Programming the PixelPusher

version 0.6

Jas Strong <jasmine@heroicrobotics.com>

0. Introduction

The PixelPusher is a simple beast. It takes pixels from the network, and it pushes them out to intelligent LEDs as fast as it can. It has some software inside it that runs a simple TCP/IP network stack, and a number of modules that tell it how to talk to different LED control chips.

The sophistication in this system comes from the design of the network. PixelPusher is designed to be plug-and-play; to abstract the complexity of the task away from the user; to allow artists and architects to concentrate on the interesting part of the design equation.

We provide a software stack that is comprised of three components:

1. *Firmware*

The firmware is a piece of software that is loaded into the flash memory of the PixelPusher controller cards. It contains about 100 kilobytes of machine code. It contains a realtime operating system, a network stack, a FAT filesystem, a USB stack, and a signalling stack to talk to the strips or pixels you have attached.

2. *The library*

The library is a piece of Java code which runs on the controlling computer. It consists of a set of classes which collectively find, make orderly, and talk to one or more PixelPusher controllers.

3. *Examples*

The examples are provided in the library package. They are generally quite simple implementations of the required software to make a PixelPusher array work. If you have installed the library, you can find them in Processing's "Examples..." dialogue, under "Contributed Libraries".

1. To begin...

1. The first thing you'll need is a computer. The software is based on Processing, so head to http://processing.org/ and pick up the right version for your machine. It is available for PC, Mac and Linux, so you should be pretty well covered.

2. Next you need to install the library. You can find instructions on how to install libraries into Processing at http://wiki.processing.org/w/How_to_Install_a_Contributed_Library - PixelPusher is listed under "Hardware". You can also find the PixelPusher library at http://forum.heroicrobotics.com/ - note that it is frequently updated, so be sure to keep up to date.

3. Thirdly, you should update the firmware on your PixelPusher. It is quite likely that new firmware versions have come out since your product shipped, as we follow a continuous development methodology. Don't worry- it's quite easy to do, and all you need is a mini USB cable. You can find instructions on how to do this in the "How To Push Pixels" document, also available from the Heroic Robotics forum.

4. And lastly, you will need to write a configuration file for each PixelPusher in your system, save each one onto a USB memory stick, and insert it into the appropriate PixelPusher's USB host port. Then, to make them read their configuration, press the reset button or power the PixelPusher off and then on. The configuration file is, again, explained in the "How To Push Pixels" document.

You'll need to connect all the PixelPushers to the same network your computer is on, of course.

Once you've done that, you can open one of the PixelPusher examples in Processing, hit "run" and you should see it pick up all your PixelPushers in a few seconds.

2. How does all this work?

PixelPusher is designed to automatically configure the array, and it does this using two types of message. One is sent from a controlling computer to the PixelPushers, and the other is sent from the PixelPushers to the controlling computer. Skip the next page if you're not curious about how this works.

A technical aside:

The Universal Discovery Protocol

The messages that are sent from the PixelPusher to the controlling computer follow a format we call the Universal Discovery Protocol. These packets are sent once a second by all the PixelPushers on the network. They contain several pieces of information about the PixelPusher, and are picked up and stored in a database by the library. The pieces of information include:

- the PixelPusher's IP address and MAC address
- two numbers called the "group ordinal" and "controller ordinal", assigned by the user
- how many strips are attached to this PixelPusher
- how many pixels are in each strip
- how long it takes this PixelPusher to process each packet of data
- how many packets the PixelPusher missed in the last second
- how many strips can fit into a single datagram (packet).
- a flag byte for each strip, describing the pixel layout (RGB or RGBOW, 24-bit or 48-bit, renormalized or linear).
- the software version the PixelPusher is running
- a flag word for the PixelPusher, denoting whether it requires packed strip datagrams

The library uses this information to build a representation of the array in memory. This representation is populated with PixelPushers as they are seen, and kept up to date as discovery packets arrive. The application software can then ask the library for pieces of this representation, and modify the pixels in the model; the library handles sending updates out to the PixelPushers automatically and asynchronously. To do this, it uses the second form of traffic.

