SECRETS OF THE UNIVERSE

AN INTERACTIVE ART INSTALLATION AND PERFORMANCE
CONTROLLED WITH RASPBERRY PI

NICHOLAS SQUIRES

ARBITRARYY.COM

2015

Secrets of the Universe

Secrets of the Universe (SOTU) was an art installation and musical performance that explored various topics in physics and cosmology. The opening exhibition and performance was the culmination of Simonne Jones' 6-month artist's residency at Platoon Kunsthalle in Berlin, Germany, August 2013.

Secrets of the Universe was a series of six technology infused mixed media paintings that created a unique, interactive visual arts and musical experience. I was commissioned by Simonne to provide a system that allowed her and observers to interact with her paintings. I created a system that was motion-activated, LED lit, and wirelessly controlled hardware and software system built on the Raspberry Pi platform.

This article describes how the Raspberry Pi was used to control the SOTU lighting system as an example of what an exceptional, low-cost computer it provides for interactive arts and musical applications.

Figure 1 – "Secrets of the Universe". Platoon Kunsthalle, Berlin, DE 8/2013

System Operation and Requirements

Conceptually, the system was simple with just two modes of operation: Exhibition and Performance. In Exhibition Mode, LED lights attached to the paintings were to activate when observers came within a predefined proximity of each painting. In Performance Mode, Simonne would use the system as an interactive, real-time lighting tool during her musical performance where she could activate the lights on the paintings using her musical instruments.

Simonne and I first defined the system and operating mode requirements as described in the following sections. The details of the lighting and color schemes, trigger devices, etc. were subsequently defined during the integration phase.

System Operating Modes

Exhibition Mode: Observer Interaction with the Paintings

During the exhibition phase of the show people were allowed to traverse the stage and interact with the artwork. When they entered the Activation Zone (AZ), a configurable area in front of painting, its lights would illuminate and run through a configurable lighting scheme until they exited this zone. If an observer stepped within a configurable Warning Zone (WZ) of a painting, the lights would erratically flash red until they exited this area.

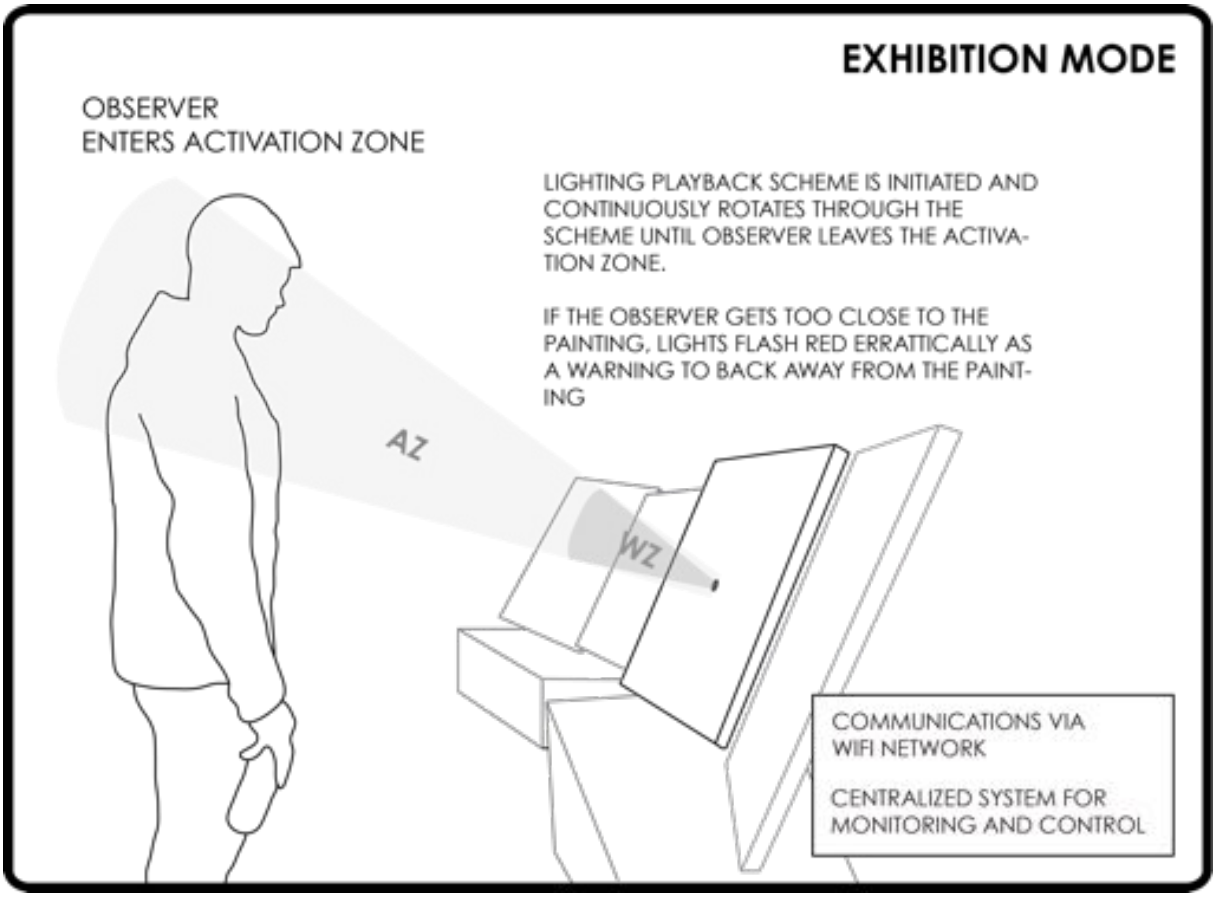

Figure 2 - Exhibition mode operational diagram

In exhibition mode SOTU functioned in accordance with the following requirements:

1> The painting's LED lights shall activate a predefined lighting scheme when an observer is within a given range in front of a painting and turn off completely when no observer is present in this area
2> The system shall be capable of creating and playing back custom lighting schemes
3> The LED lights on each painting shall enter "warning mode", by erratically flashing red, when a user is within one foot of a painting

People stood in front of the paintings, stepped in and out of the activation zone, waved their hands in front of the sensor to activate the warning mode, or just stood in front of the painting for minutes on end and marveled at the painting accentuated by its mesmerizing lighting display.

Performance Mode: Live Musical Interaction with the Paintings

In the performance phase of the show, the paintings were to operate seamlessly as part of Simonne's live musical performance where she would use them as a performance tool, to illuminate the paintings by dancing in front of them and activate them to the rhythm of her music using her array of musical instruments

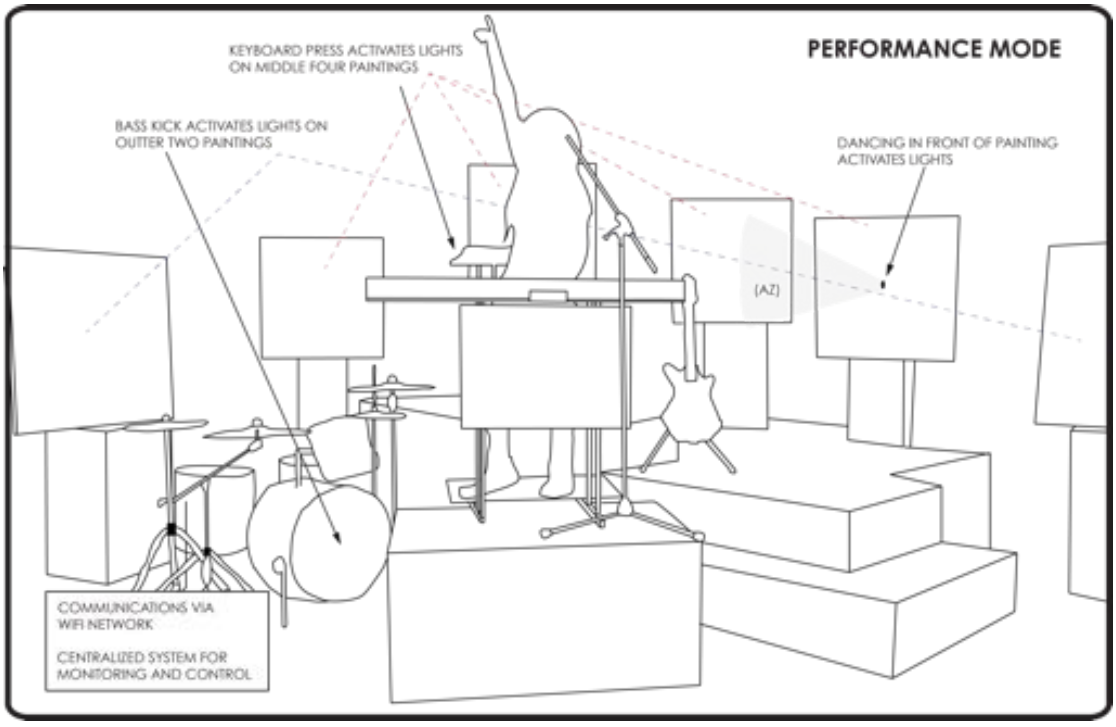

Figure 3 - SOTU performance mode operational diagram

During this phase of the show, SOTU was set into performance mode and the system operated in accordance with the following requirements:

1> The LED lights on each painting shall be capable of being independently activated via key pushes on a MIDI keyboard and when the bass drum is kicked
2> The LED lights on each painting shall be capable of being independently activated when within 2 feet of any given painting

Functional Requirements

With the system's operational concept complete, I next developed the following system-level requirements.

