Tyler Altenhofen
Martin Deng

3D Persistence of Vision Display

Microcontrollers and Embedded Systems

University of Pennsylvania
ESE 350 Final Project
5/14/2015

Introduction:

Persistence of Vision (POV) is the property of an afterimage appearing to persist for a small amount of time after the image is removed. This is reason that you do not notice the black spaces between movie frames. Even though there is a small amount of time where there is no image, your eye and brain process the series of images as one continuous view. A traditional POV display flashes lights at precise times, while scanning across a 2D field to use the property of the persisting afterimage to create the illusion of 2D image. Below in figure 1 you can see an array of LEDs attached to a bicycle wheel and then in Figure 2 you can see those light flashing at exact intervals to create an image. Another property of this extremely fast scanning across a field is that the arm that does the scanning can be made so thin, and move so fast that the less bright arms appear to be completely invisible. You can observe this property in Figure 2 where the spokes and LEDs are invisible because of their high speed.

The idea that stemmed from this was that instead of having the LEDs scan across a 2D field, we could have a series of LEDs scan across a 3D volume. Because each arm becomes invisible at high speeds, we can precisely time the lights to generate a floating 3D image. In figure 3 below you can see the design we implemented to scan across a 3D area.

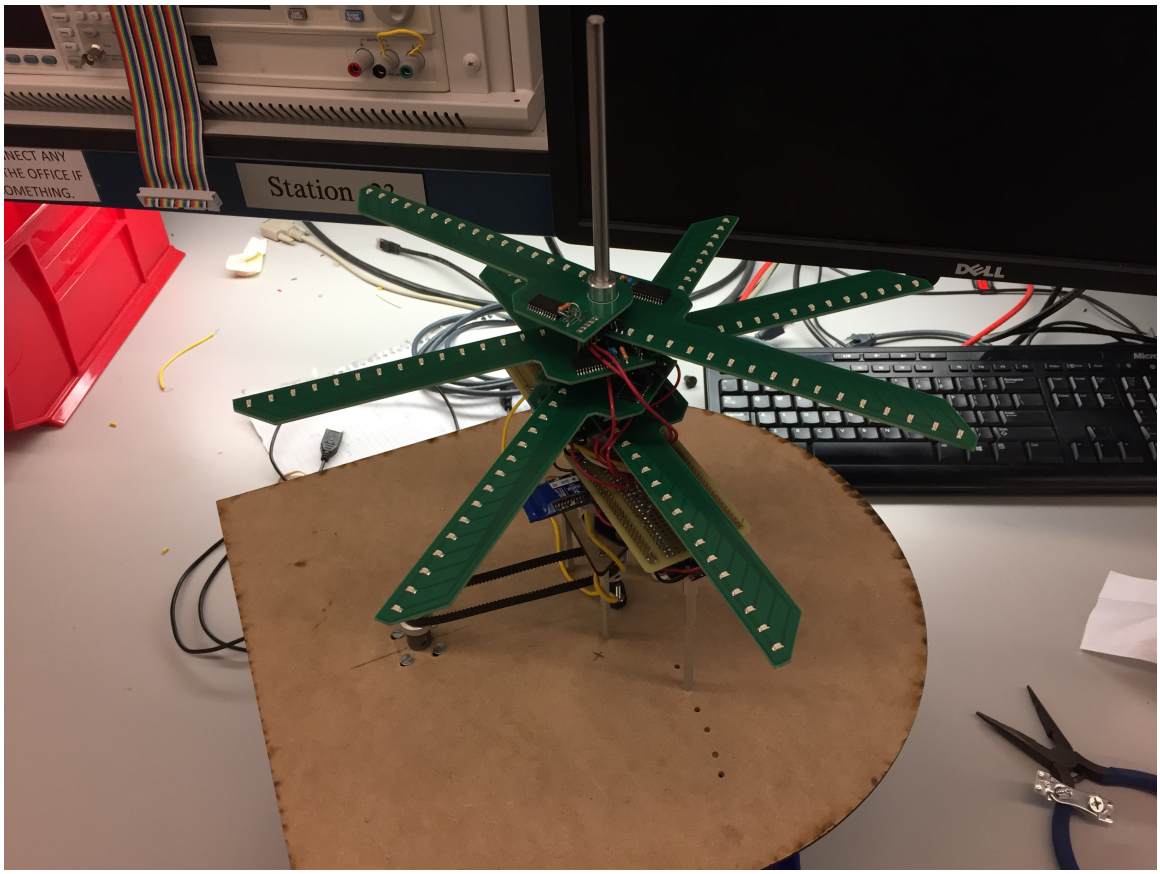
The requirements for our proposed project were that we create a closed loop display capable of creating interesting patters while interacting with the user. Our reach goals

were to implement the ability to traverse a 3D topological map of a pre-generated area. This would function like Google maps but in 3D.

