Table of Contents

Overview 3
- What is Processing?
- Prerequisite Guides
- Parts

Circuit Diagram 7

Installing Processing on a Raspberry Pi 8
- Prepare Your Raspberry Pi
- Processing

Coding the Raspberry Pi Video Synth 10
- Install the Required CircuitPython Libraries
- Download the Code and Image Files
- Running the Python and Processing Code

How the CircuitPython Code Works 16
- Socket Setup
- The Loop
- Update the NeoSlider NeoPixels
- Reading the Rotary Encoder
- Reading the NeoSliders
- Reading the VL53L4CD

How the Processing Code Works 19
- A Lot of Variables
- The Setup
- Reading the Socket
- Changing Animations
- The Animations

Constructing the Mounting Plate 27
- Template Ideas

Assembly 30
- Cut the NeoSlider's I2C Address Jumpers
- Connect the STEMMA Boards
- Connect to the Raspberry Pi
- Mount the Components
- Assembly Complete!

Usage 38
- Going Further
Overview

Video synthesizers can create fun and funky visual effects on screens. In this project, you'll learn how to use Processing and Blinka, the CircuitPython compatibility library for single-board computers running Linux, to display animations and control them with hardware.

STEMMA boards communicate over I2C, which makes interfacing with the Raspberry Pi simple.
There are four animations included in the Processing sketch. You can change which one is actively playing by using the rotary encoder.

You can change the colors and attributes of each animation with the NeoSliders and VL53L4CD time of flight sensor.

What is Processing?

Processing is a piece of open-source software designed for beginners to code visuals with Java. It can run on a variety of platforms, including Windows and macOS. In this project, you will be running it on a Raspberry Pi 4.
Prerequisite Guides

There are a few guides that will come in handy while you're working on this project.

- CircuitPython Libraries on Linux and Raspberry Pi
- Adafruit NeoSlider
- Adafruit I2C QT Rotary Encoder
- Adafruit VL53L4CD Time of Flight Distance Sensor

Parts

**Raspberry Pi 4 Model B - 4 GB RAM**
The Raspberry Pi 4 Model B is the newest Raspberry Pi computer made, and the Pi Foundation knows you can always make a good thing better! And what could make the Pi 4 better...

[https://www.adafruit.com/product/4296](https://www.adafruit.com/product/4296)

**Adafruit NeoSlider I2C QT Slide Potentiometer with 4 NeoPixels**
Our family of I2C-friendly user interface elements grows by one with this new product that makes it plug-n-play-easy to add a 75mm long slide potentiometer to any microcontroller or...

[https://www.adafruit.com/product/5295](https://www.adafruit.com/product/5295)
Adafruit I2C Stemma QT Rotary Encoder Breakout with NeoPixel
Rotary encoders are soooo much fun! Twist em this way, then twist them that way. Unlike potentiometers, they go all the way around and often have little detents for tactile feedback....
https://www.adafruit.com/product/4991

Adafruit VL53L4CD Time of Flight Distance Sensor - ~1 to 1300mm
The Adafruit VL53L4CD Time of Flight Sensor is another great Time of Flight distance sensor from ST in the VL5 series of chips, this one is great for...
https://www.adafruit.com/product/5396

STEMMA QT / Qwiic JST SH 4-pin Cable with Premium Female Sockets
This 4-wire cable is a little over 150mm / 6" long and fitted with JST-SH female 4-pin connectors on one end and premium female headers on the other. Compared with the chunkier...
https://www.adafruit.com/product/4397

4 x STEMMA QT / Qwiic JST SH 4-Pin Cable
STEMMA QT / Qwiic JST SH 4-Pin Cable
50 mm STEMMA QT cable
https://www.adafruit.com/product/4399

1 x Black Nylon Machine Screw and Stand-off Set – M2.5 Thread
M2.5 screws and stand-offs
https://www.adafruit.com/product/3299
1 x Official Raspberry Pi Micro HDMI to HDMI Cable
- 1 meter long
Micro HDMI to HDMI Cable

1 x Official Raspberry Pi Power Supply 5.1V 3A with USB C - 1.5 meter long
Raspberry Pi Power Supply 5.1V 3A with USB C

1 x SD/MicroSD Memory Card (8 GB SDHC)
MicroSD Memory Card

Circuit Diagram

This project consists of STEMMA boards that are connected to each other using STEMMA QT cables. QT to QT cables are used for the NeoSliders, rotary encoder and VL53L4CD. A QT to socket cable is used to connect to the Raspberry Pi.

