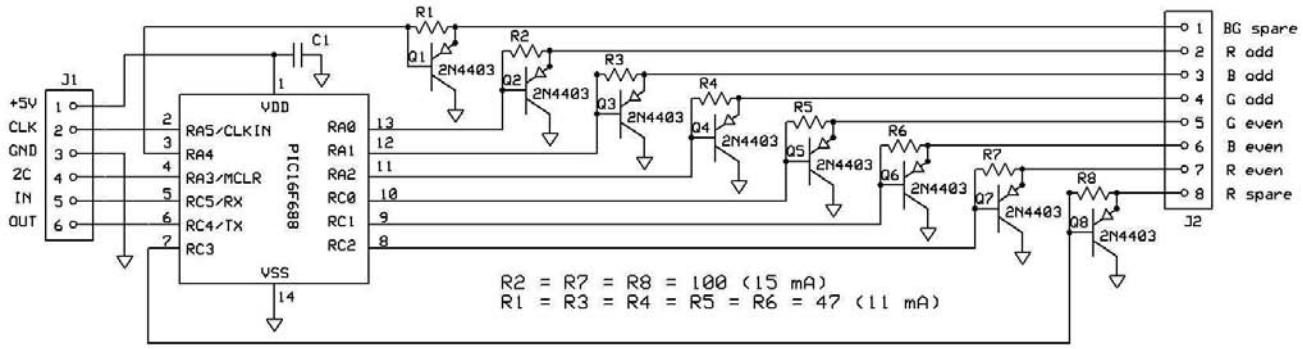
To verify that the charliedplexed LEDs would work as intended and that I had chosen correct resistor values, I prototyped a chipiplexed Renard-HC circuit and ran a small hard-coded PIC program to sequence through all the primary colors for each pair of RGB LEDs. It took several tries to get the LEDs to light up in the correct order until I mapped out which pair of I/O pins went to each LED. I also modeled the circuit using LTSpice[11] as an additional sanity check. Here's my "high-tech" prototyping setup: ☺

[10] According to Rev C or later of the datasheet, http://ww1.microchip.com/downloads/en/devicedoc/41203d.pdf
[11] A free download from http://www.linear.com/designtools/software/ltspice.jsp; There is also a Yahoo group, LTspice.



My prototyping setup

The test circuit for the chipiplexed Renard-HC is shown below (the 16 RGB LED circuit was shown earlier):



Chipiplexed Renard-HC test circuit (common cathode/PNP)

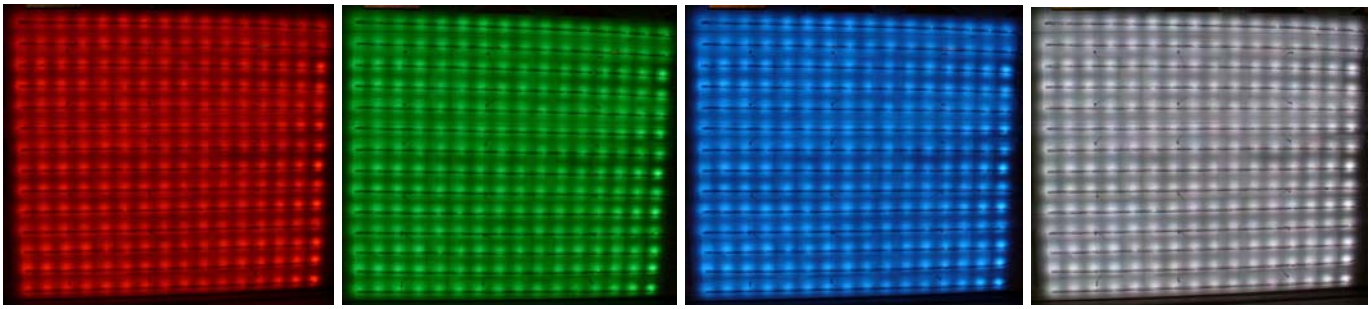
To avoid burning out everything, I doubled the resistor values for the first few tests, then measured the voltages and currents at various places in the circuit to check for correct operation. Since the measured numbers were within the expected range, I reran the test with the real resistor values. Still no smoke. ☺

LED pair color	PIC total	RA0		RA1		RA2		RA4		RC0		RC1		RC2		RC3	
		#1	#2	#1	#2	#1	#2	#1	#2	#1	#2	#1	#2	#1	#2	#1	#2
Off	1.2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Red	27.8	6.8	12.2	0	0	0	0	13.1	0	0	0	0	14.5	13.1	12.2	0	0
Green	20.3	6.7	0	0	0	9.4	8.8	0	0	9.4	9.4	0	13.9	0	0	0	0
Blue	22.9	6.7	0	10.4	9.9	0	0	0	0	0	0	10.8	14.1	0	0	0	9.7
Magenta	44.8	7.1	11.6	9.5	8.6	0	0	12.4	0	0	0	9.6	15.1	12.4	11.7	0	8.4
Cyan	37.9	7.0	0	9.8	9.0	8.6	7.7	0	0	8.5	7.8	9.9	14.9	0	0	0	8.8
Yellow	42.5	7.1	11.7	0	0	8.4	7.5	12.4	0	8.3	7.5	0	15.0	12.5	11.7	0	0
White	57.1	7.2	11.1	8.9	8.1	7.6	6.8	12.0	0	7.6	6.8	8.9	15.4	11.8	11.2	0	7.8

Sample test circuit measurements, in mA (R with 100Ω, G and B with 47Ω)

The table above shows the actual measurements I observed in the test circuit for 2 of the LED pairs (during different test runs). The results are similar, except for when a different I/O pin became the common cathode (RA0 for #1, RC1 for #2) or the I/O pin was unused (RC3 for #1, RA4 for #2). So on average, it looks like the red LEDs drew around 13 - 14 mA each, the green LEDs drew 9 - 10 mA, and the blues were 10 - 11 mA. This was very close to my targets of 15 mA red, 11 mA green, and 13 mA blue, so I was happy with these resistor vales. I suppose I could have fine-tuned them to get a little more current/brightness out of the LEDs, but I wanted to leave safety padding to allow for parts variations and resistor tolerances. This is probably as close as I can get by just using a simple series resistor rather than a "constant current" type of circuit.

Although the numbers varied somewhat, the LEDs seemed to be fairly uniform in brightness, as shown in the photos below:



LED brightness comparison, R, G, B, W (R+G+B)

Colors which had 2 or all 3 of the R, G, and B LEDs on would obviously be brighter than single-LED colors. Within a color there is slight variation, but I think there are several contributing factors besides the chipiplexing circuit itself: ridges in the garage door surface, and pixel alignment. The pixel centers are also brighter than the edges (due to diffusion), so there is variation even within each pixel. I think these photos make it look worse than it was in person - the camera was not that great with low-light sensitivity. Anyway, my sequences did not have the entire grid filled with one solid color for any length of time, so I don't think the slight variations really mattered.

I used a cheap digital multi-meter to measure the voltages directly, but it didn't measure current, so I inserted a 1 Ω resistor in series and then used the measured voltage drop across it and Ohm's Law to get the current. The little PIC test program turned on each LED combination for about 4 – 5 seconds, long enough to get a reading on the multi-meter before moving on to the next one.

From the numbers above, it looks like there is some current leakage between the various LED paths (which is to be expected, since transistors and LEDs are not perfect), but the overall numbers indicated that the transistors were doing their job of taking most of the load from the I/O pins, and that the resistor values were giving approximately the desired target current in the LEDs. A final validation came when I removed the delay factor in the PIC program and was able to see the whole string light up, and the pixels looked like they were fairly uniform in brightness.

4.3.3. Returning to a Mystery

During some chipiplexing tests last year (with AC SSRs), the chipiplexing circuit did not work as expected (the transistors did not seem to turn on), so I turned them around and it worked, and I ran it that way for the rest of the 2009 season. Out of curiosity, I tried that again with the charlieplexed RGB LEDs this time, and it also worked - so the chipiplexing appears to work for the RGB LEDs when the transistors are in either orientation, but for the optocouplers in the AC SSRs it only worked when the transistors were "backwards". I could only come up with 2 possible reasons for this: either I connected something incorrectly last year, or the chipiplexing circuit needs a certain minimum current in order to start working (optos ran at ~ 5 mA, RGB LEDs are running at 10 – 15 mA). I can't go back and check the first case. I suppose the second case might make sense given

that the voltage drop across the chipiplexing resistors needs to be large enough to turn the transistors on (needs a 0.7V drop), and with smaller currents this might not have been true.

Now the transistors did not run quite as efficiently "backwards" as when connected the "correct" way (that is, according to the EDN article), but they were within about 20%. Since I wanted to drive the RGB LEDs as hard as possible, I turned the transistors to the orientation shown in the EDN article for the final circuit.

4.4. Parts List

The parts I used for each string of 16 RGB pixels are listed below:

Description	**Cost of Parts**	**Cost with tax, S&H**
16 RGB LEDs (I used PLCC6 5050s)	16 x 0.19 = 3.04	3.04
8′ of Cat3 (Cat5 would also work)	8/1000 x 85.00 = 0.68	0.68
One SIP8 socket (could also use RJ45)	0.14	0.18
Plexiglass strip 6′ L x 5/8″ W x 1/8″ thick	6 x 0.28 = 1.68	1.99
Small piece of stripboard (1 x 8 holes)	1/46 x 1.00 ≈ 0.02	0.02
4″ x 7′ plastic wrap	7/900 x 3.00 ≈ 0.02	0.02
Total	**≈ 5.58**	**≈ 5.93**
Per pixel cost	**0.35**	**≈ 0.37**
Compare: ADSIC12RGB (LPD6803)	**0.59**	**≈ 0.75**

The actual numbers for the quantities that I bought are shown above. I tried to factor in the tax and shipping, but since unit prices and shipping costs can vary according to the quantity, there's some variability when trying to do comparisons. For example, many suppliers charge shipping according to graduated weight thresholds. Taking these factors into account, it looks like dumb RGB pixels are cheaper than smart pixels for now. Of course, you also need to factor in the assembly time and effort. However, that's true of any build-vs.-buy hobbyist trade-off. The commercial pixels are probably more tolerant of weather conditions, although we had a lot of rain in December and my DIY pixel strings still work (except for 1 pixel with a bad solder joint that went out, which could have been due to garage door movement or vibration).

My basic cost was under $0.40 per RGB pixel, excluding the controller. This is about 2/3 the cost of LPD6803-based pixels available from China, or about ½ the cost if you add in shipping. To make it a completely fair comparison, you would also need to add the controller costs, but there again, a Renard-HC controller seems to cost less than a DMX controller + LPD6803 decoder (that part of the comparison is in the controller chapter).

I used plexiglass strips as the backing material of the RGB strings because they are flat, fairly rigid, light-weight, and transparent. Low weight is important for larger number of strings – in the past I've had to add counter-weights to the garage door when I mounted something on it that was too heavy. I wanted a transparent material so that reflected light from the garage door would show through, to make the strings themselves almost invisible. The flat shape was very handy for holding the PLCC6 LEDs in alignment, because they were also flat.

4.5. Construction

When I saw AussiePhil's elegant RGB string construction technique, I thought that a charlieplexed string wouldn't be too bad either. What I didn't fully appreciate at the time was that charlieplexed RGB LEDs are more like a parallel circuit than a series circuit. It turns out that the PLCC6 LEDs are well-suited for series applications, because their pins are already pointing in the direction that the connecting wires need to go. Not so with charlieplexed RGB LEDs … ☹

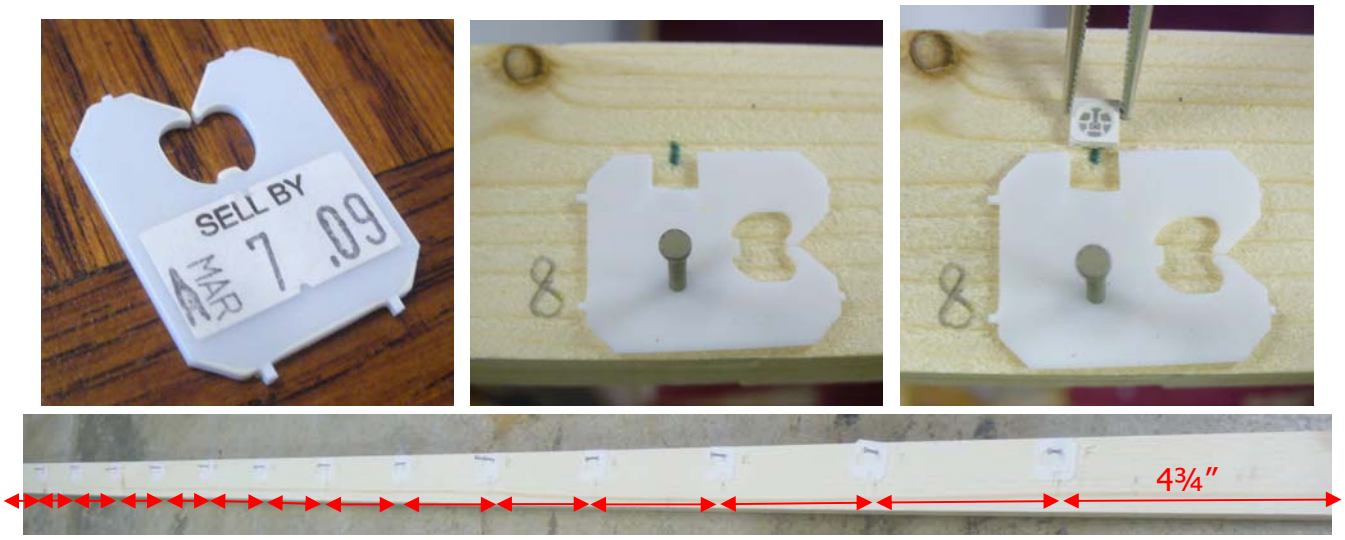
Building the charlieplexed RGB pixel strings was a rather tedious process. The first string took me over 4 hours to solder 16 RGB LEDs, but I think that was mostly because I was still learning how to do it. By about the 5th or 6th string, I was down to about 2½ hours, and by the 10th string I had it down to under 2 hours. From a practical stand-point, this is obviously not very scalable. I wouldn't want to try this for 1000s of pixels,

but for a smaller "prototype" pixel grid like the one I built with a few hundred pixels, it was doable. My thought was that if I could verify that this technique would even work, then hopefully later I could find a commercial product that could do something similar at a comparable price. I was also thinking that if a commercial product did not appear, then I could use a punch-block technique, or find or make snap-fit sockets for the PLCC6 LEDs, to avoid the tedious soldering. I still believe that's possible, especially with the 3D printers that are now available, but I haven't followed through on this yet. Not withstanding, I still consider this project to have been a useful exercise, and I was quite happy with the final appearance of the grid.

There are probably easier ways to make the LED strings, but below are the construction steps I used.

Setup

Since I would be building a number of strings and I wanted the pixels to be aligned nicely, I first built a jig to ensure uniform spacing. The jig also helped to hold the LEDs in place during soldering. I used very high-tech parts for the jig: an 8 foot 1" x 2" and bread wrapper holders. ☺



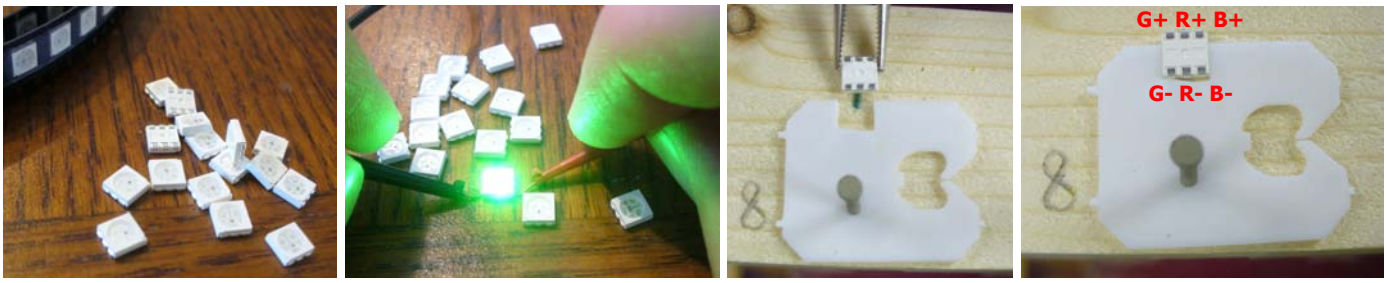
Jig with LED holders

I cut a notch out of each bread wrapper holder, then fastened them to the 1" x 2" at 4¾" intervals, to give an overall length of about 6' for each RGB string. If I had been using 4-pin T1¾ RGB LEDs, I think I would have just drilled holes into the 1" x 2" to hold the LED bodies while soldering, and/or just drilled holes through the plexiglass backing material and insert the LED leads and not even use the "LED holders" at all.

Step 1: Prep

For each 6' string, I cut an 8' length of Cat3, allowing 2' for connecting the strings to the controller. I used Cat3 instead of Cat5 because it is less work to untwist, and it can be a little cheaper. However, Cat5 will also work since it's the same wire, just twisted more.

Due to the number of solder joints, it was too tedious to leave the outer jacket on and cut through it when needed, so I removed it. Then I found it too difficult to hold the 8 wires in place, so I gave up and untwisted them, and then just built the RGB strings one wire at a time. This actually made the process a lot easier (easier to hold one wire in place at a time, fewer mistakes, etc). I untwisted them by first separating the pairs, then the wires in each pair, by holding one in each hand and rolling between my fingers. I suppose an electric drill or other automated technique would be easier, but untwisting by hand was pretty easy too.

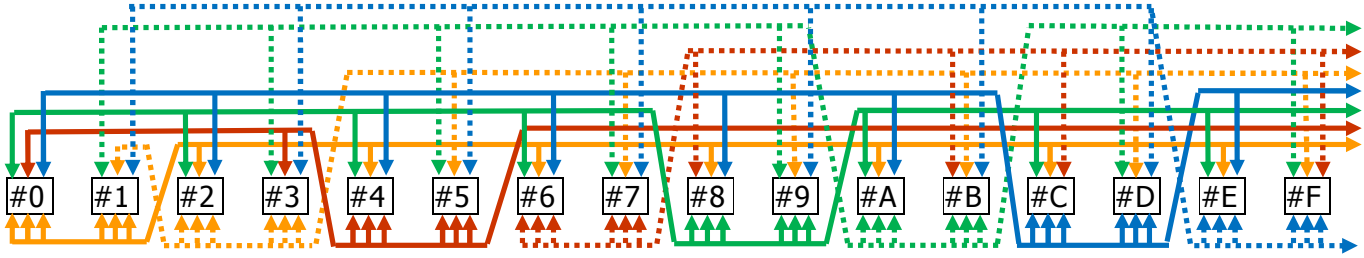


Testing LEDs, then loading them into the jig

Before soldering, I tested each LED. I found no bad ones, so the quality of LEDs from the group buy was good. Then I loaded each set of 16 LEDs into the jig. It's important to use the same orientation for each LED, because it's a lot easier to solder them correctly the first time than to redo it (don't ask me how I know this). ☹ I put the LEDs face down in the jig because the leads are more accessible from the back when soldering.

Step 2: Wiring the LEDs

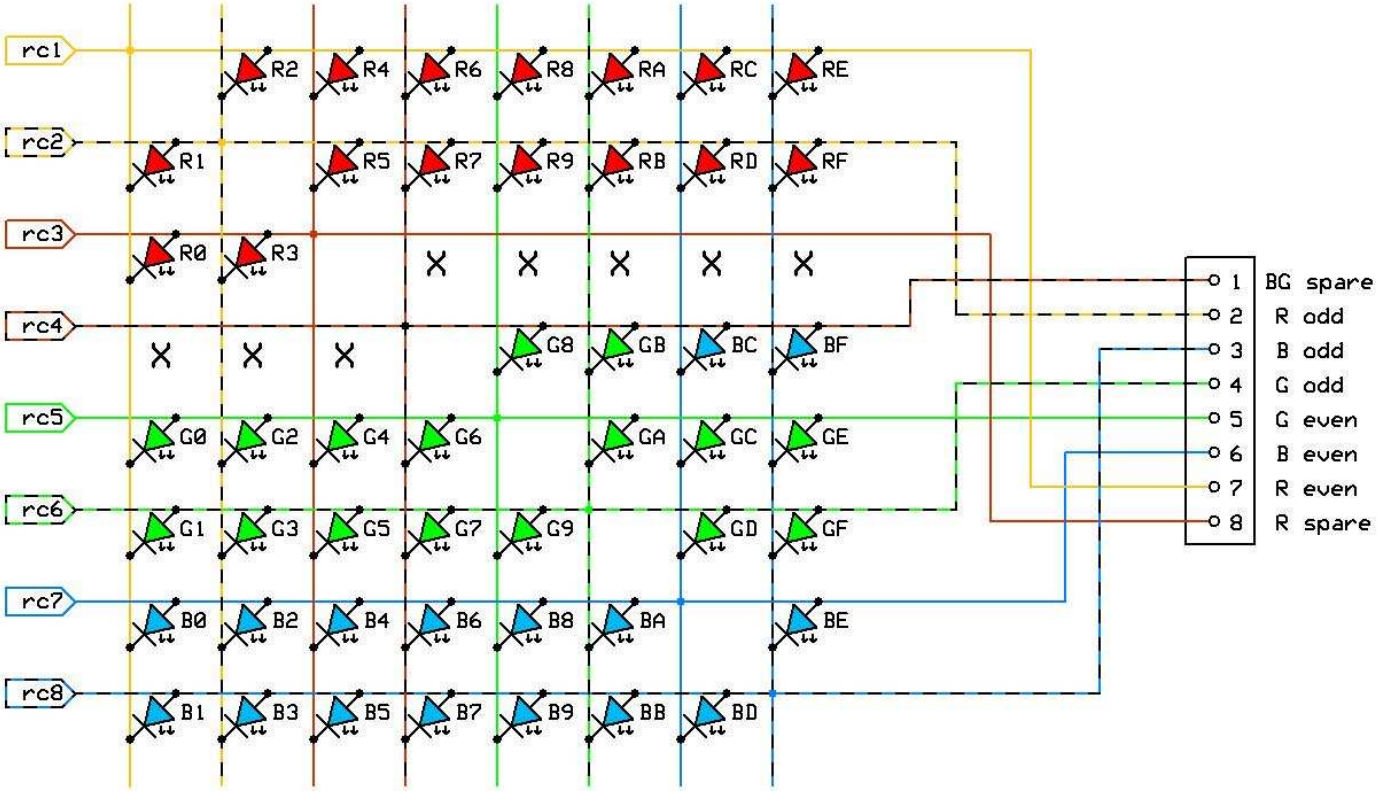
The physical wiring arrangement I used is as follows:



Charlieplexed 16 RGB string wiring diagram

The LEDs are shown from the back, since they were face-down when soldering. Since there are 16 LEDs, I used hex numbering in the diagram above. LED #0 is farthest from the controller, #F is closest.

The solid lines above represent the solid-colored Cat3 wires and go to the even-numbered LEDs, while the dotted lines represent the white wires with colored bands and go to the odd-numbered LEDs. To make it easier to remember, I used orange to supply the red LED anode, green and blue to supply their respective LED anodes, and the brown pair as the "extra" lines that provide the red, green or blue wires when those have already been used as the charlieplexed common cathode (refer to the circuit diagram below). Each wire serves as the common cathode for one pair of RGB LEDs in turn, which is why the RGB LEDs are controlled in pairs. Except for that, they are in parallel, with one red (orange), one green and one blue wire going to each LED (solid to the even LEDs, dotted to the odd LEDs) except when one of those is already used for charlieplexing a row. This actually sounds more complicated than it is - sorry about my poor explanation. For reference, the charlieplexed RGB LED circuit is repeated below with color-coded lines to match the wiring diagram above (common-cathode shown):



Charlieplexed RGB LEDs with color-coded wires

So anyway, I added one wire at a time to the LEDs, working from one end of the jig to the other. I found that there seemed to be a natural order to follow when making the connections:

- Start at the end furthest from the controller (LED #0), and work toward the controller end (LED #F).
- Start with the orange wire (even red LEDs) and then white/orange wire (odd red LEDs), so that the middle anode is soldered before the one on either side of it.
- Next connect the brown (alternate reds) and then white/brown (alternate greens/blues) to finish off the middle anodes, so each LED has at least one wire.
- Then connect the green wire (even green LEDs) and then white/green wire (odd green LEDs).
- Finally, connect the blue wire (even blue LEDs) and then white/blue wire (odd blue LEDs).

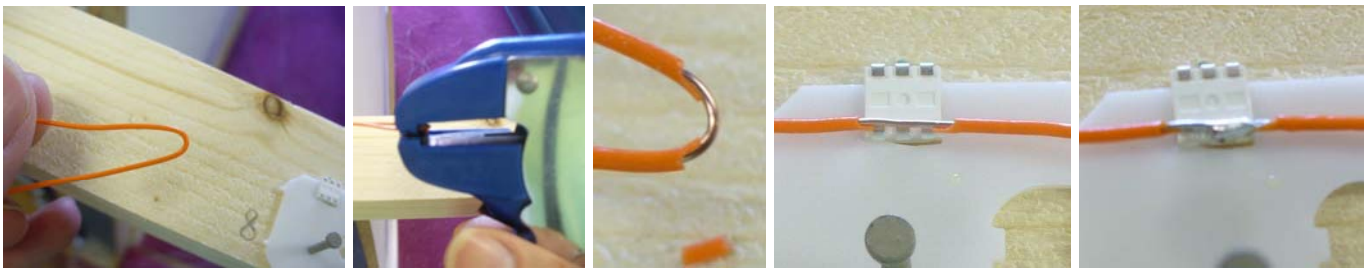
Using this order, the common cathodes of the LEDs will be connected left-to-right (labeled 0 to F, above).

