The Aurora LED wall: a 4 foot by 8 foot iPad-controlled, music-responsive canvas of color

This project began as a need to fill an empty 6’ by 12’ wall in my living room above the couch. Since I have an attraction to flashing things, I thought, “what better to fill that space than lots of lights making cool colorful patterns?” Thus the idea for the Aurora LED wall was born. This is my largest and most involved project to-date (involving soldering, woodworking, and microcontroller as well as PC programming), so I wasn’t sure how it would turn out. But it ended up coming out much better than I expected. If you like the vids/pics/description here or have some thoughts on improvements to the wall, please add some comments below!

The lights in the Aurora consist of a grid of 544 RGB LEDs capable of ~16 million colors attached to a 4’ by ~8’ wood board. The LEDs are controlled by a microcontroller, which is in turn controlled by a program running on a laptop. The computer is the brain of the system, being responsible for creating the graphics that appear on the wall, and the microcontroller relays the messages to the LEDs. The PC software connects to software running on an iPad, which allows interaction with the moving patterns. Also, the Aurora has a mode where the builtin programs respond to music, bouncing and flashing with the beat, turning it into a VJ of sorts.

There are several programs implemented so far. One is the Belousov–Zhabotinsky reaction, implemented as a 2D cellular automata (kind of like Conway’s game of life but less pixelated and more colorful). The idea for the graphics come from a cycle of chemical reactions that oscillate as some chemicals are used up and others are created. The resulting patterns look like smooth waves of color that sometimes repeat in a loop and other times diverge. With the iPad, you can perturb the automata with your finger and see what happens, control the speed of the visuals, and sharpen the edges of the pattern. It’s worth checking out the Wikipedia page and the video above as it makes some awesome patterns. Another visualization uses Perlin noise to create constantly changing flowing blobs of color. Perlin noise is a type of computer-generated noise that injects randomness with a harmonic structure so that the smoothness or roughness of the blobs can be controlled. Interaction via the iPad lets you control the speed of the blobs, their size and their complexity. Yet another program displays a flickering fire, kind of like the old TV channels, except almost 8 feet wide. It allows you to customize the color scheme so the flames can range from the standard yellow, orange and red to something like flickering pink with a green background. There are other programs as well (such as a mode where you can paint with your finger) and the code is structured so that it’s easy to create new ones that take advantage of the color palette features described below, the iPad control and the music responsiveness.

The color palette used for the graphics is highly configurable, allowing the different visualizations to change mood, so to speak. Using the iPad, you can choose from popular color schemes found on Adobe Kuler (an online repository of color palettes), or use a random starting color with a number of different color theory strategies (e.g. complementary, tetrad, analogous, etc.) I didn’t know anything about color theory before this project but it turns out you can intelligently find colors that go well together. This is especially useful since I’m not so great at choosing colors myself. If you look at the video, you can see the different types of color schemes available.

One of the components of the wall that both added complexity and improved the aesthetics was diffusion of the LEDs with translucent plastic. This made the wall take on a more organic form. In my opinion and based on comments I’ve received from past projects, bare LEDs look very digital, in part because they create discrete points of light. Diffusing them results in one continuous canvas instead of a grid of little bright point light sources. If you’re curious, you can see some pics below of what the LEDs look like without the diffusion [PIC]. To do the diffusion, I attached some prismatic acrylic lighting panels (the ones found in front of fluorescent ceiling lights) to the board about 5-6” away from the LEDs. I figured this distance out empirically: too close and not much diffusion occurred; too far and the pixels melded together. It was tricky (at least for me with my lack of carpentry skills) to keep the flexible plastic at this distance; you can see the build details below if you’re curious how it ended up being working out.

One difficult design decision I encountered was whether to use a microcontroller (e.g. an Arduino) to generate the graphics or a PC. I originally leaned towards the microcontroller route because I wanted the Aurora wall to be of a self-contained unit that didn’t require booting up and plugging in my laptop every time I wanted to use it. The flipside is that microcontrollers are obviously not as powerful as PCs, so the visualizations wouldn’t look as good. Honestly, they’re also just harder to program. I ended up deciding on the PC route after I realized that I could replace my laptop with a Raspberry Pi (assuming I can get one!).