The Strip Update Message Protocol

Strip updates are packed into datagrams and sent by this protocol. The packet contains information relating to specific strips, including:

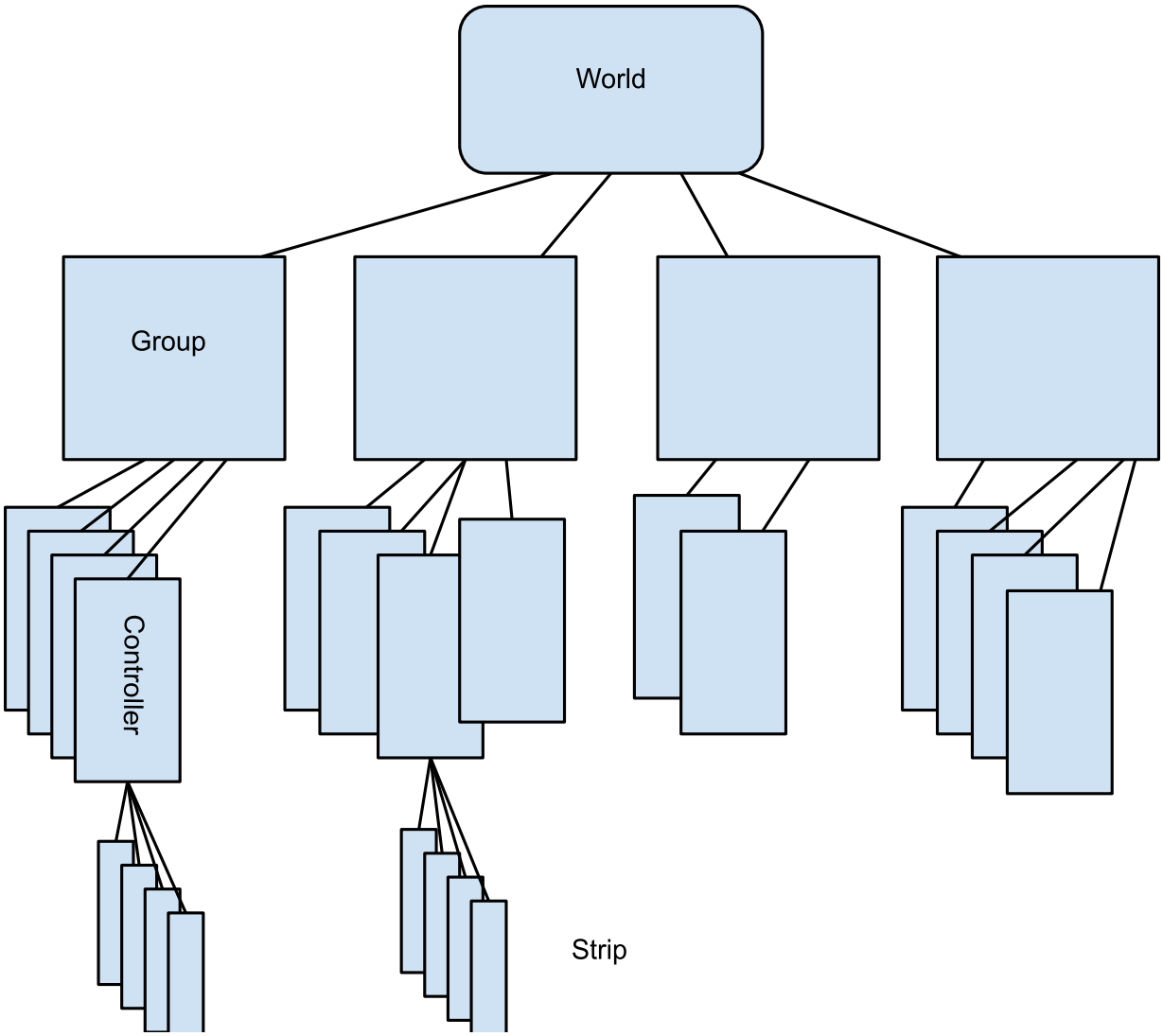
- a per-card sequence number for the current packet
- a strip number from zero to seven
- a number of RGB triplets appropriate to the number of pixels in a strip

Since the memory available to the ethernet controller on PixelPusher is limited, only a limited number of pixels can be sent in a single datagram. With our usual maximum of 240 pixels per strip, two strips can be sent in each datagram. Each datagram has a single sequence number, followed by a variable number of repetitions of the strip number followed by the pixel data. The timing information and the missed packet data from the discovery packet are used to throttle the packets being sent to the available system bandwidth.