I was a little concerned about soldering the PLCC6 LED contacts since they are so short and close to the LED chips, but I did not burn out any LEDs while soldering so I guess my soldering iron temperature was correct – hot enough for quick soldering but not hot enough to cause damage. The LED datasheet said the LEDs could withstand 260° for 5 seconds, but most of the time the solder adhered to the LED contact in less than a second after initial contact with the copper wire. I was also concerned about melting the Cat3 insulation – a few times wires were clamped together too tightly during soldering and it melted through the insulation, but since there was only one exposed wire when I separated them, this was not a problem.

Due to the parallel nature of charlieplexing, there are actually 2 wires that need to be connected to most of the LED pins (one to the previous LED and one to the next). Now 16 RGB LEDs x 6 pins x 2 wires = 192 connections, which is a lot just for 16 RGB LEDs. However, I found some tricks to make this a lot less work (described below). Also, good lighting, standing up with the jig mounted at elbow height or above, and standing facing the common lead side all seemed to make it a little easier.

The first trick allows 3 common cathode (or anode) connections to be made at the same time. Rather than cutting multiple shorter lengths of wire, I kept the 8' lengths as-is, then formed a loop and "mid-stripped" it to

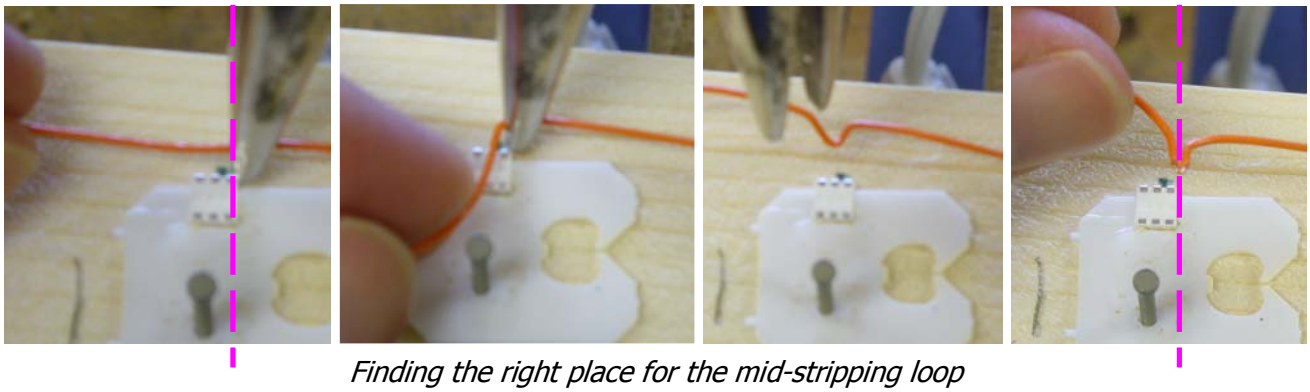
expose a short section of copper which could then be soldered across all 3 cathodes of the PLCC6 LEDs at the same time. I left the common cathode loops quite round and open before mid-stripping because the wire needed to be straightened out again and stretched across the 3 common cathode pins before soldering.



Mid-stripping a wire and soldering 3 common cathode leads at one time

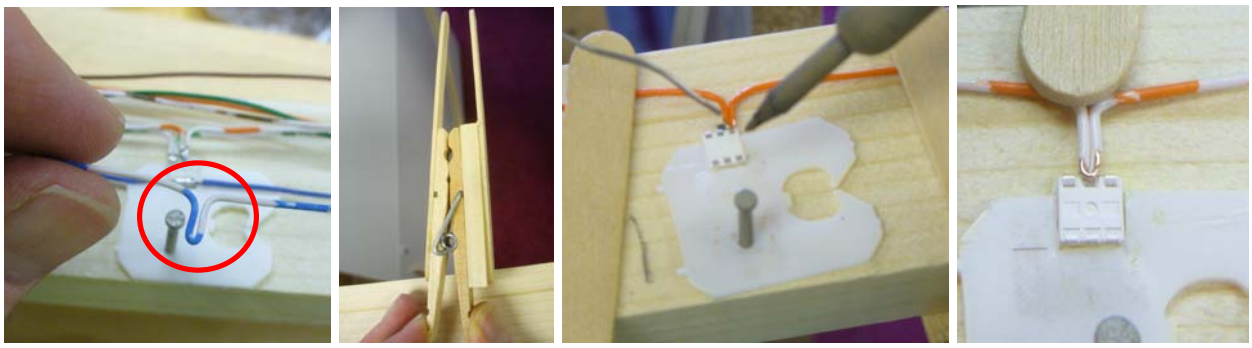
It look a little practice to get the mid-stripping just right (correct stripper tension is important), but then I was able to do it fairly quickly and consistently. Out of 32 strings (over 1500 anode connections), I only cut through one wire when trying to mid-strip it (stripper tension was too high), so that's a pretty good average. It was easy to repair – there's just one additional solder joint on that RGB string.

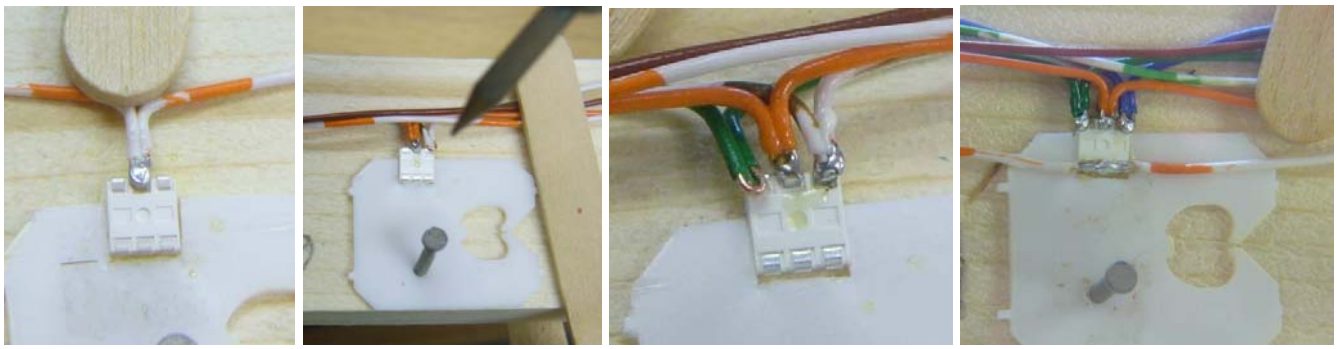
Since mid-stripping the wire allowed me to use one continuous length of wire, I just started at one end of the jig, connected the first LED, then on the next, working my way down the jig one LED at a time. I did not expect to be able to strip the wire in all the correct places exactly if I did it ahead of time, so I mid-stripped the wire as I went along. I formed each loop so it would line up with the LED pin to be connected next:



Finding the right place for the mid-stripping loop

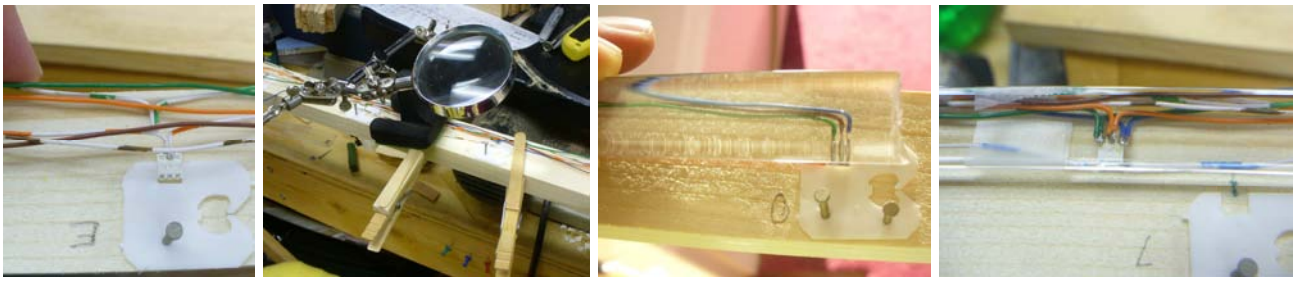
For the individual anode connections, I made the loop smaller/closer than for the common cathode triple connections, then pinched it tightly closed after mid-stripping because the loop needed to be as narrow as possible in order to leave room for the 2 other connections beside it.





Mid-stripping individual anode loops, pinching closed and soldering

To connect the mid-stripped wire, I clamped it to the 1" x 2" using my hi-tech, low-tension clamps, which held the copper loop on top of the LED lead, then applied the solder. This produced a surprisingly solid connection. I found it easier to do the middle lead first (the red LED) when there was nothing to either side, then the leads on either side after it (green and blue). The common cathode was soldered either before or after the anodes, depending on the position of the LED within the string, but neither interfered with the spacing of the other.

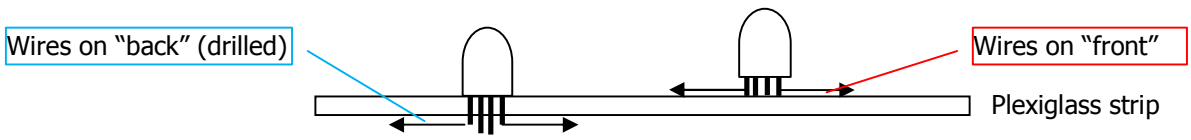


Tucking in wires, inspection, attaching structural support

Depending on which leads were already soldered, on some LEDs I had to "tuck" the next mid-stripped loop under wires that were already there. After all connections were made, I did a visual inspection and movement test for bad connections (especially looking for cold solder joints). When all was okay, I laid a plexiglass strip on top of the LEDs and temporarily attached it using tape (more details in later section).

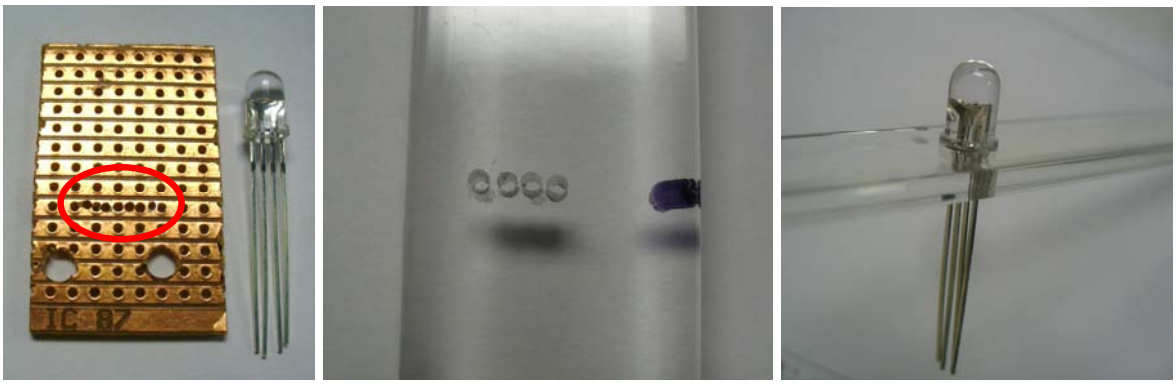
4.5.1. Alternate Process – 5mm T1¾ LEDs

The above process would have been similar, perhaps a little easier, if I had used 4-pin T1¾ LEDs rather than PLCC6 LEDs – there would only be one common lead instead of 3 to solder for the common cathode, and the leads would all be in one row perpendicular to the wire direction, rather than 2 rows of leads parallel to the wires as with the PLCC6 LEDs. The LED leads could be bent and fastened to the surface of the plexiglass strips similar to the PLCC6 technique above, or the plexiglass could be drilled so the LED leads could be inserted and soldered on the back side to hold them in place:



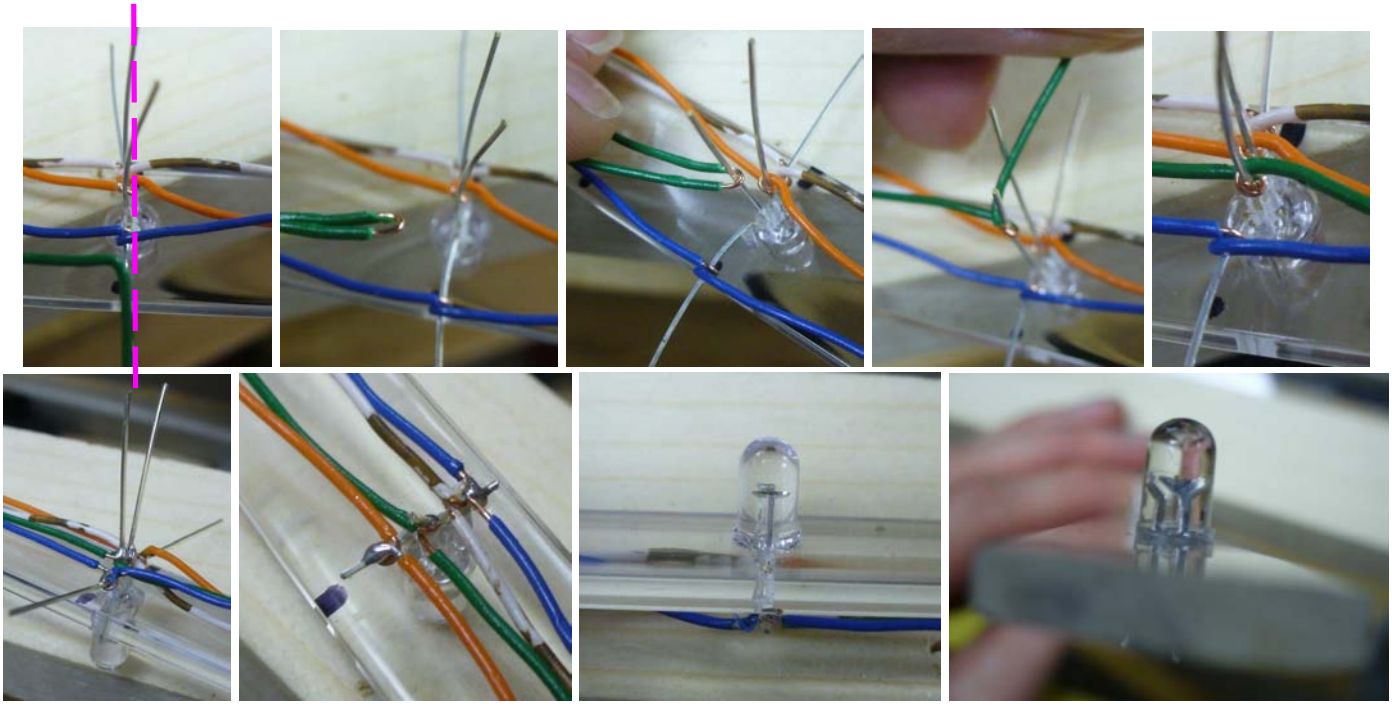
T1¾ LED mounting options

I think I probably would have drilled holes in the plexiglass and inserted the LED leads so the LEDs would be held in place. The T1¾ LED leads can be spread out to give a little more room for soldering purposes.



Using a drill guide to drill holes for LED leads

The same basic mid-stripping technique could be used on the connecting wires – form a small loop so it lines up with the leads, then mid-strip it and wrap it around the LED lead, then solder it and trim the leads:

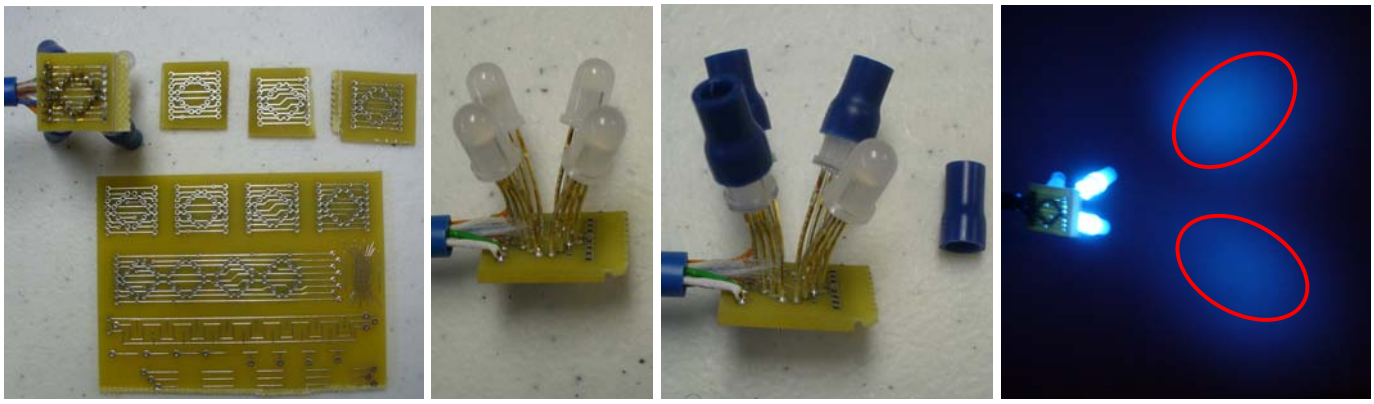


Mid-strip wire, slip it onto LED lead and wrap, then solder and trim leads

The T1¾ body sits nicely on the plexiglass strip, so it can be held in place firmly and aimed precisely, but I would probably need to use a blob of hot glue for weather-proofing, rather than the plastic wrap approach described later.

4.5.2. Alternate Process – 4x4 Clusters

Another arrangement I tried using was 4 clusters of T1¾ RGB LEDs, with the LEDs aimed at an angle:

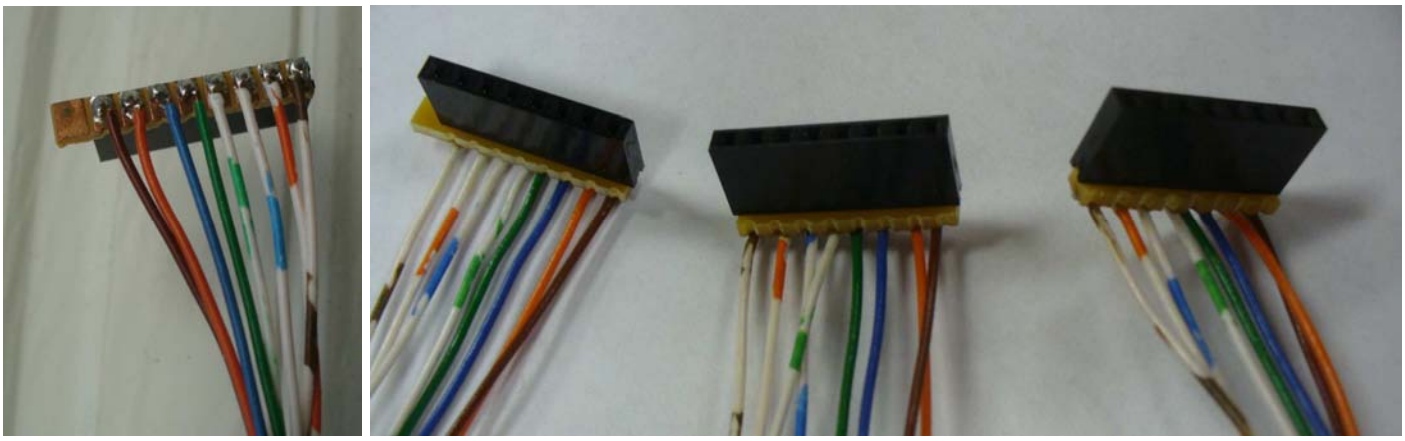


4x4 LED clusters, elliptical shape

This approach would have been less effort to build because the connections are consolidated onto 4 small PCBs so there are only 4 lengths of wire between them, instead of spread out as 16 separate LEDs. However, the projection angle became a problem – the LEDs had to be 5" or less from the garage door in order to clear the opening as the door opened and closed, but when the LEDs were that close to the door they needed to be aimed at more of an angle, which distorted the reflected pixels into an elliptical shape. If the pixels had been further away from the garage door surface, this probably would have looked okay. I used diffused LEDs in the example shown above, so I also tried "cover tubes"; the elliptical shape is more prominent with non-diffused LEDs.

Step 3: Adding a Connector

After soldering all the LEDs, the ends of the Cat3 wires did not line up so I trimmed them and then added a connector. I used SIP8 sockets on the strings and headers on the controller board because they were slimmer and a little less expensive, but RJ45 jacks would also work (the Renard-HC PCB accepts either). To give the wires more support, I used a small piece of stripboard (1 x 8 holes) between the wire and the SIP8 sockets.



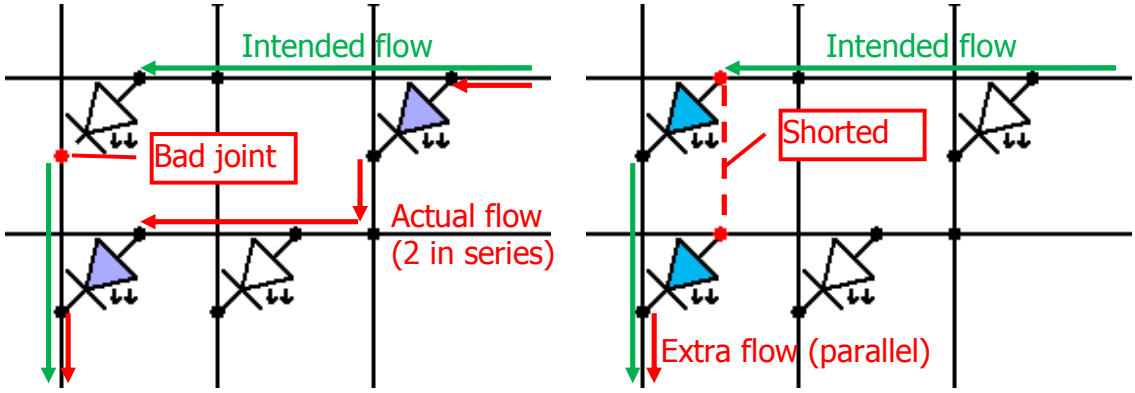
SIP8 sockets, with 1 x 8 stripboard for more support

I connected the wires in the wrong order (backwards) because I didn't first check the orientation of the strings relative to the controller. However, I did all the strings the same way, so I was able to compensate for this by reordering them within the Vixen Renard-HC plug-in (hard-coded). I guess I should make this a config option in future. Since I was using SIP8s, I also could have just plugged them in the other way around, instead of reordering them within the plug-in, but I did not allow a long enough connection wire for that in some cases.

Step 4: Troubleshooting

After building the strings but before the final weather-proofing step, I tested the strings by connecting them a few at a time to one of the controller PCBs. Then I ran a little hard-coded sequence to step through each color on each LED. There seemed to be 2 basic patterns for the strings that didn't work correctly.

The first pattern was that one color of an LED did not light, but a couple of other LEDs would light up dimly in its place. This was due to an open connection in the charlieplexed circuit – the current would find an alternate path though 2 other LEDs in a series path (the pair of LEDs with the lowest effective resistance, but if multiple pairs are close, more than one extra pair will light dimly). This can occur on either the individual anode or the common cathode side, and a few strings actually had multiple occurrences, so the test sequence to slowly step through each LED one at a time was helpful in isolating those. I resoldered the joint to fix this problem. This pattern would also occur if one of the LED chips was actually burned out, but fortunately that did not happen.

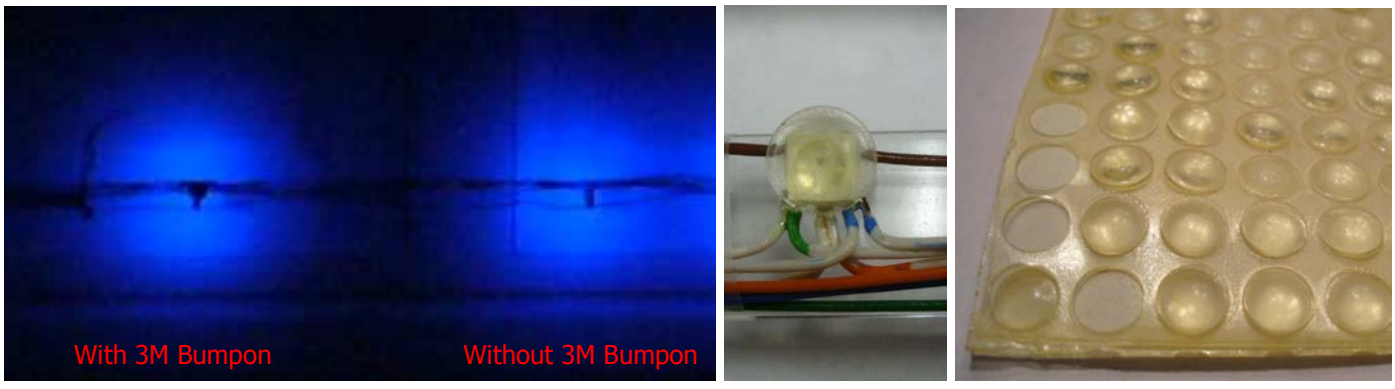


LED light-up patterns with bad solder joints (open or shorted) ☹

The other pattern was that all LEDs would light up when they were supposed to within the test sequence, but an extra LED would also light up. This turned out to simply be a short between 2 adjacent anodes, and re-soldering/re-shaping the connections corrected this problem.

Step 5: Lenses (optional)

I wanted the light from the LEDs to diffuse and fill the 4¾" area for each pixel, but this made the outer edges of the pixels dimmer and less well-defined. I tried sticking a 3M Bumpon over each LED to focus the light a little more, but after the strings were wrapped with plastic wrap (next step), the 3M Bumpons did not seem to make much difference. They were not noticeable from a distance (see left photo below).