Anyway, hope you enjoy the Aurora LED wall! Be sure to leave some comments about your impressions below.

Technical details:

-Built with 544 addressable RGB LEDs with WS2801 ICs (8 bit x 3 channels allowing ~16 million colors) in a 32x17 matrix.

-Uses a 350W power supply and can consume up to 60mA \* 544 = 32 Amps at 5V

-The LEDs are driven by a ChipKit Uno32 microcontroller running at 80Mhz that communicates with a custom Processing sketch over serial at 921600 bps allowing a ~45 fps refresh rate

-The Processing sketch communicates via OSC (Open Sound Control) to an iPad app called TouchOSC with a custom control panel layout [img]

-The audio responsiveness uses fast fourier transforms (FFT) on line in audio to detect beats on low, middle and high frequency bands

-Different color theory strategies for building the color palette are supported such as complementary, tetrad, compound, and others in addition to downloading colors from Adobe Kuler

-Uses a gamma table with gamma of 2.5 to adjust the LED brightness

-Built with a 4’ x 8’ piece of ¾” plywood (solid!) and 2x3 beams to form the frame

-The diffuser is made of 4 2’x4’ clear prismatic acrylic lighting panels distanced ~6” from the LEDs

Credits:

-toxiclibs colorUtils and colorLibs for Processing

-TouchOSC & oscP5 for Processing

-Simon Dixon for the concept for part of the beat detection code from "ONSET DETECTION REVISITED" Proc. of the 9th Int. Conference on Digital Audio Effects (DAFx-06), Montreal, Canada, September 18-20, 2006

-Ryan Govostes and Jim Bumgardner for the code from which the Perlin noise program was adapted

-Alasdair Turner for code from which the Belousov–Zhabotinsky reaction program was adapted

-Mick Hornbeck for help with the woodworking design

Below is a somewhat detailed description of how the Aurora LED wall was built. There are two many steps to go through all the background details of each part of the build, so I’ve by necessity kept the description fairly high level. Thus it’s targeted more at the intermediate to advanced maker.

Step 1: The grid

The core of the Aurora LED Wall is a 32 (width) by 17 (height) array of RGB LEDs (544). The LEDs I used are pixel strand LED modules and each is controlled by a WS2801 integrated circuit (IC). Each pixel is really 3 LEDs (red, green and blue), which allows it to form 256x256x256 colors, which is a lot! So virtually any color, although there are some limitations when you take the gamma adjustment into account. The pixels are attached [pic] in series and can be individually addressed such that each can have any of the possible colors at any given frame. The way this happens is beyond the scope of this description unfortunately, but it has been described elsewhere [link].

These pixels [pic] are 12mm in diameter so to form the array I drew a grid on a 48” by 80” piece of plywood with about 2.25” separation between pixels with some room to spare on the edges. This turned out to be a pain because I’m not all that handy, but I worked it out with the help of a 48” aluminum right angle ruler.

Step 2: Drilling the holes

I chose ¾” thick plywood to hold the LEDs because I wanted the final product to be solid and to last. Also I figured it would be less likely to warp, which I discovered is probably true in comparison to thinner plywood options, but is not completely so. It did end up warping a bit so I bought some 2x3 beams to form a frame for it, which I attached to the plywood with 5/16” hex bolts. With the frame, it won’t warp but that also obviated the need for the ¾” thick plywood; but by then I was committed. Probably ¼” plywood would do and would be much lighter.

To drill the holes, I used an AC powered drill with a XXX drill bit, which is worth spending money on if you are drilling 544 holes. This took a while but not as long as you might think. It does make a mess though so don’t do it in your living room like I did! [pic] I also ended up sanding all the holes with sandpaper to remove all the splinters of wood created by the drilling.