The NeoSlider addresses will be set on the Assembly Page.

- NeoSlider 0x30
  - SCL (yellow wire) to NeoSlider 0x31 SCL
  - SDA (blue wire) to NeoSlider 0x31 SDA
  - VIN (red wire) to NeoSlider 0x31 VIN
  - GND (black wire) to NeoSlider 0x31 GND

- NeoSlider 0x31
  - SCL (yellow wire) to NeoSlider 0x32 SCL
  - SDA (blue wire) to NeoSlider 0x32 SDA
  - VIN (red wire) to NeoSlider 0x32 VIN
  - GND (black wire) to NeoSlider 0x32 GND
• NeoSlider 0x32
  ◦ SCL (yellow wire) to Rotary Encoder SCL
  ◦ SDA (blue wire) to Rotary Encoder SDA
  ◦ VIN (red wire) to Rotary Encoder VIN
  ◦ GND (black wire) to Rotary Encoder GND

• Rotary Encoder
  ◦ SCL (yellow wire) to VL53L4CD SCL
  ◦ SDA (blue wire) to VL53L4CD SDA
  ◦ VIN (red wire) to VL53L4CD VIN
  ◦ GND (black wire) to VL53L4CD GND

• VL53L4CD
  ◦ SCL (yellow wire) to Pi SCL
  ◦ SDA (blue wire) to Pi SDA
  ◦ VIN (red wire) to Pi 3V
  ◦ GND (black wire) to Pi GND

---

Installing Processing on a Raspberry Pi

[Diagram of Processing software]

Welcome:
Welcome to this page documenting how to use the Processing software on the Raspberry Pi and similar Linux-based single-board computers.

Prepare Your Raspberry Pi

First, install the latest version of the Raspberry Pi OS Bullseye onto an SD card. You can refer to the [CircuitPython on Linux and Raspberry Pi](#) guide for more help setting it up.
Once you have everything set up, you will need to open a terminal and install Blinka. Refer to the [Installing CircuitPython Libraries on Raspberry Pi](#) page to quickly get up and running.

**Processing**

On your Raspberry Pi, open the Chromium browser and navigate to the [Download](#) page of the [Processing for Pi](#) website.

At the bottom of the Download page, you will see a TGZ archive for Processing 3.5.3. Click on it to begin the download.

After downloading, move the .tgz file to your home directory. Open a new terminal window, and enter the following command:

```
tar xvfz processing-3.5.3-linux-armv6hf.tgz
```

This extracts the archive package that you downloaded and installs Processing onto your Raspberry Pi.

You can run Processing from the terminal with these commands:

```
cd ~/processing-3.5.3/
./processing
```
Coding the Raspberry Pi Video Synth

Install the Required CircuitPython Libraries

You will need to have a few libraries installed before the script will run on your Raspberry Pi.

Install the required CircuitPython libraries with the terminal:

```
pip3 install adafruit-circuitpython-seesaw
pip3 install adafruit-circuitpython-vl53l4cd
pip3 install adafruit-circuitpython-simpleio
```
Download the Code and Image Files

Once you've finished setting up your Raspberry Pi with Blinka and the library dependencies, you can access the Python code, Processing code, and two .png image files by downloading the Project Bundle.

