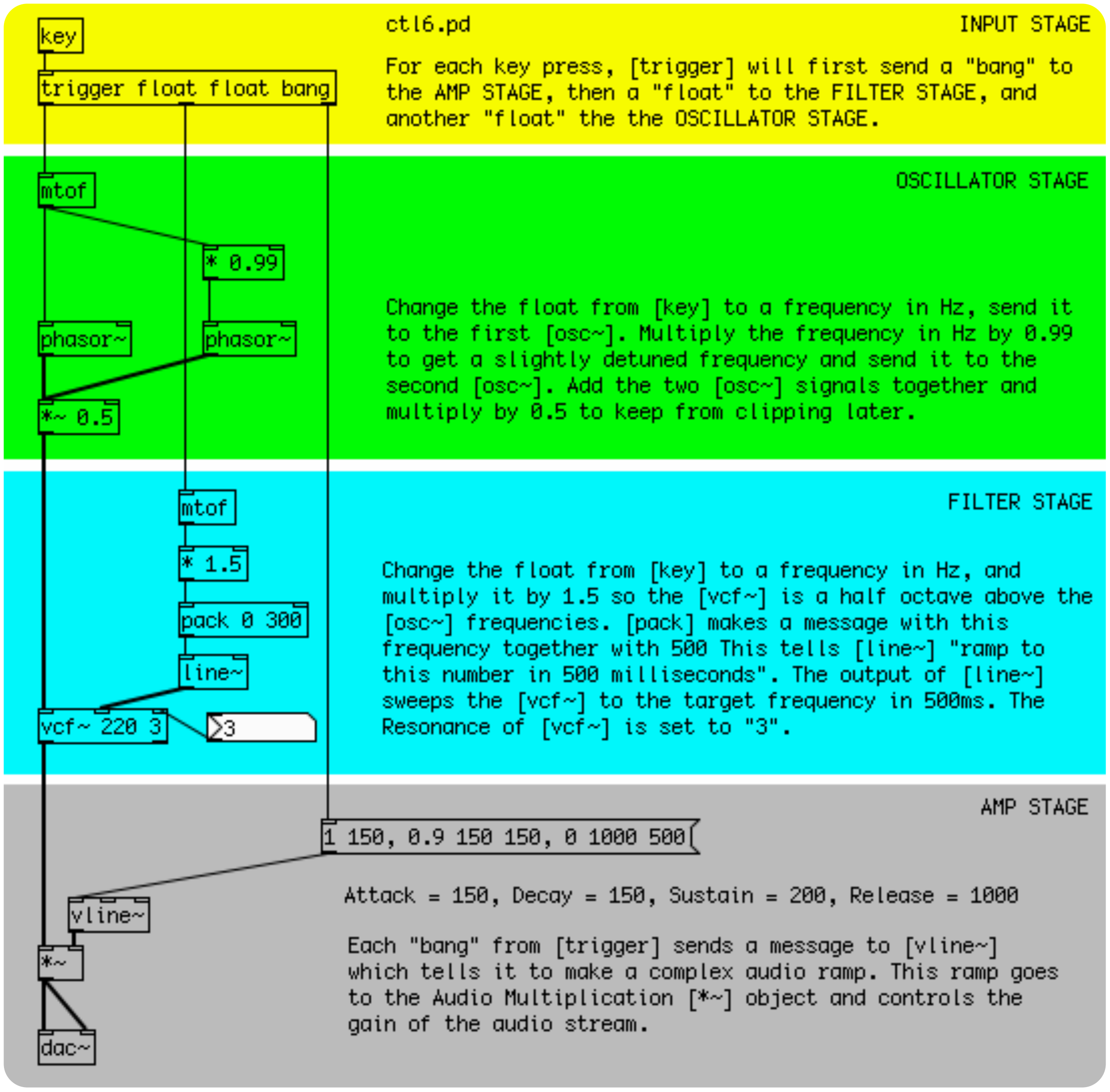
Building a Pure Data Sound Synthesizer

John Talbert, May 2021



ctl6.pd

INPUT STAGE

For each key press, [trigger] will first send a "bang" to the AMP STAGE, then a "float" to the FILTER STAGE, and another "float" the the OSCILLATOR STAGE.

OSCILLATOR STAGE

Change the float from [key] to a frequency in Hz, send it to the first [osc~]. Multiply the frequency in Hz by 0.99 to get a slightly detuned frequency and send it to the second [osc~]. Add the two [osc~] signals together and multiply by 0.5 to keep from clipping later.

FILTER STAGE

Change the float from [key] to a frequency in Hz, and multiply it by 1.5 so the [vcf~] is a half octave above the [osc~] frequencies. [pack] makes a message with this frequency together with 500 This tells [line~] "ramp to this number in 500 milliseconds". The output of [line~] sweeps the [vcf~] to the target frequency in 500ms. The Resonance of [vcf~] is set to "3".

AMP STAGE

Attack = 150, Decay = 150, Sustain = 200, Release = 1000

Each "bang" from [trigger] sends a message to [vline~] which tells it to make a complex audio ramp. This ramp goes to the Audio Multiplication [*~] object and controls the gain of the audio stream.

A Four Stage Filtered Additive Synthesizer

Table of Contents

What is Pure Data

http://write.flossmanuals.net/pure-data/introduction2/

Pure Data (or **Pd**) is a **real-time graphical programming environment** for **audio**, **video**, and **graphical processing**. Pure Data is commonly used for live music performance, VeeJaying, sound effects, composition, audio analysis, interfacing with sensors, using cameras, controlling robots or even interacting with websites. Because all of these various media are handled as digital data within the program, many fascinating opportunities for cross-synthesis between them exist.

Programming with Pure Data is a unique interaction that is much closer to the experience of manipulating things in the physical world. The most basic unit of functionality is a box, and the program is formed by connecting these boxes together into diagrams that both represent the flow of data while actually performing the operations mapped out in the diagram. The program itself is always running, there is no separation between writing the program and running the program, and each action takes effect the moment it is completed.