After plastic wrap, 3M Bumpons don't make much difference

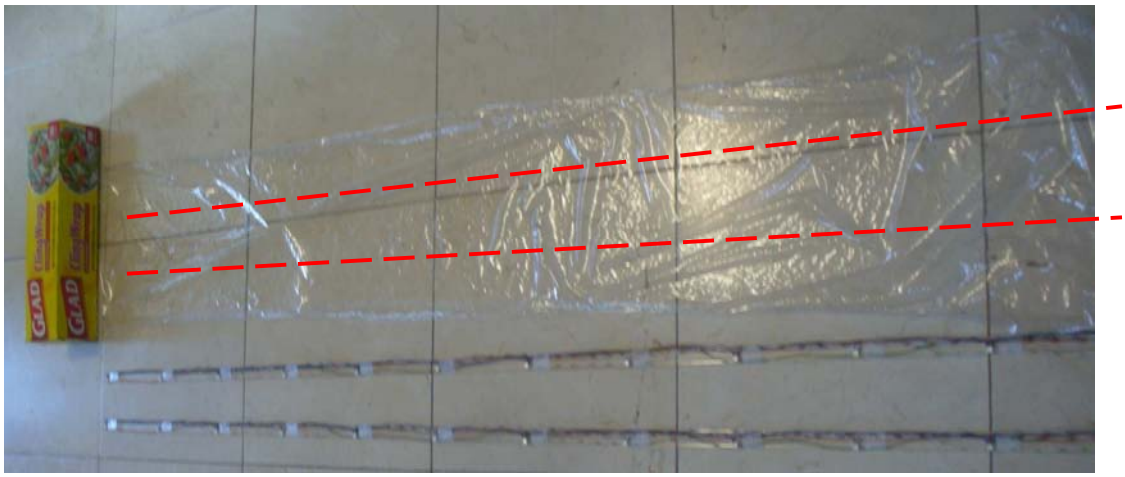
Due to the extra cost and minimal improvement, I decided to skip the Bumpons, but if I had not used the plastic wrap for weather-proofing, I might have used the Bumpons.

Step 6: Backing Material and Weather-proofing

After soldering, I attached the RGB LEDs to a rigid backing material in order to keep the wires straight and to avoid straining the solder connections. I used 6' lengths of 5/8" W x 1/8" H plexiglass strips because they are transparent, and the rectangular cross section also made it easier to ensure that the LEDs were perpendicular to the garage door surface. The plexiglass strips are not quite as rigid as I would have liked, but if they are supported securely they will remain straight. OTOH, since they can bend I may be able to use them for my 2011 project as well.

Buying pre-cut plexiglass strips in bulk and having them shipped was about the same cost as buying a sheet of plexiglass locally and then cutting it myself (4' x 8' sheet), so I opted for the pre-cut strips to avoid the extra work of cutting. It is very unlikely that I would have gotten nice, straight strips if I tried to cut them myself, anyway. ☹

For protection during testing, I just fastened the RGB strings to the plexiglass strips in a few places temporarily using tape. After testing/fixing, I fastened them more permanently by wrapping the strings in a couple of layers of plastic wrap. I chose the plastic wrap for weather-proofing purposes, but I also liked the way it held the LED strings nicely in place and would stick to itself securely without glue or tape, but it could also be unwrapped again or cut and rewrapped for repairs.



Cut plastic wrap length-wise and then wrap RGB strings for weather proofing

I just used the cheap stuff from the grocery store; the shipping and packing type would probably be more durable, or even heat-shrink tubing. I unrolled a 7' length on a hard, flat surface and then cut it length-wise into 3 pieces – the roll was 1' wide, but I did not want too many layers on the strings in order to avoid extra diffusion of the light. Using 4" wide strips was enough to wrap the strings 2 – 3 times lengthwise.

Moisture build-up inside the plastic-wrap

We had a lot of rain in December, so moisture built up inside the wrapped strings. I think this was actually condensation rather than rain leaking in, although it might have been both. The plastic wrap was actually too

much protection, since it trapped moisture inside the strings, but the moisture seemed to be mostly between the outer layers of plastic wrap, with only a little actually inside where the LEDs were. This did not cause any electrical problems, though, since the solder connections were actually raised up higher within the air pocket. Since the plexiglass strips were mounted horizontally, I probably could have just wrapped a few inches wide around the area containing the LEDs, rather than the entire 6' length of plexiglass strip, or maybe used clear heat-shrink tubing to cover the LEDs.

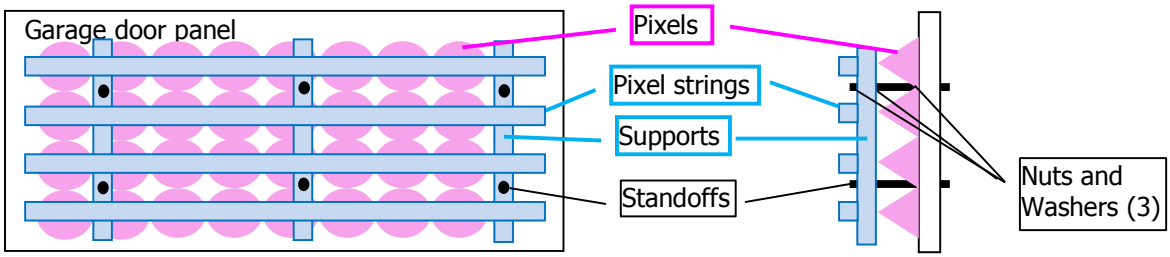
If I had used T1¾ LEDs instead of PLCC 5050s, the plastic wrap would not have been as convenient, due to the higher profile of the T1¾ bodies. I suppose a blob of hot-melt glue over the connections and leads would be sufficient to seal them, rather than wrapping them with plastic.

4.6. Mounting

I used the following materials to mount each group of 5 pixel strings (80 RGB pixels):

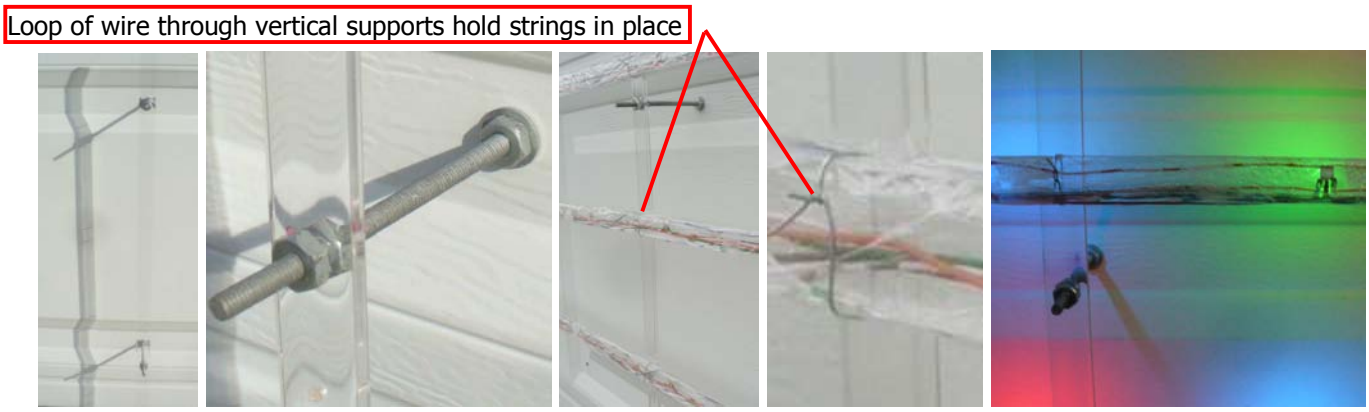
- three 1½' L x 5/8" W x 1/8" H plexiglass strips
- six #8-32 x 2½" machine screws or bolts
- eighteen #8 washers and nuts (3 per bolt)

The cost for these supplies was around $2 to $3 for 5 strings, so the incremental cost per pixel was negligible. The parts above are enough for half of a garage door panel (1 string horizontally); it would have been double this for an entire panel (2 strings horizontally).



I mounted the RGB strings horizontally on the garage door panels, 4 or 5 strings per panel. Since the pixels were already attached to plexiglass strips, I used shorter vertically-mounted plexiglass strips to support each string in 3 places, as shown above. The machine bolts served as stand-offs, with nuts and washers holding the strings at the desired distance from the garage door surface. Adjusting the position of the nuts allows the pixel size to be adjusted; I used a standoff distance of 2" in order to get a 4¾" diameter pixel reflection.

At first my wife was in shock when she discovered that I had drilled holes in our almost-new garage door (for the stand-offs), but she calmed down after a while. In trying to justify my actions, I pointed out that we share the garage door, and in fact I only drilled into my 1% of the door, leaving her 99% of the door intact (actually, by surface area, 48 x 1/8" Dia holes < 1"² < 0.01% of the garage door surface area, so I drilled less than 1% of my 1%). I thought it would also be easier to ask for forgiveness than to get permission. With the grid removed, one unexpected benefit of the holes is that at night the door gives a starlight "twinkle" effect as the inside garage lights shine through the holes as you approach it. ☺



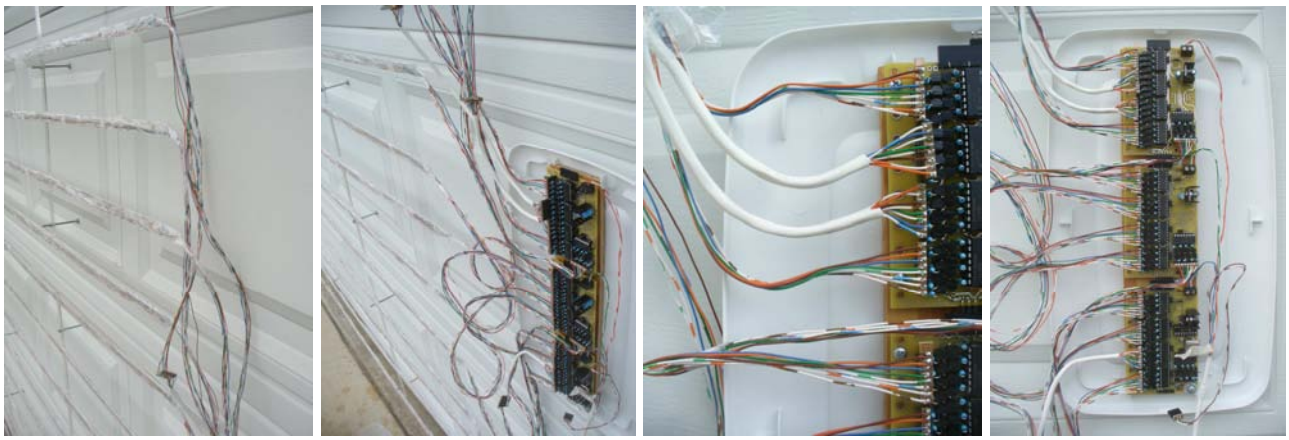
Vertical supports with adjustable stand-offs hold pixel strings in place

To attach the pixel strings to the shorter vertical support strips, rather than glue I just drilled a pair of small holes into the vertical strip and then ran a loop of thin wire through it to hold the horizontal RGB string in place. This allows a string to be easily adjusted or removed. It also allows me to flip the pixel strips over, if I want a point-source effect rather than diffused pixels. The mounting strips are nearly invisible, but the stand-offs cast a slight shadow.

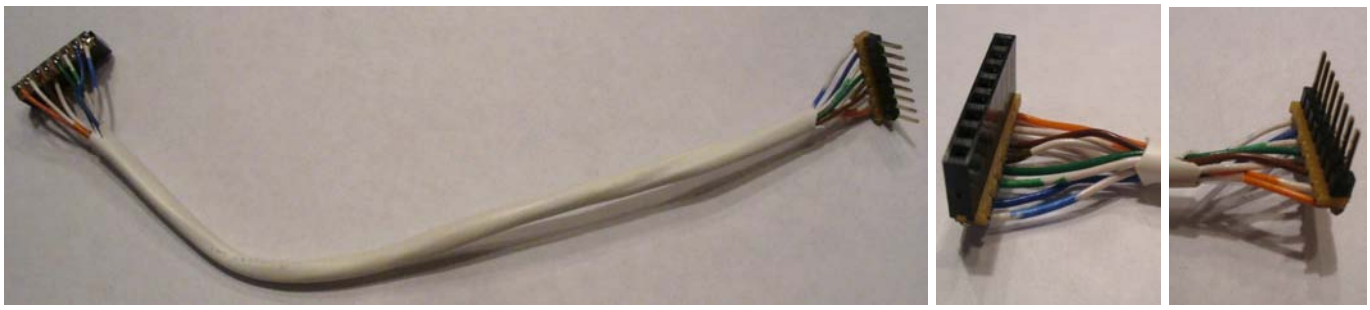
I originally considered making a long narrow "shadow-box" out of white Coroplast (corrugated plastic sheet) to fit on each garage door panel, with dividers to keep the pixels separated, but the plexiglass strips seemed like less work. If the garage door had been a dark color or non-reflective, or if my wife had caught me before I drilled the holes for the stand-offs, I probably would have used the Coroplast approach, and just hung the shadow boxes on each garage door panel.

4.7. Hook-up

With the controller mounted off to the side of the pixel strings (details in the Controller chapter), all I had to do was connect the SIP8 sockets on each RGB string to the SIP8 headers on the controller (or RJ45s if I had used those instead).



Connecting the pixel strings to the controller mounted on the garage door



RGB string "extension cord"

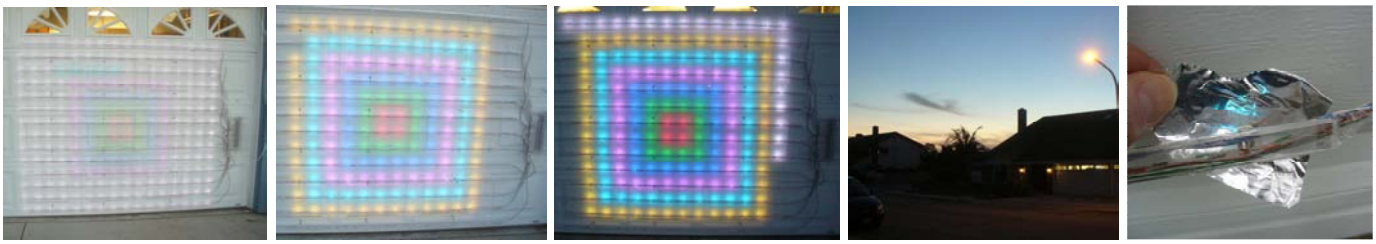
Since the garage door panel height was not an even multiple of the pixel size, the vertical position of the strings varied from one door panel to another. I thought I had allowed enough wire to reach the controller, but a few of the strings did not quite reach, so I had to make a few little "extension cords", with a SIP8 socket on one end and header on the other. ☹

4.8. Testing

The strings and controller all worked fine after testing them individually, so I was looking forward to seeing the whole grid light up for the first time. Big disappointment ☹ – no pixels lit up when I ran a test pattern. I looked over everything and didn't see any obvious problems such as forgetting to connect something or connecting something backwards, so I disconnected and reconnected a few wires and tried it again, hoping that it would magically work this time. 😉 Still nothing, so it looked like this was going to be a painful process of tracking down an implementation error, rather than just a stupid mistake (the circuit diagrams had been used successfully elsewhere). At least there was no smoke. ☺

My bewilderment grew as I measured voltages on the controller and they were all correct. While trying to decide how to divide up the problem into smaller steps, I finally leaned over and actually looked directly at the LEDs, and was surprised to see that they were actually lit. It turns out the grid had been working for I don't know how long - I just couldn't see it.

I was working during daylight hours in order to see what I was doing, but the grid pixels were so dim that they couldn't be seen in daylight (at least not when they face backwards and reflect off the garage door). However, they are bright enough to easily see at night.



Grid gets brighter as ambient sunlight decreases; using tin foil as "mirror" to check

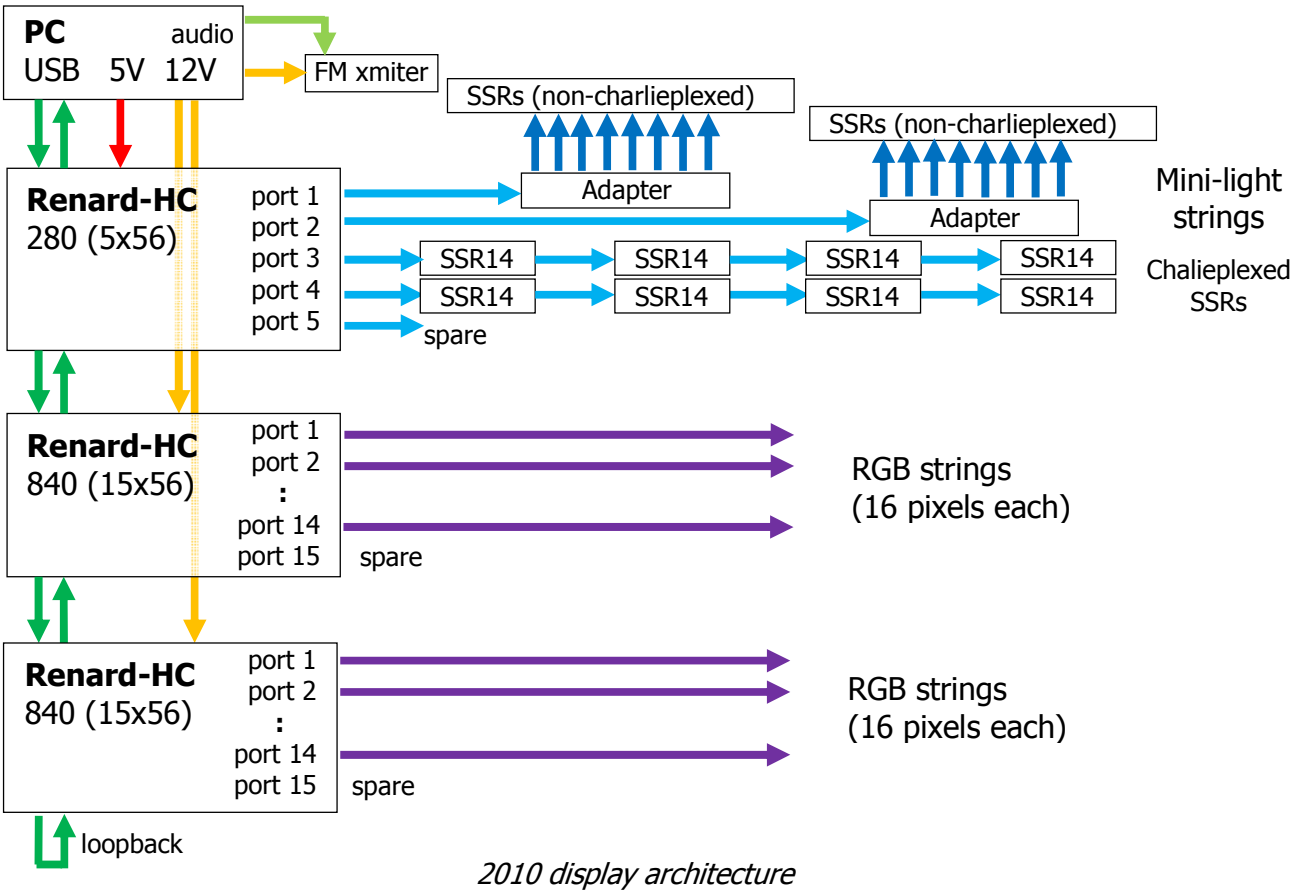
The above photos show how the apparent brightness increases as it gets darker out. Just after sunset is when the grid really becomes visible. To avoid wasting more time looking for non-existent problems, when I need to check the pixels during daylight I now use a piece of tin foil as a mirror so I can see if the LEDs are actually on or not.

Even though the grid is not very bright, it's very visible at night, so I consider this prototype/experiment to have been successful. My conclusion is that dumb, charlieplexed RGB LEDs can be used as grid pixels, even when diffused and with a 1/8 duty cycle, as long as the LEDs themselves are bright enough compared to the ambient light. Now the wiring is tedious, but at least this provides another viable option on the cost-vs-effort spectrum.

I was also reasonably pleased with the graphics capabilities of the grid. Even a small 16 x 14 grid can display simple graphics and text fragments, so this type of grid could be used as a large RGB version of LEDTRIKS. The firmware still has some bugs to fix (a bad flicker problem during some types of updates), but since a PIC16F688-based Renard-HC was able to drive them, then I felt that another major goal was successful: to intermix graphics and discrete props on the same COM port, and use the same type of controller for both.

5. Renard-HC Controller

My planned display architecture for the 2010 season was as follows:



2010 display architecture

A 280 channel Renard-HC was to drive all of my AC SSRs, and then a pair of 840 channel Renard-HCs would drive the pixel grid. However, due to time constraints, I only used one of the 840 channel Renard-HC controllers and half the number of grid pixels, and I ended up using my original (non-chipiplexed) DIYC Renard controller to drive the non-charlieplexed AC SSRs instead of running them via de-charlieplexing adapters connected to the 280 channel Renard-HC, due of voltage compatibility problems[12].

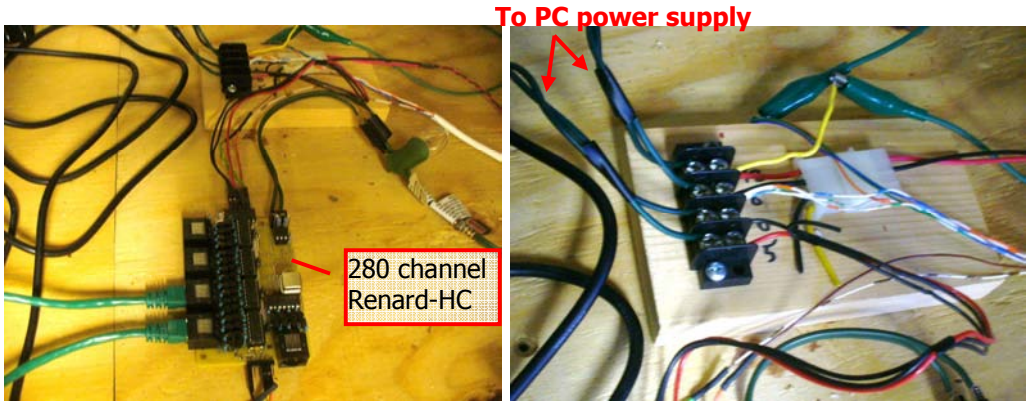
The main purpose of this chapter is to describe the 840-channel Renard-HC controller that I used to drive the pixel grid, its construction and operation. However, below is a brief description of some of the other aspects of the display architecture shown above.

[12] Voltage drop between I/O pins is less, which was not quite enough to drive the optos with 680 Ω resistors.

Power Supply

In previous years, I just used a USB port to supply +5V to the controller (the non-PWM Renard firmware uses very little power). This year, however, the 840 channel Renard-HC controllers were going to draw 2 – 3 A each for the grid pixels and the significantly higher channel count, so I powered the controllers from the PC power supply instead.

I used a spare 4-pin power supply connector from inside the PC and ran it out to a terminal block. The controller that ran the AC SSRs was right next to the computer (centralized controller model), so I just ran a short 5V line from the terminal block to the controller. Since the FM transmitter needed 12V and was also near the PC, I also ran a short 12V line from the terminal block to the FM transmitter.



Terminal block distributes +5V or +12V from PC power supply

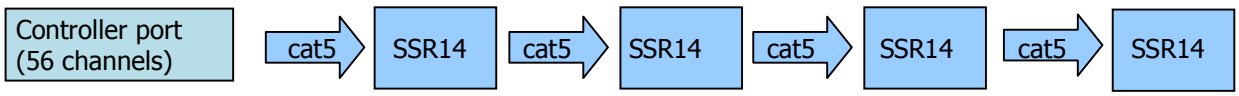
The 840 channel Renard-HC controller for the grid pixels was a further distance from the PC, mounted on the garage door (decentralized controller). To avoid voltage drop problems due to the heavier current over the longer power leads, I ran 12V from the terminal block over to the controller and then used on-board 5V regulators rather than running the 5V directly from the PC power supply. I used 3 of the 4 pairs of a Cat3 cable, with the remaining pair used to carry the serial in/out line to the remote controller.

As I add additional high-channel count props in future, it looks like I will be transitioning to more and more of a distributed controller architecture with the controllers closer to their props than the computer, so I will need to pay more attention to power distribution issues.

AC SSRs

I have a mixture of charlieplexed and non-charlieplexed AC SSRs. The charlieplexing happens on the controller side, so the SSRs themselves are very similar. They all use the same circuit diagram as the DIYC AC SSRs, except they have 7 channels[13] instead of 4, so they use all 8 wires in the Cat5. Also, the charlieplexed SSRs have no resistors on the input side of the optos (they are on the Renard-HC controller instead).

The charlieplexed SSRs have 2 RJ45 connectors each, so they can be daisy-chained to reduce cabling. This allows up to 8 x 7 = 56 channels to be controlled with one cable run from each controller port. I connected 2 chains of 4 SSR14s (dual SSR7s) to 2 of the ports on the central 280 channel Renard-HC to run some of my larger props such as the ArchFans, which take 16 channels each.



4 SSR14s can be daisy-chained on each Renard-HC port (56 channels per port)

[13] I'm not superstitious – there are 7 wires remaining in a Cat5 cable after you use one as a common.