To do this, click on the Download Project Bundle button in the window below. It will download as a zipped folder.

```python
# SPDX-FileCopyrightText: 2022 Liz Clark for Adafruit Industries
# SPDX-License-Identifier: MIT

import socket
import time
import board
import simpleio
import adafruit_vl53l4cd
from adafruit_seesaw import seesaw, rotaryio, neopixel
from adafruit_seesaw.analoginput import AnalogInput

# VL53L4CD setup
i2c = board.I2C()  # uses board.SCL and board.SDA
# i2c = board.STEMMA_I2C()  # For using the built-in STEMMA QT connector on a microcontroller
vl53 = adafruit_vl53l4cd.VL53L4CD(i2c)

# rotary encoder setup
encoder = seesaw.Seesaw(i2c, addr=0x36)
encoder.pin_mode(24, encoder.INPUT_PULLUP)
rot_encoder = rotaryio.IncrementalEncoder(encoder)

# neoslider setup - analog slide pot and neopixel
# 0x30 = red control
# 0x31 = green control
# 0x32 = blue control
red_slider = seesaw.Seesaw(i2c, 0x30)
red_pot = AnalogInput(red_slider, 18)
r_pix = neopixel.NeoPixel(red_slider, 14, 4)

g_slider = seesaw.Seesaw(i2c, 0x31)
green_pot = AnalogInput(g_slider, 18)
g_pix = neopixel.NeoPixel(g_slider, 14, 4)

b_slider = seesaw.Seesaw(i2c, 0x32)
blue_pot = AnalogInput(b_slider, 18)
b_pix = neopixel.NeoPixel(b_slider, 14, 4)

# rotary encoder position tracker
last_position = 0

# neoslider position trackers
last_r = 0
last_g = 0
last_b = 0

# VL53L4CD value tracker
last_flight = 0

# rotary encoder index
x = 0
```
# VL53L4CD start-up
vl53.inter_measurement = 0
vl53.timing_budget = 200

vl53.start_ranging()

# HTTP socket setup
s = socket.socket()
print("socket check")

port = 12345
s.bind(("", port))
print("socket binded to %s" % (port))

s.listen(1)
print("listening")
time.sleep(10)

c, addr = s.accept()
print('got connected', addr)

while True:
    # reset the VL53L4CD
    vl53.clear_interrupt()

    # rotary encoder position read
    position = -rot_encoder.position

    # VL53L4CD distance read
    flight = vl53.distance

    # mapping neosliders to use 0-255 range for RGB values in Processing
    r_mapped = simpleio.map_range(red_pot.value, 0, 1023, 0, 255)
    g_mapped = simpleio.map_range(green_pot.value, 0, 1023, 0, 255)
    b_mapped = simpleio.map_range(blue_pot.value, 0, 1023, 0, 255)

    # converting neoslider data to integers
    r_pot = int(r_mapped)
    g_pot = int(g_mapped)
    b_pot = int(b_mapped)

    # set neopixels on neosliders to match background color of Processing animations
    r_pix.fill((r_pot, g_pot, b_pot))
    g_pix.fill((r_pot, g_pot, b_pot))
    b_pix.fill((r_pot, g_pot, b_pot))

    # rotary encoder position check
    if position != last_position:
        # rotary encoder is ranged to 0-3
        if position > last_position:
            x = (x + 1) % 4
        elif position < last_position:
            x = (x - 1) % 4
        # send rotary encoder value over socket
        # identifying string is "enc"
        c.send(str.encode(' '.join(["enc", str(x)])))
        # reset last_position
        last_position = position

    # sliders only update data for changes >15 to avoid flooding socket
    # red neoslider position check
    if abs(r_pot - last_r) > 2:
        # send red neoslider data over socket
        # identifying string is "red"
        c.send(str.encode(' '.join(["red", str(r_pot)])))
        # reset last_r
        last_r = r_pot

©Adafruit Industries
# green neoslider position check
if abs(g_pot - last_g) > 2:
    # send green neoslider data over socket
    # identifying string is "green"
    c.send(str.encode(' '.join(['green', str(g_pot)])))
    # reset last_g
    last_g = g_pot

# blue neoslider position check
if abs(b_pot - last_b) > 2:
    # send blue neoslider data over socket
    # identifying string is "blue"
    c.send(str.encode(' '.join(['blue', str(b_pot)])))
    # reset last_b
    last_b = b_pot