Figure 1 (LEDs spinning platform)

Figure 2 (Spinning LEDs)

Figure 3 (our 3D POV Display)



Mechanical Design

The first challenge was designing a rig capable of moving multiple LEDs across of volume. I went through several different ideas on how to do this most efficiently. We could have a single arm and change its angle relative to an axis. We could have many vertical arms all spinning around an axis, and have each one be closer to point of rotation than the one before it.

I decided on using the design seen above in figure 3 for a few reasons. The outward arms allow us to balance the spinning so that the center of mass is right above the point of rotation. This would minimize shaking resulting from extremely fast rotation. The fact that all of the arms are very far apart from each other also means that when looking at it from an angle there will be very few times when an arm blocks the view of a

lower arm. This design also allowed us to keep all of the heavy and complex components close the shaft and as low as possible. This means that there is a smaller moment of inertia, and that most wobble from being unbalanced will be close the base where it will be stabilized. As you can see, we were able to extend the LEDs out significantly farther than the onboard chips, and MBED, which remain tight up against the shaft. Final this design allowed us a whole lot adaptability in our design. We were able to get a single arm working before implement the rest. We were also able to add extra electrical components down at the bottom when needed without drastically changing the center of gravity or the stability. This came in very useful when the ZigBee was acting up and we implemented Bluetooth control protocol.

Electrical Design:

To get power onto the shaft we implemented a brush system. There are two contacts made from simple brass bushings on the shaft, which then make contact with stationary electrical brushes similar to the ones you can find in DC motors. Because we have an outside power source we don't have to worry about draining a battery or holding all of that extra weight, but we do now have to deal with the changes in voltage that result from brushes making contract with fast moving surfaces. The speed at witch we were rotating the shaft meant that there were the occasional loss of power at very high speeds. After some tweaking we got the number of times power was lost to be very low, but there was still room for improvement. So solve this we implemented a voltage regulator, which uses a couple of large capacitors to maintain and smooth the voltage through bumps. This means we never lose power even when spinning at very high speeds.

For the arms we used printed circuit boards because they are strong, thin, light and especially consistent. The very small differences in weight between the boards meant if we spin two opposite each other, their centripetal forces will completely cancel each other out. You can see our design in figure 4.

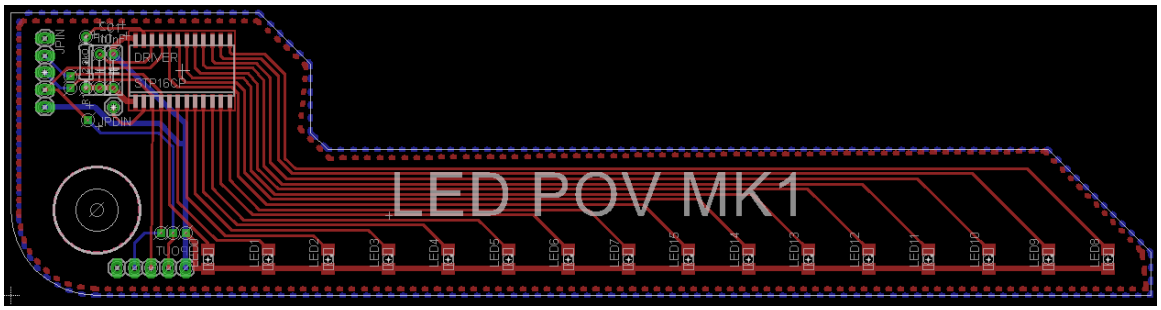
On each of these arms was an STP16CPC26 LED sink driver. This chip can drive up to 16 LEDs at a time and is controlled by 5 digital signals, serial data input, clock input,

latch input, output enable, and external resistor input. For each of the 8 drivers, output enable is tied to ground to constantly display the bits held by the driver, while the external resistor input is connected through a 20 kO resistor to ground to limit the current through each of the LEDs to 20 mA.

Each of the LED drivers contains a latch register and output register. LEDs will display the bits currently held in the output register if output enable is driven low, and bits are loaded into the latch register by toggling the clock input and serial data input pins. When the latch input is driven high, the led driver loads the latch register to the output register, simultaneously switching all the LEDs. Since all 8 blades use the same power, ground, and latch enable pins, these are connected together through the 8 blades. Each blade's serial data input pin is connected to a unique pin on the mBed.