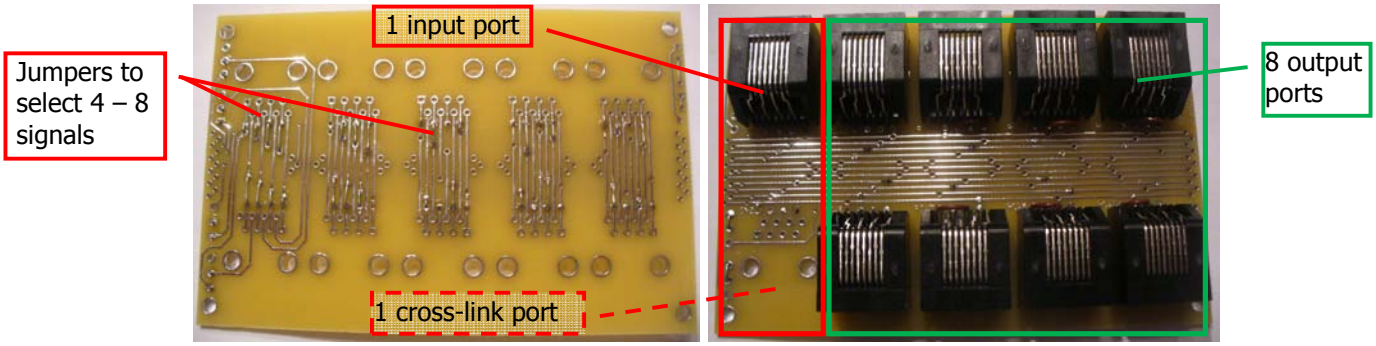
My older SSRs are on stripboard, but now I have a nice little PCB for this. The SSR14 PCB can be used as-is, or snapped apart for a pair of SSR7s, which can be used separately or as building blocks to build up larger SSRs (up to 56 channels, in multiples of 7 channels).

SSR14 PCB can be used as-is, or snapped apart or used as a building block

The SSR14 PCB accommodates 1 or 2 RJ45 or SIP8 connectors. The second connector is only used for daisy-chaining - it rotates the 8 wires by one position so the next SSR in the chain uses different channel numbers.

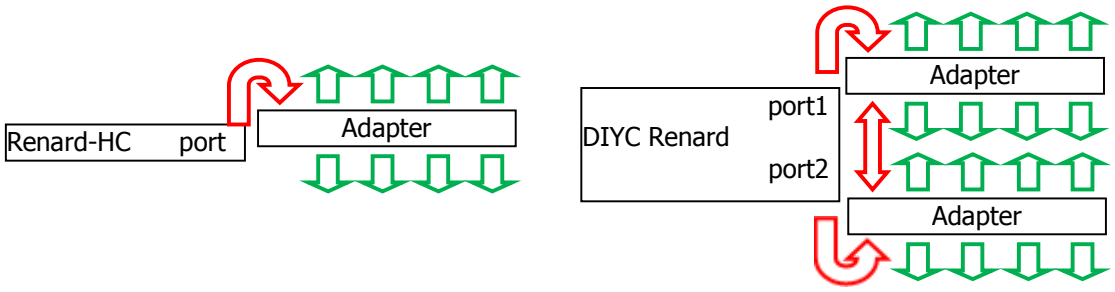
I use RJ45 sockets on the ends that I intend to connect to the controller or another daisy-chained SSR, and I use SIP8 sockets and headers on the ends that I intend to connect together as building blocks for a larger SSR. I can also use both SIP8 and RJ45 (SIP8 mounted on the underside of the PCB, RJ45 on top), for cases where I may want to use the SSR7 either way. If I will never want to separate the SSR7s, I can just hard-wire them together. Otherwise, the SIP8 sockets and headers allow them to be easily separated without cutting joiner wires.

This past season, I tried to use a Renard-HC controller (designed to drive charlieplexed SSRs) to run my non-charlieplexed SSRs (which have input side resistors), but the lower voltage drop from the controller was not enough to reliably trigger the SSRs. I ran out of time on the retro-fit, so I just used my DIYC Renard controller from last year to run the non-charlieplexed SSRs. For next season, I will probably convert my non-charlieplexed SSRs for use with charlieplexing. Since my SSRs already have 7 channels and use all 8 wires in the Cat5, the conversion is simple – just bypass the 680 Ω resistors on the input side of the optos. However, since the non-charlieplexed SSRs only have 1 RJ45 connector, they cannot be daisy-chained.



De-charlieplexing adapter: compatible with Renard-HC or DIYC Renard

I wanted to run 56 channels from each Renard-HC port, so I made a little "de-charlieplexing" adapter PCB. It basically provides the second daisy-chain connector that would be needed for each SSR, and also rotates the control signals one position for each connector so that each SSR7 will use a different set of channels, and a single Renard-HC port can drive 8 SSR7s.



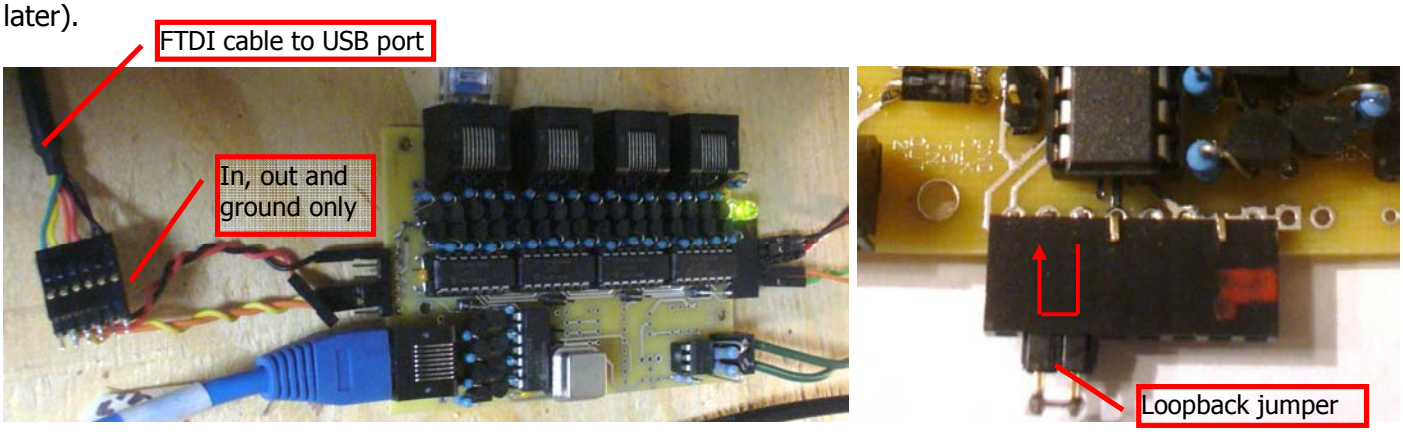
Adapter uses one Renard-HC port, or 2 DIYC Renard ports (total of 8 control bits)

I use all 8 conductors on my controller ports for SSR control, so I will only need to run one Cat5 cable into the adapter. For a standard DIYC Renard controller with 4 control signals in the Cat5, 2 adapter boards would be needed (4 control signals going into each adapter), and then the adapters cross-linked to provide the missing 4 signals to each other (8 control signals are needed for a complete adapter). There are also holes for jumpers to select whether each inbound or outbound RJ45 uses 4 or 8 connections (or anything in between).

COM Ports

I was able to run the pixel grid as well as discrete incandescent light strings from the same USB COM port, since I used Renard-HC controllers to drive both types of channels. With the data compression rates that I was getting on my sequences, it looked like there would be enough bandwidth for at least 2K channels at 115K baud and 50 msec refresh rate (which is only a 4:1 compression ratio).

I used a 5V FTDI TTL 232R cable to connect the serial in and out from the USB port on the PC to the first controller (280 channel Renard-HC), and then daisy-chained the serial in and out from there to the next controller (840 channel Renard-HC). I used the 4th pair of wires in the Cat3 cable to carry serial in and out from the first Renard-HC to the second (the other 3 pairs were used to carry +12V to the controller). I originally planned to chain another 840-channel Renard-HC controller after that, but I didn't get all the pixels installed. After the 840 channel Renard-HC, I added a loopback jumper to send the serial in back to the serial out, to complete the loop. The Renard-HC PCB and plug-in support a full feedback loop in order to allow diagnostic info or run-time statistics from the controller to be sent to Vixen, or a test Putty window (described later).



FTDI cable (using In, Out and Ground only), loopback jumper

For testing or low-power operation, the FTDI cable is very convenient because it can be connected directly to the SIP6 header on the Renard-HC PCB, but since I was not taking power from the USB port this time, I only connected serial in and out and ground. I may not have needed the ground since there was already a ground coming from the same PC.

I originally planned to run the entire display from a single USB port, but due to time constraints and the voltage level problem in going from a Renard-HC to non-charlieplexed SSRs, I ended up running my non-charlieplexed SSRs with my older DIYC Renard controller on a separate USB port.

5.1. Parts List

I built two 840-channel Renard-HC controllers, one to drive each half of the grid, but since I only ended up with half of the grid installed, I only used one of those controllers.

The parts are mostly the same as with the DIYC Renard controller, just different quantities and combinations. The parts I used for each 840 channel controller are listed below (prices are shown in $USD):

Qty/Description	Source/Part#	Cost of Parts	Cost with tax, S&H
15 x PIC16F688	Mouser 579-PIC16F688-I/P	15 x 1.36 = 20.40	24.06
15 x .1 uF 50V ceramic cap.	Mouser 581-SA105E104MAR	15 x 0.06 = 0.90	1.06
120 x 2N4403 PNP transistor	Mouser 512-2N4403TAR	120 x 0.037 = 4.44	5.24
45 x 100 Ω resistor ¼ W	Mouser 271-100-RC	45 x 0.02 = 0.90	1.06
75 x 47 Ω resistor ¼ W	Mouser 271-47-RC	75 x 0.02 = 1.50	1.77
3 x Renard-HC5x56 PCB	ExpressPCB **OR** SeeedStudio? (estimated)	3 x 10.14 = 30.42 3/10 x 40.00 = 12.00	34.72 **OR** 15.00
21 x SIL8 breakable header (OR 15 can be RJ45 sockets to match RGB strings) **OR** 5 x SIL40 break. header	Mouser 855-M20-9990845 **OR** Futurlec HEADS8 **OR** Mouser 855-M20-9994045 **OR** Futurlec HEADS40	21 x 0.15 = 3.15 **OR** 21 x 0.07 = 1.47 **OR** 5 x 1.50 = 7.50 **OR** 5 x 0.19 = 0.95	3.71 **OR** 1.88 **OR** 8.84 **OR** 1.22
Base config sub-total		**(41. to 66.) 60.03**	**(49. to 77.) 69.79**
15 x 14-pin socket (optional)	Mouser 571-1-390261-3	15 x 0.11 = 1.65	1.95
3 x 7805 5V 1A regulator	Mouser 512-LM7805CT	3 x 0.63 = 1.89	2.23
3 x TO220 heatsink (opt.)	Mouser 532-577102B00	3 x 0.22 = 0.66	0.78
3 x HS256.500C bolt	(Local) Dowpak	3 x 0.03 = 0.09?	0.09?
3 x HN256C nut	(Local) Dowpak	3 x 0.03 = 0.09?	0.09?
6 x 1 uF 50V elec. radial cap	Mouser 647-UVY1H010MDD	6 x 0.07 = 0.42	0.50
3 x 5mm 2-pin terminal block	Mouser 538-39890-0302 **OR** Mouser 571-2828372	3 x 0.43 = 1.29 **OR** 3 x 0.36 = 1.08	1.52 **OR** 1.27
3 x 1N4001 diode (optional)	Mouser 625-1N4001-E3	1 x 0.10 = 0.10	0.12
1 x 18.432 MHz clock	Mouser 520-TCH1843-X	1 x 2.11 = 2.11	2.49
1 x 8-pin socket (optional)	Mouser 571-1-390261-2	1 x 0.13 = 0.13	0.15
1 x T1¾ 5mm LED (opt)	(any)	0.10?	0.10?
1 x 1K resistor ¼ W (opt)	Mouser 271-1K-RC	0.10?	0.10?
3 x SIL8 socket (optional)	Mouser ? **OR** Futurlec FHEADS8	? **OR** 3 x 0.14 = 0.42	? **OR** 0.54
1 x 16pin DIP socket (opt)	Mouser 571-1-390261-4	1 x 0.13 = 0.13	0.15
2 x RS485 driver (optional)	Mouser 511-ST485BN	2 x 1.24 = 2.48	2.92
1 x 13½" x 8" 6 Qt Enclosure	(Local) Home Depot	≈ 1.00	≈ 1.00
Optional parts sub-total		**(0 to 13.) 9.98**	**(0 to 15.) 11.61**
Total		**(41. to 79.) 70.01**	**(49. to 92.) 81.40**
Per pixel (240 pixels)	**15 x 16 = 240 pixels**	**≈ (.17 to .33) 0.29**	**≈ (.20 to .38) 0.34**

The Renard-HC PCB can be configured in several different ways, so there are a lot of "optional" parts listed above. Here are some notes regarding the options I chose:

- The Renard-HC PCB can accommodate either RJ45 sockets or SIP8 headers. I chose the SIP8 headers because they cost less and were more compact. I'm not sure why there is such a big price difference between Futurlec and Mouser on the SIP sockets and break-away headers.
- I used "partial-ICSP" headers: 2-pin headers to supply Clock and Data[14], with the remaining 3 signals (VPP, +5V and ground) supplied via some of the pins on the SIP6 header at the "top" of the PCB.

[14] http://doityourselfchristmas.com/forums/showthread.php?11269 Flash PIC16F688 In Circuit, dirknerkle and budude

- I used sockets for the PICs so that I could replace them easily if they burned up. ☹ Also, in case the ICSP headers did not work as intended, I would be removing and inserting the PICs several times.
- Since I was using the PC's power supply for the grid controllers and they were over 20' away from the power, I ran 12V over 3 pairs of a Cat3, and then used on-board 5V regulators to avoid voltage losses. The 5V regulator in my test circuit became quite warm when driving only 5 RGB strings, so I decided to put a regulator on each Renard-HC PCB separately, and also use a heatsink.
- I was using common cathode RGB strings, so I used PNP transistors; use NPN for common anode.
- To protect against stupid mistakes (connecting the power backwards), I placed a protection diode in series with the power before the 5V regulator.
- I did not need the ZC detector, so I used the 2-pin terminal block to feed 12V DC into the 5V regulator.
- I used 100 Ω resistors to drive the R LEDs, and 47 Ω resistors for the B and G LEDs (calculations are shown in the previous chapter on RGB Strings). A shorted LED may cause a PIC to burn out if not detected soon enough, so I used sockets for the PICs.
- If getting the SIL headers from Futurlec, it appears to be cheaper to get 40s and break them down; if getting them from Mouser, the 8s look cheaper. (The 8s can also be broken down into 6s and 2s).
- I chose not to populate the power indicator LED, in order to avoid an additional "pixel" being visible.
- The serial I/O seemed to work okay over the 20' distance, so I did not populate the RS485 line drivers
- Since the central Renard-HC controller was using 115K baud, I also used 115K baud for the grid controllers because they were on the same line. This data rate seems to require the use of an external 18 MHz clock, which I shared across 3 PCBs.

I found David Jones' PCB design tutorial[15], posted by a DIYC member, to be very helpful since I had not designed a PCB before; I had only used stripboard for past DIYC electronics. In addition, several DIYC members (N1ist, David_AVD, P. Short, AussiePhil, and Budude) were very helpful in providing some additional PCB design comments on an initial PCB design[16]. For the final design, I probably broke a lot of PCB design rules or did not arrange parts optimally, but at least the PCB worked and did what I needed it to do. I was surprised to find many similarities between PCB design and model railroad layout design, particularly for the routing of traces and the layout of parts.

ExpressPCB may not be the cheapest source of PCBs, but since I was new to PCB design, and the ExpressPCB design software and ordering process is very simple to use, I just used them for my initial PCB development. Their mini-board service offers a fixed PCB size of 2.5" x 3.8" (which is why the Renard-HC and related PCBs are all that size). The service has quick turn-around time, which was helpful since I had to go through a few revisions of the Renard-HC PCB. I did not buy silk screens, due to the extra cost and since these were prototype PCBs anyway.

Now that the Renard-HC PCB is working, if I were to switch to another PCB supplier I'm guessing the Renard-HC PCB would cost something like \$40 setup ÷ 8 PCBs + \$0.60 per square inch[17] x 9.5 "² ≈ \$11.00 per PCB (based on a quantity of 8 PCBs, which would be enough for a 32x15 grid + 280 AC SSR channels + and one spare PCB), although it sounds like SeeedStudio might be even less expensive[18] (10 boards up to ~ 3.9" x 3.9" for \$40 would be ~\$4 each).

There was also a little miscellaneous hardware and power/data/USB cables, but those or equivalent would be needed regardless of the type of controller, so I did not include them in the above list. I used a 5V FTDI TTL 232R-5V cable running to the first Renard-HC, and Cat3 from there to the 840 Renard-HC for the pixel grid.

[15] http://doityourselfchristmas.com/forums/showthread.php?10897 Old man thinks he wants to learn PCB … mmulvenna
[16] http://doityourselfchristmas.com/forums/showthread.php?11038 Will this PCB design work okay?
[17] Example price from http://doityourselfchristmas.com/forums/showthread.php?7206 Free PCB Design Software
[18] http://doityourselfchristmas.com/forums/showthread.php?7206 Free PCB Design Software

Cost Comparison

Since my display only needed very basic RGB control, I was curious to compare the Renard-HC costs to higher quality DMX-based controllers, to see if my compromises in functionality were worth the savings.

I am not too familiar with using DMX to run RGB pixels, but from what I've read about it in a thread about the LPD6803-based ADSIC12RGB pixels[19], it sounds like a DMX dongle handles 512 ÷ 3 = 170 RGB pixels and a proprietary LPD6803 decoder handles 120 RGB pixels, and both of these are needed to run the LPD6803 pixels. This would add something like $0.83 per pixel (see the table below), which makes the effective cost of running LPD6803-based pixels about $1.58 per pixel (second table below).

Item	Bare cost	Cost + shipping	Cost per pixel
DMX controller (170 pixels)	$60	$70?	$0.41
LPD6803 decoder (120 pixels)	$40	$50	$0.42
Total per pixel			**$0.83**
Compare: Renard-HC			**$0.20 to 0.38**

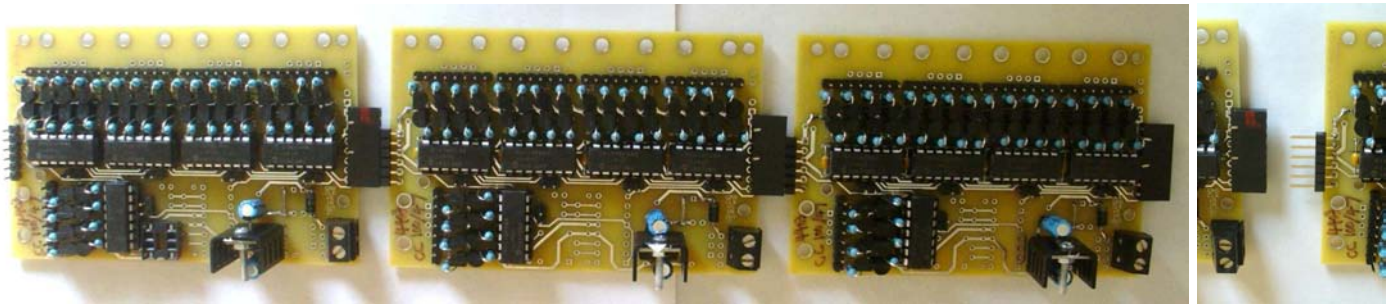
Now the cost of the LPD6803-based smart pixels is higher than dumb charlieplexed pixels, but the pre-built pixels are brighter, probably more durable, and there is no tedious construction time. Also, the DMX/LPD6803 decoder cost is higher than the Renard-HC controller, because it can control pixels more precisely. However, for my rather limited graphics needs, there's no reason why I can't inter-mix these approaches, combining the most attractive parts from each. For example, if each PIC on a Renard-HC PCB is used to drive four 50 ct pixel strings[20], then a Renard-HC PCB can drive 5 PICs x 4 strings/PIC x 50 ct/string = 1000 LPD6803 pixels, so the per-pixel cost of the controller would drop to about $27 ÷ 1000 pixels < $0.03.

Type of pixel	Type of controller	Cost per pixel
Dumb charlieplexed	Renard-HC (chipiplexed)	0.37 + 0.38 = 0.75
LPD6803	DMX + LPD6803 decoder	0.75 + 0.83 = 1.58
LPD6803	Renard-HC (using SPI)	0.75 + 0.03 = 0.78

The LPD6803 + Renard-HC combination would allow me to run higher quality, pre-built LPD6803-based pixels at about the same cost as the dumb pixels, without the tediousness of hand-building the pixels.

5.2. Construction

I used the Renard-HC PCB as a building block, connecting them together in groups of 3 to make each grid controller. Six PCBs were enough to run 2 controllers x 3 PCBs/controller x 5 ports/PCB = 30 strings of 16 RGB pixels. I only planned to run 28 strings instead of 30, so this allowed one spare port per controller.



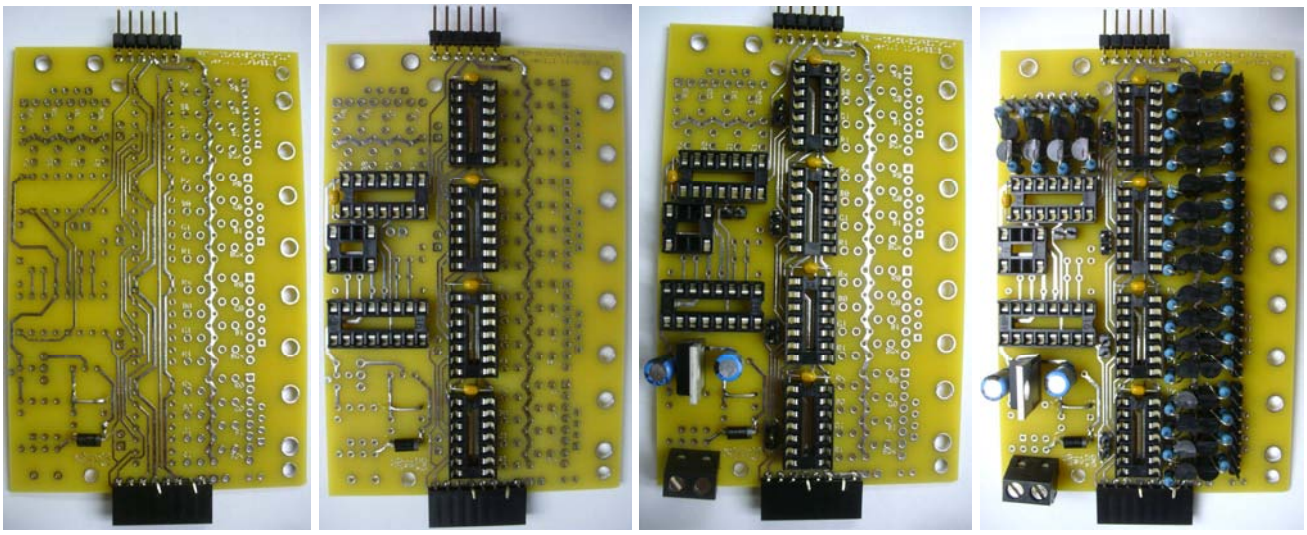
840 channel Renard-HC: 3 PCBs per controller; PCBs can be separated if connectors are used

[19] http://forums.auschristmaslighting.com/index.php/topic%2c120.0.html RGB Pixel Strings from Ray Wu's Online Store…

[20] There are 8 available I/O pins; only 2 are needed to drive a string of LPD6803s (I already have some prototype code)

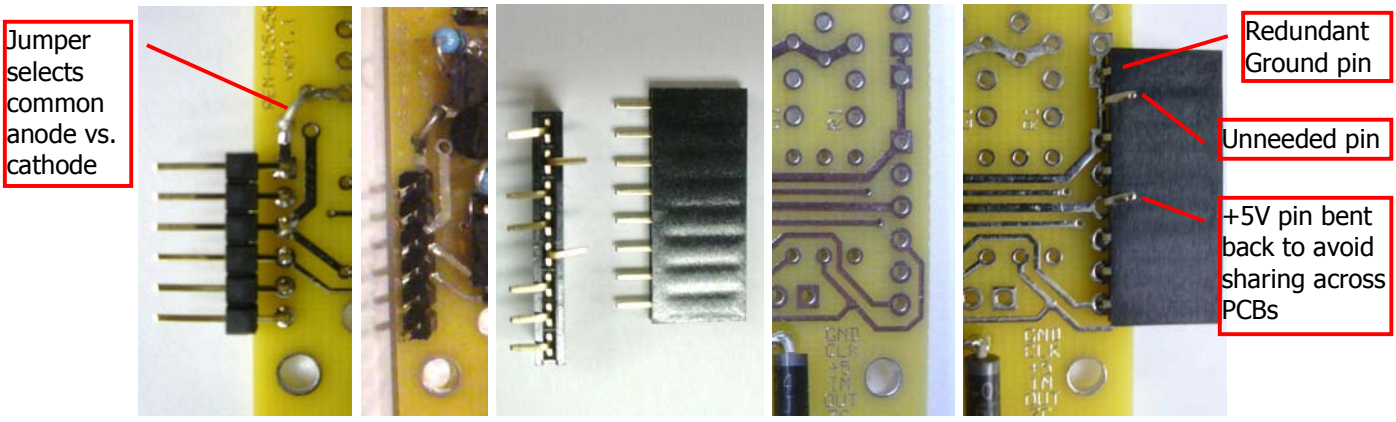
I followed the standard construction rules such as soldering 1 or 2 corner pins before the rest of the part, and adding parts according to their height, shortest to tallest, except that I added the voltage supply parts before all the transistors and resistors, even though they were slightly taller.