Step 3: Soldering the LED pixel strands together

The pixel strands came in lengths of 50 in each so there was a fair amount of soldering to put them all in one giant strand of 544. At each junction, I stripped the 4 wires on either side (making sure the connect IN to OUT) and soldered the same colored wires together. This was a fairly long and somewhat frustrating process if I remember correctly, but you get better at it. A trick I use for soldering loose wires together is to tape them with magic tape to a surface (such as a desk) that you don’t mind soldering iron burns on. Tape them to the surface close enough so that they touch, so that you’re removing one dimension of space in which they can escape (much easier than using a third hand IMO).

One important aspect of LED strands is that they require power injection, by which I mean you need to attach 5V/GND every so often along the strand. If you don’t do this, the voltage will keep dropping as you move along what you think is the 5V wire. It will eventually (rather soon actually) reach a point where it can no longer supply enough voltage to the ICs to function properly and they will not do what you want. For this reason, you are recommended to inject power every 50-75 pixels or so. For this project, I inject power every 64 pixels because my rows are 32 pixels wide and then I can keep the power supply connections all on the same side of the wall. Thus, I added inputs for 5V and ground every 64 pixels.

The power supply I used [pic] was 350W (better to err on the higher side) and is rather large at about 2”x6”x10”. It’s nothing special, but does happen to have a fan that switches on automatically at a certain temperature. This is somewhat annoying actually because the fan is loud, but it doesn’t come on often. I plan to replace it with something quieter.

Step 4: Putting in the LEDs

This was pretty straightforward, but was harder than I expected. I chose a drill bit that was slightly smaller than 12mm [check] because I couldn’t find a metric one and I wanted the LEDs to fit snuggly. Some of the holes turned out to be smaller than others and a number were downright painful to push the LEDs through. I put the LED pixel strand in with the last pixel in the bottom right corner (facing the LEDs) where the power supply and microcontroller were going to go. I placed them in rows (as opposed to columns) for reasons explained above, so that the first LED (i.e. the last one to receive the color bits) was in the upper left corner. The power injection points were also on the right.

Step 5: Attaching the standoffs and the diffusing panels

The standoffs hold the diffusing panels 5-6” away from the wall. My original plan was to use 6” long 5/16” bolts with a bunch of nuts and washers to hold the acrylic panels (which are pretty light) the right distance away. On the suggestion of my friend Mick, I added thin conduit pipe between the acrylic panel and the plywood holding the LEDs. [pic] This allows a nut and washer on the back of plywood to tighten the whole system. Thus from back of the wall to the front, the order of the components is:

nut -> washer -> plywood -> acrylic panel

Then the bolt slides into a hole drilled in the acrylic panel from the front and is tightened by a nut in the back. Since the acrylic panels were quite flexible, I used 6 standoffs per panel [pic] for a total of 24 standoffs (since there are 4 24”x48” panels). This design ended up being quite sturdy, especially with the conduit. The bolt heads do extrude in the front in the center of the LED display but these turned out to be inconspicuous in comparison to the LED patterns.

Holes need to be drilled into the plywood in strategic places that match up with the corners of the acrylic panels and matching holes need to be drilled into the panels themselves. It was a bit tricky to get these to line up correctly and I ended up taping the acrylic panels to the plywood so I could align the holes in the right places. Drilling the acrylic was a pain at first since the stuff is prone to cracking. Eventually I figured out that if you put pressure at all points around the hole you’re drilling, you can avoid creating cracks. The trick to doing this is to take two blocks of wood, drill a hole in the first one, then put the panel in between the blocks and line up the hole in the wood with the hole you want to make in the acrylic. Then drill away! Once you’ve made all the holes, attaching the panels is pretty easy. There may be some small gaps between the panels, but you can remove those by shortening the gaps between the standoffs with zip ties.