1> The system shall operate in Exhibition and Performance modes as specified in the exhibition and performance mode requirements
2> The system shall have a centralized *Monitoring and Control System ("Control Center" CC)* that provides a Graphical User Interface (GUI) for central control and monitoring of each system
3> Each painting shall house an independently powered and controlled RGB LED lighting system accessible over a WiFi network via Secure Shell (SSH), a web application, or the monitoring and control system
4> Each painting shall contain a sensor with sufficient accuracy to determine an observer's distance from a painting when they are within 10 feet of it
5> The system shall use Open Sound Control (OSC) as its primary communications protocol
6> The system shall be capable of accepting and converting MIDI messages to OSC messages
7> Each painting shall be independently controllable using external MIDI/OSC subsystems
8> Each painting shall house an LCD screen capable of displaying the painting's full name
9> Each painting shall report its health and operational status to a centralized monitoring and control system at a configurable time interval during all operating modes
10> The software shall be able to be updated remotely over the SOTU wireless network
11> The system shall provide the capability to programmatically create custom lighting schemes
12> The system shall provide an end-user web interface (accessible via a wireless local area network) that provides the user basic lighting controls

SOTU System Overview

The Raspberry Pi has captured the attention of computer enthusiasts since its release. Along with microcontrollers like Arduino, it has lowered the entry barriers for those with the desire to venture into the world of physical computing. The Raspberry Pi's

computing power and Linux operating system coupled with Arduino's real time processing capabilities create a small, low cost, mighty computing duo that was ideal for this application. A diagram of the SOTU system is shown below.

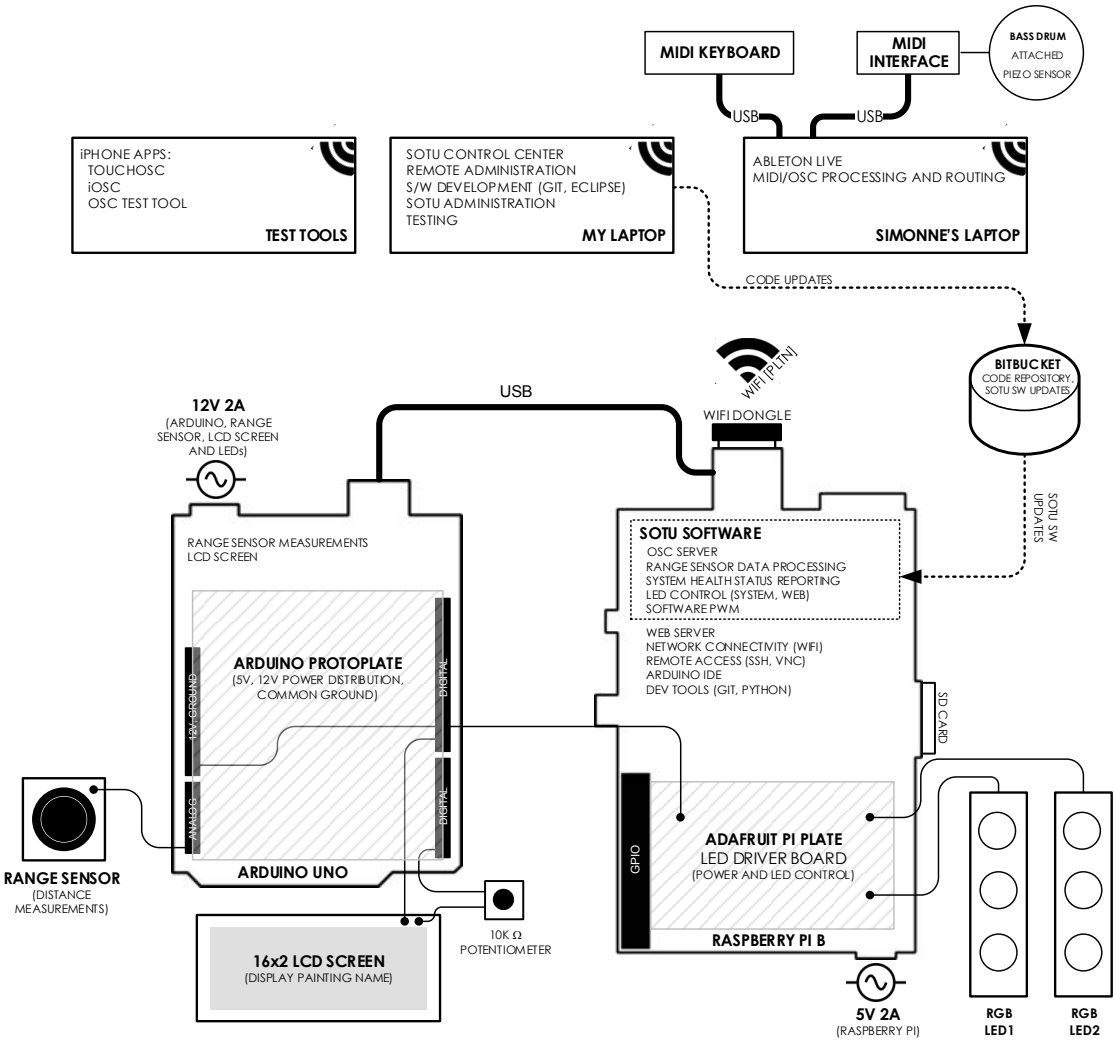

Figure 4 - SOTU system overview

Hardware

Each SOTU system was composed of the following components:

- 1 x Raspberry Pi Model B
- 1 x Adafruit Pi Plate
- 1 x Arduino Uno R3
- 1 x Edimax USB 802.11n/g/b WiFi Dongle
- 1 x 4GB SDHC Card Class 4
- 1 x 10K potentiometer

- 1 x Sparkfun Arduino Protoplate
- 1 x Maxbotix LV-EZ1 Range Sensor
- 1 x Sparkfun 16x2 LCD Display
- 2 x Analog RGB LED Strips (30 LED/per strip)
- 1 x 12V 2A Power supply (Arduino, LCD, Range Sensor, and LED power)
- 1 x 5V 2A Power Supply (Raspberry Pi, WiFi dongle power)

An assembled unit is shown below.

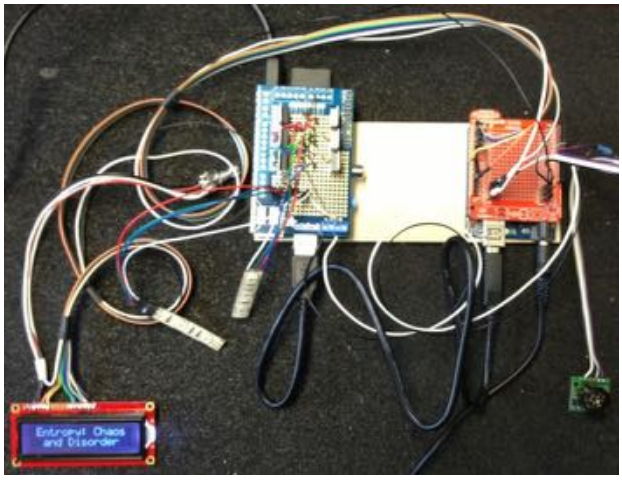

Figure 5 - Assembled SOTU computing unit

Raspberry Pi

The Raspberry Pi supported the majority of the system functions.

1> Open Soud Control (OSC) Server – Primary communications software. Processed all incoming OSC messages.
2> Webserver – Apache httpd. Served web based LED control web pages.
3> Remote Access – SSH and VNC. SSH for remote software development on the Raspberry Pi and VNC for development in the Arduino IDE.
4> WiFi – Wireless access to the SOTU WiFi network
5> LED Strip Control – Software Pulse Width Modulation (PWM) for LED control

LED Driver Board

The LED driver board was a circuit of transistors and resistors arranged onto an Adafruit Pi-Plate. This attached to the Raspberry Pi I/O panel and was the connection point for the LED strips. There were separate circuits for each LED strip that were both connected to the common system ground.