First I added jumpers, sockets, and other low-profile parts, then the voltage supply parts, then the transistors, resistors, and then SIP8 headers for the strings. I used SIP8 headers instead of RJ45 sockets because they were more compact and cheaper.



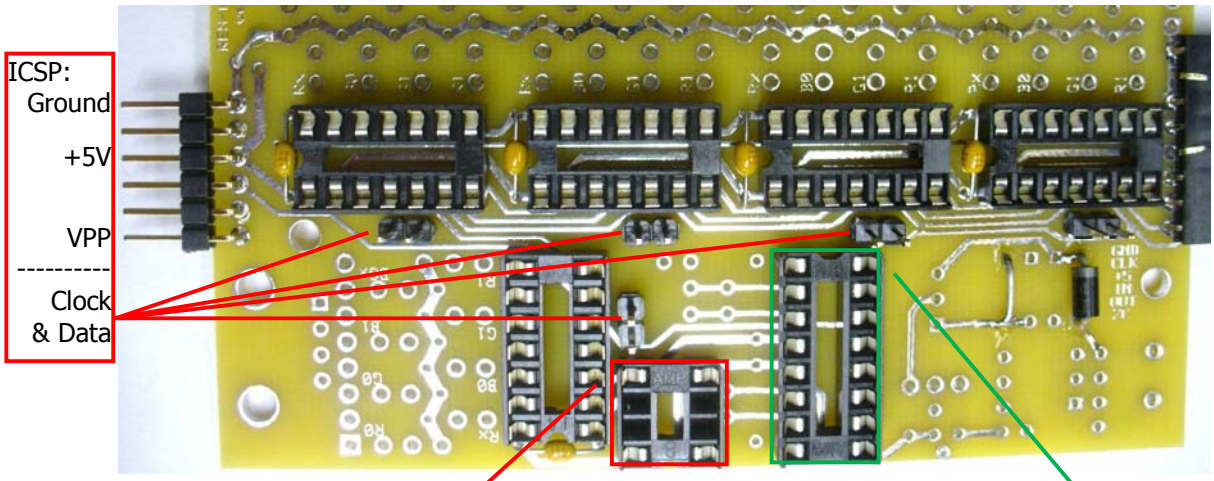
A Renard-HC PCB in various stages of completion, parts mostly added shortest to tallest

First I added the common cathode jumper (PNP transistors, collectors grounded), to match the RGB strings - the Renard-HC PCB can be used with either common anode or common cathode LEDs or SSRs. Then I added a SIP header and socket for connecting the PCBs together to form a larger controller. Rather than hard-wiring the PCBs together, I used SIP sockets and headers to allow the PCBs to be easily separated for testing and maintenance purposes. On the first PCB I used an upright SIP6 header rather than a right-angle header, so it would be more compact within the enclosure. (Only the first PCB can have an upright header, which prevents it from connecting to another one). For the SIP sockets, I didn't have right-angle sockets (I didn't plan that far ahead when ordering parts), so I just bent the pins manually. I only had SIP8 sockets, so I bent back 2 pins that were not needed – the eighth pin actually fit into an LED hole, so I only needed to bend back the one beside it, and also the +5V pin (with a 5V regulator on each PCB, they did not need to share the +5V line).



SIP header (right-angle or upright), and SIP socket (unused pins bent over)

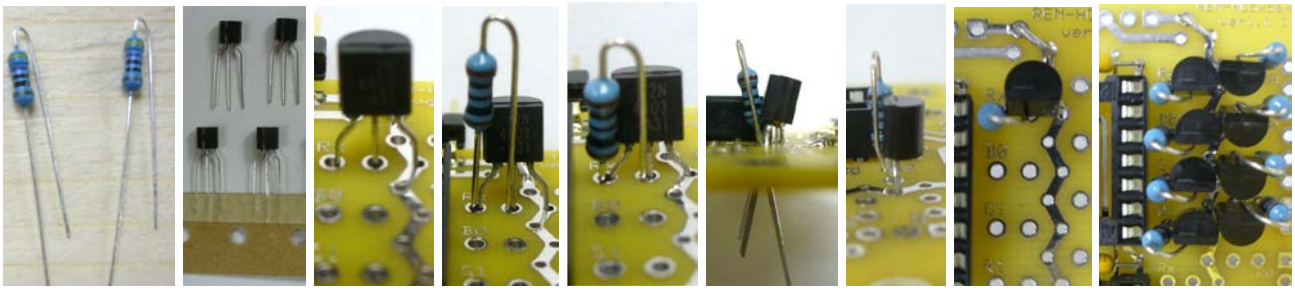
Next I added sockets and bypass capacitors for the PICs, and partial-ICSP headers.



IC sockets, PIC bypass caps, partial-ICSP headers

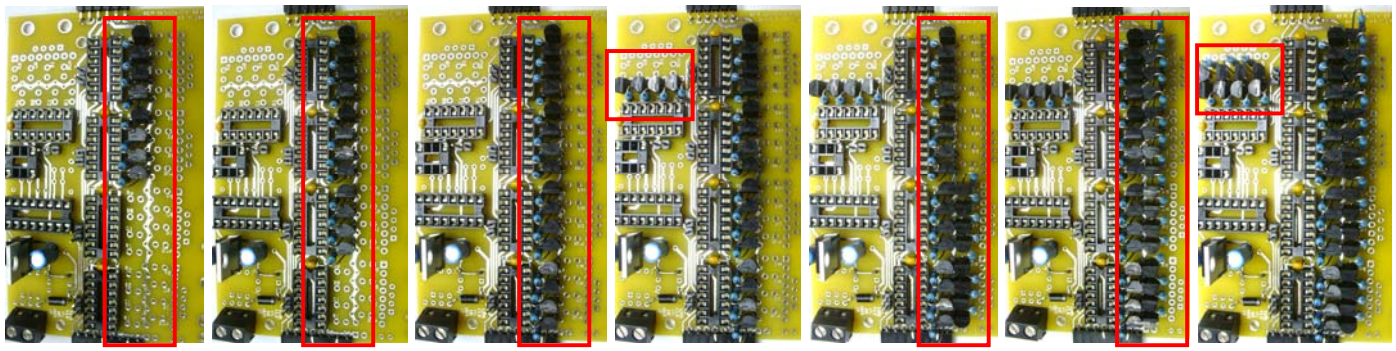
I used sockets for the PICs so I could replace any that burned out during testing (I expected some mistakes), and in case the ICSP headers did not work and I needed to remove the PICs to program them. I also used a socket for the external clock (4 of 8 pins removed) so I could change the speed or remove it if not needed, but I only did this on 1 out of 3 PCBs because it was shared between them. Similarly, I added a 16-pin socket to hold 2 RS485 drivers in case the controller was too far away from the PC (this didn't seem to be needed), but again only on 1 out of 3 PCBS because it was shared. The PCB did not have enough room for a full 5-pin ICSP header near each PIC, but I was able to squeeze in a 2-pin header for at least the Clock and Data lines. This worked out okay, because the VPP, +5V and Ground can be shared between PICs using the SIP6 header, and I use a custom connector for ICSP anyway (My Prototyping Setup photo shown earlier).

After adding the voltage regulator parts next, then I added all the chipiplexing transistors and resistors. The resistors must be installed end-wise (bend over one lead). I don't know if transistors always come this way from Mouser, but I happened to pick a part# that gave pre-bent transistor leads, which saved a little effort. In order to stay within the ExpressPCB Mini-board prototyping service restrictions (PCB size and hole count), the transistors and resistors share holes. I found it easier to partially insert a transistor first, then a resistor, then push them down together, but other methods probably also work. The holes are sized to easily hold both a transistor and a resistor, but are snug enough that they won't fall out when the PCB is turned over to solder.



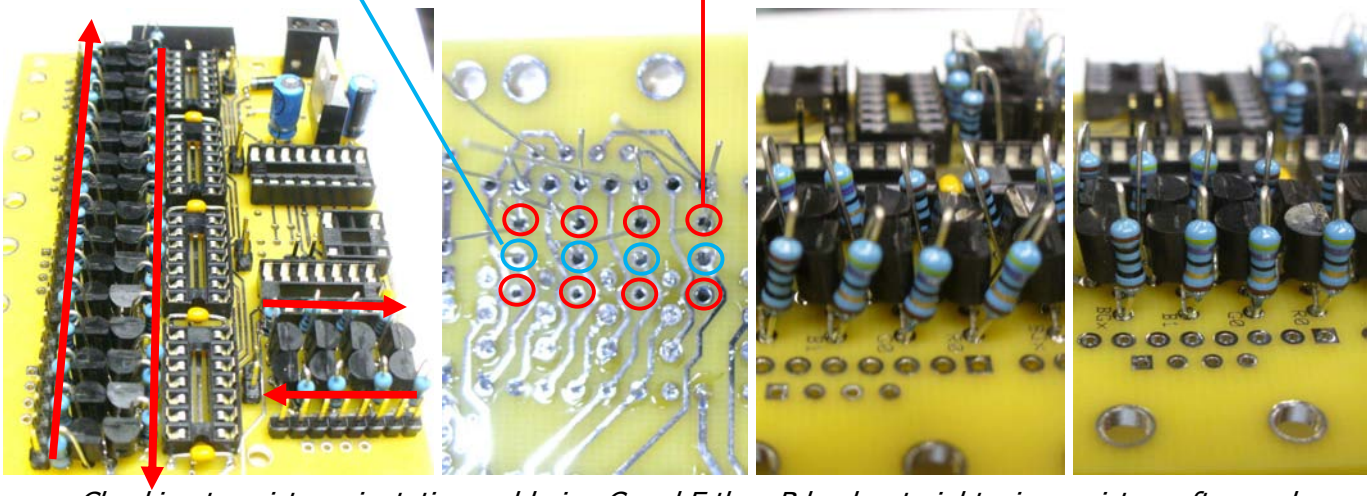
Resistor and transistor prep, inserting into shared holes, straighten up (tight fit)

I found it safer to insert and solder/clip those closest to the PICs first, working my way down the PCB from one end to another. A few times I accidentally swapped some of the resistors (R LEDs are 100 Ω, B/G LEDs are 47 Ω) or got some transistors turned around, so I think doing all the same type and orientation first helps to avoid those mistakes, and then double-checking before soldering also helped me catch some more errors.



Filling in transistors and resistors, working from one end of PCB to the other by PIC

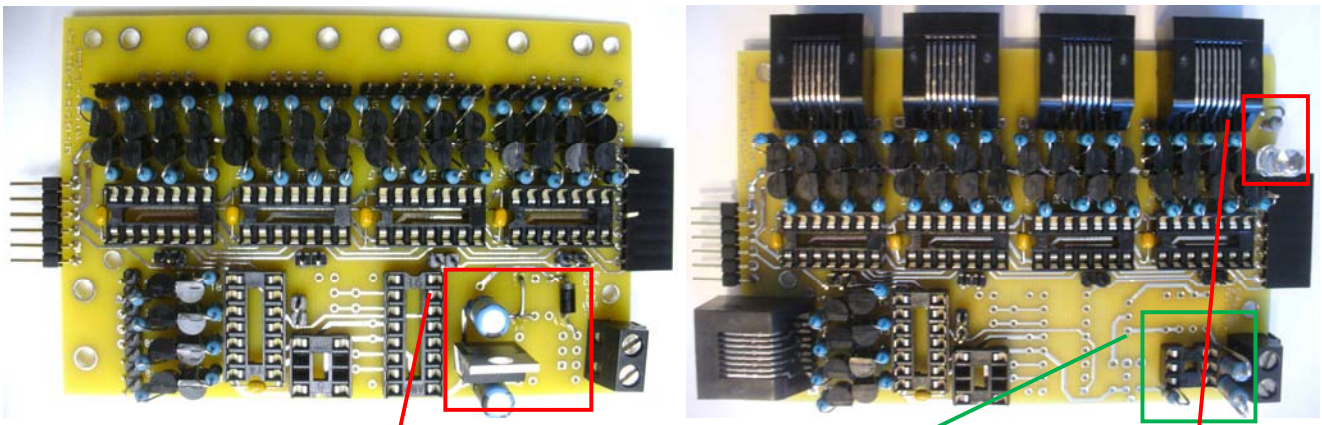
The orientation of the transistors can be checked quickly by looking at them from above; they should all be facing the same direction within a row, and facing the opposite direction within the other row. Since there are many leads in a tight space, I solder the outer leads first (transistor emitters and collectors), then trim those and solder the inner leads (transistor bases). After adding the transistors and resistors, I straightened them up a little, then finished off by adding the SIP8 headers.



Checking transistor orientation, soldering C and E then B leads, straightening resistors afterward

For the first few PCBs, I tested each one before building the next one, so that I could correct any mistakes and minimize wasted parts. This was also why I used sockets on all the chips and between the PCBs – with the PICs, ZC and/or clock removed, there are only a few dollars of parts left wasted on a dead Renard-HC PCB. The testing procedures I used are described in the next section.

I like the versatility of the Renard-HC PCB, because I can use it for AC SSRs as well as grid pixels. I built an additional PCB to drive AC SSRs for my discrete incandescent props, using the same general steps as above, except for a few differences.



Configured for RGB LEDs *Configured for AC SSRs*

Instead of voltage regulator parts, I populated the ZC detector (needed for AC line sync), I used different resistor values (5 mA optos instead of 10 – 15 mA for RGB LEDs), I added a power indicator LED (since there were no other RGB LEDs right near this controller), and I used RJ45 sockets instead of SIP8 headers (for the Cat5 cables going out to the AC SSRs rather than to my DIY RGB strings).

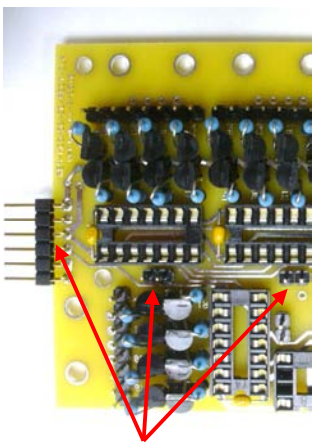
5.3. Programming/Testing

Even though I had already breadboarded and tested the Renard-HC circuit, my first few tries at using a PCB still had problems. These were due to incorrect spacing, hole sizes or layout of parts on the PCB, rather than the circuit itself. Even with the very helpful PCB Design tutorial I read, I guess there's still a lot to learn about PCB design. ☹ I also had various software problems, so those were also fun to work through. 😉

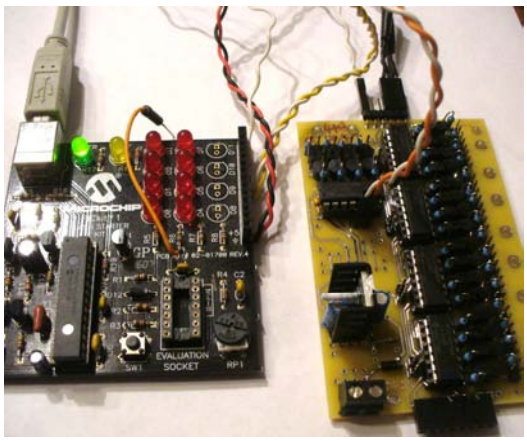
When testing the first Renard-HC PCB, I programmed the PICs with a hard-coded program to slowly step through the LEDs. This allowed me to verify that the populated PCB was working, and also provided a way to test the RGB strings. After working though those problems, I was able to use the real Renard-HC firmware to test the PCBs. I have now standardized on one working version of Renard-HC firmware, so it's easy to switch between test and real modes, change baud rate, or add/remove the clock without reprogramming the PICs.

I used ICSP to program the PICs. This avoids the wear and tear of pulling the PICs out over and over again (which I did many times, due to firmware bugs ☹) and is more convenient, so I am glad that I was able to squeeze it onto the Renard-HC PCB.

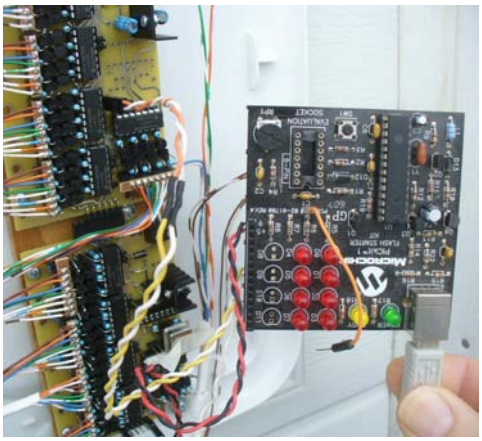
To program the PICs, I connected the VPP, +5V and Ground ICSP signals to the corresponding pins on the SIP6 header on the PCB (VPP goes to ZC). Then I connected the ICSP Clock and Data signals to the 2-pin partial-ICSP header beside each PIC – this allows one PIC to be programmed at a time. The programming seems to work even when an RGB string is connected to the PIC, which allows them to be easily programmed in-place, but this doesn't seem as reliable with multiple PCBs connected together (probably because there are too many PICs on the VPP or +5V line). The SIP6 headers and sockets on the Renard-HC PCBs allowed me to separate them if I needed, for ICSP programming or other purposes.



Partial-ICSP connections



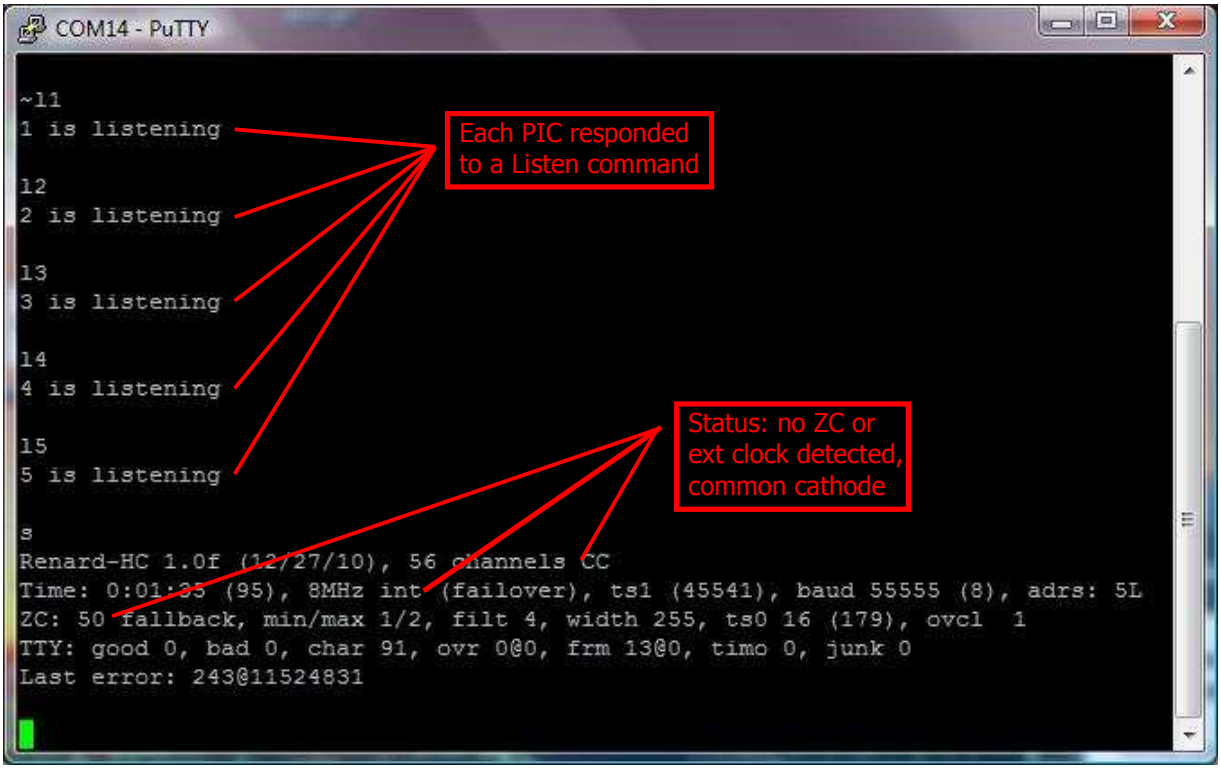
Initial ICSP programming



In-place ICSP

For testing the PCBs prior to installation on the garage door, I just connected them to a USB port using the FTDI cable. This is very convenient because the In, Out, +5V and Ground pins on the Renard-HC SIP6 header match the corresponding pins on the FTDI connector (ZC and Clock are not used for this type of test).

Then I used Putty to verify that the PICs actually got programmed correctly. I used the Listen command to check that each PIC responded – if they have not been connected to Vixen since power-up, L1 will address the first PIC, L2 the second PIC, etc. (otherwise the addresses will be as assigned by the Renard-HC plug-in). If I also want to check if I am using the correct version of firmware or if the external Clock or ZC are detected, then I will use the Status command. For this test to work, a loopback jumper also needs to be connected across the last Serial In and Serial Out pins (so the output from the PICs can get back to the USB port).



Putty test: check if PICs are programmed using Listen command

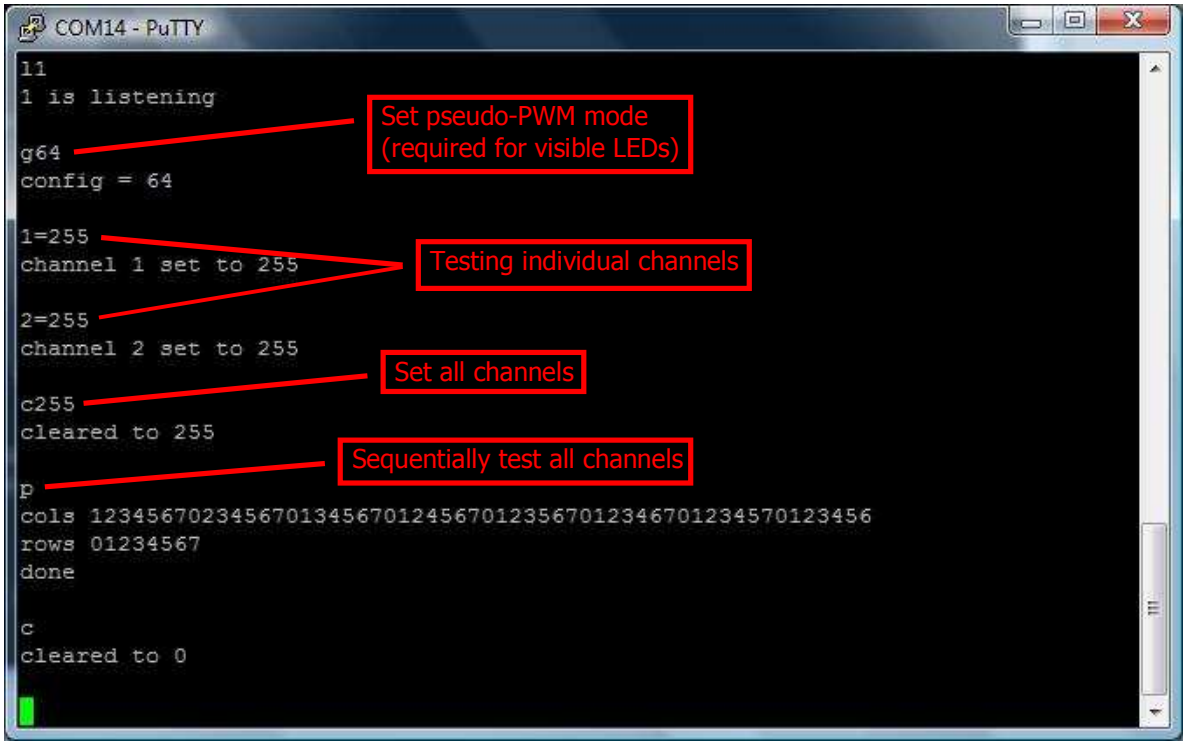
If the PICs respond correctly, then I connect an RGB string to one of the ports and try to turn on a few of the pixels. To do this type of test, I needed to know how the channels are arranged relative to the RGB LEDs, so I mapped them out by trying each one and then made the following chart as a reference:

Pin/bit	RA4 0x80	RA2 0x40	RA1 0x20	RA0 0x10	RC3 0x08	RC2 0x04	RC1 0x02	RC0 0x01
RA4 0x80	none	1 = G#7	2 = B#7	3 = R#7	4 = unused	5 = R#6	6 = B#6	7 = G#6
RA2 0x40	8 = G#B	none	9 = B#B	10 = R#B	11 = unused	12 = R#A	13 = B#A	14 = G#A
RA1 0x20	15 = B#F	16 = G#F	none	17 = R#F	18 = unused	19 = R#E	20 = B#E	21 = G#E
RA0 0x10	22 = unused	23 = G#3	24 = B#3	none	25 = R#3	26 = R#2	27 = B#2	28 = G#2
RC3 0x08	29 = unused	30 = G#5	31 = B#5	32 = R#5	none	33 = R#4	34 = B#4	35 = G#4
RC2 0x04	36 = unused	37 = G#1	38 = B#1	39 = R#1	40 = R#0	none	41 = B#0	42 = G#0
RC1 0x02	43 = B#C	44 = G#D	45 = B#D	46 = R#D	47 = unused	48 = R#C	none	49 = G#C
RC0 0x01	50 = G#8	51 = G#9	52 = B#9	53 = R#9	54 = unused	55 = R#8	56 = B#8	none

I/O pin to channel and RGB LED mapping

The R, G, and B LEDs are numbered 0 – F above. You can see the general pattern of an odd and even R, G and B LED in each row. For example, setting RC2 low (row) and RC3 high (column) is channel 40 on the PIC (when using common cathodes), which should turn on Red LED #0 (the first one) in the RGB string.