Voila! If you’re following along at home you’re now done with the hardware and have a fully built LED wall! On to the software…

Step 6: Communicating with the LED wall

As I mentioned above, I was hesitant at first to use a computer to run the LED Wall because it just makes it more cumbersome to use. That is because it needs to be setup and connected every time. The advantages of using a PC though are significant; it makes the programming much easier and the algorithms involved can be much more complex. Although it is very difficult to get one at the moment, Raspberry Pis [link] will eventually become acquirable and could serve as a dedicated host. (One potential issue is whether sound will work in Processing running on ARM; anyone know anything about this?)

With the PC in the mix, the flow of information is from a Processing sketch to the microcontroller to the LEDs. The Processing sketch does the lion’s share of the CPU work involving deciding at each frame what the pattern is and assigning colors to each LED. The sketch sends this information in the form of a simple packet to the microcontroller. I considered using an Arduino at first but ended up opting for a ChipKit Uno32 since it runs at 80MHz and allows much faster serial communication with the PC (921600 bps vs 115200 for the Arduino). The microcontroller then passes the color of each LED in turn using the protocol specified by the WS2801 with the ChipKit DSPI library. This allows about a 45 fps refresh rate, which should be flickerless.

Step 7: Programming the graphics sketches

This part of the project was kind of nice because the programming could all be done in Processing, which is pretty straightforward. I found some inspiration from sketches on Processing.org and OpenProcessing.org and both significantly adapted some and created others to make the visualizations you see in the videos. There are a number of tabs in the sketch:

-ledWall (the main loop and the iPad communication over OSC)

-BeatDetect (the code to allow responsiveness to music)

-CircularArray (a utility class for circular arrays used by the beat detection)

-Console (a class to display console output in a window when running as a standalone executable)

-Drawer (a base class for providing common utilities to each visualization program including doing gamma correction, accessing the current color palette, the current beat information and settings from the iPad)

-PaletteManager (a class to create and manage color palettes making it easy to shape the color scheme)

-Pixels (class to draw to the screen and/or the LED wall if it’s connected)

-SerialPacketWriter (utility class for creating packets to send to the microcontroller)

I’ve implemented a number of programs but X of them worked out well enough to deserve talking about. Since these inherit from the base Drawer class, they have access to the different parameters controlled by the iPad including the color palette to use, animation speed, color cycling speed (how fast to cycle through the color palette), custom settings for each animation, brightness, and various audio settings to make them beat sensitive:

-AlienBlob: uses Perlin noise to create smooth blobs of color that bounce around; different settings (controllable from the iPad) allow zooming in and out to change the size of the blobs as well as increasing the harmonics of the Perlin noise to alter the complexity of the blobs.

-BouncingBalls2D: a customizable number of balls of different sizes bouncing around a box in 2-dimensions including gravity and using physics to calculate fully elastic collisions.

-Paint: use your finger(s) to paint color-changing pictures on the wall from the iPad or just create ribbons of color that slowly (or quickly) disappear

-Fire: a flickering fire covering the wall with standard fire colors or a custom palette giving unreal effects.

-Belousov–Zhabotinsky reaction: a 2-dimensional cellular automota implementation of the cycling chemical phenomenon called the BZ reaction (adapted from [www.aac.bartlett.ucl.ac.uk/processing/samples/bzr.pdf](http://www.aac.bartlett.ucl.ac.uk/processing/samples/bzr.pdf)); this program displays rippling arc-like patterns that form in a somewhat chaotic way and tend to settle into repeating patterns that can be disturbed with finger touches to the iPad and with gradients that can be “sharpened” to increase contrast

Step 8: Communicating with the iPad

The communication with the iPad is a key feature because it enables a ton of interactivity, which I really wanted for this project. I started out thinking of using sensors, but there were none that were really appropriate except for something like the Kinect (perhaps for the next project…). The iPad communicates with Processing over the Open Sound Control (OSC) protocol, with an existing iPad app called TouchOSC ($5 on the app store); I’m sure there is something similar out there for all you Android users. This app allows the creation of custom layouts of buttons, sliders, multitouch inputs, etc and saved me from having to write my own app, so hats of to those developers. Once I decided on a layout, the implementation was pretty straightforward since there’s a nice Processing library for OSC (called oscP5) and the protocol is pretty simple. Most of the trickiness is finding what parameters in the visualization programs have a significant visual effect and deserve a role on the iPad interface; i.e. the fun part. :)