Pure Data is a graphical programming environment. What this means is that the lines of code, which describe the functions of a program and how they interact, have been replaced with visual objects which can be manipulated on-screen. Users of Pure Data can create new programs (**patches**) by placing functions (**objects**) on the screen. They can change the way these objects behave by sending them **messages** and by connecting them together in different ways by drawing lines between them.

The real advantage of Pure Data is that it works in "real time". That means that changes can be made in the program even as it is running, and the user can see or hear the results immediately. This makes it a powerful tool for artists who would like to make sound or video in a live performance situation.

The core of Pure Data is written and maintained by Miller S. Puckette (http://crca.ucsd.edu/~msp/) and includes the work of many developers (http://www.puredata.org/), making the whole package very much an open source community effort. Pd runs on **GNU/Linux**, **Windows**, and **Mac OS X**, as well as mobile platforms like **Maemo**, **iPhoneOS**, and **Android**.

Learning Pure Data

Pure Data is an ideal platform for creating Electronic Music Synthesizers. The title page show some Pure Data code taken from the Pure Data Tutorial on the Floss Manuals Website. This four stage filtered additive synthesizer is a good example of what Pure Data Code looks like.

Here is a list of books and tutorials on Pure Data, with an emphasis on Electronic Music Generation.

Miller Puckette PD Creator	http://msp.ucsd.edu/Pd_documentation/index.htm
PD Website Tutorials	https://puredata.info/docs/tutorials
Floss Website Manual	http://write.flossmanuals.net/pure-data/
"Loadbang" by Johannes Kreidler	http://www.pd-tutorial.com
"Designing Sound" by Andy Farnell	http://aspress.co.uk/ds/pdf/pd_intro.pdf

"Pure Data: Electronic Music and Sound Design" by Blanchi, Alessandro, Maurizio

PD Synthesizer Design

It is possible to build an entire Music Synthesizer from just Pure Data, complete with all the Slider and Switch objects needed to operate it. However, the problem with this setup is that only one controller at a time can be operated from a mouse or trackpad. What is really needed is a bank of actual pots, sliders, and switches and maybe even light sensors, motion detectors, accelerometers, foot switches, etc. This paper will detail a way to incorporate as many hardware controllers as you like into your Pure Data software sketch.

Basically, an Arduino microprocessor will be tasked with the job of collecting data from all the controller devices and transmitting it to the Pure Data sketch. From among the many communication protocols designed to send data between an Arduino microprocessor and a main computer, USB MIDI was chosen for its simplicity and speed.

All Arduino boards provide multiple ADC pins to connect the pots, sliders, and other sensors mounted on your controller box. They also all have digital pins for connecting any switches and other on/off type sensors.

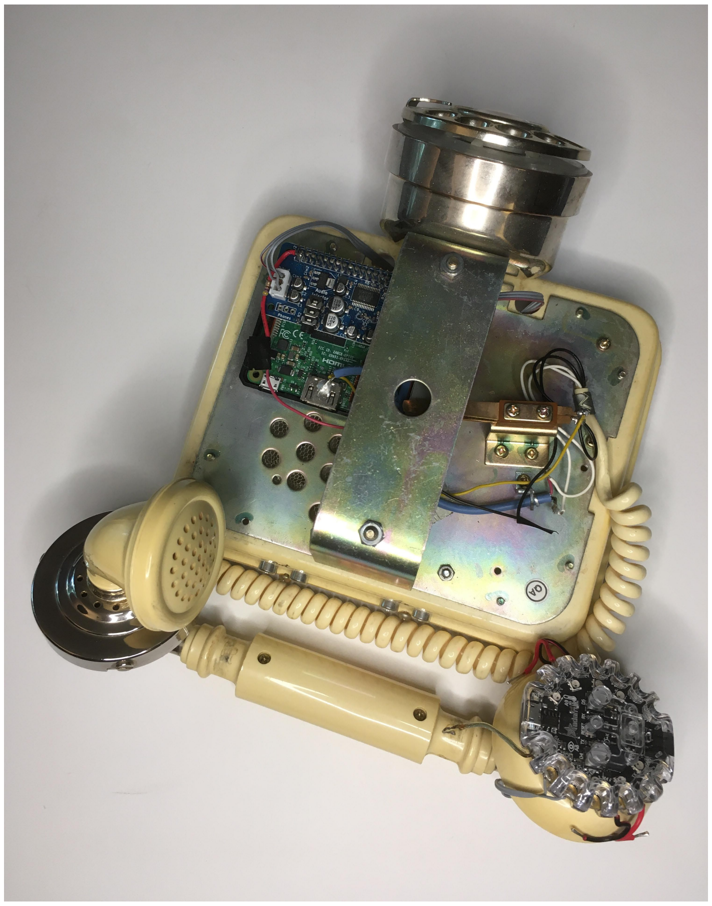
The Arduino MIDIUSB library used in this project only works on microcontrollers with native USB capabilities. That limits your Arduino board choices to the Leonardo, Micro, Due, Nano and MKR boards. With the MIDIUSB library you can just plug your microprocessor into your personal computer with a USB cable, and it will appear to your Pure Data sketch as a MIDI controller. However, in this case, the Note-On MIDI commands transmitted back and forth over the USB cable will not be carrying the usual music keyboard data. Instead, they will be programmed to carry any changes happening among the pots, sliders, switches and sensors making up your controller device.

The Controller

By now you probably have your own vision of what a controller device would look like, maybe a box with rows and rows of slide pots, joysticks, big fat pushbuttons and the like. The programs outlined here on both the Arduino microprocessor side and the Pure Data sketch side can easily accomodate whatever you have in mind.

However, I'm going to go a little wild and crazy here and build a controller that nobody has imagined. Be assured, though, that program design detailed here can easily be adapted to your own ideas.

Meet the French Continental PD Phone Synth !