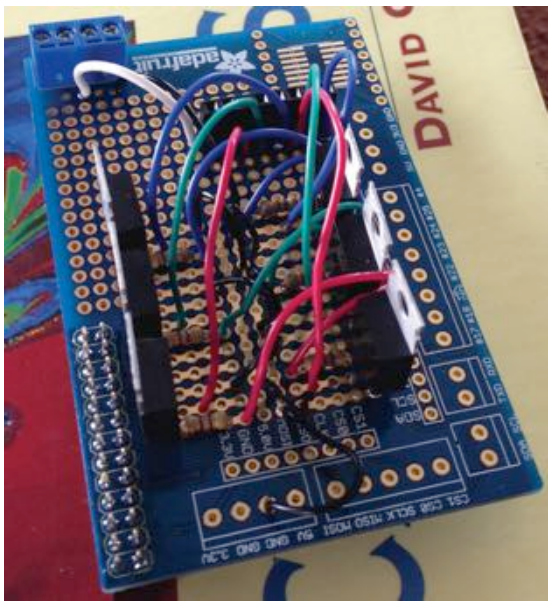

Figure 6 - LED driver board

Arduino and Protoshield

The Arduino and its attached protoshield implemented the following functions:

1> Range sensing and distance measurements – Measured the distance of an observer from the front of the painting.
2> 12V and 5V power distribution – Powered the Arduino, Raspberry Pi, LEDs, LCD screen, and range sensor
3> LCD Screen – Displayed the painting name

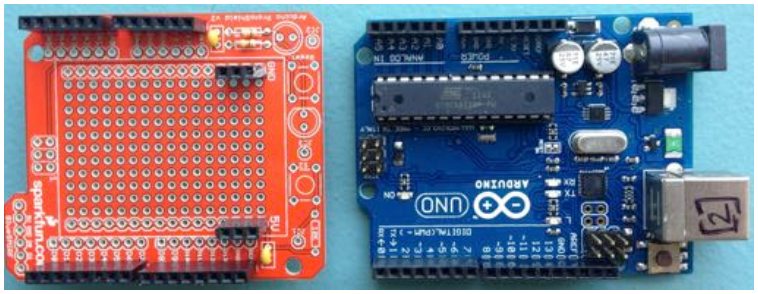

Figure 7 Arduino and Protoshield

Range Sensor and LCD Screen

Holes were cut in the canvas of each painting where the range sensor (center of painting) and LCD screen (bottom right) were then placed. The range sensor was used

to determine an observer's distance from the painting. The LCD screen displayed the name of the painting.

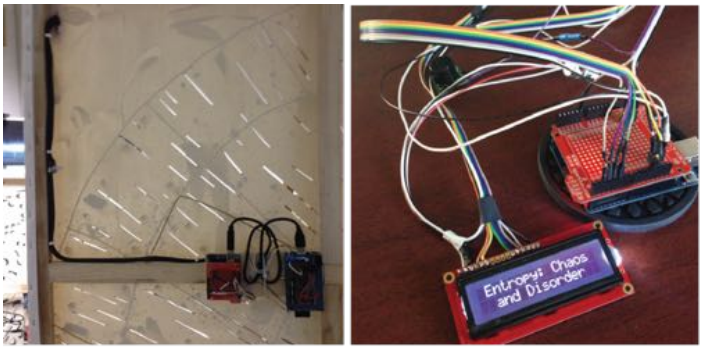

Figure 8 – Left: SOTU computing hardware mounted to the frame. Wiring was secured in mesh cable management material and fastened to the frame using wire nails. Right: LCD screen displays the painting name "Entropy: Chaos and Disorder"

Figure 9 - 16x2 LCD screen displays the painting name, "Big Bang: Birth of the Universe". Range sensor is hidden in the black paint at center of the canvas.

Software

SOTU is a software intensive, Raspberry Pi centric system. The Raspberry Pi ran the Raspbian Wheezy operating system. I wrote the majority of the code in Python and used Eclipse + PyDev as my primary development environment. Arduino software

development was done using the Arduino IDE. All of the SOTU code was maintained in Bitbucket repositories.

The SOTU software system was made up of an Arduino program, range sensor data processing, programmatic LED lighting controls and effects libraries, a communications layer, user interfaces, system health monitoring, maintenance scripts, OSS/COTS tools, test tools, and web controls. A diagram of the software architecture is shown below and will be discussed in detail in the following sections.

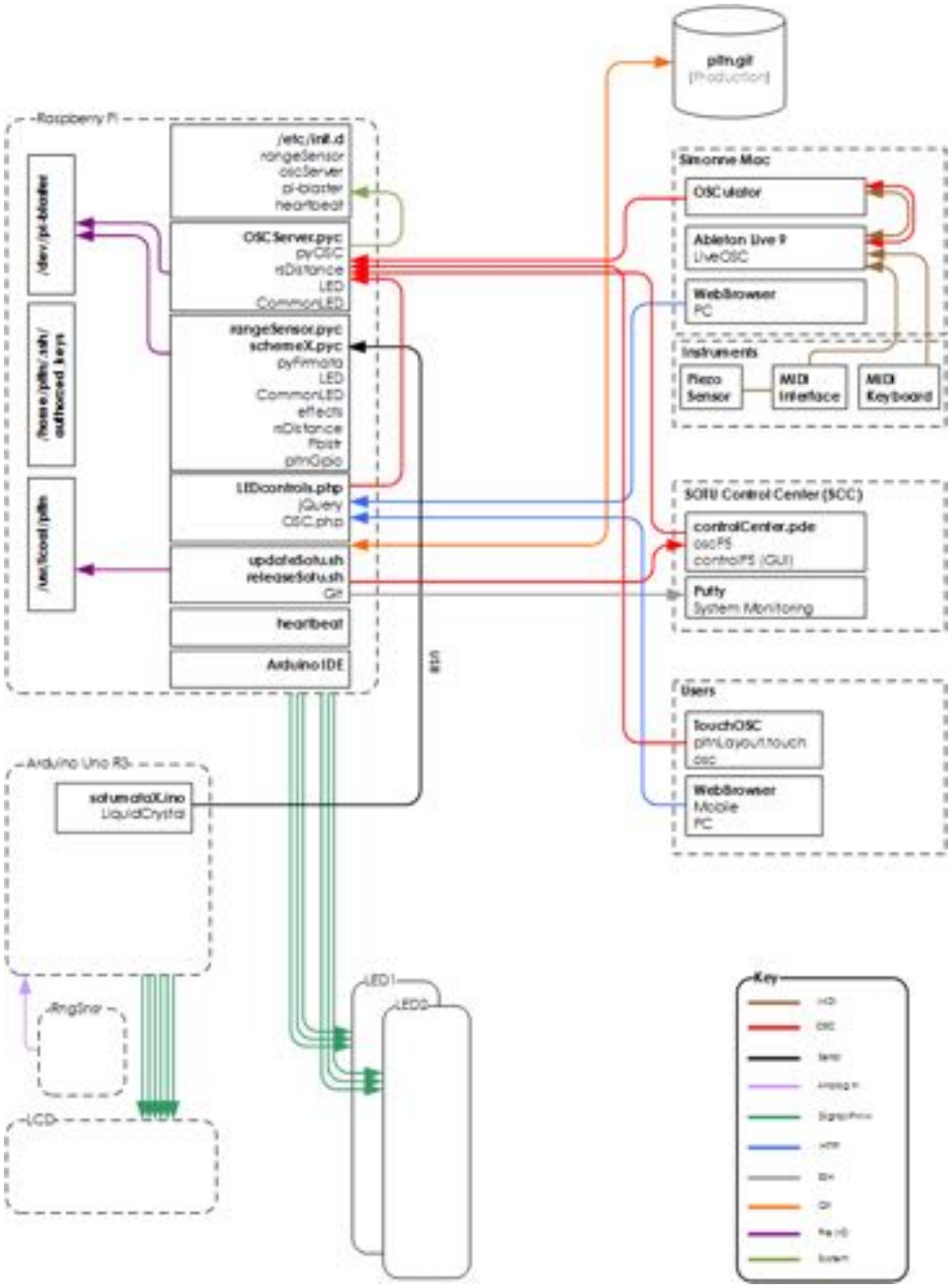

Figure 10 – SOTU software architecture

Arduino

The range sensor and LCD screen were physically connected to analog/digital pins on the Arduino. I ran a modified version of the `Firmata.ino` software; `Firmata.ino` is an Arduino implementation of the Firmata protocol that allows you to send/receive analog/digital set points to/from and Arduino over serial USB. My customizations to it were minimal; I added in the Arduino LCD libraries and custom code to display the painting names on the screens and branded it "sotumata".

Distance Measurements

The Raspberry Pi was connected to the Arduino with a USB cable that permitted bidirectional serial data flow between them. On the Raspberry Pi, I used `pyFirmata` (a python interface for the Firmata protocol) to access raw range sensor readings from the Arduino. The readings from the Arduino were voltage measurements that were converted to distance measurements on the Pi using the scaling factor specified in the Maxbotix data sheet.