# VL53L4CD value check

# setting max value of 45
if flight > 45:
    flight = 45
last_flight = flight
if abs(flight - last_flight) > 2:
    print(flight)
    # send VL53L4CD data over socket
    # identifying string is "flight"
    c.send(str.encode(' '.join(['flight', str(flight)])))
    # reset last_flight
    last_flight = flight

After downloading the Project Bundle, unzip the folder. Your Raspberry Pi should have the following files in the Raspberry_Pi_Video_Synth folder:

- BlinkaRaspberryPiVideoSynth.py
- Raspberry_Pi_Video_Synth.pde
- cat.png
- pizza.png
Downloading with `wget`

You can also use the terminal to access the files. You can create a new directory called `Raspberry_Pi_Video_Synth` and use `wget` to access the files individually from GitHub.

```
mkdir Raspberry_Pi_Video_Synth
cd Raspberry_Pi_Video_Synth
```

Running the Python and Processing Code

Open Thonny, or your preferred Python IDE, on your Raspberry Pi. Open the `BlinkaRaspberryPiVideoSynth.py` file.

Open Processing 3 on your Raspberry Pi. Then, open the `Raspberry_Pi_Video_Synth.pde` file with the Processing IDE.
The Python and Processing files are communicating with each other through a socket. A socket is a port that is opened between two nodes that allows for sending data.

In this case, the Python script is sending data from the NeoSliders, rotary encoder and VL53L4CD to the Processing script. The Processing script's animations are then affected by these inputs.

Run the BlinkaRaspberryPiVideoSynth.py in the IDE. When the script begins running, the socket is opened and waits for a connection.

You should see *listening* print out to the REPL. This lets you know that the socket has been opened and is waiting for the Processing script to connect.

Then, run the Raspberry_Pi_Video_Synth.pde script in the Processing IDE.

After the socket connection is established, you'll see the Processing animations running full screen on your Raspberry Pi. You should be able to change parameters with the various STEMMA board inputs.
How the CircuitPython Code Works

Before the loop, the NeoSliders, rotary encoder, and VL53L4CD are setup.

```python
# VL53L4CD setup
vl53 = adafruit_vl53l4cd.VL53L4CD(board.I2C())

# rotary encoder setup
encoder = seesaw.Seesaw(board.I2C(), addr=0x36)
encoder.pin_mode(24, encoder.INPUT_PULLUP)
rot_encoder = rotaryio.IncrementalEncoder(encoder)

# neoslider setup - analog slide pot and neopixel
# 0x30 = red control
# 0x31 = green control
# 0x32 = blue control
red_slider = seesaw.Seesaw(board.I2C(), 0x30)
red_pot = AnalogInput(red_slider, 18)
red_pix = neopixel.NeoPixel(red_slider, 14, 4)

red_slider = seesaw.Seesaw(board.I2C(), 0x31)
green_pot = AnalogInput(g_slider, 18)
green_pix = neopixel.NeoPixel(g_slider, 14, 4)

b_slider = seesaw.Seesaw(board.I2C(), 0x32)
blue_pot = AnalogInput(b_slider, 18)
blue_pix = neopixel.NeoPixel(b_slider, 14, 4)

# rotary encoder position tracker
last_position = 0

# neoslider position trackers
last_r = 0
last_g = 0
last_b = 0

# VL53L4CD value tracker
last_flight = 0
```
x = 0

# VL53L4CD start-up
vl53.inter_measurement = 0
vl53.timing_budget = 200
vl53.start_ranging()

Socket Setup

This is followed by the setup for the socket. The socket allows communication between the Python code and the Processing code.

First, `s` is setup as the socket object. Then, a port is created called 12345. It can be any number, since this is a virtual port that only exists to link the Python and Processing scripts.

The socket begins listening on port 12345 using `s.listen(1)`. While the socket is listening, the Processing code will be able to connect to the Python code.