Here we have a fancy old style dial phone. The phone innards was relieved of its massive solenoid and bell to provide space for a Raspberry Pi computer board running the Pure Data programming app. The base of the phone was drilled to accomodate Audio stereo input and output jacks connected to an audio codec board (AudioInjector.net) riding on top the Raspberry Pi. A rotary pot glued to the Rotary Dial of the phone and the handset cradle switch were co-opted for future audio schemes.

To top off this brilliant device, Adafruit's Circuit Playground Express was chosen as the controller microprocessor. This micro is encased in a small circular disc that mounts perfectly on the handset earpiece. Best of all, the USB cable that connects the Circuit Playground Express (CPE) to the Raspberry Pi fits nicely inside the coils of the handset cable.

Circuit Playground Express (CPX)

https://learn.adafruit.com/adafruit-circuit-playground-express

The Circuit Playground Express by Adafruit is a miniature microprocessor device incorporating a number of sensors and LEDs, all mounted on a small circular board. Its ARM processor is compatable with the Arduino IDE programming app and with the Arduino MIDIUSB library. The microprocessor includes a light sensor, a three axis motion sensor, a microphone sensor, ADC connections, Digital Input connections, Capacitive Touch connections, on-board switches, and ten RGB LEDs.

A USB cable connection is used to program and interact with the CPX from a main computer. That main computer can an Apple Mac computer running Arduino's IDE programming app or Pure Data. Alternatively, that USB cable could easily be switched over to a Raspberry Pi programming environment at any time.

The programming required is two-fold. First, the Circuit Playground must be programmed to send its changing sensor data to the main computer and, perhaps, also receive LED data from the main computer to light the ten RGB lights spaced around the Circuit Playground. All data will be sent and received through the serial USB cable connecting the Circuit Playground to the Main Computer. This communication program, once written and tested from an Arduino IDE app, can permanently be loaded onto the Circuit Playground to service any Pure Data sketch needing a controller device.

Second, the Pure Data sketch, running on either the Raspberry Pi or the Apple Mac, must be able to receive this specially formatted sensor data from the Circuit Playground and do something interesting with it. The receiving subroutines in Pure Data can be set in stone, but the main music synthesizer program can be changed at will.

Setup

The Circuit Playground has a flash drive that should appear as a “CPLAYBOOT” drive on your Mac. Unfortunately, MacOS version 10.14.4 and above will prevent the flash drive from appearing. An older MacOS or a Windows or Linux machine (Raspberry Pi?) is needed to update the system on the Circuit Playground Express.

https://learn.adafruit.com/adafruit-circuit-playground-express/updating-the-bootloader

After updating the bootloader, the Arduino IDE programming app should work fine even if the flash drive does not appear on the desktop. Double clicking the button at the center of the Circuit Playground will reset the processor causing all the NeoPixels to turn green.

The next step is to program the Circuit Playground from the Arduino IDE app (Integrated Developement Environment). Make sure your Mac is on a WiFi Network. Start up the Arduino IDE. Select the Menu Tools/ManageLibraries. Do a search for “circuit playground” and install/update this library. Try out the example programs under File/Examples/AdafruitCircuitPlayground.

https://learn.adafruit.com/adafruit-circuit-playground-express/set-up-arduino-ide

You can also download the complete Library from: **https://github.com/adafruit/Adafruit_CircuitPlayground**

Your Arduino program should include both the Circuit Playground and MIDIUSB libraries:

```
#include <Adafruit_CircuitPlayground.h>
#include <MIDIUSB.h>
```

Note that some of example programs in the Playground Library were written for the older Classic Playground so the sensor addresses (A1, A2, …) may need to be changed to the newer Playground Express addresses.

https://learn.adafruit.com/adafruit-circuit-playground-express/pinouts

Function	Express	Classic
Button A	D4	D4
Button B	D5	D19
Slide Switch	D7	D21
Red LED	D13	D13
10 NeoPixels	D8	
Accel Interrupt	D27	
IR Transmit	D25	
IR Receive	D26	
Light Sensor	A8	A5
Temp Sensor	A9	
IR Proximity	A10	
Microphone	Internal	A4
Speaker Out	A0/D12	A11/D12
Cap Touch	A2/D9	A9/D6
Cap Touch	A3/D10	A10/D10
Phone Dial	A4/D3	
Phone Switch	A5/D2	
Cap Touch	A6/D0	A0
Cap Touch	A7/D1	A1

Once your Arduino sketch is loaded and running, start up the Pure Data application on the main computer. Inside the Pure Data application you must first find the serial port number that the Circuit Playground occupies. Click on the "devices" message box to get a list of serial ports in the PD Console window. Choose the one that shows up when you plug in the Circuit Playground USB and enter that in the green "serial port #" selection box. When the correct port is selected, the running lights on the Playground Express will stop, indicating that serial communications has been set up. Messages in the PD Console Window will also confirm this.

Finally, go to "MIDI Setting…" under the Pure Data "Media" menu and choose "Circuit Playground Express" for the input and output devices.

Communication

Firmata, a well established communications protocol, is included in the Arduino Library for the Circuit Playground. It uses MIDI protocol, byte-size serial communication through USB cables. The MSB (most significant bit) bit of each byte is used to mark each byte as a command or 7-bit data. A zero in the most significant bit marks the 7 lower bits as data. A "1" in the most significant bit marks the lower 7 bits as commands. Firmata is designed to be all-encompassing and as such can be unwieldy.

There is a public Firmata written specifically for the Circuit Playground Classic. The Firmata code was written for the older Classic Circuit Playground and does not support some of the Express devices such as the Accelerometer or the Capacitive Touch sensors. On the PD side, Pure-Data has public Firmata scripts for most Arduino Boards but the scripts use the PD extended version and, as of now, there is no public PD script coded specifically for the Circuit Playground Express.

Other communication schemes, such as Simple Message System, converts the data to ASCII numbers for the data and ASCII letters for the commands, all separated by special ASCII characters such as Carriage Return or Space. The ASCII bytes are sent through the serial USB lines. Inside Pure Data, the ASCII then needs to be converted back to actual numbers. This scheme is simple but slow.