During testing I observed that the range sensor had an error ratio of 1:7. I used a simple averaging function to convert these readings into a reliable source of distance measurements. The code below is from the `rsDistance` module I wrote to get these reliable range sensor readings. When `arduinoMeasure()` was invoked, *n* raw range sensor readings were taken and stored in a list, converted to distances, and then averaged using the median function from Python's `numpy` library. These range sensor measurements were then used as needed to determine an observer's presence/distance from the front of the painting.

<rangeSensor.py>

```
1.  import pyfirmata
2.  import time
3.  import numpy as np
4.
5.  #setup serial connection to Arduino
6.  board = pyfirmata.Arduino('/dev/ttyACM0')
7.
8.  # start an iterator thread so that serial buffer doesn't overflow
9.  it = pyfirmata.util.Iterator(board)
10. it.start()
11.
12. #set the number of raw range sensor readings
13. numMeasures = 10
14.
15. # Set up the RPi to read the analog values from pin 0 on the Arduino
16. pin0=board.get_pin('a:0:i')
17.
18. def arduinoMeasure():
19.         while pin0.read() is None:
20.             print "passing"
```

```
            pass
        #Calculate the distance. Uses raw voltage and scaling factor per maxbotix spec
        return distance = pin0.read()*512

def measureAvg():
        #Number of measurements to take
        dataPoints = []
        #take numMeasures measurements and put into array for avg calculations
        for num in range(0,numMeasures):
            #measure the range numMeasures times and put them into an array
            #Data sheet allows distance measurements at 50ms intervals
            distance=arduinoMeasure()
            time.sleep(0.01)
            dataPoints.append(distance)
        return avg = np.median(dataPoints)
```

Programmatic LED Control

Each painting had two RGB LED strips mounted to the inside frame of its canvas. The strips were connected to the GPIO pins on the Raspberry Pi via the LED driver board.

Pulse-width Modulation (PWM) is required to illuminate these analog RGB LEDs. PWM regulates the amount of power supplied to DC electrical devices such as analog LEDs. In spite of all of its great capabilities, the Raspberry Pi lacks sufficient native PWM capable outputs for this application with only one GPIO pin that is hardware PWM capable. Applications like this that require more than one PWM capable output can use software PWM as an alternative.

There are numerous software libraries available in the open source community for software PWM. `RPi.GPIO v0.5.2a` (a Python package, very outdated version now) and `pi-blaster` are two open source packages that enable software PWM on the GPIO pins. The Kernel in Raspbian wheezy is not intended for real-time applications; jitter is likely when using software PWM. During tests the jitter was much more prominent with RPi.GPIO versus pi-blaster, so I used pi-blaster. Note that when the pi-blaster service is running, ALL GPIO pins on the Pi are locked in as PWM outputs and cannot be configured differently.

With Pi-blaster the power supplied to a given GPIO pin is controlled by writing a decimal value of 0.0 – 1.0 to the device file `/dev/pi-blaster`. Writing a value of 0.0 for a given pin represents 0% brightness (off) while a value of 1.0 represents 100% brightness (full brightness). As an example, to set pin 2 to 20% PWM (turn connected LED on to 20% brightness) run the following command:

```
1.  echo "2=0.2" > /dev/pi-blaster
```

I wrote the following `pi-blaster` wrapper makes it easier to programmatically set the pin values

<PBlstr.py>

```
from os import system

class Pblstr:
    def __init__(self):
    #Pi-Blaster device file
    self.devFile = '/dev/pi-blaster'

    def write(self,gpioPin,val):
    """
    This function writes to the pi-blaster device file
    INPUT:
      - gpioPin = GPIO pin to write to
      - val     = Set the pin to this value
    """
    system("echo \"{0}={1}\" > {2}" .format(gpioPin,val,self.devFile))
```

A Python script or another class could control the PWM of any GPIO pin:

```
from PBlstr import PBlstr as pb

{do stuff}

pb.write(1,0.3)
```

This would set GPIO pin 1 on the Raspberry Pi to 30% PWM which would cause the LED attached to that pin would turn on to 30% brightness.

`PBlstr` was the building block for all other LED functions. For example, `CommonLED` implemented essential LED methods for converting the analog RGB values to acceptable pi-blaster values, setting individual GPIO pin values, turning all lights off/on, and setting the color for an entire LED strip. An example is the `setColor` method from `CommonLED` shown below.

<commonLED.py>

```
from decimal import Decimal,getcontext
from PBlstr import PBlstr as pb

#Set the decimal precision to two decimal places
getcontext().prec = 2

def setColor(self,RGB):
    """
    Set RGB color passed to it
    RGB - array of R, G, B values to set
    """
        i = 0
```

```
        #gpioPinsList is a configurable list of GPIO pin
        #numbers where the RGB terminals on the LED strip are connected
        for gpioVal in gpioPinsList:
        gpioVal = rgbToPb(gpioVal)
                self.pb.write(gpioVal, RGB[i])
                i += 1

def rgbToPb(self,rgbVal):
    #Scale from raw RGB values to pi-blaster allowed values
    return Decimal(rgbVal)/Decimal(255)
```

I was then able to set the RGB value for either strip on the painting as shown below.

```
from CommonLED import CommonLED as cLED

    {do stuff}

cLED.setColor(1,[255,0,0])
```

The `setColor()` function is now a fundamental capability of the system. Continuing to build upon these classes, I wrote the `effects` class that implemented the following lighting effects:

- `solid()` – Set an LED strip to an RGB value
- `fade()` – Fade from one color to another
- `rotate()` – Rotate through a predefined array of colors
- `pulse()` – Switch back and forth between two given colors at a specified rate for a specified number of iterations
- `flashFade()` – Set an LED strip to an RGB value then fade it to off

With these all of the LED classes completed I used them to create lighting playback scripts that contained instructions on how the lights would behave. As an example, the following playback script was run on the *Big Bang: Birth of the Universe* painting, which was meant to simulate the Big Bang event. This scheme would be activated when an observer entered the Activation Zone.

<bigBang.py>

```
#!/usr/bin/python

from time import sleep
from Effects import Effects as ef
from rsDistance import rsDistance

allOff     = [0,0,0]
```

```
#Dim white
startColor = [10,10,10]
#Full brightness
endColor   = [255,255,255]
#RGB colors to rotate between.  Colors taken from false-color
#representations of nebulae pictures
universeColors = [[28,30,68],[40,93,144],[123,32,144],[67,47,103]]
#Activate scheme while observer is between 10 and 30 inches in front of the painting
actionZone = [10,30]

while True:
    #Take a distance measurement. If observer is in action zone run the following to completion
    distance = rsDistance.measureAvg()
    if actionZone[0] <= distance <= actionZone[-1]:
        """fade in from dark to dim white (dim white represents universe singularity)
        """
        ef.fade(allOff,startColor,1,0)

        """pulse at dim white for 5 seconds
        """
        ef.pulse(startColor,0.2,5,5,1)

        """pause before explosion
        """
        sleep(2)

        """simulate Big Bang explosion by rapid jump to 100% full white brightness
        """
        ef.fade(startColor,endColor,49,0.01)

        """pause after expansion
        """
        sleep(5)

        #Fade to the universeColors[] and rotate through them after expansion to
        #represent the formation of galaxies, nebulae, etc
        ef.fade(endColor,universeColors[0],1,0.05)
        ef.rotate(universeColors,1,1,0.001)

        #Fade out from the last color in the universe array
        ef.fade(universeColors[-1],allOff,1,0)

        #check if observer still in AZ and if so, re-run scheme, otherwise turn allOff
        distance = rsDistance.measureAvg()
```

A video demonstration of the Big Bang playback script running on my SOTU prototype here: http://youtu.be/7-34Rvw3Aqs

Open Sound Control (OSC)

OSC is a communications protocol that enables musical instruments, Digital Audio Workstations (DAWs), computers and other multimedia tools to communicate with one another on a network. A useful implementation of the OSC protocol is a simple client server mechanism; an OSC "client" sends OSC messages, and an OSC "server" receives and processes them. With this simple communication mechanism, default integration

with modern DAWs and a great Open Source python implementation called pyOSC it was the best choice to use as the primary communication protocol.

Raspberry Pi Control Using OSC

Using pyOSC I developed oscServer, a Python program that ran on each Raspberry Pi that listened for, received and processed incoming OSC messages. Using pyOSC's callback mechanism the incoming messages executed functions in oscServer providing me the capability to execute LED lighting functions or even Raspbian system commands by simply sending the associated OSC message.

The code below illustrates my OSC server implementation. In this example, I instantiated an OSC server, added a message handler for the OSC message /osc/led, (which served as the base OSC address for LED functions) and defined a callback function led() to invoke functions in the CommonLED and Effects libraries to perform desired LED actions.