Once a connection is established on the port, `got connected` is printed to the REPL. Then, you’ll see the Processing animations begin to run full screen on the Raspberry Pi. The Python script also begins running the loop.

```python
# HTTP socket setup
s = socket.socket()
print("socket check")
port = 12345
s.bind(('', port))
print("socket binded to %s" %port)

s.listen(1)
print("listening")
time.sleep(10)

c, addr = s.accept()
print('got connected', addr)
```

The Loop

`position` and `flight` are setup to hold the incoming values from the rotary encoder and the VL53L4CD.

The NeoSliders have a default value range of 0 to 1023. The `map_range()` function is used to change their range to 0 to 255. This allows them to send RGB
values directly to the Processing sketch since they will be affecting the background color of each animation.

```python
# reset the VL53L4CD
vl53.clear_interrupt()

# rotary encoder position read
position = -rot_encoder.position

# VL53L4CD distance read
flight = vl53.distance

# mapping neosliders to use 0-255 range for RGB values in Processing
r_mapped = simpleio.map_range(red_pot.value, 0, 1023, 0, 255)
g_mapped = simpleio.map_range(green_pot.value, 0, 1023, 0, 255)
b_mapped = simpleio.map_range(blue_pot.value, 0, 1023, 0, 255)

# converting neoslider data to integers
r_pot = int(r_mapped)
g_pot = int(g_mapped)
b_pot = int(b_mapped)
```

**Update the NeoSlider NeoPixels**

The NeoSliders' NeoPixels are set to match the background color of the Processing animations.

```python
# set neopixels on neosliders to match background color of Processing animations
r_pix.fill((r_pot, g_pot, b_pot))
g_pix.fill((r_pot, g_pot, b_pot))
b_pix.fill((r_pot, g_pot, b_pot))
```

**Reading the Rotary Encoder**

The rotary encoder controls which animation is active in the Processing script. `c.send()` sends the rotary encoder's position over the socket to Processing.

The data is encoded as a string with `str.encode()`. `enc` is sent at the first part of the string so that Processing can identify that it is receiving data from the rotary encoder. You can think of it as sending a variable name. `x` holds the encoder's position and is sent as the second part of the encoded string.

```python
# rotary encoder position check
if position != last_position:
    # rotary encoder is ranged to 0-3
    if position > last_position:
        x = (x + 1) % 4
    if position < last_position:
        x = (x - 1) % 4
    # send rotary encoder value over socket
    # identifying string is "enc"
    c.send(str.encode(' '.join(["enc", str(x)])))
```
Reading the NeoSliders

Each slider is setup to compare the current value to the previous value. When that value changes above a certain threshold, the new value is sent over the socket.

Similar to the rotary encoder, each NeoSlider has an identifying string label that acts as a variable in Processing.

```python
# sliders only update data for changes &gt; 15 to avoid flooding socket
# red neoslider position check
if abs(r_pot - last_r) &gt; 15:
    # send red neoslider data over socket
    # identifying string is "red"
    c.send(str.encode(' '.join(["red", str(r_pot)])))
    # reset last_r
    last_r = r_pot
```

Reading the VL53L4CD

The `flight` and `last_flight` are setup to compare the current value to the previous value of the VL53L4CD. When that value changes above a certain threshold, the new value, `flight`, is sent over the socket.

A maximum value of 45 is setup so that the range of expected data from the VL53L4CD can be setup in the Processing script. Just like the other hardware interfaces, the VL53L4CD has an identifying string label, `flight`, that acts as a variable in Processing when its data is sent over the socket.

```python
# VL53L4CD value check
if abs(flight - last_flight) &gt; 2:
    # setting max value of 45
    if flight &gt; 45:
        flight = 45
    # send VL53L4CD data over socket
    # identifying string is "flight"
    c.send(str.encode(' '.join(["flight", str(flight)])))
    # reset last_flight
    last_flight = flight
```

How the Processing Code Works

The Processing code begins by importing the Network library and setting up the HTTP client as `myClient`. This is part of the setup for communication over the socket with the Python script.
import processing.net.*;

// HTTP client
Client myClient;

A Lot of Variables

Before the setup, the variables for the various animations are declared. Their purpose is commented in the code.