OSC (Open Sound Control) uses Ethernet or WiFi for communications which is super fast. Because of the speed, this data is sent simply as text. There are standard OSC text formats for different types of data; however, in reality, the text data can be loosely formatted to anything both sender and receiver can agree upon. Pure Data has some UDP or TCP objects designed to receive and send Ethernet data.

Solution

My final solution involves two components, an Arduino programmed sketch for the Circuit Playground Express, and a set of Pure Data objects designed to receive controller data and send LED Pixel data. These two sketches will be documented in full at the end of the paper.

For this project I have chosen to use MIDI messages, in particular the Note-On command, to transfer data between the Circuit Playground Express and Pure-Data on the Raspberry Pi. In this case the data transfered through MIDI is not the traditional musical note information; it is sensor data from the Circuit Playground.

Three bytes are sent in a MIDI Note-On command - the Command/MIDI-Channel byte, the Pitch byte, and the Note Velocity byte.

1. ***Command Byte****. 4 bits of the Command Byte identify the Midi command as Note-On. They must be set to 1001. The 4-bit MIDI-Channel data in the command byte will specify which sensor is sending data, for as many as 16 sensors.*

2. ***Data Bytes****. The two Note-On data bytes will be used for the sensor data - 7 bits from Pitch and 7 bits from Velocity. This allows the data to have any value from 1-bit switch data to 14-bit sensor data (a maximum value of 16,383).*

For data numbers larger than a 7-bit 127 maximum, set Pitch equal to the value bit-shifted right by 7, and set Velocity equal to the value ANDed with 127 (Binary 0111 1111).

MIDI over USB is used as it has a much higher speed than regular MIDI over MIDI cables. Since the MIDI communication uses the same USB lines as the Arduino IDE program loading, there can be problems interrupting the current MIDIUSB program to load a new one from the IDE app. If your program refuses to load, follow this procedure:

The IDE load command starts by "Compiling" the Program and then "Uploading" it. Wait for the "Uploading" to start and then double click the center RESET button on the Circuit Playground to interrupt the current program and put the CPX into bootload mode. Timing this action can be tricky, but you should be able to eventually get the load to work after several attempts.

On the Pure Data side, simply use several MIDI "notein" objects to receive the CPX sensor data. Adding a "MIDI Channel" argument (1 to 16) to notein will cause that object to only receive messages from a particular sensor, as set in the Arduino sketch. If both the Pitch and Velocity outlets are combined to receive a sensor value larger than 127, the left outlet value must be shifted left by seven bits and then added to the right outlet. See the illustrated Pure Data sketch at the end of this paper.

Inside the Arduino Circuit Playground sketch each CPX Sensor is abitrarily assigned a MIDI Channel (0-15 here, 1-16 in PureData)

0	A6 Capacitive Touch
1	A7 Capacitive Touch
2	A2 Capacitive Touch
3	A3 Capacitive Touch
4	Phone Cradle Switch

5	Right Button
6	Left Button
7	Slide Switch
8	Phone Rotary Dial
9	Light Sensor
10	Sound Pressure
11	
12	
13	Accelerometer X
14	Accelerometer Y
15	Accelerometer Z

Received in PureData using "notein x" objects with x = Sensor (MIDI CHNL#)

Sensor Data Collection

The main job of the Circuit Playground sketch is to collect Sensor data. This commonly involves the function analogRead(pin) for the rotary and slide pots, and the function digitalRead(pin) for the switches. In this project, the rotary dial pot and the cradle switch use these functions. For most controller designs with banks of sliders and switches, these two functions are all you need.

In this project, specialized functions for the sensors found in the Circuit Playground Library are added to the program.

```
readCap(pin)
rightButton()
leftButton()
slideSwitch()
lightSensor()
soundSensor()
motionX()
motionY()
motionZ()
```

Each sensor value on your controller is read once each time through the main loop. Its value is then sent to the Pure Data sketch over USB MIDI, but only if there is a change in the value. The entire main loop program is repeated over and over again for as long as the Circuit Playground has power. That means that each sensor's value must be stored in a variable. Any reading from the current loop is then compared with the value stored from the last loop. If there is a change, the new value is sent out and its variable is updated with the new value. If there is no change, then nothing happens. This action is easily accomodated in code with the common "if-then-else" construct.

Sometimes the sensor outputs will naturally fluctuate by small amounts resulting in a constant rapid flow of MIDI data for that sensor. This can be minimized by averaging out the fluctuations. To do this in the code, several past readings are saved, summed together with the current reading, and then divided by the number of readings summed. This average value is then saved and used as the sensor value for the if-then-else decisions described above.

In code, the averaging is accomplished by saving a set number of past readings in a circular array. A circular array has a pointer to the positions of the stored array readings. The pointer always points to the oldest sensor reading. Each time through the loop the following steps are taken:

```
1. Use the loop pointer to get the oldest reading in the array.
2. Subtract that reading from a stored sum of all the array readings.
3. Add the new reading to the stored sum.
4. Enter the new reading into the array at the pointer's location, replacing the oldest reading.
5. The pointer is now at the newest reading. Increment it so it again points to the oldest reading.
6. Make the array circular - if the incremented pointer goes past the end of the array, reset it to the
zero element.
```

This averaging calculation is employed for many of the analog sensors. The size of the arrays, or number of past reading saved, can easily be changed in the code. The larger the array the less flucuations in the sensor. Greater smoothing out of the sensor readings also slows down the sensor changes; so there is a compromise involved in setting the size of the averaging arrays too large. It is also interesting to note that, due to the design of our coding here, increasing the size of the array does not increase the time needed to do the averaging calculation.

The touch sensors proved to be a special problem. The touch sensors were implemented by simply hanging a bare wire from the Circuit Playground solder pads. They were to act as simple digital switches - high value when touched and low value otherwise. However, the "readCap()" function does not put out a digital value, it actually puts out an analog value. It was found that the readings are fairly constant when the wire is not being touched, but fluctuate wildly when touched -- nothing at all like a proper switch.