<oscServer.py>

```
1.  #Import the pyOSC OSC server libraries
2.  from OSC import OSCServer
3.  from CommonLED import CommonLED as cLED
4.  from Effects import Effects as ef
5.
6.  #Define OSC server port and traceback IP
7.  OSCPort = 4567
8.  OSCIP   = "0.0.0.0"
9.  #Instantiate server
10. oscSrv = OSCServer((OSCIP,OSCPort))
11.
12. def led(path, tags, args, source):
13.     """
14.     Process message sent from an OSC client. When server receives /osc/led
15.     Address, use the CommonLED and Effects classes to do
16.     make the lights do stuff. args is an array of the
17.     arguments passed in the OSC message so
18.     """
19.     oscProg  = args[0]
20.     #Turn LED strip 1 and 2 on to RGB 255,255,255
21.     #using the CommonLED function
22.     if oscProg == 'allOn':
23.         cLED.setColor(1,[255,255,255])
24.         cLED.setColor(2,[255,255,255])
25.
26.     if oscProg in gpioPins.keys():
27.         #Check if oscProg is a GPIO pin (as defined in gpioPins)
28.         #if it is we only want to perform an operation on a single
29.         #pin.  The pinValue is then taken and some action is applied
30.         # to it.
31.         pinValue = args[1]
32.         action   = args[2]
33.         #search gpioPins dict for pin value. Exit when found
34.         for dictColor,gpioPin in gpioPins.iteritems():
```

```
            if oscProg == dictColor:
                break
        #This calls the flash method in the Effects module
        #which sets gpioPin to high then sets to low after
        #"slp" seconds
        if action =="flash":
            ef.flash(gpioPin,slp)
        elif action == 'solid':
            cLED.setPinValue(gpioPin,pinValue)

#Message Handlers and Callback functions
oscSrv.addMsgHandler("/osc/led",led)

while True:
    #listen for OSC messages until Python script is terminated
    oscSrv.handle_request()
```

Sending the following OSC message to this server:

```
/osc/led allOn
```

would cause both strips in the painting to turn on to 100% white brightness.

As another example, to turn LED strip 1 on to 100% red brightness and strip 2 to 30% blue brightness send the following messages:

```
/osc/led r1 1 solid

/osc/led b2 0.3 solid
```

The flexibility in `pyOSC`'s callback mechanism also made it possible to control Linux system commands such as starting/stopping services or a graceful shutdown. The example below shows how I implemented remote shutdown of the Pi with OSC. Note that for (some) security `authzKey` must be sent as the second argument of the message.

`<oscServer.py>`

```
from subprocess import call

    def rpi (path, tags, args, source):
        #get the acommand to run and the authorization key from
        #the arguments passed in the OSC message
        cmd = args[0]
        key = args[1]
        #check that the off command is sent along with the proper
        #authorization key.  The authzKey is defined in a variable
        #not shown here
        if cmd == 'off' and key == authzKey:
            print "RPi received shutdown command. Shutting down now.
            #issuing Linux shutdown command
            call(["sudo", "shutdown", "-h", "now"])
        else:
            print "Bad function/Authorization Key provided"
```

```
17.
18. oscSrv.addMsgHandler("/osc/rpi",rpi)
```

To shut down the Raspberry Pi gracefully (assuming I sent the proper authzKey) I sent:

```
/osc/rpi off <authzKey>
```

which would shut it down with:

```
sudo shutdown –h now
```

An Aside on Linux Services

Those that are familiar with Debian based Linux distributions are also probably familiar with the service command. `service` allows you to run System V (SysV) init scripts that are stored in `/etc/init.d`. SysV scripts allow you to start/stop system services, check their status, process ID and enable them to run at various run-levels.

The SysV system provides the framework to extend custom scripts into Linux system services. A quick tutorial on creating an init script for a custom service is here:

http://www.stuffaboutcode.com/2012/06/raspberry-pi-run-program-at-start-up.html

And more detailed information about SysV init scripts can be found here:

http://www.debian.org/doc/manuals/debian-reference/ch03.en.html#_sysv_style_init

I wrote SysV scripts for each essential SOTU function including `rangeSensor`, `oscServer` and `heartbeat`. With these I could start, stop, or status `rangeSensor` and `heartbeat` with:

```
service <serviceName> (start|stop|status)
```

I next extended `oscServer` to accept OSC messages to control these services. The callback function below processes OSC messages of the form

```
/osc/service <serviceName> (start|stop|status)
```

`<oscServer.py>`

```
1.  def srvc(path, tags, args, source):
2.      """
3.      Callback function to handle all RPi related functions
4.          OSC Msg: /osc/srvc <srvcName> start|stop
5.      """
6.      #list of allowed services and values. Basic security
7.      #to prevent disallowed services from being controlled
8.      allowedSrvcs = ["pi-blaster","ssh", "httpd", "rangeSensor","heartbeat"]
9.      allowedCmds  = ["start","stop"]
10.     srvcName = args[0]
11.     value    = args[1]
12.     #check if this is an allowed command
```

```
13.     if srvcName in allowedSrvcs and value in allowedCmds:
14.         call(["sudo", "service", srvcName, value])
15.     else:
16.         print "{0}: \"{1} {2} {3}\" Not allowed" .format(localtime,path,srvcName,value)
```

I was then able to control any *allowed* system service (configured by `allowedSrvcs` in line 8) (i.e. `pi-blstr`, `ssh`, `httpd`, `rangeSensor`, `heartbeat`) using OSC messages.

The SOTU OSC Message Address Space

The OSC server connected external systems to the most important SOTU and Linux system functions. Abstracting these capabilities as OSC messages was essential for system command and control. The table below defines the SOTU OSC message address space.

System Function	Address	(Data Type) Argument	Description/Example
LED Control			Control LED function
	`/osc/led`	(string) **`allOff`**	Turn all lights off
		(string) **`allOn`**	Turn all lights on to 100% brightness
		(string) **CS** (float32) **B** (string) **E** (C)olor= `r`, `g`, `b` (S)trip = `1`, `2` (B)rightness = `0.0 – 1.0` (E)ffect = `solid`, `flashFade`, `flash`, `rotate`, `pulse`	Turn color **C** (red, green, blue) on strip **S** (LED Strip 1 or 2) to % brightness **B** with effect **E** Ex: **`/osc/led r1 0.4 flash`** Turns red LEDs on strip 1 to 40% brightness with flash effect
Raspbian Service Control			Control allowed services in `oscServer`
	`/osc/srvc`	(string) **S** (string) **A** (S)ervice = `pi-blaster`, `rangeSensor`, `httpd`, `ssh`, `heartbeat` (A)ction = `start`, `stop`	Start or stop an allowed service **S** Ex: **`/osc/srvc heartbeat stop`** Stops the heartbeat service; health status no longer be published to the control center
Raspbian Functions			Control other Raspbian functions (non services)
	`/osc/rpi`	(string) **A** (string) **K** (A)rgument = `off` Authorization (K)ey = `authzKey`	Gracefully turn off the Raspberry Pi when valid authorization key (K) is provided.
System Health Monitoring			Reports system health information from each Pi

/osc/heartbeat	(string) **H** (string) **S** (string) **P** (H)ostname (S)ervice Linux (P)rocess ID	While the heartbeat service is running, each painting sends its hostname, and a list of tracked services (defined in **heartbeat**) along with their associated Process IDs (PID) to the control center

Figure 11 - SOTU OSC address space

Health Monitoring and the SOTU Control Center (CC)

During early system testing it was challenging to manage and monitor the six separate systems. It was critical to be able to monitor all these systems in real-time during shows, I developed a common UI that allowed me to control and monitor all of the paintings through a single interface.