3. Groups and Controllers

We anticipate that you might want to have more than one section to your PixelPusher array. Maybe you're lighting different areas, or you want to have a big billboard-sized video display that's separate from your 3d sculpture. That's why we implemented groups and controllers.

When the library enumerates a PixelPusher it reads from it two numbers, which are called *group* and *controller*. Each group is a collection of controllers, and within each group they are ordered by the controller number. You can select controllers by group, or you can just select all of them at once, in which case they are ordered first by group and then by controller- so, for example, group 2 controller 1 comes before group 2 controller 2, but group 1 controller 37 comes before either. If two controllers have the same number, the ethernet hardware address is used as a tiebreaker. So there is a three level hierarchy:



4. Nuts and bolts

As you'll see from our examples, there are a few bits and pieces you'll need in your Processing sketch in order to make the system work. Here's a quick rundown.

- The imports

PixelPusher defines a number of classes that provide essential features. At the top of the sketch you'll see a block that looks like this:

```
import com.heroicrobot.dropbit.registry.*;
import com.heroicrobot.dropbit.devices.pixelpusher.Pixel;
import com.heroicrobot.dropbit.devices.pixelpusher.Strip;
import java.util.*;
```

This imports the registry, which is the software component that listens for the discovery packets, the PixelPusher abstractions for Pixels and Strips, and the Java standard utility classes which are used for collections.

- The registry

The next thing we do is to declare the DeviceRegistry.

- The observer

PixelPusher's registry uses an Observer model to tell the sketch when PixelPushers appear and disappear. There's a simple implementation of this below.

```
class TestObserver implements Observer {
  public boolean hasStrips = false;
  public void update(Observable registry, Object updatedDevice) {
        println("Registry changed!");
        if (updatedDevice != null) {
          println("Device change: " + updatedDevice);
        }
        this.hasStrips = true;
    }
};
```

Then we declare one, like this:

```
TestObserver testObserver;
```

As you can see, it's not a big class, and just implements a member you can use to check when there are strips available to the system. The observer will get called and updated whenever the array's shape or behaviour changes significantly.

- Setup

Then there are some bits and pieces we need to do in the sketch's setup() method. This is a method that runs before the sketch starts in earnest, and it's a good place to instantiate the bits and pieces we need for PixelPusher. The code we use looks like this:

```
  registry = new DeviceRegistry();
  testObserver = new TestObserver();
  registry.addObserver(testObserver);
```

This instantiates the registry, getting it running and listening for PixelPushers to appear. It instantiates the observer, and then tells the registry about it.

- Pushing those pixels

Having done all this setup, we're good to go. We do our actual pushing in the sketch's `draw()` method. First we have to wait until there actually are some strips in existence! It might take a second or two after the sketch starts until a PixelPusher announces itself. We wait until some make an appearance.

```
if (testObserver.hasStrips) {
      //...  the PixelPusher code is wrapped up in here
}
```

When they make their appearance, we make a call like this to make the registry start talking to the PixelPushers:

```
registry.startPushing();
```

The next thing we need to do is to get the strips. The Strip class is a representation of a string of LED pixels or a flexible LED strip. As we saw earlier, there are groups which contain controllers each of which contains strips. Since walking this hierarchy is tedious, we provide a way that you can bypass this and have the registry do it for you.

```
List<Strip> strips = registry.getStrips();
```

This returns a list of all the strips known to the registry. Alternatively, if you only want to talk to one particular group, you can do this:

```
List<Strip> strips = registry.getStrips(some_group_number);
```

and it will return a list of all the strips belonging to controllers whose group number is given.

Then all you need to do is to loop through the strips and set the pixels within them to the appropriate colour. Be warned that not all the strips may be the same length! The Strip class gives you a method to determine the length. `strip.getLength()` will return the number of pixels in the strip. To set the colour of a pixel, you can call `strip.setPixel(color, number)` where colour can either be a Processing color object or a PixelPusher Pixel object.