To solve this problem, I decided to use the same averaging scheme described above but averaging not the actual sensor values but the absolute value of the differences between the readings. When touched, the average difference is large, while the untouched average difference is close to zero. You then compare the difference average with some chosen threshold value to decide on a final switch value of 1 or zero. Problem solved but with a lot of calculation.

I suspect that the touch sensor pads need to be mounted close to but insolated from a ground plane to tame them.

CPX Pixels

The Circuit Playground Express has 10 multi-colored RGB Pixel Lights. The CPX library has a function designed to light up just one Pixel at a time by sending it three 8-bit color values - CircuitPlayground.setPixelColor(pixel#, red, green, blue).

The MIDI “noteout” object is used from Pure Data to manipulate the Pixels. This object sends 3 bytes to the Circuit Playground Express - rx.byte1 with a 4-bit MIDI Channel number, rx.byte2 with 7-bits of data, and rx.byte3 with 7-bits of data. With this limited data space in a MIDI NoteOut command, we can't use it to fully load an RGB pixel's three full bytes of red, green, and blue data. Instead, the NoteOut command will be used to choose from several Pixal operations defined in the Arduino sketch.

The 4-bit MIDI Channel number in rx.byte1 is used in a “switch/case” function to define 16 different Pixel operations. In general, the first 8 operations play with three program variables labeled red, green, and blue. Each is an 8-bit number, variable between 0 and 255. The remaining 8 operations load the color variables onto one or more of the 10 Pixels. The chart below details the 16 operations created for this Circuit Playground sketch.

Pure Data Operation	rx.byte2	rx.byte3
1	0a00 000c	0ddd dddd
Set red: If a=1, then red = a random value, else red = cddd dddd		
2	0a00 000c	0ddd dddd
Set green: If a=1, then green = a random value, else green = cddd dddd.		
3	0a00 000c	0ddd dddd
Set blue: If a=1, then blue = a random value, else blue = cddd dddd.		
4	0a00 0000	0000 0000
Red Pixel: If a=1, then red = a random value, else red = 255, green = 0, blue = 0.		
5	0a00 0000	0000 0000
Green Pixel: If a=1, then green = a random value, else green = 255, red = 0, blue = 0.		
6	0a00 0000	0000 0000
Blue Pixel: If a=1, then blue = a random value, else blue = 255, red = 0, green = 0.		
7	0a0b bbbg	0ggg rrrr
Set All: If a=1, then red, green, blue = random xxxx xxxx values, else red = rrrr 0000, green = gggg 0000, and blue = bbbb 0000.		
8	0mpp ppcc	0xxx xxxx
Fill Matrices: Two redx, greenx, and bluex arrays were created to hold RGB values for all 10 pixels. One RGB Pixel color is loaded in this operation (one of 30 per matrix). pixel color = xxxx xxx0, RGB select = cc (red=0, green=1, blue=2), pixel number = pppp (0 to 10), matrix = m (one of two).		
9	0m00 0000	0000 0000
Load Matrix: matrix select = m (one of two). Load 30 RGB values for all 10 Pixels from matrix.		
10	0000 0000	0000 0000
Load Random: Load all 10 Pixels with random color		
11	0000 0000	0000 0000
Load Random Variation: Keep higher 4 bits of red, green, and blue, and load a random value for their lower 4 bits. (cccc rrrr)		

12		0000 0000	0000 0000
None:			
13		0000 0000	0000 pppp
Load One Pixel:	Load one Pixel (equal pppp) with values in the variables red, green, blue		
14		0000 0ppp	0ppp pppp
Load Any Pixels:	Each “p” represents one of 10 pixels. Load all those where p=1 with the values in the variables red, green, blue.		
15		0000 0000	0000 0000
Load All:	Load all 10 Pixels with same color from red, green, blue.		
16		0000 0000	0000 0000
Clear All:	Turn off all Pixels. CircuitPlayground.clearPixels( )		

The simplest way to use “noteout” in Pure-Data for these LED operations is to create several messages populated with three rx.byte LED operation values and connect them to the left input of the “noteout” object. Click on the messages you want sent to the Circuit Playground.

| rx.byte2 rx.byte3 operation# (

CPX Sketch

```
/*
   Many thanks to George Mandis for his circuit-playground-midi-multi-tool

   Circuit Playground Express Sensors are sent as MIDI Note-On data
   LED Pixels can be set with Received MIDI Note-On data

   Serial.print statements can be commented out except when debugging.

   Each Sensor is assiged a MIDI Channel (0-15 here, 1-16 in PureData)

   0   A6 Capacitive Touch
   1   A7 Capacitive Touch
   2   A2 Capacitive Touch
   3   A3 Capacitive Touch
   4   Phone Cradle Switch
   5   Right Button
   6   Left Button
```