Finally we have added an MBED, a ZigBee, a Bluetooth chip, and a hall sensor to the bottom board so that we can control each of the LED drivers. We use the hall sensor to sense when we cross the magnetic field of a magnet held stationary on the board. This helps us to extrapolate the current position.

Figure 4 (Eagle CAD PCB arm design)



Software Design:

The final software design was loosely based on the Model View Controller architecture pattern.

The Model consisted of points in radial form, points in Euclidian form, 3D blocks, and a set of 3D blocks associated with a topological map of buildings in a city. All of these classes built upon each other and they each allowed for a different mode of operation.

The View was simply the display. (although not staying quite true to MVC) You could give it points, Euclidian points, or blocks and it would add them to the display. You could also clear the display, or ask the display to update the frame that it is currently displaying. This updating of a frame correlates to each display of lights for every degree in the rotation. The View would compensate for the difference in arm angles when displaying a frame without the controller having to explicitly do anything. Updating the frame would then drive a series of clock signals with different serial outputs in between them to all of the on arm chips and then push the register to display.

The data structure that holds the state of each led for each frame has to be some sort of 3D data structure. The problem with that is it necessarily takes $O(n^3)$ memory which is a lot for an microcontroller. I got around this by creating a 2D array, which held chars. The first degree of the array dictated an angle the frame should be displayed at, The second dictated the distance from the center, and finally to get the status of an LED at a certain height you would & that char with a bit at specific position. For instance, if I wanted to get the status of an LED at degree 90, radial distance 4, and height 5, I would ask the data array for [90][4] and then & that char with the char 00010000. This would give me 0 if the light is off and some non-zero number if it should be on.

The Controller was simply the main class. It dealt with all of the input signals, and needed interaction with both the view, and the model. The first input signal it had to deal with was the hall sensor, which occurred at an exact position every rotation. It would start a timer when it triggered an interrupt, and then measure the difference from the last interrupt to get the rotational speed. After that it created a ticker, which would call a function to update the frame of the view once every 360th of the rotation time.

It would also deal with taking the input signals from the Bluetooth. If it received and ‘m’ it would change modes and clear the view. After that ‘w’, ‘a’, ‘s’, ‘d’, ‘p’, and ‘l’

were used to control different aspects of the view. In radial line mode it would keep a reference to a radial point and use ‘w’, ‘a’, ‘s’, ‘d’, ‘p’, and ‘l’ to move the point, in, out, around, or up and down as needed. It would then have the view draw the point. In Euclidian it would hold a reference to a Euclidian point and move it with ‘w’, ‘a’, ‘s’, ‘d’, ‘p’, and ‘l’ where each button correlated with out, left, in, right, up, and down. It would simply move the Euclidian point and then give it to the view to draw. The last mode was map mode, where a set of boxes would be drawn on the screen. When ‘w’, ‘a’, ‘s’, or ‘d’ were pressed it would call a method on each of the boxes to move them in the correct direction and then it would clear the view and add all of the boxes again. This seems like a lot of computation but moving a box is a low cost constant time operation, and the view would only convert the points that were close enough to appear on screen. This means the Euclidian and radial points that had to be instantiated for the box were only instantiated if the box was on screen. This lazy instantiation takes the upper bound for the complexity of the computation down from O(All of the boxes) to O(The area of the POV view). This means we can put an extremely large map into the memory and it shouldn’t have trouble updating the display.

Conclusion

In the end the project was a success. We reached our base goals and our reach goals.

For our base goals we created a working display with a couple of modes to edit the state of the display. It would hold a stable 3D image on a screen and then there were drawing modes that functioned a lot like a 3D etch a sketch. These modes allowed us to use a computer to control the movement of a cursor and create 3D lines that would be drawn along the movement of cursor. We could move this cursor in radial coordinates, or in Euclidian.

We were able to get a rudimentary version of our reach goals up and running. We can hard code in a map with blocks and lines, and then use the direction keys to move the map around as we wanted. This would remove the parts of the map that were off screen

without crashing, and would move each object in the correct direction while maintaining the layout.

From here the project can take a whole bunch of different directions. The software has been designed with adaptability in mind so a future version would preferably have more arms to give a greater z-axis resolution. Additionally I would like to create a design where two rigs of similar design would revolve in such a way that their arms overlap the same area without hitting each other. You can think of this like a gear system where they never come into direct contact but some sections move over the same area. This would allow us to make the display as large as we wanted without requiring huge spinning components that would get very dangerous.