5. Every day I'm scrapin'

In most of our examples, we scrape pixels from the Processing sketch window and send them to the array directly. The code to do this looks like this:

```
int stripy = 0;
List<Strip> strips = registry.getStrips();
int yscale = height / (strips.size());

for(Strip strip : strips) {
    int xscale = width / strip.getLength();
    for (int stripx = 0; stripx < strip.getLength(); stripx++) {
         color c = get(stripx*xscale, stripy*yscale);
         strip.setPixel(c, stripx);
    }
    stripy++;
}
```

6. Tweaking the rate

Normally, the registry keeps track of how many packets are being lost (they get dropped if there isn't enough bandwidth or if you're using a wireless network and there's noise) and how long each PixelPusher is saying it takes to push out a complete packet to the strips, and adjusts the rate at which packets are sent automatically. However, if you're on a very slow network, like a very long wireless link or a cellular modem, you might want to add an extra slowdown. That's what the `registry.setExtraDelay()` method is for.

```
registry.setExtraDelay(milliseconds);
```

This adds an extra delay to every packet that is sent by the registry's worker threads. If you set it to zero, only the standard rate limiting applies. With full-length Heroic Robotics strips and the default update rate, each PixelPusher consumes between 5 and 10 megabits per second of bandwidth. In the special turbo mode, a PixelPusher with two strips running at maximum speed will consume about 52 megabits per second of bandwidth.

If you're on a network with a high error rate, like some wireless networks, or poorly installed ethernet, you might find that you get persistently high error rates and the update frequency drops uncontrollably. In this case, you may want to disable the autothrottling entirely. We provide a registry method to do this. `registry.setAutoThrottle(true)` will turn autothrottling on, and `registry.setAutoThrottle(false)` will turn it off.

As of 2013-03-25, the library defaults to autothrottling being turned off, so you will need to make the above method call to turn it on.

7. Getting the colours right

If you're using video source material and everything looks washed out, you may want to apply the antilog curve correction. PixelPusher faithfully sends everything you tell it to out to the strips, otherwise. As of the 2013-06-08 version of the library, there is a call to tell it to correct the luminance curve: `registry.setAntiLog(true)` will set the curve correction on. The value `false` will turn it off. Thanks are due to Rus Maxham for contributing this code.

8. A note on architecture

The way PixelPusher's library actually works is that there are several threads, all running asynchronously. The data structures used internally are all lockless, so good realtime performance is possible. The threads are named, so if you want to use VisualVM to profile what's using CPU time, you can easily identify them. The different processes are these:

- The UDP callback (aka DiscoveryListener)

There's a small routine that gets called every time a discovery packet arrives, which updates the PixelPusher data structures. It keeps the structures up to date, and also notes how long it's been since any given PixelPusher has been seen. If they disappear for more than a few seconds, they are removed from the structure. If they say that they have missed packets (as

indicated by discontinuities in the sequence numbers) then this routine is responsible for changing the appropriate rate limiting variables.

- The preening thread

This runs every few seconds to clean the data structures and remove any sadly departed PixelPushers from the system.

- The marshalling thread (aka SceneThread)

This runs continuously, and takes care of the worker threads, making sure that there is one worker thread for every PixelPusher. It's responsible for killing threads where the PixelPusher has disappeared, and starting new ones when new PixelPushers appear.

- The worker threads (with names like "CardThread for 00:04:13:a3:a6:5c")

Each PixelPusher card has an associated worker thread. The worker thread is simple- it checks the Registry to see how the card is doing, and then it checks the strip structures associated with its card. If any of them are marked as updated, it walks through them, sending them to the card at the rate that has been calculated, and marks them as not updated.

There are several data structures internal to the library which are used to keep track of which cards are which, which strips belong where, and which strips are which length. It is strongly recommended that you stick to using the software interfaces listed in this document, rather than modifying any of the structures yourself, since these have changed in the past and will likely change in future- which would require you to write new code!

9. Multi-head operations

Because the worker threads only send packets to update pixels that have changed, it is possible to have multiple controlling PCs (or Macs, Android devices, etc.) on a single PixelPusher array, as long as they are each controlling separate subsets of PixelPushers. It is strongly recommended that you restrict this sharing to one controlling PC per PixelPusher, since otherwise there are likely to be timing conflicts and you may end up flooding one poor PixelPusher with twice as much network traffic as it deserves. If you just can't keep the architecture this simple, use the Extra Delay feature described a few pages ago to reduce the signalling rate. You will also need to disable the autothrottle system, since the sequence numbers will not be synchronized between different controlling PCs.