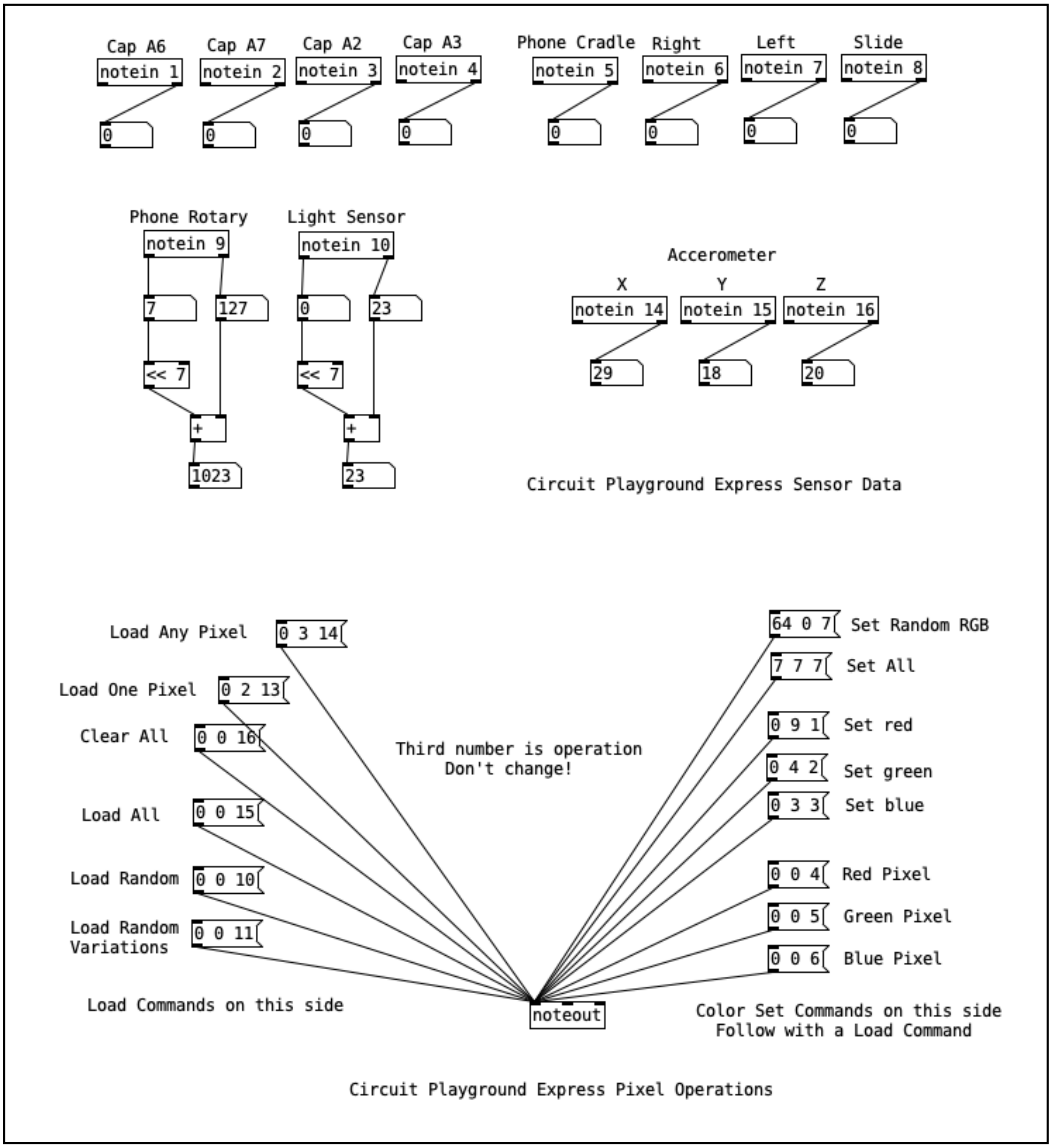
```
    7   Slide Switch

    8   Phone Rotary Dial
    9   Light Sensor
    10  Sound Pressure
    11
    12
    13  Accelerometer X
    14  Accelerometer Y
    15  Accelerometer Z

    Receive in PureData using "notein x" objects with x = Sensor (MIDI CHNL#)

*/

#include <MIDIUSB.h>
#include <Adafruit_CircuitPlayground.h>

uint8_t pads[] = {0, 1, 9, 10};  // "A" pin numbers used for Capacitive Touch
//These are Circuit Playground Classic pins (Actual Express pins are A6, A7, A2, A3)

uint8_t padValues[] = {0, 0, 0, 0}; // Capacitive Touch On/Off states

const int numChannels = sizeof(pads); // number of pins set up for Capacitive Touch
const int numReadings = 3;  // number of diffs used to calculate average

int readingDiffs[numChannels][numReadings];   // the difference readings from caps
int total[numChannels];  // added total of diffs
int lastReading[numChannels];  // last actual Cap reading for calculating diff

int diffSensitivity = 100;
int capacitiveRead;
int readIndex;
int average = 200;

bool rightButtonPressed = 0;
bool leftButtonPressed = 0;
bool slideSwitchPressed = 0;
bool phoneButtonPressed = 0;

const int rotaryNumReadings = 3;
int rotaryReadings[rotaryNumReadings];
int rotaryTotal = 0;
int rotaryIndex = 0;
int rotaryRead = 0;
int rotaryAverage = 0;

const int lightNumReadings = 5;
int lightReadings[lightNumReadings];
int lightTotal = 0;
int lightIndex = 0;
int lightRead = 0;
int lightAverage = 0;

const int soundNumReadings = 10;
int soundReadings[lightNumReadings];
int soundTotal = 0;
int soundIndex = 0;
int soundRead = 0;
int soundAverage = 0;

int X;
int Y;
int Z;
int color;
int red = 0;
int green = 0;
int blue = 0;
byte redx[2][16];
byte greenx[2][16];
byte bluex[2][16];
byte matrix;

int xxx;

midiEventPacket_t rx;

/*
 * **********************SETUP***********************
 */

void setup() {
 CircuitPlayground.begin();
 CircuitPlayground.setAccelRange(LIS3DH_RANGE_8_G);
 Serial.begin(9600);

 pinMode(2, INPUT_PULLUP);   //Phone Cradle Switch
 randomSeed(analogRead(A4));

}

/*
 * ********************FUNCTIONS*******************
 */

void noteOn(byte channel, byte pitch, byte velocity) {
  midiEventPacket_t noteOn = {0x09, 0x90 | channel, pitch, velocity};
  MidiUSB.sendMIDI(noteOn);
}

/*
 * *******************MAIN LOOP*******************
 */

void loop() {

 /* **************Capactive touch to MIDI**************
  *
  *  When touched the capacitive readings display wide erratic values.
  *  While untouched, the readings are fairly constant.
  *  To get simple On/Off switch type results, The reading differences are averaged.
  */

 for (int chan = 0; chan < numChannels; ++chan ) {  // number of Touch controls

    capacitiveRead = CircuitPlayground.readCap(pads[chan]);

    total[chan]= total[chan] - readingDiffs[chan][readIndex]; //subtract oldest diff
    readingDiffs[chan][readIndex] = abs(capacitiveRead - lastReading[chan]); //calculate diff
    total[chan] +=readingDiffs[chan][readIndex];  //add latest diff to running total

    lastReading[chan] = capacitiveRead; //save reading for next diff calculation
    average = total[chan]/numReadings;  //calculate average value of Cap reading differences


        if (average > diffSensitivity)  {  //readings all over the place = touched
          if (padValues[chan] != 1) {
  //         Serial.print(pads[chan]);
  //         Serial.print(" - ");
  //         Serial.print(average);

            noteOn(chan, 1 + pads[chan], 1 );
            MidiUSB.flush();
            padValues[chan] = 1;
  //         Serial.println(" - Note on");
            delay(10);
          } // end of if padValues
        }   // end of if average


        else {                          // fairly constant Cap readings = not touched
          if (padValues[chan] == 1) {
   //        Serial.print(pads[chan]);
   //        Serial.print(" - ");
  //         Serial.print(average);

            noteOn(chan, 1 + pads[chan], 0);
            MidiUSB.flush();
            padValues[chan] = 0;
    //       Serial.println(" - Note off");
            delay(10);
          } //end of if padValues


  } // end of else
  } // end of for

    ++readIndex;
    if (readIndex >= numReadings){ readIndex = 0; }

/*
 * *****************Switches to MIDI******************
 */
    if (CircuitPlayground.rightButton() && !rightButtonPressed) {
        rightButtonPressed = true;
        noteOn(5, 0, 1 );
        MidiUSB.flush();
    //    Serial.println("Right Button On");
    } else if (!CircuitPlayground.rightButton() && rightButtonPressed){
        rightButtonPressed = false;
        noteOn(5, 0, 0);
        MidiUSB.flush();
    //    Serial.println("Right Button Off");
    }

   if (CircuitPlayground.leftButton() && !leftButtonPressed) {
        leftButtonPressed = true;
        noteOn(6, 0, 1 );
        MidiUSB.flush();
    //    Serial.println("Left Button On");
   } else if (!CircuitPlayground.leftButton() && leftButtonPressed){
        leftButtonPressed = false;
        noteOn(6, 0, 0);
        MidiUSB.flush();
   //     Serial.println("Left Button Off");
    }

   if (CircuitPlayground.slideSwitch() && !slideSwitchPressed) {
        slideSwitchPressed = true;
        noteOn(7, 0, 1 );
```

```
        MidiUSB.flush();
   //     Serial.println("Slide Switch to Right");
   } else if (!CircuitPlayground.slideSwitch() && slideSwitchPressed){
        slideSwitchPressed = false;
        noteOn(7, 0, 0);
        MidiUSB.flush();
    //    Serial.println("Slide Switch to Left");
    }

   if (digitalRead(2) && !phoneButtonPressed) {
        phoneButtonPressed = true;
        noteOn(4, 0, 1 );
        MidiUSB.flush();
   //     Serial.println("Phone Button Off");
   } else if (!digitalRead(2) && phoneButtonPressed){
        phoneButtonPressed = false;
        noteOn(4, 0, 0);
        MidiUSB.flush();
    //    Serial.println("Phone Button On");
    }


/*
 * *******************Analog to MIDI*************
  rotaryRead = analogRead(A4);