// variables for receiving data from Blinka socket
int index;
String inString;

// holding the red, green, blue data
String[] r;
String[] g;
String[] b;
int red;
int green;
int blue;

// holding the VL53L4CD data
String[] f;
int flight;

// cat and pizza images
//emojis are from the OpenMoji emoji library (https://openmoji.org/)
PImage cat_img;
PImage pizza_img;

// colors for Circles animation
color c_red = color(255, 0, 0);
color c_green = color(0, 255, 0);
color c_blue = color(0, 0, 255);
color c_yellow = color(255, 125, 0);
color c_aqua = color(0, 125, 255);
color c_purple = color(255, 0, 255);

IntList colors;

// float for Cube animation
float i = 0.0;

// variables for Circles animation
int rad = 60;
float xpos, ypos;

float xspeed = 2.8;
float yspeed = 10;

int xdirection = 1;
int ydirection = 1;

// variables for pizzaCat animation
int pizzaCount = 32;
int catCount = 32;
int emojiCount = 32;

PImage[] cats = new PImage[catCount];
PImage[] pizzas = new PImage[pizzaCount];
float[] moveX = new float[emojiCount];
The Setup

To run the animations full screen, `fullScreen()` is called with `P3D`. P3D is a renderer in Processing that allows for a z-axis parameter. The Cube animation requires this.

An integer list called `colors` is used in the Circles animation. `colors.append()` is called to add the colors to the list that were defined before `setup()`.

The `loadImage()` function loads the .png image files that are used in the Pizza Cat animation.

Three `for` statements are used to setup the Pizza Cat animation. The `cat_img` and `pizza_img` are loaded into arrays and the starting coordinates and speeds for each emoji instance is setup.

Finally, `myClient` is setup to connect to the socket on port `12345`.

```java
void setup()
{
  // setting animations to run fullscreen with P3D
  fullScreen(P3D);
  // RGB color mode with value range of 0-255
  colorMode(RGB, 255);
  ellipseMode(RADIUS);
  // setting xpos and ypos in center
  xpos = width/2;
  ypos = height/2;

  // creating array of colors
  // this is used for the Circles animation
  colors = new IntList();
  colors.append(c_red);
  colors.append(c_yellow);
  colors.append(c_green);
  colors.append(c_aqua);
  colors.append(c_blue);
  colors.append(c_purple);
```
// loading the cat and pizza images
cat_img = loadImage("cat.png");
pizza_img = loadImage("pizza.png");

// adding pizza and cat emojis to their arrays
for (int slice = 0; slice < 15; slice++) {
pizzas[slice] = pizza_img;
}
for (int claw = 16; claw < catCount; claw++) {
cats[claw] = cat_img;
}

// creating arrays of coordinates and speed for pizzaCat
for (int z = 0; z < emojiCount; z++) {
x_dir[z] = random(width);
y_dir[z] = random(height);
x_speeds[z] = random(5, 20);
y_speeds[z] = random(5, 20);
moveX[z] = x_speeds[z];
moveY[z] = y_speeds[z];
}

// connecting to socket to communicate with Blinka script
myClient = new Client(this, "127.0.0.1", 12345);

Reading the Socket

In Processing, **draw()** is the loop. The code contained in **draw()** will run forever.

**draw()** begins by checking if any data is coming in over the socket. **inString** is used to hold the data coming in over the socket as a string. If you recall in the Python code, the data from the hardware is being encoded as a string before it is sent over the socket.

```processing
// void draw() is the loop in Processing
void draw()
{
  //if data is coming in over the socket...
  if (myClient.available() > 0) {
    //string data is stored in inString
    inString = myClient.readString();
  }
```

Right now, the string coming over the socket is one long string. For example, if the rotary encoder was set to **2**, then the string would come in as "**enc 2**". For this information to be useful for Processing, the **2** needs to be parsed out from the string and changed to an integer.