Step 9: Using the color palettes

Using flexible color palettes was something I thought of early in the project and I’m glad I did. Changing palettes, along with the other iPad custom settings) allows the same program to take on many different manifestations and represent different moods, making the wall much more entertaining. I used two Processing libraries (toxiclibs.colorUtils and colorLib) to implement different ways to create palettes with colors that match. colorLib [link] provides an interface for downloading sets of colors that were most highly rated on Adobe Kuler [link], which is a repository for people’s favorite color schemes. Toxiclibs.colorUtils [link] implements different color theory strategies for finding matching colors from an randomly selected starting color; these strategies include complementary, tetrad, triad, analogous, monochrome and others. I won’t get into the details of how those work but you can read up on them at the above link if you’re curious. Since there is a common base class for all the programs, it was easy to implement access to the current color palette by the programs and equally easy to change the color palette without having to reimplement this feature for each program.

Step 10: Programming the beat detection

After showing the LED wall to some friends, I heard some feedback that I expected: “You should make it music-responsive like your last project” (which will hopefully be posted here in the not too distant future). I could not resist the challenge. There are different ways to make visualizations music responsive, which include: analyzing the time-domain signal corresponding to the change in pressure representing the sound, or analyzing the frequency-domain signal resulting from doing a Fourier transform; following the amplitude of the sound, or looking specifically for sharp transitions representing beats. There are others, of course, but those are a few. For this project, I was keen on analyzing beats in different frequency ranges (i.e. bass, midrange, and treble) because I thought it would translate best in a synesthetic way to visual patterns (this was from spending a lot of time staring at iTunes visualizations ☺).

Converting the audio signal from the time signal to the frequency signal (i.e. the amplitudes at each frequency) is straightforward with the Processing minim.FFT package. The second step in the approach I used to detect beats is to multiply adjacent frequency intensities together for mid to high-range frequencies. Since there are lots of relatively small post-FFT frequency bands at higher frequencies (more than we need really), I decided to take advantage of the fact that these neighbors should have similar changes if the changes are representative of a real beat. Thus, I multiply the intensities of between 5 and 7 neighboring channels together so that the result will be highest where the neighboring bands agree and will be low when they don’t agree. Thus noise is reduced. Note, that I don’t do this on the low frequency bands because the frequency of those bands is on the same order as the difference between neighboring frequencies.

The third step is to apply a metric called spectral flux that comes from a paper by Simon Dixon ("ONSET DETECTION REVISITED" Proc. of the 9th Int. Conference on Digital Audio Effects (DAFx-06), Montreal, Canada, September 18-20, 2006. <http://www.dafx.ca/proceedings/papers/p_133.pdf>). Briefly, the idea is to take the difference in the intensity at a given timepoint of a given frequency band from the previous band, kind of like an instantaneous difference. The value of the spectral flux will be high when there is sharp change upward so by looking for peaks of this metric, we can find the onset of beats in different frequency bands.

The fourth step is to smooth this still quite noisy signal (especially for bass frequencies) with an exponential moving average (but not too much or you’ll lose your signal!). The spectral flux value is then compared to an adaptive threshold to see if a beat has occurred. This threshold is defined as a configurable number of standard deviations from recent spectral flux values above the mean from an exponential moving average. If we assume a normal distribution, the threshold at three standard deviations above the mean will be exceeded by a random process ~0.1% of the time, so even higher values of this will filter out many of the potential false beats.

That’s it for my short treatise describing the Aurora LED wall; you’ve made it to the end! Be sure to add a comment if you have any questions or if this description ended up being useful to you for your project.