  rotaryTotal = rotaryTotal - rotaryReadings[rotaryIndex];
  rotaryReadings[rotaryIndex] = rotaryRead;
  rotaryTotal += rotaryRead;

  ++rotaryIndex;
  if (rotaryIndex >= rotaryNumReadings){ rotaryIndex = 0; }

  if (int(rotaryTotal/rotaryNumReadings) != rotaryAverage){
    rotaryAverage = int(rotaryTotal/rotaryNumReadings);
    noteOn(8, rotaryAverage >> 7, rotaryAverage & 127);
    MidiUSB.flush();
 //   Serial.print("Rotary Dial  ");
 //   Serial.println(rotaryAverage, DEC);
  }

 //**********************************************

  lightRead = CircuitPlayground.lightSensor();
  lightTotal = lightTotal - lightReadings[lightIndex];
  lightReadings[lightIndex] = lightRead;
  lightTotal += lightRead;

  ++lightIndex;
  if (lightIndex >= lightNumReadings){ lightIndex = 0; }

  if (int(lightTotal/lightNumReadings) != lightAverage){
    lightAverage = int(lightTotal/lightNumReadings);
    noteOn(9, lightAverage >> 7, lightAverage & 127);
    MidiUSB.flush();
 //   Serial.print("Light Sensor  ");
 //   Serial.println(lightAverage, DEC);
  }

 //**********************************************
/*
   // Take 10 milliseconds of sound data to calculate
  soundRead = CircuitPlayground.mic.soundPressureLevel(10);
  //soundRead = CircuitPlayground.soundSensor();
  soundTotal = soundTotal - soundReadings[soundIndex];
  soundReadings[soundIndex] = soundRead;
  soundTotal += soundRead;

  ++soundIndex;
  if (soundIndex >= soundNumReadings){ soundIndex = 0; }

  if (int(soundTotal/soundNumReadings) != soundAverage){
    soundAverage = int(soundTotal/soundNumReadings);
    noteOn(10, soundAverage >> 7, soundAverage & 127);
    MidiUSB.flush();
    Serial.print("sound Sensor  ");
    Serial.println(soundAverage, DEC);
  }
*/
//****************Accelerometer: TBD*****************

 // Accelerometer: TBD
  X = CircuitPlayground.motionX() + 20;
  Y = CircuitPlayground.motionY() + 20;
  Z = CircuitPlayground.motionZ() + 20;

  noteOn(13, 0, X);
  noteOn(14, 0, Y);
  noteOn(15, 0, Z);
/*
  Serial.print("X: ");
  Serial.print(X);
  Serial.print("  Y: ");
  Serial.print(Y);
  Serial.print("  Z: ");
  Serial.println(Z);
*/


//*********Receive MIDI to Play With LED PIXELS**************

// Pixel control from MIDI Note-On commands (rx.header == 9)
// Pixel 8-bit data (0 to 255) -- 7 bits in byte3, 1 bit in byte2 (LSB)
// Pixel operations set by MIDI Channel value in byte1
// Warning!! PureData MIDI Channel values are 1+ (1-16) the case values here (0-15)


 rx = MidiUSB.read();

      if (rx.header == 9) {               //if Note-On received
/*
        Serial.print("Received: ");
        Serial.print(rx.header, DEC);
        Serial.print("-");
        Serial.print(rx.byte1, DEC);
        Serial.print("-");
        Serial.print(rx.byte2, DEC);
        Serial.print("-");
        Serial.println(rx.byte3, DEC);
*/
        switch (rx.byte1 & 15){  //the MIDI Channel bits control what happens


          case 0:     //set red
            if (rx.byte2 >= 64) { red = random(255); }
            else {
            red = ((rx.byte2 & 1) * 128) + rx.byte3; }
            break;
```

```
case 1:     //set green
  if (rx.byte2 >= 64) { green = random(255); }
  else {
  green = ((rx.byte2 & 1) * 128) + rx.byte3; }
  break;
case 2:     //set blue
  if (rx.byte2 >= 64) { blue = random(255); }
  else {
  blue = ((rx.byte2 & 1) * 128) + rx.byte3; }
  break;

case 3:     // set color = red
  if (rx.byte2 >>7) { red = random(255); }
  else { red = 255; }
  blue = 0;
  green = 0;
  break;
case 4:    // set color = blue
  if (rx.byte2 >= 64) { blue = random(255); }
  else { green = 255; }
  red = 0;
  blue = 0;
  break;
case 5:     // set color = green
  if (rx.byte2 >= 64 { green = random(255); }
  else { blue = 255; }
  green = 0;
  red = 0;
  break;

case 6:    // set all 3 colors from 4 bit values
  if (rx.byte2 >= 64) {
    green = random(255);
    red = random(255);
    blue = random(255);
    }
    else {
      red = rx.byte3 << 4;
      green = rx.byte3 + ((rx.byte2 & 1) * 128);
      blue = (rx.byte3 & 30) << 3;
    }
    break;

case 7:   // load RGB element of color matrixes.
          // 7 bit RGB values for each of 10 pixels
    if ((rx.byte2 & 3) == 0) {     //redx matrix
      redx[rx.byte2 >> 6][(rx.byte2 >> 2) & 15] = rx.byte3 < 1;
    }
    if ((rx.byte2 & 3) == 1) {      //greenx matrix
      greenx[rx.byte2 >> 6][(rx.byte2 >> 2) & 15] = rx.byte3 < 1;
    }
    if ((rx.byte2 & 3) == 2) {      //bluex matrix
      bluex[rx.byte2 >> 6][(rx.byte2 >> 2) & 15] = rx.byte3 < 1;
    }
    break;

case 8:  //load all 10 Pixels from one of 2 color matrixes
    matrix = rx.byte2 >> 6;
    for (int x = 0; x < 10; x++) {
    CircuitPlayground.setPixelColor(x, redx[matrix][x],
    greenx[matrix][x], bluex[matrix][x]); }
    break;

case 9:   // load Random colors
```

```
          for (int x = 0; x < 10; x++) {
          CircuitPlayground.setPixelColor(x, random(255),
          random(255), random(255)); }
          break;

     case 10:  //load Random variations of colors
          for (int x = 0; x < 10; x++) {
          CircuitPlayground.setPixelColor(x, (red & 0xF0) + random(0xF),
          (green & 0xF0) + random(0xF), (blue & 0xF0) + random(0xF)); }
          break;

     case 11:
          break;

     case 12:     //load one Pixel
         CircuitPlayground.setPixelColor((rx.byte3 & 15), red, green, blue);
         break;

     case 13:    //load any number of Pixels with red, green, blue values
         xxx = rx.byte3 + (rx.byte2 << 7);
         for (int x = 0; x < 10; x++) {
         if (bitRead(xxx, x)) {
         CircuitPlayground.setPixelColor(x, red, green, blue);
         }
         }
        break;

     case 14:    //load all 10 Pixels with same color
         for (int x = 0; x < 10; x++) {
        CircuitPlayground.setPixelColor(x, red, green, blue); }
        break;

     case 15:    //clear all 10 Pixels
        CircuitPlayground.clearPixels();
        break;

   }  //end of switch

  } //end of if rx.header


}   // end of loop
```