Health Monitoring

The following system health data was sent from each painting to the control center using the program **heartbeat** every 5 seconds (configurable setting):

- Linux process IDs (PID)
  - oscServer (osc)
  - ssh (ssh)
  - httpd (web)
  - health service (hst)
  - pi-blaster (pib)
- SOTU operational mode (mod): performance or exhibition
- Range sensor reading (rng) in inches

In **heartbeat**, I used the **pyOSC** client method to send the data to the OSC server that was running on the control center (see next section for CC description). The code generates heartbeat messages. An OSC client is instantiated, the OSC message is constructed with the data points listed above and sent to the control center:

<heartbeat.py>

```
1.  import subprocess
2.  from time import sleep
3.  from OSC import OSCClient, OSCMessage
4.
5.  #List of Linux services
6.  srvcs       = ["pi-blaster","oscServer",“httpd”,”ssh”,”heartbeat”]
7.  pgrepOutput = []
8.  queryTime   = 5 #Number of seconds between heartbeat check
9.  #The IP and port of the CC OSC server
10. OSCPort = 12000
11. OSCIP   = "X.X.X.X"
12.
13. #Define the heartbeat OSC address
```

```
statusAddr = "/osc/heartbeat"

#instantiate an OSC Client
srvcClient = OSCClient()
srvcClient.connect( (OSCIP, OSCPort) )

try:
    while True:
        #Get the RPis hostname
        h = subprocess.Popen(["hostname"], stdout=subprocess.PIPE)
        hOut, hErr = h.communicate()
        #For each service in array srvcs, construct the OSC message, log the output
        #Then send it to the OSC server that is running on the control center
        for srvc in srvcs:
            #Get the PIDs of each service in srvcs
            p = subprocess.Popen(["pgrep", srvc], stdout=subprocess.PIPE)
            pOut, pErr = p.communicate()
            pOut.rstrip('\n')
            #construct the OSC message with the OSC address,
            #Linux process ID, and the service name
            oscMsg = OSCMessage(statusAddr)
            oscMsg.append(hOut.rstrip('\n'))
            oscMsg.append(srvc)
            #Log the output to std Out
            if pOut:
                oscMsg.append(1) #there was a service running
                print "{0}: {1} " .format(srvc,"Running")
            else:
                oscMsg.append(0)
                print "{0}: Stopped" .format(srvc)
            #Send the OSC Message
            srvcClient.send(oscMsg)
        sleep(queryTime)

except KeyboardInterrupt:
        pass
```

SOTU Control Center (CC)

The Control Center is a user interface that I developed in Processing (a popular visual programming language) to aggregate the painting's health data and to provide some basic system controls. Processing is a popular programming language that is used for visual programming. The CC was developed using:

- ControlP5 – Graphical UI development library, sliders, buttons, etc
- oscP5 – OSC server/client library

A screenshot of the interface is shown below.

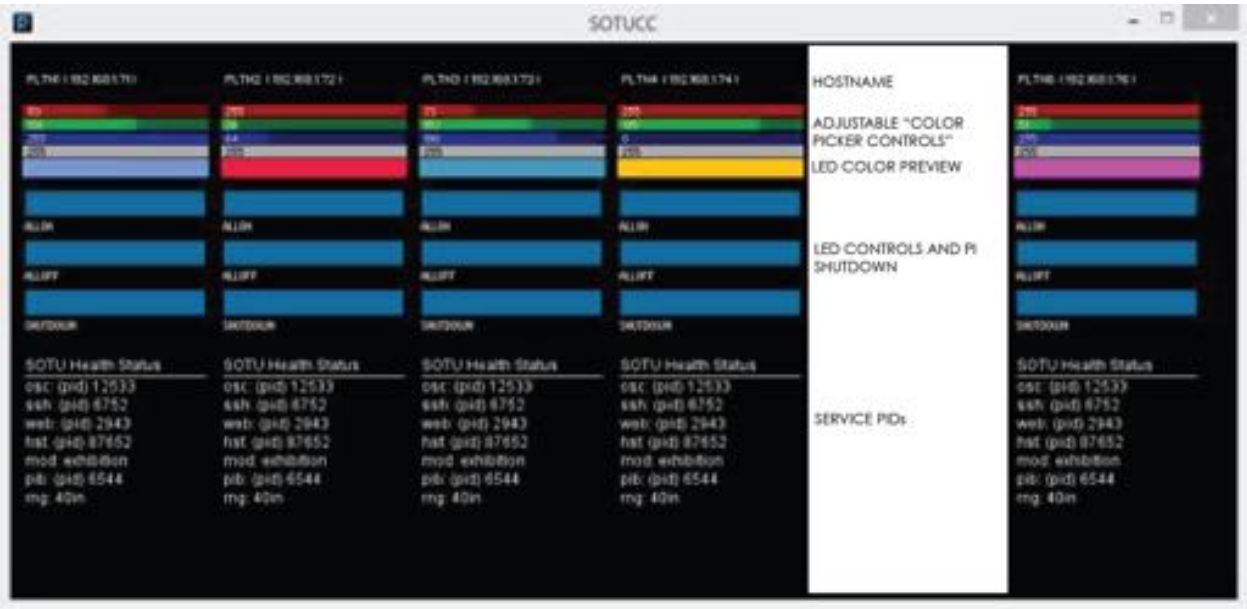

Figure 12 – CC GUI interface was built with Processing. Note this is "dummy" data for illustration purpose.

On the backend of the CC UI was an OSC server implemented with `oscP5`. This server processed the heartbeat messages from each Pi and organized them into the UI shown above. If a service went down on a painting I would be alerted, allowing me to take action to bring the system back to normal operation.

I built in a simple self-healing capability. If a service died, it would try to restart automatically. I would be alerted (at a configurable time interval) if the service restarted successfully or if it failed and I needed to take further action. If the service didn't come back online after retrying for 10 seconds I restarted the service manually using a PuTTY terminal.

I was also able to control some basic lighting and system functions with the CC. Button clicks and slider changes triggered `oscP5` client events that would send associated OSC messages to a painting.

- Color Pickers – Set the color of each painting using an RGB value. LED strip color is displayed in the preview bar below each set of sliders.
  - `/osc/led x1 <value> solid`
- AllOn – Set the LEDs on a painting to all white
  - `/osc/led allOn`
- AllOff – Turn LEDs off
  - `/osc/led allOff`
- Shutdown – Gracefully shut down the Pi
  - `/osc/rpi shutdown <authzKey>`

I didn't get around to programming in Linux service controls into the CC that would've been useful; it is a straightforward extension from what was discussed in this section.

Software Updates

I used a separate Git repository on Bitbucket for the production version of SOTU software. This repository held compiled Python programs, system init scripts, SOTU web application, Arduino software, and SOTU S/W release and S/W update scripts. Updating software on both controllers was simple and another example of the Arduino/Raspberry Pi synergy.

Raspberry Pi

At the end of a development cycle I used `releaseSOTU`, a script I wrote to generate a software build from my local clone of the development Git repository. This script compiled and copied all of the necessary Python modules (`oscServer.pyc`, `Effects.pyc`, etc) and scripts to the runtime directory `/usr/local/sotu.`

I then ran the script `updateSOTU.sh -push` to send this new release to the production Git repository; it was now available to all paintings.

I updated the other paintings using `updateSOTU.sh -pull` that simply ran a series of Git commands:

1> `push`
   a. Stage changes: `git add *, git add -u *`
   b. Commit: `git commit -m "Updating SOTU repository on ${DATE}"`
   c. Push to Master: `git push`
2> `pull`
   a. Pull latest: `git pull`

Arduino

The Arduino was connected to the Pi via USB and was configured in the Arduino IDE (installed on each Pi) as a serial device. The Arduino software was updated as follows,

1> Pull the latest release of the SOTU software using `updateSOTU.sh -pull`
2> Connect to the Pi with VNC to display back the Linux desktop environment
3> Open the latest version of the `sotumata.ino` sketch with the Arduino IDE
4> Upload the sketch to the Arduino using the IDEs upload tool

System Testing

I developed an assortment of test tools to test the systems' most critical functions. A rudimentary PERL script using the OSC library `Net::OpenSoundControl::Client` was initially sufficient to send messages to the server. I used the following script to test the addresses defined in the OSC address space.

<oscTest.pl>

```
#!C:\perl\bin\perl.exe
#
use lib 'C:\perl\lib';
use Net::OpenSoundControl::Client;

if (@ARGV != 2) {
    print "\n\tusage: perl sendOSC.pl <server IP> <port#>\n\n";
    exit;
} else {
       #Specify the server IP address and OSC listening port
    $server_ip = $ARGV[0];
    $port = $ARGV[1];
}

#Create an OSC client object
my $client = Net::OpenSoundControl::Client->new(
        Host => $server_ip, Port => $port)
        or die "could not start client: $@\n";

#Construct the OSC address and send the OSC message with required
#parameters for that address
my $osc_addr = "/osc/led";
$client->send(["$osc_addr" , 's', "g2", 'i', 0, 's', "solid"]);
```

This script became challenging to use with the growing complexity of the OSC address space and parallel OSC server testing. I needed a tool that was easier to swap between OSC server IP addresses, one that provided a simple means to program in multiple OSC commands and one that was mobile friendly. After trying various iOS applications including iOSC, OSC Test Tool, and Control, I decided to try TouchOSC.