I only connect one RGB string because the FTDI cable can only supply enough current for 1 string. First I tell the PIC with the RGB string on it to Listen, then I use the Set Channel command to turn on a channel. After setting various channels, if one of the LEDs lights up each time (one color only), then that tells me that at least the chipiplexing transistors and resistors are working. The LED will be very dim unless pseudo-PWM is turned on first (G64). For a quick test if all channels are working, the Clear command can be used to turn them all on (C255). To test if all the LEDs work individually, then the Playback command (P) is also useful.



Putty test: turning on 1 channel, all channels, or sequential test

Setting channels manually is also a way to check the "dimming curve" for the LEDs. Although the Renard-HC handles 256 dimming levels per channel, I have not checked how exact the pseudo-PWM is, and the relative brightness also depends on the LEDs and is typically not linear.

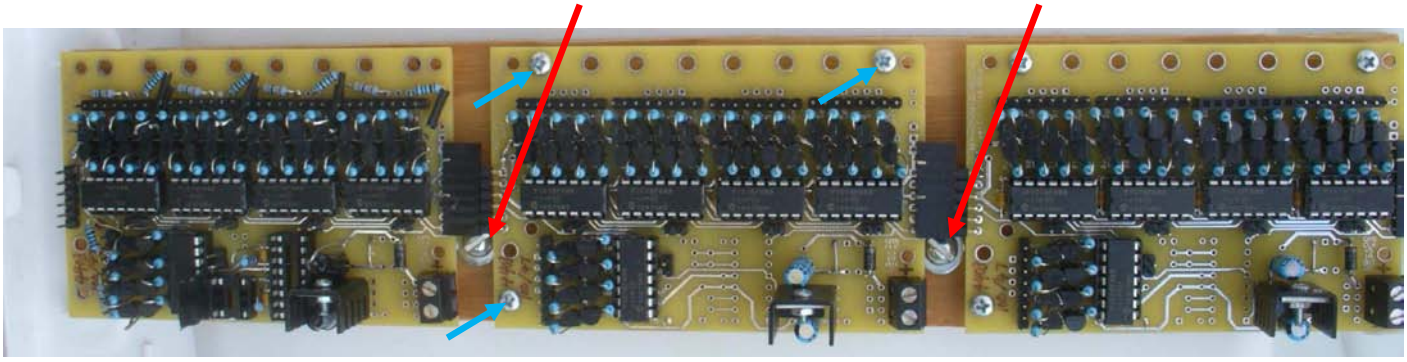
If a stand-alone test with Putty works, then Vixen can be used for a more complete test. Several steps are needed to set this up; these are described in the chapter on Sequencing.

5.4. Installation

I used the following materials to install the 3-PCB pixel controller (it would have been double this if I had used both controllers and all the pixels):

- one 6 quart plastic container from Walmart or Home Depot
- two #8-32 x 1½" machine bolts, nuts, and washers
- a ~3" x 12" piece of 1/8" plywood
- nine #4 x ½" metal screws
- about 2" of 1/8" (inner diameter) plastic tubing
- one nylon cable clamp

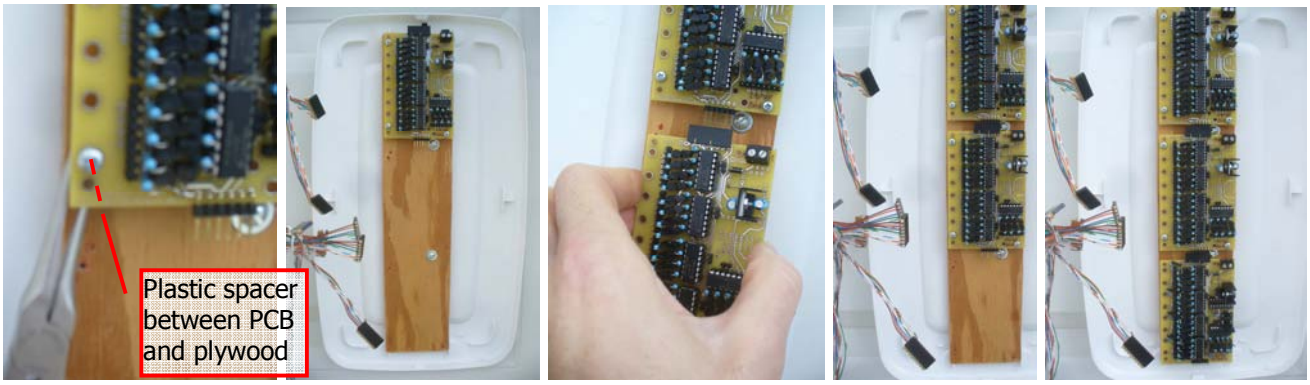
First I drilled mounting holes through the plywood and container lid. I positioned the controller (3 PCBs) on the plywood and marked where the gaps between the PCBs were, then drilled one mounting hole between each pair of PCBs so that the bolts would still be accessible after the PCBs were installed:



Hole positions for mounting bolts (red arrows) and stand-offs (blue arrows)

I also drilled smaller holes for the stand-offs, 3 per PCB. The screws I had were a little too big for the PCB mounting holes, so I just used some of the RJ45 holes since those were vacant (I used SIP8s instead).

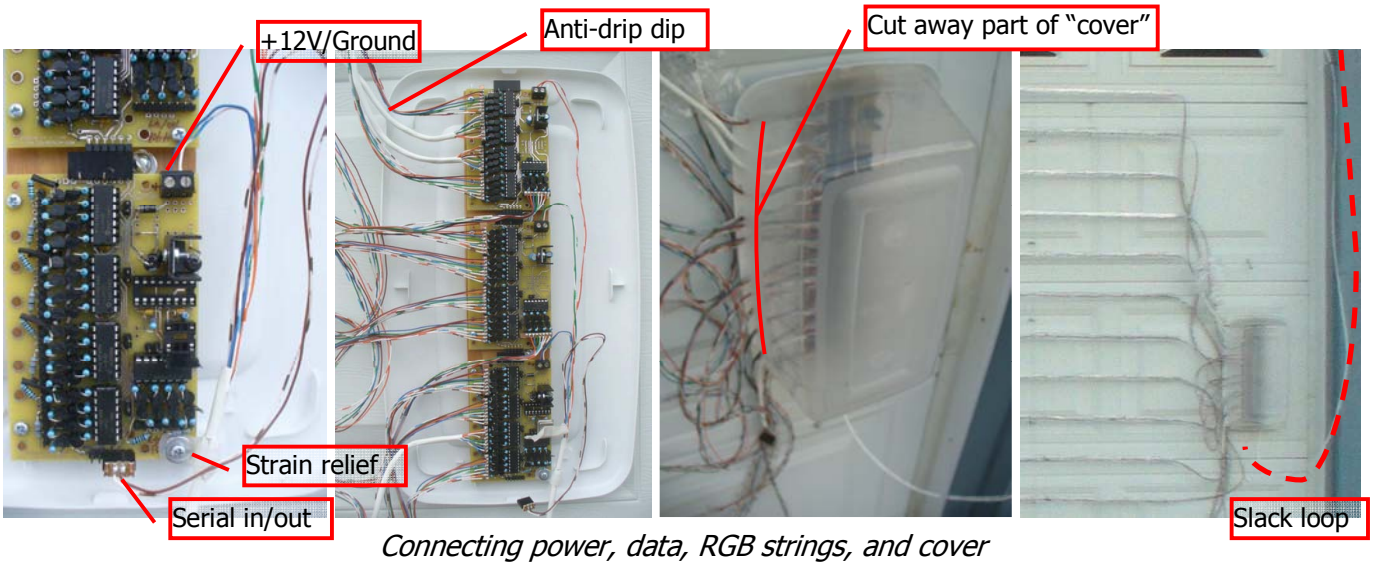
Then I used the plywood as a template to drill 2 mounting holes in the garage door (in my part of the door, not my wife's). The controller would be mounted to the side of the RGB strings, on the middle garage door panel so that the strings from the panels above and below would also reach it. I inserted the 2 mounting bolts through the washers, plywood and container lid, then attached the nuts on the back-side of the door and tightened them. This created a sturdy "base" on which to mount the PCBs.



Attaching the PCBs one at a time to the base, using stand-offs

I oriented the PCBs so the SIP8 connectors were closest to the pixels. On the right half of the garage door, the PCBs were actually "upside down" (serial data flowed against gravity) – there was no noticeable decrease in baud rate, though ☺ (on the left half of the door the serial data would flow the same direction as gravity).

I attached the PCBs to the plywood base using the stand-offs. I added one PCB at a time, plugging them together using the SIP headers and sockets. I cut the plastic tubing into 3/16" lengths for use as spacers, then held the plastic spacer tubes in place with pliers, inserting the stand-off screws one at a time.



Connecting power, data, RGB strings, and cover

I ran Cat3 to the controller, with 3 pairs carrying +12V from the PC power supply (one pair to each PCB), and the fourth pair carrying serial in/out. The controller was close enough to the up-stream controller (Renard-HC running AC SSRs) that I didn't need the RS485 drivers, but I had put a socket on the first PCB in case they were needed. I connected the serial in and out to the corresponding pins on the upright SIP6 header of the first PCB, with a loopback jumper on the last PCB to complete the loop.

The Cat3 comes out from the upper right corner of the garage door opening, so I left a loose loop of Cat3 hanging at the edge of the garage door to prevent it from being pulled tight when the door is all the way opened or closed. It took a few tries to get the correct length – I started with a longer loop and then shortened it if there was too much slack with the door fully opened AND fully closed. I also anchored the Cat3 to one of the standoffs for strain relief using a nylon cable strap.

I connected the SIP8 sockets on the RGB strings to the SIP8 headers on the PCBs. A few of them didn't reach, so I had to make a couple of SIP8 "extension cords" ☹. With the enclosure lid secured to the door, the enclosure body becomes the weather-proof cover – it just snaps into the lip. I had to cut away a strip on the string side to allow space for the RGB cables to enter. However, that created a drip hazard – rain from wires higher than the controller could run down the cable into the controller. To avoid this, I make sure that the cables going into the controller are horizontal and then bend a dip in the cable before it gets to the controller (the photo above was shot before I had finished making those adjustments). If there is not enough slack in the cable to do that, then I suppose wrapping something absorbent around the cable will work for smaller amounts of moisture.

Since the cover was translucent white and I used Cat3 with a white jacket, they blended in well with the garage door, making them less noticeable. They gave a fairly neat appearance, although the colored Cat3 wires going to the RGB strings was a little more noticeable.

6. Sequencing

I used Vixen 2.1.4.0 with a couple of custom plug-ins to run sequences on the Renard-HC controllers this past season. These plug-ins are described in the sections that follow, and I'll include them along with this writeup. I also used standard plug-ins such as Adjustable Preview and Renard, but those are already documented.

I'll also briefly describe the process I used to sequence the dumb pixel grid. I found this process useful for testing the RGB strings and Renard-HC controller, as well as for actually creating the show sequences. This process will probably change as I rework the plug-ins before the next display season.

I was a little nervous about using over 2K channels in Vixen, since I had used nowhere near that number of channels before. I was relieved to find that it worked pretty well. The only problems I had were a run-time error at the beginning of the show Program each night (probably due to memory), and sluggishness when first starting the Program - it looks like Vixen tries to load the entire Program into memory first before running it. The run-time error was just an annoyance, because the Program continued to run the rest of the evening okay; I just had to restart Vixen after the Program was over for the night.

The sluggishness was due to an under-powered PC, but since the sequences actually ran okay I just lived with the initial delay when loading. The show computer was an old 2.2 GHz AMD with 512 MB RAM running Windows 2000. *:embarrassed:* I turn off Adjustable Previews when running the sequences for the actual show, so all Vixen has to do is read an XML file and spit out bytes over a serial port, and it was able to do this comfortably even on such an old PC. For the actual work of creating/editing the sequence, I use a more up-to-date dual-core Intel laptop with a few GB of RAM, and sequences load fairly quickly on that.

After reading that AussiePhil was able to run over 8K channels in Vixen[21], I feel comfortable that I still have some head-room left in Vixen. If I ever do hit the ceiling, I will probably switch over to Linux since several DIYC forum members have had success running Vixen sequences there[22].

6.1. Renard-HC Plug-in

Since I was using the same basic Renard-HC circuit and firmware from last year, I was able to reuse my existing Renard-HC plug-in, although I did make a few adjustments to it.

The first change was related to channel reordering. In past years, I've just connected the light strings to the SSRs in whatever order was most convenient based on their physical layout and power consumption, then used the channel reorder function in Vixen to rearrange the channels into their correct order by swapping pairs of channels until they are all in order. For a few hundred channels, this is a little tedious, but still manageable. However, with the higher channel count of a pixel grid, I found this approach was not practical any more. Instead, I added a little logic into the Renard-HC plug-in to reorder the channels automatically for each PIC, so that no manual reordering of the grid is needed at all. This logic also skips over the 8 unused channels within each group of 56, so I didn't need to skip over channels when editing within Vixen.

Another change I made was for better I/O performance. The Renard-HC plug-in will now only send out data to the PICs whose channels changed during the previous interval, rather than sending out all the channels updated by Vixen (Vixen sends all the channels assigned to a controller, even if only a few of them change). This was only a minor code change, but had a huge impact on serial I/O bandwidth. For example, with some simple "follow the moving pixel" test sequences, this resulted in a compression ratio of about 100:1 (11 data bytes sent out, compared to 1120 bytes passed in by Vixen). For a more realistic sequence, the compression would not be as high as this, but it is still a major improvement from last year. The larger the grid, the more significant the bandwidth savings can be.

[21] http://doityourselfchristmas.com/forums/showthread.php?14713 Aussiephil 2010 Show Videos
[22] http://doityourselfchristmas.com/forums/showthread.php?8526 Linux Ubuntu 9.04 with Vixen

I also hard-coded a temporary change into the Renard-HC plug-in to allow mixed RGB pixel grids/arrays and discrete light strings on AC SSRs, since I was running both over the same USB port. I think I have since removed that change, but be aware of this if you try to use this plug-in for a mixed environment. I need to formalize this into a configuration option that allows the choice of charlieplexed dumb RGB pixels (with 8 unused channels out of each group of 56), vs. monochrome pixels where all channels can be used. I recently identified yet another option that I would like to add (described later in the Future Plans chapter).

6.2. Channel Reorder Wizard

I also started work on a Channel Reorder wizard Add-in, to allow easier semi-automated channel reordering within Vixen. The idea was that it would sequence through a designated range of channels one at a time, announcing the channel and allowing a quick, 1-click correction, somewhat like an insertion sort algorithm. However, I put this on hold after I realized how much work it would be to manually reorder an RGB grid.

Instead of pursuing this plug-in further, I hard-coded the reordering logic into the Renard-HC plug-in, and then allowed the overall order of channels to be controlled via mouse direction in the Grid Editor. I may come back to this again later, after more critical RGB grid functions are all working.

6.3. Grid Editor Plug-in

Channel-by-channel RGB prop sequencing can be rather tedious in Vixen, because each RGB pixel is represented by 3 separate channels. The RGBLED Add-in helps a little with this, but seems to be oriented more towards smaller numbers of RGB pixels. So, I started working on a Grid Editor plug-in to allow easier setup and sequencing of RGB pixel grids or arrays. I didn't get as far on this as I would have liked, but I did get it running well enough to be able to add some simple animated graphics into my sequences this past year.

The Grid Editor is (supposedly) independent from the actual controller hardware, so it can be used in different environments. It is intended to serve 5 purposes: a quicker grid setup, frame-by-frame graphics editing, effects generator, playback control, and allow graphics to be embedded directly within the sequence. First I started with Vixen's existing Adjustable Preview plug-in and then added playback control (pause/resume, prev/next frame, etc). Then I started adding Windows Paint capabilities (I didn't get too far with this yet). I'll probably also add text handling functions similar to the LEDTRIKS editor, and then also an Effects Generator for fancier graphics maneuvers. I'm not trying to replace other software tools that do this – I'm just trying to collect up these functions and package them so I can use them directly from within Vixen with an RGB grid.

I've identified a number of additional features I'd like to add to this plug-in, so I'll probably rewrite it for the next display season. If anyone would like to use the current version as-is in the mean time, you're welcome to play around with it, and I'll answer questions about it, but I probably won't be fixing any bugs until the rewritten version is available.

6.4. Grid Sequencing Steps

The steps I used to add graphics into my sequences are described in the remainder of this chapter. I used these steps for testing, as well as for the actual show sequences. Reader familiarity with Vixen is assumed, and in some places the descriptions are a little vague to encourage further reader experimentation.

6.4.1. Plug-in Installation (Once Only)

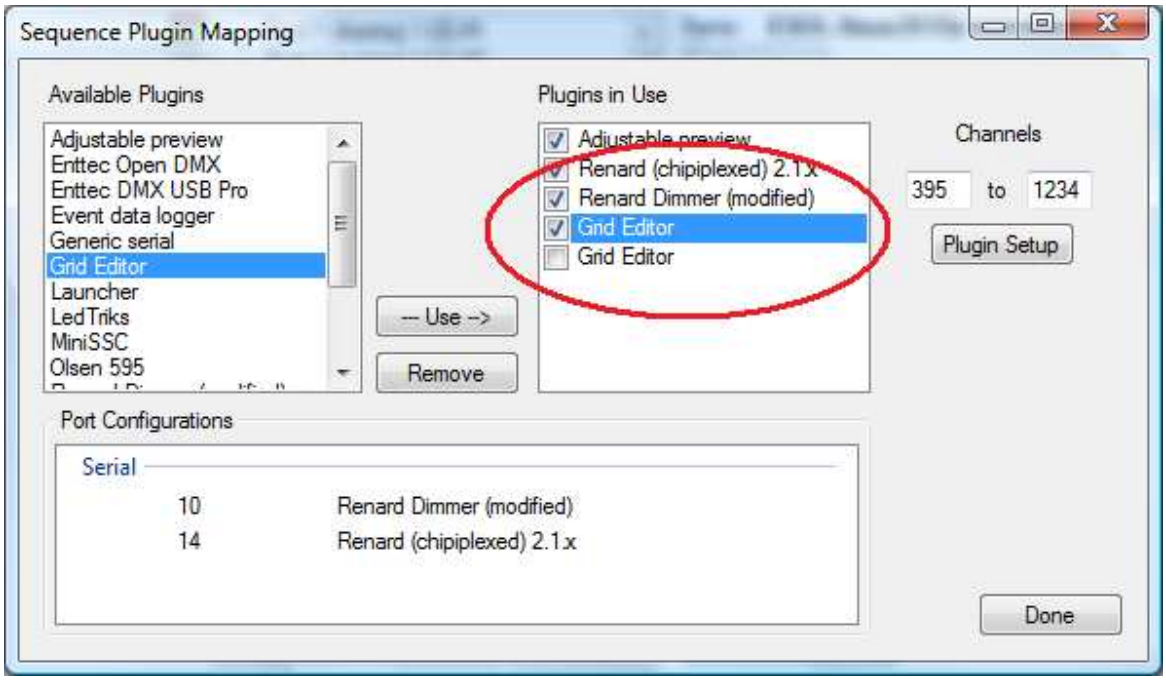
To "install" the Grid Editor plug-in, I copied GridEditor-21.dll into Vixen's Plugins/Output folder. Vixen will automatically recognize DLLs in this folder if they export a class implementing the IEventDrivenOutputPlugIn interface. There are 2 versions of the plug-in (both built from the same #develop solution): one for Vixen 2.1.x and the other for 2.5.x. I haven't tested the 2.5.x version recently but it's the same source code so it might work. If anyone tries to use it and has problems, let me know and I'll try to take a look at it.

Installation of the Renard-HC plug-in is similar – just copy the -21.dll or -25.dll into Vixen's Plugins/Output folder.

6.4.2. Plug-in Setup: Adding Channels

After the plug-ins were installed, I opened a profile and added channels for the RGB grid. Although I used a Vixen profile with my sequences, the same technique works for sequences without a profile.

Since I had planned to run a 32 x 15 grid (14 rows + 1 spare), and each 16-pixel RGB string uses 56 channels in the Renard-HC (3 x 16 + 8 wasted = 56 channels), I added 2 x 56 x 15 = 1680 channels to my Vixen profile. I planned to use a separate 840-channel Renard-HC controller for each side of the grid, so I split the grid channels into 2 ranges, and added the Grid Editor plug-in to the profile twice (once for each side). This made it easier to manipulate each half of the grid separately, since I was unsure if I would have enough time to get the second half of the grid actually installed and running. Vixen seems to only allow 1 plug-in per COM port, so I only added the Renard-HC plug-in itself to the profile once.



Adding Renard-HC and Grid Editor plug-ins to the profile or sequence

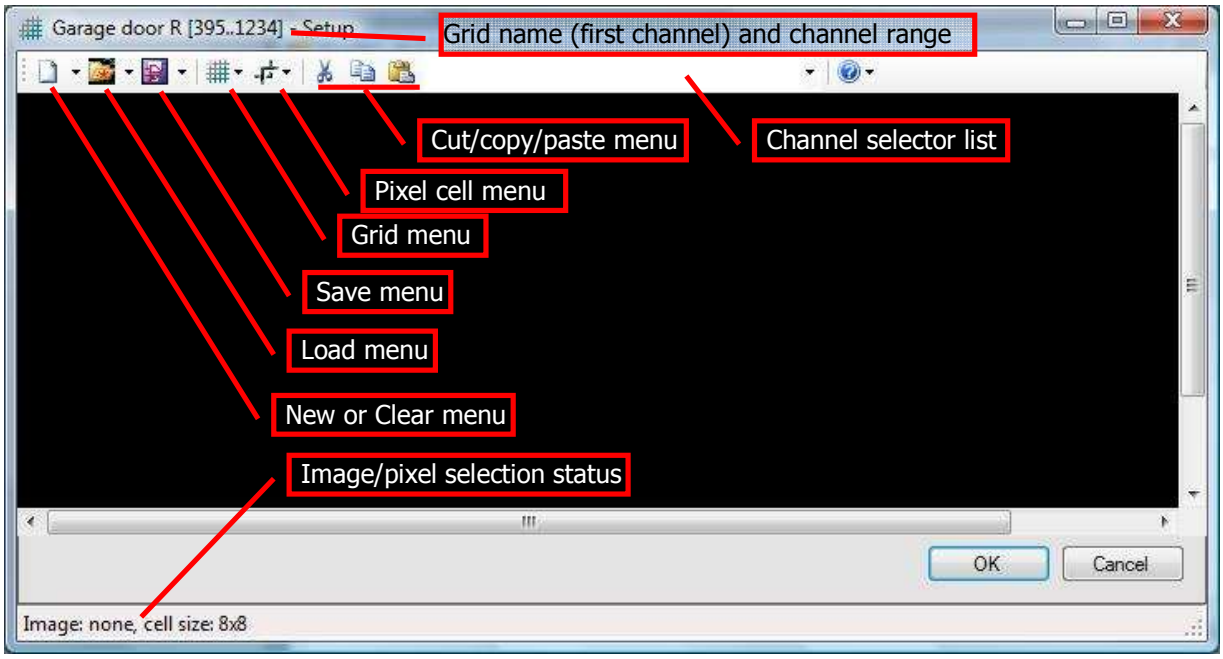
At this point, I clicked Done to force the new channels to be written to the profile, so the plug-in will see them (it re-reads the XML file to get the info). Then I renamed the first channel of each half of the grid for easier recognition – otherwise, it's easy to get lost in a sea of channels that all have similar names. The Grid Editor uses the first channel's name for the grid itself, and renames its channels using the grid name as a prefix, followed by the coordinates within the grid (to make it easier to distinguish the channels used by each grid pixel).

Next I updated the Adjustable Preview's channel range to also include the new grid channels, because I wanted the grid to show within the Adjustable Preview window as well (the Grid Editor output is compatible with Adjustable Preview).

6.4.3. Plug-in Setup: Grid Characteristics

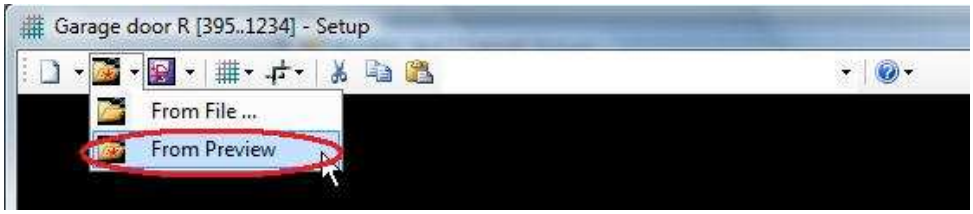
After the channels were added, I selected the Grid Editor plug-in again and clicked the Plugin Setup button to define the visual representation of the grid.