Pure Data Sketch



Raspberry Pi Loads

Sending files like the above PureData Sketch from the Mac VNC Viewer app to the Raspberry Pi:

- Set up your VNC session with the Raspberry Pi.
- Find the dropdown menu at the top center of the VNC Viewer window.
- Select the "two arrow" (—>) file transfer indicator.
- In the box that comes up click on the "Send Files" button.
- A File Selection Dialog box will show up where you can choose the file to send to the Raspberry Pi.

Installing Arduino for the Raspberry Pi:

- On the Raspberry Pi Web Browser, go to the Arduino site.
- Download the "Linux ARM 32 bits" file.
- Open the Archiver app.
- Select the Menu Archiver/Open,
- Find the downloaded Linux ARM file (in the Downloads folder) and open it.
- Select the Menu Action/Extract with "All Files" and "With Paths" selected (default).
- Go to the Terminal app and run the following to install GUI Menu paths and such.

```
cd Downloads
ls
cd arduino-1.8.9   ( or whatever version you have )
sudo ./install.sh
```

- The Arduino app should then appear under the Programming Menu.

- From the Arduino Library Manager, install Arduino IDE libraries for both the Circuit Playground Express and MIDIUSB.

https://learn.adafruit.com/adafruit-circuit-playground-express/set-up-arduino-ide