TouchOSC is an OSC server/client iOS application that functions on a WiFi network. TouchOSC *Editor* is a free tool used to build custom UIs for TouchOSC. It boasts controls such as buttons, sliders and dials that can each be individually programmed to send OSC messages. I developed the interface shown below

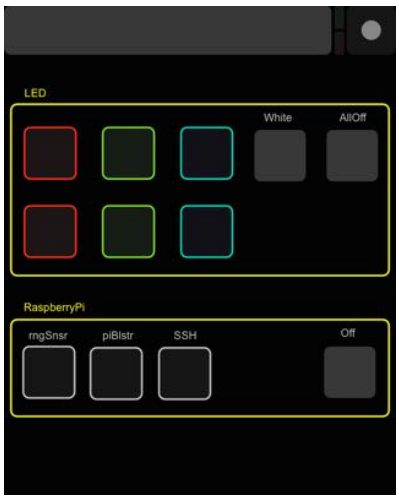

Figure 13 - TouchOSC SOTU system testing UI. LED section of buttons controlled on/off associated colors for strip 1 and 2, turn all LEDs on/off to full white. Raspberry Pi section of buttons could turn the range sensor on/off, turn SSH or Pi-Blaster Linux services on/off, or turn the Pi itself off

TouchOSC was not the perfect testing tool for SOTU. First, it didn't have the capability to send messages to multiple servers concurrently. However, it was much easier to switch between OSC servers than it was using the PERL script. Second, it wasn't capable of sending multiple arguments in an OSC message. Again, with the growing complexity of the system, I eventually grew out of TouchOSC, and ended up using a mix of test tools and scripts.

Web-based Controls

Each painting was required to be independently controllable over a wireless network. Up to this point I was only able to control them using the CC or test tools. Neither of these control methods was practical for an end user of the paintings (i.e. future painting owner).

The most practical way to implement this functionality was to develop a web application hosted by an Apache httpd server running on the Raspberry Pi. Using PHP, jQuery and a PHP implementation of OSC called OSC.php I built the simple painting control web application shown below.

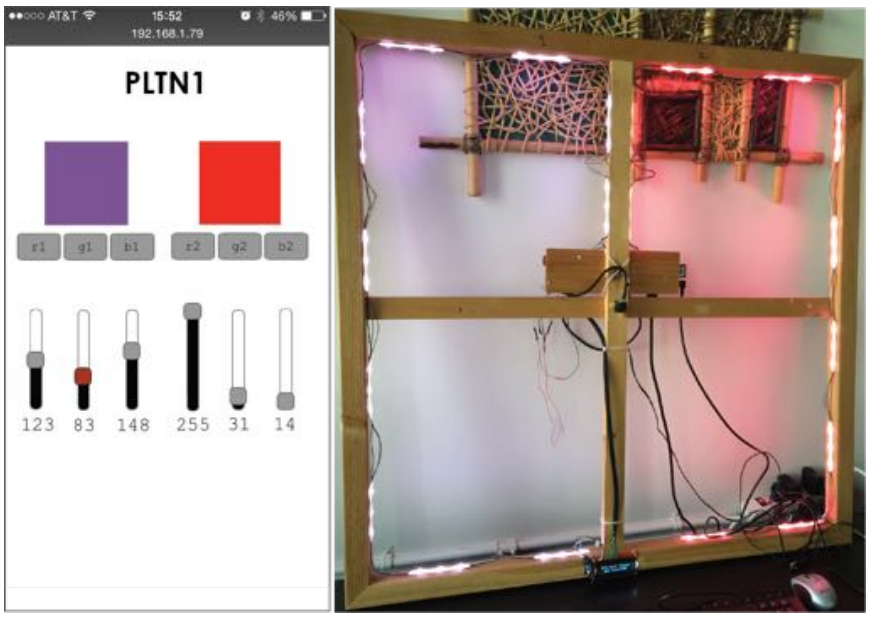

Figure 14 - Left: Web-based LED controller on an iPhone web browser. Right: lights change color based on web application

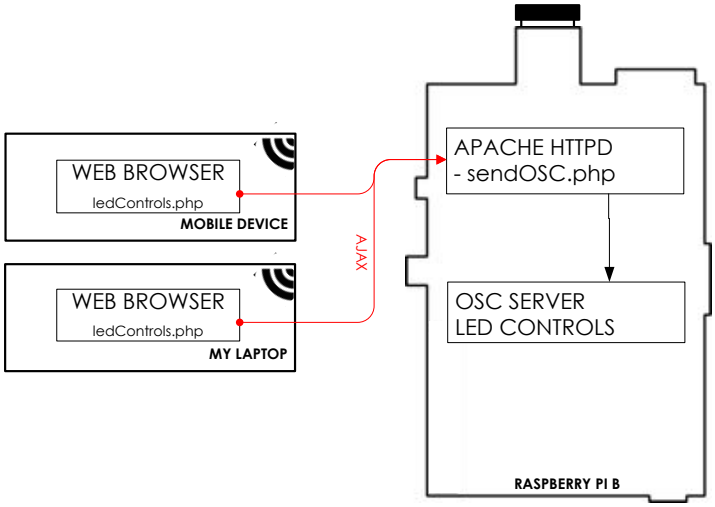

Figure 15 – Web LED application architecture. LEDs on each painting could be adjusted by accessing the ledControl.php web page individually served on each Raspberry Pi

The sliders in the UI, `ledControls.php`, implemented jQuery's AJAX methods to send red, green, and blue slider values to `sendOSC.php`. `sendOSC.php` then sent the OSC messages to the OSC server as illustrated above.

In the code snippet below, values for the red LEDs on a given strip are acquired from the web painting control interface via a `POST`, an OSC client was implemented using the

OSC.php library, and finally an OSC message was constructed using these values and sent to the OSC server for processing.

<sendOSC.php>

```
1.  <?php
2.  require_once("lib/OSC.php");
3.
4.  #values from POST
5.  echo "----";
6.  echo $_POST['rVal'];
7.  $rVal = $_POST['rVal'];
8.
9.  #Create OSC client object and parameters
10. $c = new OSCClient();
11. $c->set_destination("127.0.0.1", port);
12. $oscAddress = "/osc/led";
13.
14. #Create OSC message object
15. $r = new OSCMessage($oscAddress, array($rStrip, $rVal, "solid"));
16.
17. #Send the message
18. $c->send($r);
19.
20. ?>
```

These web-based controls eliminated the need to have access to the CC or the test tools for LED control. The LEDs on the paintings were then controllable using a web browser on any computer or mobile device connected to the same network as the painting.

System Prototype

I was developing the SOTU system thousands of miles away from the exhibition space. In order to minimize integration issues and amount of work required upon my arrival at the Platoon exhibition hall, I built a fully functioning replica of a painting. I constructed a 1m x 1m frame and mounted all of the SOTU components to it in a similar configuration to the production system.

This prototype allowed me to measure exact lengths of wires, determine required sizes of nuts and bolts, precut and predrill mounting holes, solder electrical components, determine optimal placement of the LED strips, LCD screen, and range sensors, and also test and optimize the range sensor's algorithms in its operational position.

Figure 16 - SOTU system prototype

This preparation minimized the amount of risk involved with integrating the SOTU system for the first time at Platoon and resulted in more system integration time to perfect interoperability with her live performance.

SOTU Production and Performance Integration

Production

It took about two months of procurement, assembly and testing in my home lab to have all six units ready. I had to purchase and receive all of the parts, solder all of electrical components, install operating systems and SOTU software, complete functional testing of each assembled unit, and finally label and safely package everything into boxes.

Figure 17 - Top: Assembling and testing LCD screens and LED driver boards. Bottom: Assembling SOTU units and readying for functional test

A month prior to the show I packed everything into a carry-on luggage and was on my way to Berlin.

Once in Berlin, I set up shop alongside Simonne's painting operation in the Artist Lab at Platoon where we assembled the paintings.

Figure 18 - Simonne and I working in the Platoon Artist Lab

In between Simonne's press events, we managed to test each painting, integrate SOTU with her musical instruments, and make other configuration tweaks to accommodate the environment in the exhibition space.

Performance Integration

Simonne uses the Ableton Live Digital Audio Workstation (DAW) with an assortment of MIDI controllers to trigger samples, effects and instruments during her live performance.

Ableton Live integrated seamlessly into SOTU since it can expose its internal MIDI and OSC communication making the MIDI note for key presses on her keyboard readily available. This made it easy to siphon off those messages and process them with external devices and software tools.

Getting MIDI messages from her keyboard in Ableton was simple since it was a directly connected to her computer. To send a MIDI message to Ableton when the bass drum was kicked, we connected a pressure sensor to it and connected the other end to a MIDI interface, which was then connected to her laptop. We added the MIDI interface to her Ableton session and were then able receive MIDI on/off messages each time the bass drum was kicked. Now both her MIDI keyboard and drum set were configured to send messages to Ableton.

Next I needed to translate these MIDI messages to representative OSC messages that could then be routed to the OSC servers on each Pi. Instead of developing my own

MIDI/OSC routing tool, I chose to use OSCulator. OSCulator isn't open source, but is available at a very reasonable price for all of its capabilities. It boasts an intuitive user interface, great documentation, has the ability to send OSC messages to multiple OSC servers (e.g. send commands to each painting) and integrates natively with Ableton Live. The MIDI channels can be configured to output directly to OSCulator.