The Grid Editor window resembles the Adjustable Preview plug-in, except that the menu items have been replaced with icons and there are more of them. The Grid Editor Setup window is shown below:



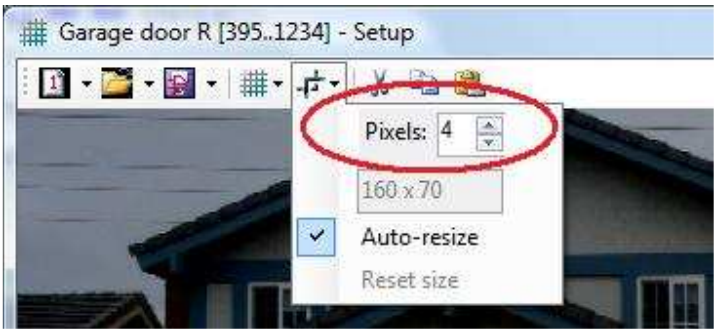
Grid Editor Setup window and menus

I usually set up the background image next. Like the Adjustable Preview, the Grid Editor can load an image from a file, but it can also just reuse the (first) Adjustable Preview's image. I generally choose this option so the grid is sized and positioned to fit correctly within the Adjustable Preview, and then I can see everything in one Adjustable Preview window later (the Grid Editor can also write back to the Adjust Preview window):



Menu items for loading a background image

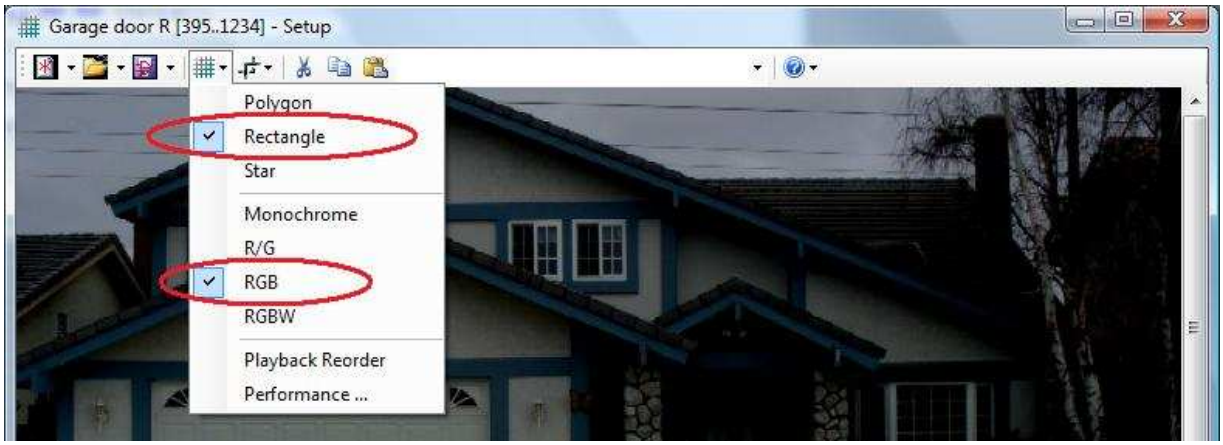
Once the background image is set, I darken the image a little using the slider at the bottom, to make it look more like nighttime (I read this tip in one of the DIYC threads), then I set up the grid and pixel characteristics. Using the Pixel menu, I set the pixel (cell) display size to match the Adjustable Preview, so the grid will line up correctly if I save it back into Adjustable Preview later:



Pixel cell size options

The other items on the Pixel menu also match the menu items and buttons that are found in the regular Adjustable Preview (they're just rearranged), so I won't describe them further.

My grid contained RGB pixels and was rectangular, so I chose those options from the Grid drop-down menu:



Grid shape and type

An RGB grid will use 3 Vixen channels per pixel. The other choices are: monochrome (1 channel per pixel), R/G (2 channels each), or RGBW (4 channels each). I've done a little testing with monochrome pixels, but I have not used the R/G or RGBW options yet, so they probably do not work.

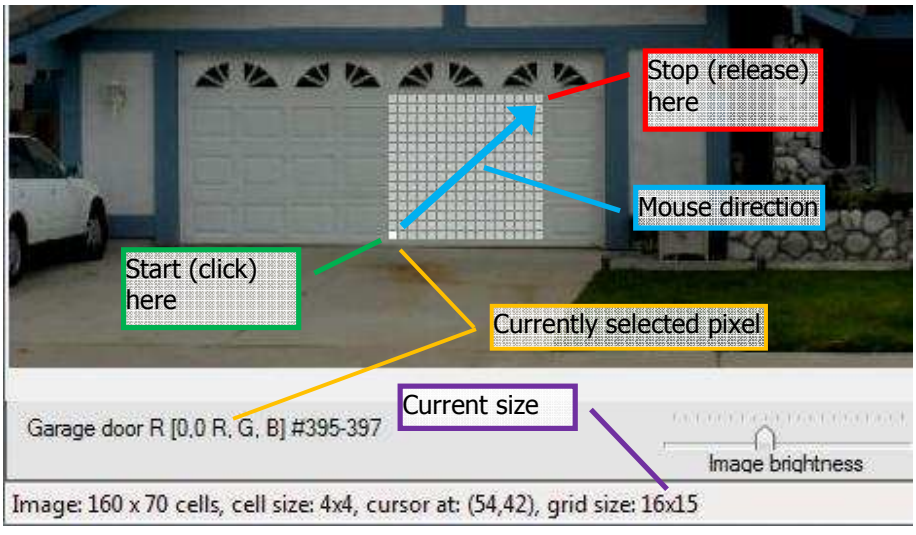
The plug-in assumes that each channel is capable of 256 levels, since Vixen passes one byte for each channel. Therefore the pixel size should be chosen according to the type of pixel, not necessarily the color depth. For example, the R/G option would be appropriate for LPD6803-bsaed pixels, since that would use 2 bytes/pixel.

I periodically back up the profile or sequence that I am editing, just in case any bugs corrupt the file.

6.4.4. Plug-in Setup: Grid Preview

After setting the pixel (cell) size and grid characteristics, my next step was to draw the actual grid itself. The Grid Editor makes this very easy - a single mouse click/drag/release movement can define all the cells for the grid, as well as setting the channel names and colors.

The controller was mounted "upside down" (serial data flowing against gravity, first row at bottom), and the strings were installed left-to-right. I wanted the grid channels in Vixen to follow this same order, so I used a bottom-to-top, left-to-right mouse movement when drawing the grid:

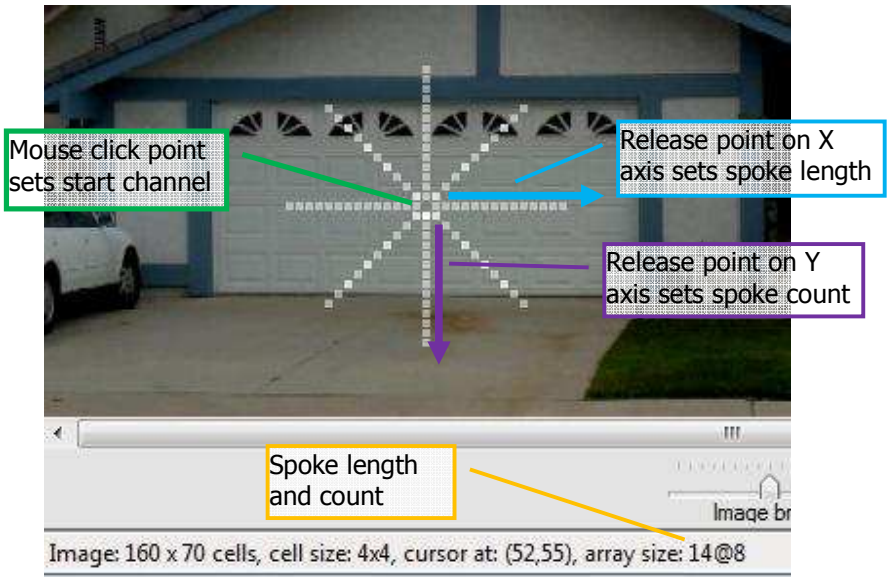


Rectangular grid: mouse direction determines channel order

This assigned the first channel in the grid to the lower left corner, and the last visible channel to the upper right corner. For RGB pixels, this also sets the color of the first channel to red, second to green, third to blue, etc, even though the cells show as white on the screen (they will have colors later during sequence editing). I will probably add an option for zig-zagged strings (assign channels in left-to-right or right-to-left order within alternating rows), so that I can use longer commercial pixel strings in future.

The currently selected grid size shows at the right of the status bar, which allowed me to check that I had selected the intended number of pixels while dragging the mouse (16 columns/LEDs, 15 rows/strings).

Channel assignment while drawing rectangular grids is fairly obvious, but the Grid Editor also supports circular or star-shaped props. With Star selected in the Grid menu, channels will be assigned in a radial pattern, outward from the initial mouse click point:



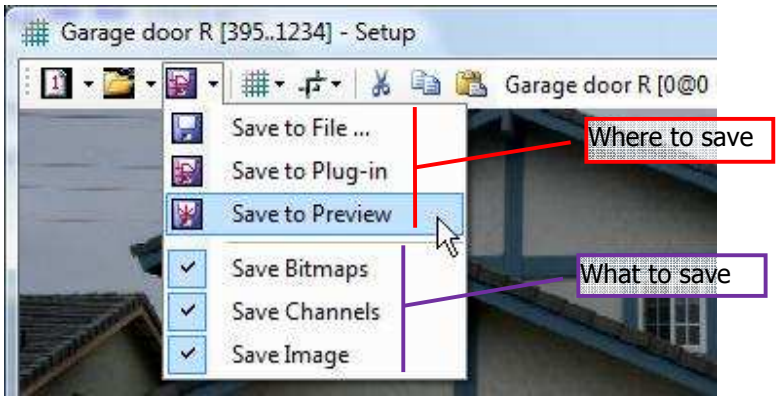
Radial channel assignment

The Y axis is used to select the number of radial spokes, and the X axis is used to select their length, up to the number of channels allocated to the grid. This takes a little practice at first, but is still a very quick way to assign channels to a round or star-shaped prop. As with Rectangle, the direction of the mouse also controls which order the channels are assigned. Star will always assign the first channel to the center, and then the channels are assigned outward from there, with the spokes assigned in a clockwise or counter-clockwise order depending on the direction of the mouse relative to the starting point. I suppose I should allow the channels to be assigned in an inward order as well as outward order.

When Polygon is selected in the Grid menu, the prop can be drawn free-hand, and channels are assigned to each cell sequentially, starting at the mouse-down point.

If no shape is selected in the Grid menu, then cells can be manually assigned to channels, similar to how channels are assigned in the regular Adjustable Preview plug-in. This is useful for precise drawing of irregular shaped props, or for going back and "touching up" a grid drawn with the Rectangle, Star or Polygon menu items – once you release the mouse those functions stop and will start all over again if you click the mouse again, so turning off all shapes is a handy way to do touch-ups.

After drawing the grid, I save it back to the sequence or profile using the Save menu. There are several options available for Save:



Save menu options

The top part of the Save menu selects where to save the grid info, and these options are sticky (the menu icon will remember the last option used). I normally save the grid definition back to the plug-in itself so I can edit it again later if needed. Often I will also save it back to the Adjustable Preview, so the grid will be displayed along with other props in the first Adjustable Preview window. Like Adjustable Preview, the Grid Editor also allows the background image to be saved to a file.

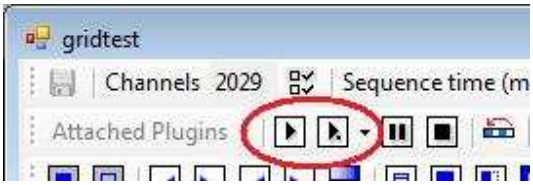
The bottom part of the Save menu allows grid data to be selectively saved. I normally leave these all on, but since drawing the grid also sets the channel names and colors, I sometimes use the Grid Editor just to set the names and colors of non-grid channels. To do this, I add a temporary Grid Editor for the range of channels that I want to rename or re-color, then draw the grid and save only the channel information (turn off Bitmaps and Image), then remove the Grid Editor again. This leaves the channel names and colors set according to grid shape and type when I drew the temporary grid. For example, on unused Renard-HC ports, I added a temporary Grid Editor with 56 monochrome channels, then drew an 8 x 7 rectangle so that the channels would be named by their charlieplexed row/column.

6.4.5. Sequence Playback

Like Adjustable Preview, the Grid Editor playback window will only be displayed during sequence playback, and hidden at other times. The Grid Editor also provides Pause and Resume functions, which allow me to stop and make adjustments to the grid as I work through the sequence.

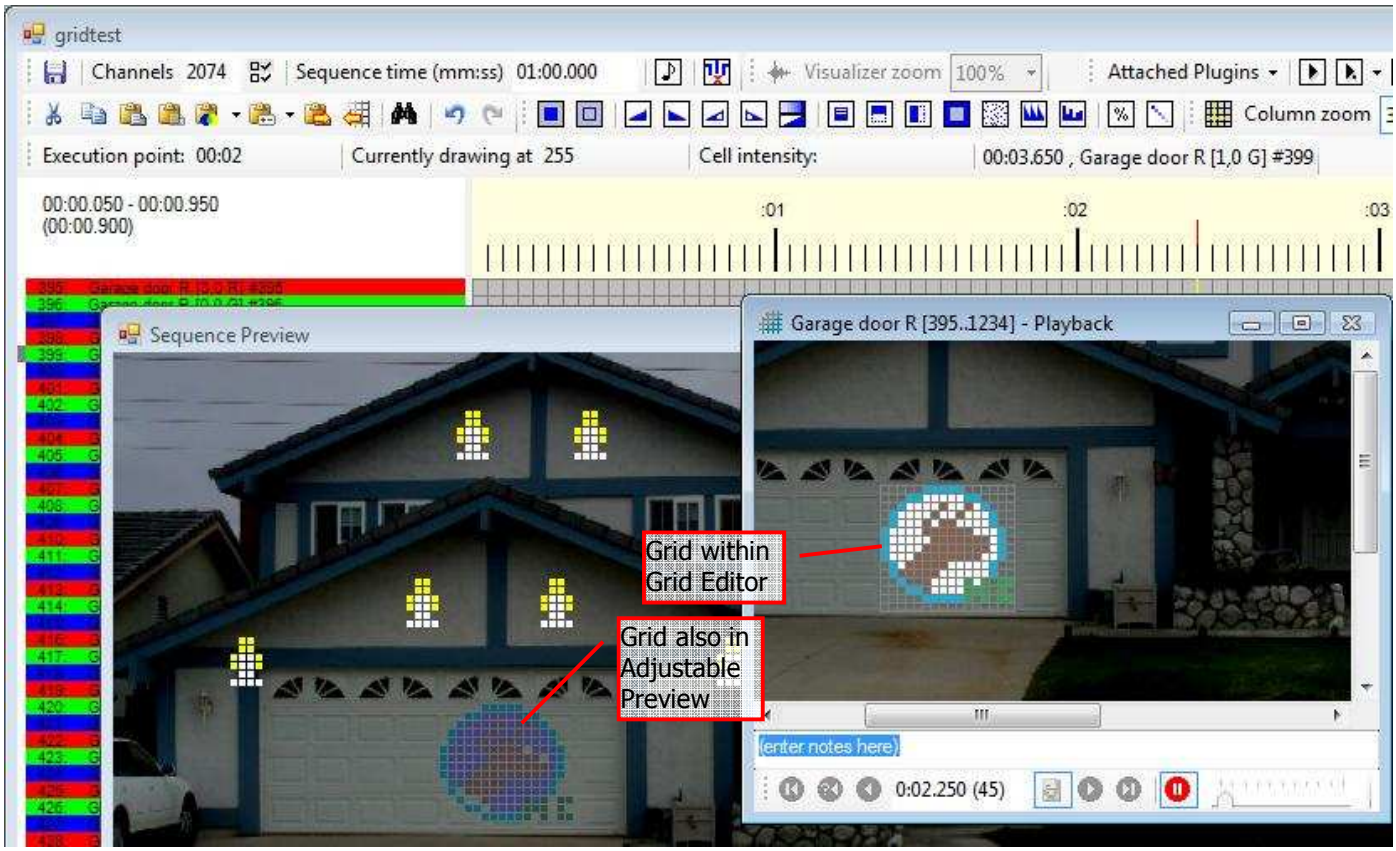
The Grid Editor plug-in should be disabled for playback of a real show because it's a CPU hog and slows down Vixen, even more so than the regular Adjustable Preview.

Playback of the sequence is initiated using the regular Vixen playback controls:



Sequence playback controls

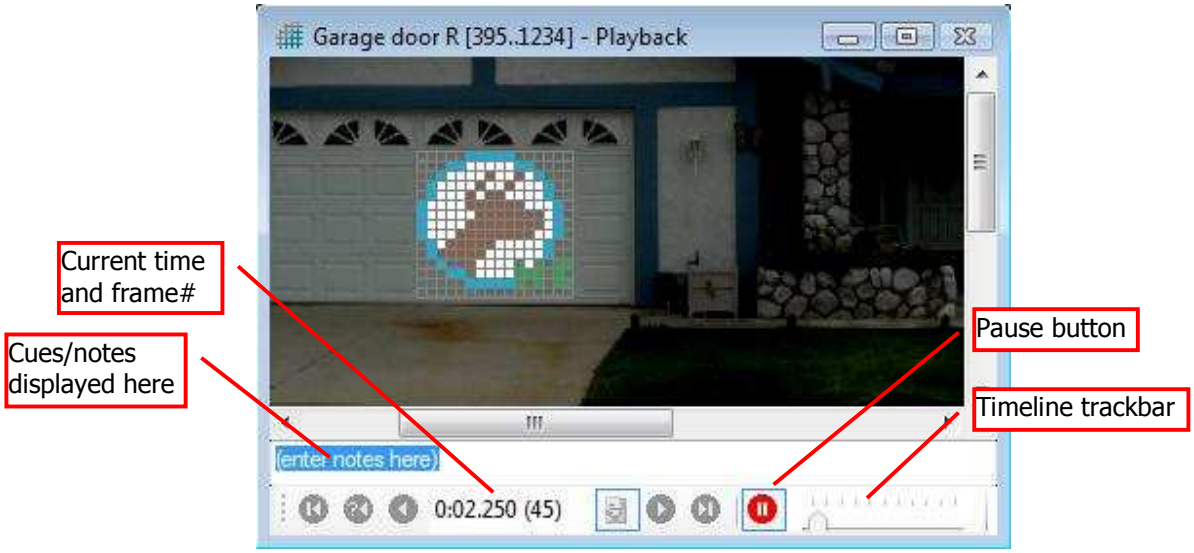
While Vixen is playing the sequence, active Grid Editor and Adjustable Preview window(s) will be visible. Each Grid Editor window remembers its last size, position and scroll settings, so they can be arranged on the screen according to what you want to see. I usually reduce the size of the Grid Editor to just show the grid, leaving more screen real estate available for the Adjustable Preview or other windows. When I also save the grid to the Adjustable Preview (as described in previous section), the grid will be visible within the first Adjustable Preview window:



Grid Editor and Adjustable Preview windows during playback

Even though the grid can be made visible within the Adjustable Preview window, the colors do not match. This is because the Adjustable Preview does not display the true RGB colors of the grid channels – it tries to blend channel colors with the background image. Since the main purpose of the Grid Editor is for RGB props, it just displays the actual RGB color of the grid pixels, without blending them with the background image.

During playback, the current time, frame number, and the latest note or cue text will be displayed at the bottom of the Grid Editor window. A trackbar also shows the relative position within the sequence:



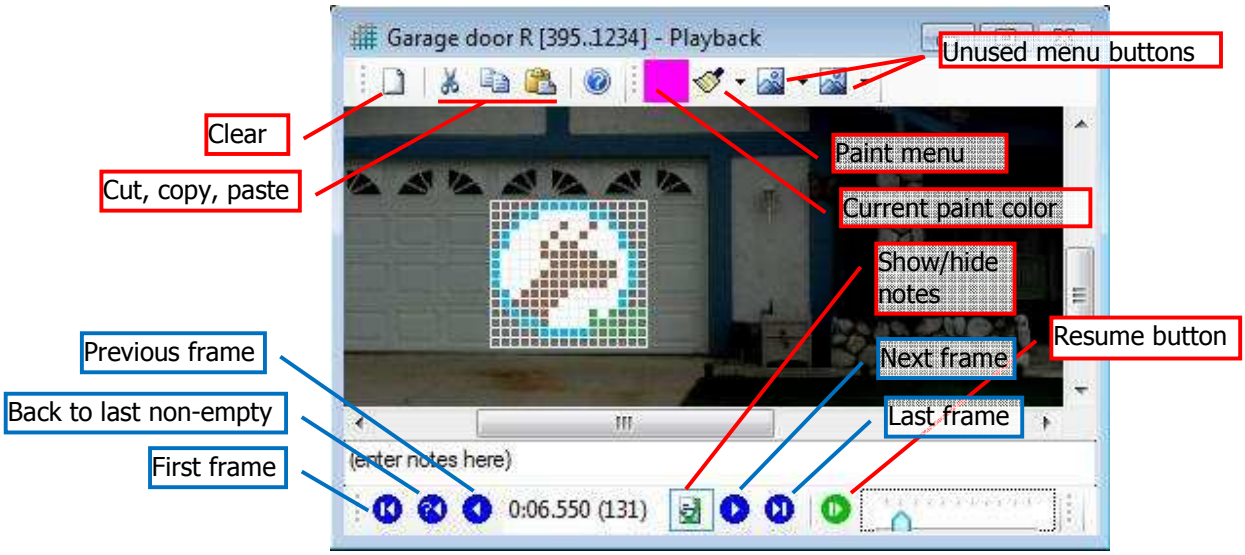
Grid Editor window in Playback mode

Since Vixen only displays the current Execution Point in seconds, the Grid Editor attempts to track the current time more precisely using an internal timer during playback. This can get out of sync with Vixen, however, so it really just serves as an approximation during playback (it also currently does not handle playback looping).

I typically resize or scroll the Grid Editor during playback, to allow only the part of the background image that is of interest to be displayed. However, it will not save the remembered size, position or scroll settings until playback is stopped in Vixen.

6.4.6. Sequence Editing

When playback gets to a part of the sequence that I want to edit, I just click the Pause button. This puts the Grid Editor in Edit mode, which changes the appearance of the window and enables various editing functions:



Grid Editor window when paused (Edit mode)

Frame Navigation

If I did not pause at quite the right place, I will use the Frame Navigation controls to move to the exact frame that I wish to edit. There are several ways to do this:

- Use the First, Previous, Next, or Last buttons to move to that frame. Advancing to a blank frame will automatically copy over the contents of the previous frame, making animation a little easier.
- If I completely overshot the part of the sequence that I have already edited, I'll use the Back to Last Non-Empty button to go back to the place where I left off editing last time.
- If I know the exact time or frame number that I wish to edit, I can just click in the Current Time and Frame box to type in either a frame number or time. The Frame Navigation controls treat a bare integer as a frame number, otherwise a time is assumed and punctuation can be used to differentiate between minutes, seconds and milliseconds.
- If I don't know the exact frame and I want to rapidly step through the sequence to find it, I will just drag the Timeline trackbar to the approximate place, and then click and hold the Previous or Next buttons to navigate frame-by-frame until I find it.

Once I have navigated to the desired frame, I can edit the grid cells. This will only assign colors to the grid cells (intensities of the individual channels for the cell); the shape or placement of the grid cells can only be changed using the Setup window, described earlier.

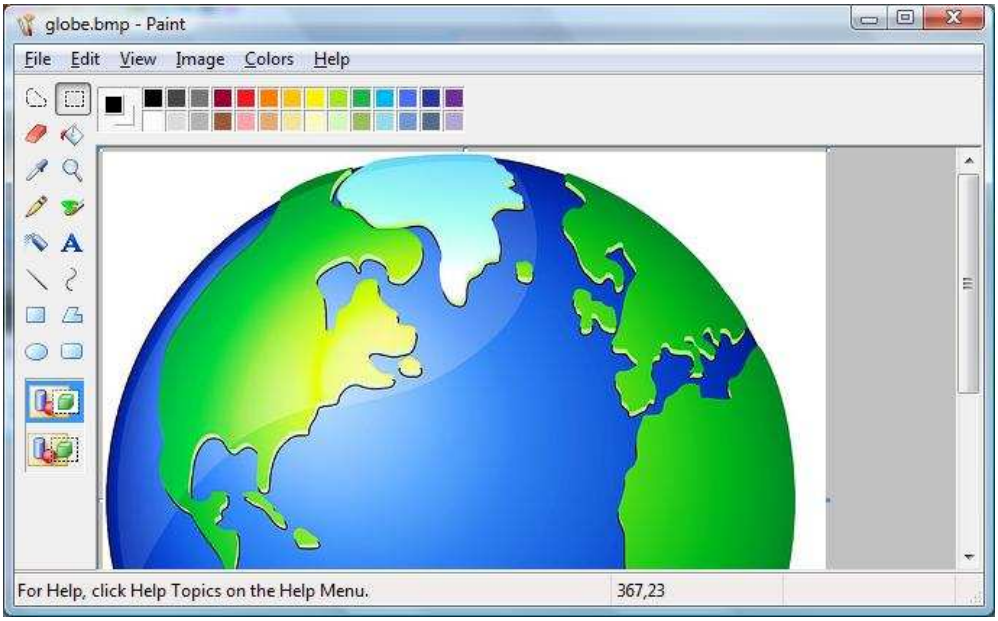
Currently, the Grid Editor has only limited editing capabilities. I will probably be adding more functions and drawing tools as I need them. The editing functions that it currently provides are:

- Clear the current frame; this sets all grid channels to 0, which is treated as black
- Cut, copy or paste the current frame to/from the Windows clipboard
- Set the color of individual grid cells

Cut/Copy/Paste Frame

The cut/copy/paste functions are useful for copying one frame to another, but I also use them to compensate for the current lack of drawing tools within the Grid Editor. For example, I can use Windows Paint or another utility to grab or draw a frame, copy it to the Windows clipboard, and then use the Grid Editor's Paste function to insert it into the sequence. Similarly, I can copy/paste a frame from the Grid Editor into Windows Paint, then edit it and copy/paste it back into the Grid Editor. Below is an example to illustrate this process.

Step 1. First, find some graphics and open or copy into a utility such as Windows Paint:



An image opened/copied to Windows Paint

Step 2. Resize and/or crop the image to fit the grid size, then set the background to black:

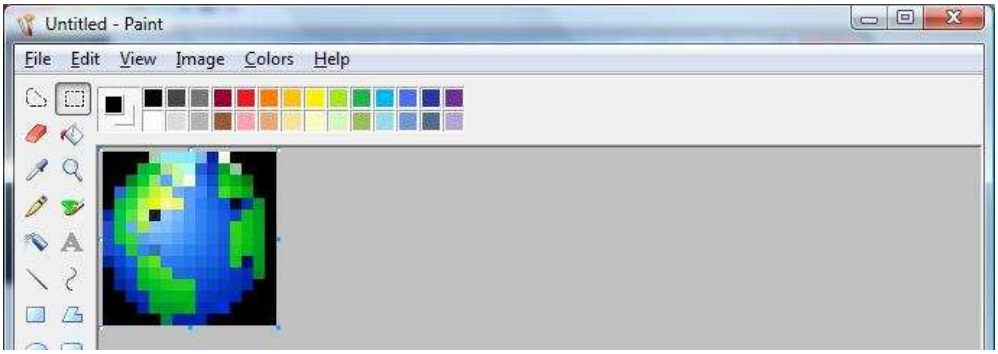
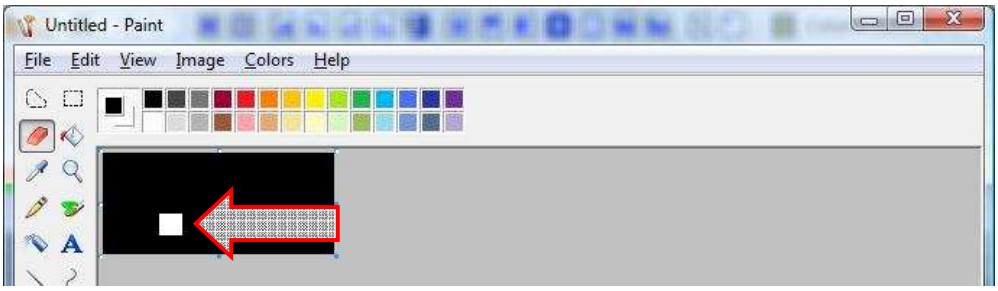


Image resized and cropped to the grid, zoomed in for detail/touch-up, then background removed

Each pixel in the image will become one grid pixel. For smaller grids like the one I used, this gives a very small image which is a little hard to see, so I use the Zoom function to display it larger if I need to see some of the detail (this does not affect the actual size or number of pixels in the image, though).

Then I can do any touch-up work, and cut out or paint the background to black. Otherwise, there will be extra pixels turned on in the grid later (black is all off).

Step 3. Copy a dummy frame from the Grid Editor and paste it into another copy of Windows Paint:



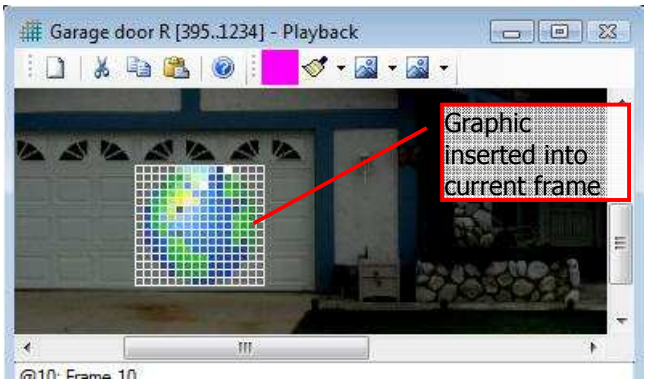
Dummy frame from Grid Editor shows grid placement within background image

The purpose of this step is to show the placement of the grid within the overall image - currently the Grid Editor uses the background image dimensions during Copy and Paste operations. Eventually I would like to implement automatic placement, but I haven't decided how to do edge detection since some of the grid pixels could be black (off). Maybe it doesn't need to do that if I add a Move function to allow the graphic to be dragged to the correct place within the background image.

Step 4. Copy and paste the graphic into the correct place within the image:

Graphic positioned correctly relative to background image dimensions

Step 5. Copy and paste the entire image back into the Grid Editor:



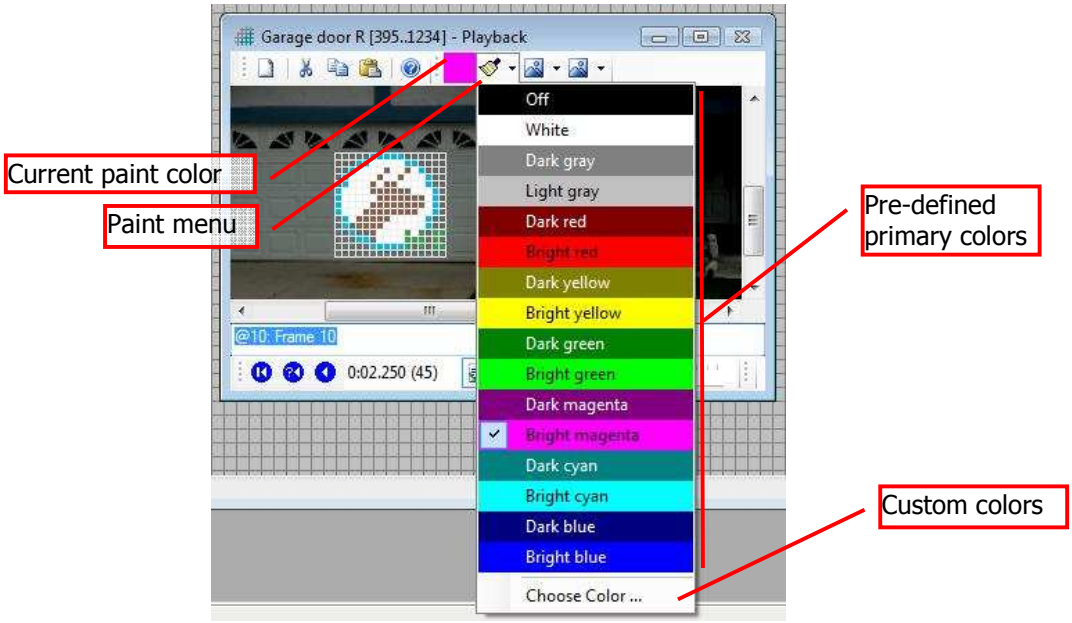
Graphic pasted into Grid Editor

The above process is a little awkward, but it wasn't too bad with the limited graphics that I used in my sequences. I will probably streamline this process in future. For example, combining the paste, resize/crop and graphic placement into a single Grid Editor function would be more convenient. If I have enough time, I may also set it up to grab frames at regular intervals from a video, then apply the resize and paste to

destination frames automatically. I have read that high-end software packages such as Madrix do this type of thing[23]. I think it would be fairly easy to add a simpler variation of that function to the Grid Editor.

Setting pixel colors

Now for setting the color of individual grid cells, the Grid Editor currently only has a freehand drawing function. This works similar to how cells are drawn in the Grid Editor Setup or in the Adjustable Preview. Left-clicking on a cell or dragging the mouse over it while the left button is pressed will set the cell to the currently selected Paint color, or using the right mouse button will turn it off again (set it to black). Cell-by-cell editing or simple line-drawing can be performed this way. First, choose the desired color from the Paint menu:



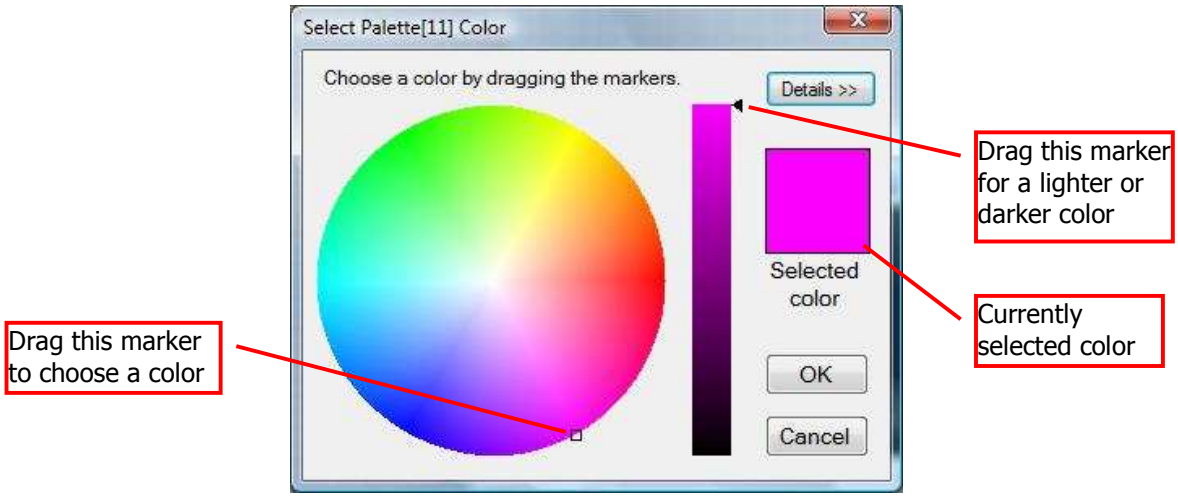
Paint menu and color palette

The Paint menu is sticky, with the last-used color remembered. Due to time constraints, the menu is currently just a drop-down list (eventually I'll probably replace it with a more compact tabular array of color buttons). The drop-down menu contains 16 pre-defined colors, which are initially set to the primary RGB colors (with 1, 2 or 3 of R/G/B set to half or full intensity), and clicking on one of these will set the Paint brush to that color.

The color menu items actually form a color palette, and can be set to new colors. Each has a checkbox which is turned on or off when the menu item is clicked. If the checkbox is on when a custom color is selected (procedure described below), then the custom color will replace the pre-defined color for that menu item.

If the desired color is not in the drop-down menu, a custom color can be selected by choosing the Choose Color menu item. This will open a color chooser window that allows an arbitrary color to be selected:

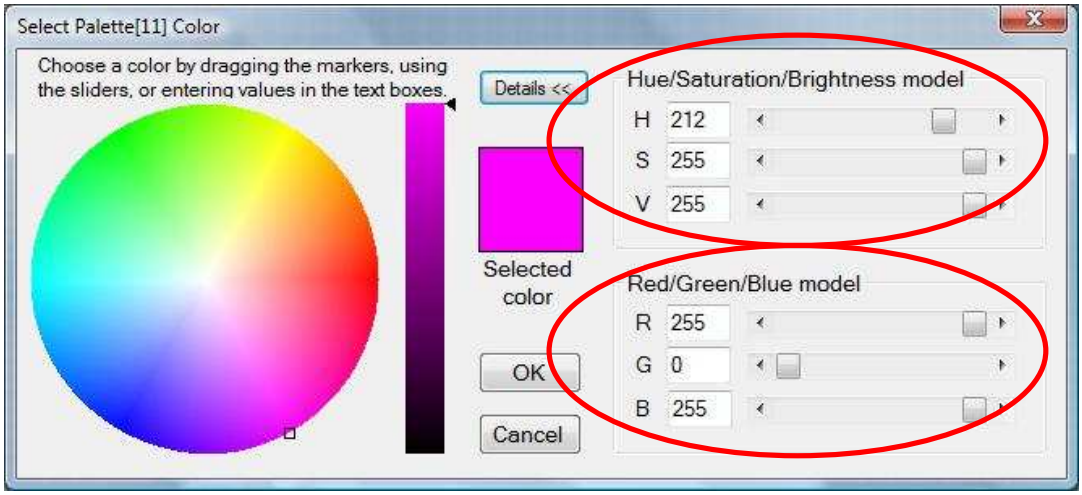
[23] http://doityourselfchristmas.com/forums/showthread.php?11468 Madrix, RGB pixel strings and the future of sequencing



Color chooser (simple mode)

To choose a color, drag the square marker to the desired color on the color wheel, or drag the triangle marker up or down to make the color darker or lighter. If a palette entry is active (its checkbox is on), then choosing a custom color will replace the pre-defined color for that menu item. Otherwise, only the Paint menu icon will be set to the custom color. In the example shown above, the custom color will replace entry #11 in the color palette – this is indicated in the title bar.

Clicking on the Details button will expand the color chooser window to show more selection controls:

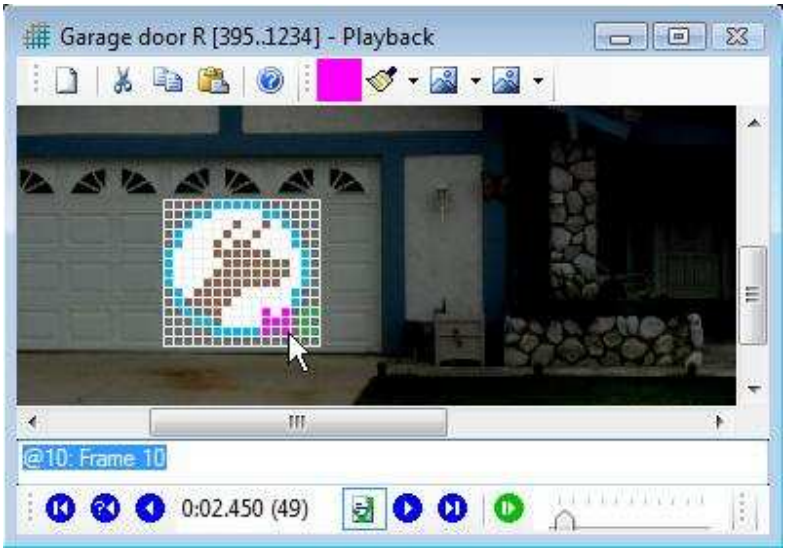


Color chooser (detailed mode)

An exact color can be selected by entering numeric values into the text boxes, or dragging the sliders to the desired position. Both the hue/saturation/brightness model and the red/green/blue model are supported. Entering or changing values in one model will adjust the equivalent settings for the other model.

Even though exact R/G/B values can be selected this way and they may look correct on the screen, the actual colors you get on the pixel hardware may not match due to the LEDs or the controller. For example, athough the Renard-HC handles 256 dimming level per channel (which would be 24-bit color depth), the LEDs that I used were non-linear in brightness (it was more like a logarithmic brightness curve, based on subjective observations). For this reason, I periodically check the actual appearance of RGB pixels in the grid to see if it's close to what I'm seeing on the screen, and then make further adjustments if necessary.

After the desired color of paint is selected, then clicking on cells or dragging the mouse over cells will set the grid cells to that color:



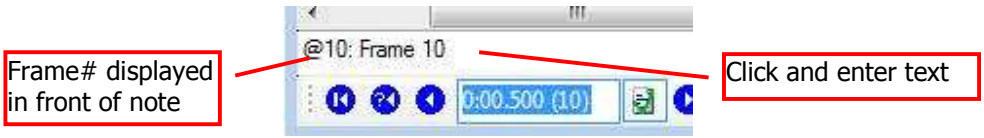
Changing grid cell color using Freehand tool

The cell changes are passed to the parent Vixen sequence window, but they are not actually written to the sequence file until Vixen's Save function is used.

Since each frame in the Grid Editor is equivalent to one column of cells in the sequence, I can also just go back into the sequence window in Vixen to manually edit the cells. This is useful for effects and functions that have not been incorporated into the Grid Editor yet (or if I encounter a bug), or for more fine-grained control of the sequence cells.

Notes/cues

I find it helpful to display text notes or cues during playback. To create a text note, click in the Notes box and type in some text:



Note editing

Although text notes are attached to an individual frame within the sequence, they will remain visible until the next note in the sequence. During playback, this prevents them from flashing on the screen and then disappearing before they can be read. This also means that you need to navigate to the frame containing a note before it can be edited or deleted. The frame number of the note is displayed in front of it, making it easier to find again for editing.

After editing, I click the Resume button to continue sequence playback from where it left off (frame navigation during editing does not affect the sequence playback position).

7. Future Plans

This past season's Renard-HC and dumb RGB pixel grid experiments had some limitations, but overall I feel that they were successful, so I would like to develop them further for next season.

The plan changes as I go along, but at the moment the projects that are attracting my interest are listed below. It seems like the limiting factor is always time. If you have a particular interest in any of these, let me know. Please note that I am just considering these, and not in any way making a commitment to actually do any of them.

Renard-HC Firmare

Lower-cost pre-built RGB pixels are starting to become available. At over $1/pixel, they are still too pricey for large-scale usage on a hobbyist budget, but they're getting closer. From recent DIYC forum threads, GE Color Effects and LPD6803- or WS2801-based pixels seem to be the best alternatives at the moment.

The GE strings seem to only be available around Christmas, which makes it difficult to experiment with them at other times of the year. The LPD6803-based pixels appear to have more consistent availability (although it sounds like there may be some US patent issues), so after you add shipping and then DMX and a proprietary decoder it looks like their effective cost is around $1.60/pixel.

Initial prototyping seems to indicate that a PIC16F688 could drive four 50 ct. strings of LPD6803 pixels (using SPI clock and data lines for each string), maybe as high as 7 strings (separate data lines, shared clock). The PIC does not have enough RAM to double-buffer that many pixels, but the LPD6803 pixels already appear to remember their data and don't flicker, so PIC RAM is not a problem. That would mean that a 5-PIC Renard-HC could drive 5 x 4 x 50 = 1K pixels (within the "good enough graphics" context described earlier). The cost of a stripped-down Renard-HC (no transistors or resistors) spread over this many pixels would make the per-pixel controller cost almost nothing ($0.03), bringing the LPD6803 effective cost back down to a competitive level.

Since 1K pixels looks like a fairly easy target, why not make it a little more interesting and set a target of 2K? At 115k baud and 50 msec refresh, it would only take a 12:1 compression ratio to achieve this (2K RGB pixels = 6K channels; at 12:1 compression this would only be 500 bytes, which is less than the 576 bytes available). I wanted to change the protocol to use a shared palette anyway, so maybe this will be the motivation.

I'd also like to rewrite the Renard-HC firmware in C. The novelty of using MPASM macros in ways they were not intended has worn off, and the existing source code is rather bulky. It doesn't look like free C compilers can do as good of a job at code optimization as hand-tweaked Assembler, but I may be familiar enough with the PIC instruction set now to be able to work around that. If not, then I might need to build a C compiler (I found some Open Source versions of Small C, Tiny C, etc that should be fairly easy to hack up).

Grid Editor Plug-in

Having read Andyb's very information thread on using Papagayo to lip-sync props[24], I think it would be very handy to be able to do this directly from within Vixen. In fact, the relationship of Papagayo voices, words, and phonemes matches another design pattern that I already wanted to add to the Grid Editor plug-in, so it looks like Papagayo could actually be implemented using the Grid Editor. This would provide an easy way to add new mouth shapes and tie them to Vixen channels. Since Papagayo is also a nice superset of the Cue Sheet functionality that I tried to add to Vixen a while back[25], this extension would be even more useful.

I also need to fill in the missing graphics and text tools in the Grid Editor, in order to avoid the need to jump out to Windows Paint. I'd also like to define an extensible way to add transition effects, similar to how other video editing packages support add-in effects. Maybe also frame-grabbing/sampling and mapping from videos to a pixel grid in Vixen.

DIY RGB String Variations

[24] http://www.doityourselfchristmas.com/forums/showthread.php?12249 Singing Pumpkin Lip-Sync andyb

[25] http://www.doityourselfchristmas.com/forums/showthread.php?7766 Vixen Cue Sheets

I'd like to go back and try to come up with a solderless, snap-in way to build DIY RGB strings. If this is possible, then building RGB strings would be less tedious, and this would make it feasible to build larger quantities for my other props.

There are some other variations of RGB pixel strings that I'd like to experiment with. For example, maybe icicle-style lights, or even put the 16 RGB LEDs into a clear flexible tube, similar to rope light or the "Cosmic ribbon" strings that I've read about. I also have another project where I'd like to use 7-segment displays – those fit well with the Renard-HC because each port could drive 8 digits (8 digits x 7 segments/digit, which matches 8 x 7 charlieplexing nicely) – I did a prototype of that last year.

At some point I might also experiment with different string lengths of charlieplexed RGB LEDs. Instead of groups of 16 LEDs using 8 wires, they could be grouped in 4s using 4 wires (4 x 3 = 12 = 4 RGB LEDs). This would increase the duty cycle from 1/8 to 1/4, allowing the LEDs to be brighter. In addition, an 8-pin PIC could be used to drive them (+5V, ground, serial in, serial out, and 4 pins to run 4 x 3 = 4 RGB LEDs), which would require a small PIC PCB to be inserted at every 4th position within a longer string of RGB LEDs, but allow a longer overall string to be built. Averaging the cost of an 8-pin PIC circuit across 4 RGB pixels would help to keep the overall pixel cost down.

AC SSR Variations

This past year, I started working on a "de-dimmer" that allows a dimmed channel to be converted back to a numeric value (8 on/off lines). This can be done, given an AC SSR signal and a ZC reference. Since the Renard-HC cannot do true PWM (because of the multiplexed rows), this would be a way to allow it to drive relays or other devices than cannot handle dimming. It would also make a handy test tool – I was thinking maybe attach a 7-segment display to it, and then be able to read the actual dimmer value that is present at an AC SSR (in order to help track down flicker problems).

I would also like to experiment more with Triac-less AC SSRs. For LED strings, incandescent C7s, or any other low-current device, the Triac doesn't seem to be needed and the opto can do the dimming directly[26]. I started working on a charlieplexed, Triac-less C7 string this past year, but didn't finish it.

I ran a few experiments to convert incandescent strings to run as half-wave, but I'd like to investigate this further. As long as the flicker is not too noticeable, SSR doubling seems like a convenient way to cut the cost of AC SSRs in half. This can add up to significant savings with higher channel counts. In theory, this is only a simple change to the Renard-HC firmware.

8. More Info

This chapter contains some misc. additional info about the RGB grid, Renard-HC, or other related background info.

This project is a work in progress, not a finished product, so any of its components are highly experimental in nature. I got it to work okay for what I was doing, but it has not been tested thoroughly under all possible conditions, and some features have not been tested at all. This should be taken into consideration if you will be trying out any of this stuff. However, please feel free to ask questions or for clarifications on the info in this article; I tried to cover too much info, and as a result probably made some mistakes along the way.

The source code is currently quite messy. Normally I would tidy it up before releasing it, but last time I did that took a while, so due to time constraints and a pending rework of major parts of it, I am releasing it as-is for now.

For questions, comments, or suggestions for improvement, please email to techguy@eShepherdsOfLight.com or just post in the DIYC forum if it's info that might be of interest to other DIYC forum members.

[26] http://doityourselfchristmas.com/forums/showthread.php?6287 Can the triac be skipped for light loads?

Release Package Files and License

In keeping with the DIYC philosophy of knowledge sharing, I am posting the following supporting files for this project:

- Howidid-Dumb RGB Pixel Grid.pdf = this article
- Renard-HC-vixen.zip = Grid Editor and Renard-HC plug-ins for Vixen 2.1 and 2.5
- Renard-HC-hex.zip = Renard-HC firmware (currently only for the PIC16F688; maybe others later)
- Renard-HC-pcb.zip = ExpressPCB schematics and PCB layouts (Gerbers tbd, if there is interest)
- Renard-HC-src.zip = Source code for Vixen plug-ins and Renard-HC firmware

These files are being released on a non-commercial, "share alike" basis. That is, you are allowed to use them for personal, non-commercial purposes, but if you make any bug fixes, changes, extensions, or adaptations, I ask that you share those with the DIYC community.

More precisely,

What about Patents?

I did not find anything that looked like a pending patent for "chipiplexing", but if there turns out to be one at some point, then a minor modification to the Renard-HC circuit would allow anything attached to it to continue to work correctly. Charlieplexing itself seems to be public knowledge since it is published by Microchip and others, so it doesn't look like there are any issues there.

I think any other info used in this project is either public knowledge, prior art, DIYC, open source, or original work, and, as such, there shouldn't be any other patent issues.

Helpful Additional/Background Info

There is too much to list it all, but here are a few selected references to helpful or related background information:

Wikipedia links to LEDs:

- http://en.wikipedia.org/wiki/LED_circuit
- http://en.wikipedia.org/wiki/Light-emitting_diode

Description of charlieplexing:

- http://ww1.microchip.com/downloads/en/DeviceDoc/chapter%201.pdf
- http://en.wikipedia.org/wiki/Charlieplexing

EDN article on chipiplexing:

- http://www.edn.com/contents/images/6615603.pdf

Microchip PIC16F688 datasheet:

- http://ww1.microchip.com/downloads/en/devicedoc/41203d.pdf

Links to a commercial DMX dongle, proprietary LPD6803 decoder, and other related items:

- http://forums.auschristmaslighting.com/index.php/topic,120.0.html

Replacement chart for common transistors:

- http://www.elexp.com/t_trans.htm

LED color chart:

- http://www.theledlight.com/color_chart.html

PLCC6 5050 LED datasheet:
- http://doityourselfchristmas.com/forums/attachment.php?attachmentid=6395&d=1269302736

LTSpice download link and Yahoo group:
- http://www.linear.com/designtools/software/ltspice.jsp
- http://tech.groups.yahoo.com/group/LTspice/

Why I did this

This project, and even this write-up itself, was a big investment of time and effort, so you may wonder why I did it. Well, there are 3 reasons:

- Christmas is the first half of a love story (Easter being the second half), when God, who created us, also reached out to save us because we could not save ourselves. I like to try to come up with new or creative ways to use Christmas lights to celebrate this.
- Hoping to help others do the same, I am trying to pass along any info that I stumble upon that might be useful.
- Tinkering with electronics and software is rather fascinating. They are really 2 aspects of the same thing, just like Christmas and Easter are. It helps me to gain a greater appreciation for just how intricate, how unimaginably complex, and yet elegantly simple our Creator made us and the world around us. If you think that random particles can just come together and do something useful like that by themselves, without a Grand Designer orchestrating the whole process, please tell me how - this would save me a lot of trial and error for my feeble little projects. ☺

Revision History

Revision	Date	Description
1.0	02/21/11	Initial version released
1.0a	02/26/11	Misc corrections (minor typos, link problems, wrong date in footer)

(eof, finally)