Once Ableton Live was configured to send the MIDI output from Simonne's instruments to OSCulator it was trivial to configure OSCulator to then route the messages to any of the paintings for LED control of the paintings.

Below is a sample of the OSCulator configuration window:

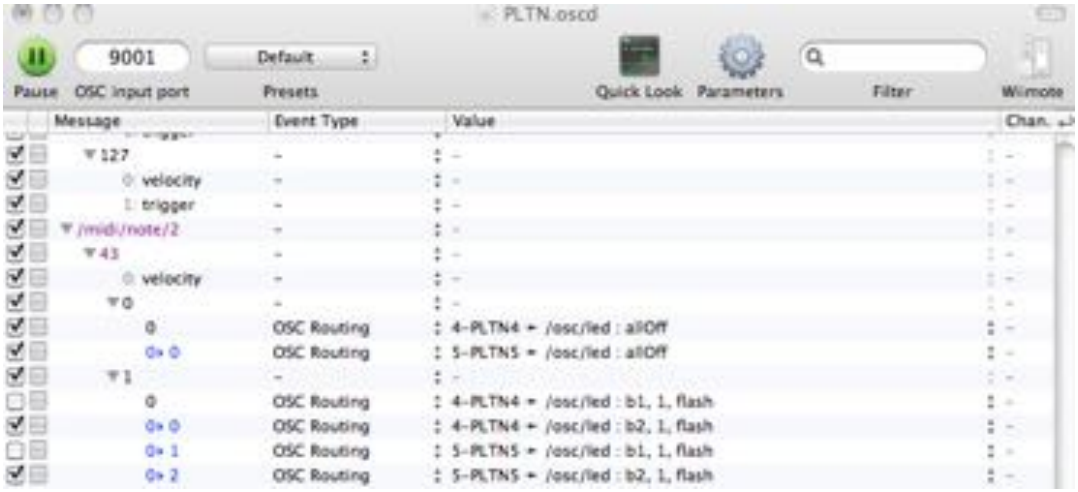

Figure 19 - OSCulator. The MIDI input from Ableton along with the routed OSC messages to paintings 4 and 5 (i.e. PLTN4 and PLTN5)

In this example, OSCulator is listening on port 9001 for any MIDI/OSC messages. If the key corresponding to MIDI note 43 is pressed on her keyboard, a value of 1 is sent. When the key is released a value of 0 was sent. In the example below, OSCulator is configured to send OSC messages

```
/osc/led b1 1 solid
```

and,

```
/osc/led b2 1 solid
```

to paintings 4 and 5 (hostnames PLTN4 and PLTN5 respectively); this would then turn both paintings blue. When the key was released on the keyboard, OSCulator would receive a value of 0 with MIDI note 43 and then send this OSC message to turn them off completely.

```
/osc/led allOff
```

The system was set into the following configuration during the performance:

- Light a painting up blue when she passed within 2 feet of any of the four central paintings
- Light a painting up pink, purple, or red when a designated key is pressed on her MIDI keyboard
- Activate blue lights on the paintings at the ends of the stage when the bass drum was kicked

A diagram of the system operating in performance mode is shown below.

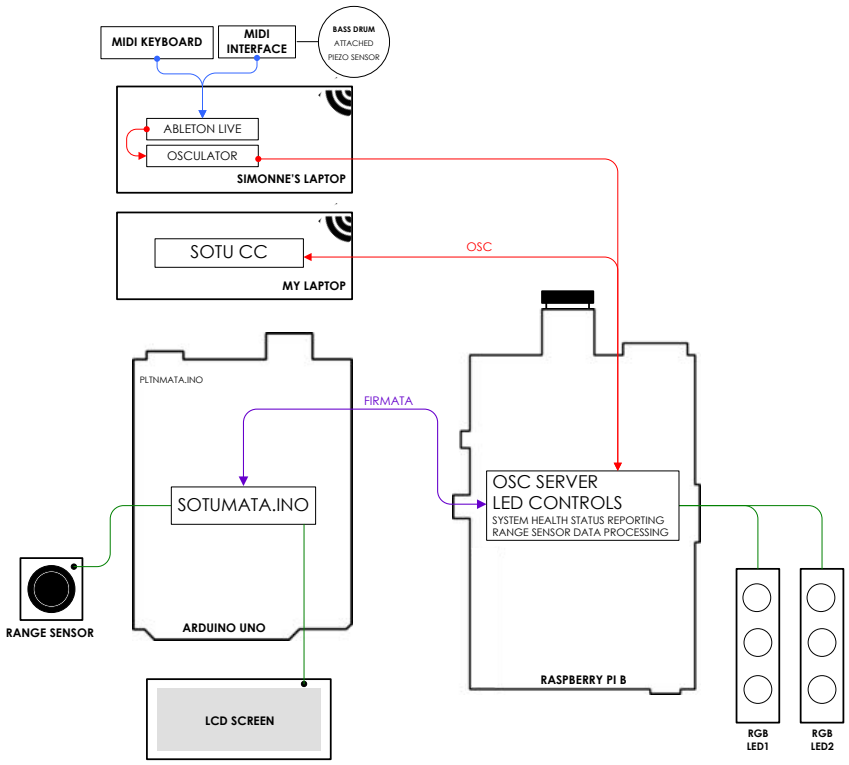

Figure 20 - OSC/MIDI message flow in performance mode. Blue lines represent MIDI, red, OSC, purple, Firmata, green, GPIO output for LED activation and Analog output for range sensor readings

For comparison a diagram for the data flow during exhibition mode is shown below.

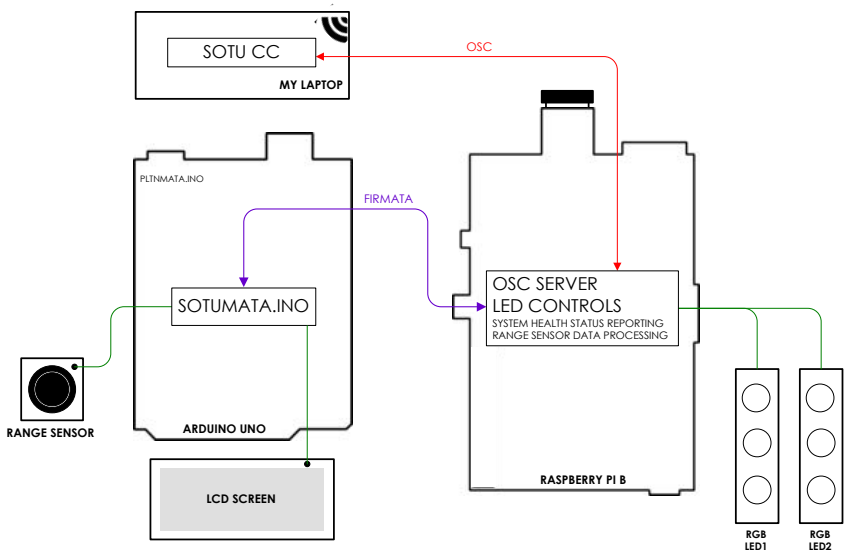

Figure 21 – Data flow during exhibition mode. Purple lines - Firmata messages over serial USB, green - GPIO output for LED activation and Analog output for range sensor readings, red - OSC.

Showtime

The paintings were displayed in a semi-circle on a stage made of milk crates that resembled a Q-bert playing field, which served as the layout for the exhibition and the stage for Simonne's live musical performance.

Figure 22 - Painting configuration on stage

Figure 23 - Simonne Jones plays her paintings during the performance at Platoon Kunsthalle

Within the following month Secrets of the Universe was also exhibited at the Berlin Remake Festival, at the Berlin Arts and Music Festival, and used during her performance in Bremen, Germany.

Figure 24 - Observers watch as the paintings act out their lighting programs at the 2013 Berlin Remake Festival

Figure 25 - Simonne Jones performs with her paintings in Bremen Germany. In this arrangement all of the paintings are triggered by her drummer's kick drum.

Listen to Simonne describe her philosophy and artistic vision for Secrets of the Universe in her interview for 3SAT TV (Germany) here: http://vimeo.com/76454151

Conclusion

SOTU performed incredibly well throughout all of the exhibitions and performances, and we did not experience a single system crash. While the Raspberry Pi is typically touted as a great hobby computer, I can attest to the fact that it is also excellent for interactive art and performance applications. I attribute a large part of our success with this project the robustness of the Raspberry Pi.

Building a system with this level of complexity is a problem filled with interface, logistic, development, operational, testing and deployment challenges. I endured all of these challenges while building SOTU with the result of me honing existing skills and acquiring many new ones. Well played Raspberry Pi Foundation, mission accomplished.

SOTU is the type of problem that unites creators from various disciplines and one that excites us as all engineers. It was an incredible project to be a part of and infinity to the power of infinity thanks go out to Simonne Jones for allowing me to help bring her vision to life.