An **if** statement is used to check if **inString** begins with the string variable from the Python script. If it's a match, then a **String[]** object is setup. **splitTockens()** is used to slice **inString** into separate strings separated by a space. Going back to the example of "**enc 2**", that means that it would be sliced to be "**enc**" and "**2**".
An integer variable holds index 1 of the `String[]` object. Index 1 is also converted to an integer from a string.

```java
//if the string begins with 'enc'
//aka is a msg from the rotary encoder...
if (inString.startsWith("enc")) {
    // the encoder pos is stored in index
    String[] q = splitTokens(inString);
    index = int(q[1]);
}
```

And that is how the data from the Python script is brought into Processing!

Similar `if` statements are setup for the three NeoSliders and the VL53L4CD.

```java
//if the string begins with 'red'
//aka is from the red neoslider
if (inString.startsWith("red")) {
    //the red value is stored in red
    String[] r = splitTokens(inString);
    red = int(r[1]);
}

//if the string begins with 'green'
//aka is from the green neoslider
if (inString.startsWith("green")) {
    //the green value is stored in green
    String[] g = splitTokens(inString);
    green = int(g[1]);
}

//if the string begins with 'blue'
//aka is from the blue neoslider
if (inString.startsWith("blue")) {
    //the blue value is stored in blue
    String[] b = splitTokens(inString);
    blue = int(b[1]);
}

//if the string begins with flight
//aka is from the VL53L4CD
if (inString.startsWith("flight")) {
    //the time of flight value is stored in flight
    String[] f = splitTokens(inString);
    flight = int(f[1]);
}
```

### Changing Animations

The current animation is controlled by the position of the rotary encoder. A series of `if` statements are setup to check on the rotary encoder's position. The animations are setup as functions outside of `draw()` so that they can be called independently.

```java
//the encoder's position corresponds with which animation plays
if (index == 0) {
    circles();
}

if (index == 1) {
    cube();
}
```
The Animations

The four included animations are designed to be fairly simple so that you can get an idea of how Processing works.

Circles

The `circles()` animation randomly generates circles in a line in the middle of the screen that are random colors from the `colors[]` list. The size of the circles is determined by the VL53L4CD.

The background color is created by the values from the NeoSliders that are held in red, green and blue. The VL53L4CD’s data is held in `int size`, which has the value range of `flight` mapped to a range of 300 to 25. This allows the circles to get larger the closer you are to the sensor and smaller the farther away you are.

```java
//the Circles animation
//colorful circles randomly appear in the middle of the screen
//background color is affected by the sliders
//the circles' size is affected by the VL53L4CD
void circles() {
  background(red, green, blue);
  strokeWeight(1);
  ypos = ypos + (yspeed * ydirection);
  if (ypos > height-rad || ypos < rad) {
    ydirection *= +1;
  }
  int size = int(map(flight, 0, 45, 300, 25));
  for (int i = 0; i < 10; i++) {
    for (int z = 0; z < 6; z++) {
      fill(colors.get(z));
      circle(width/2, random(ypos), random(size));
    }
  }
}
```

Cube

The `cube()` animation is a hollow, 3D spinning cube. Its background color is affected by the NeoSliders. The speed of the cube’s spin is affected by the VL53L4CD.

```java
//the Cube animation
//3D cube spinning
void cube() {
  background(red, green, blue);
  strokeWeight(1);
  ypos = ypos + (yspeed * ydirection);
  if (ypos > height-rad || ypos < rad) {
    ydirection *= +1;
  }
  int size = int(map(flight, 0, 45, 300, 25));
  for (int i = 0; i < 10; i++) {
    for (int z = 0; z < 6; z++) {
      fill(colors.get(z));
      circle(width/2, random(ypos), random(size));
    }
  }
}
```
speed maps the VL53L4CD's range from 10 to 0.1. The closer you are to the sensor, the faster the cube spins.

```cpp
void cube() {
  strokeWeight(5);
  float speed = map(flight, 0, 45, 10, 0.1);
  background(red, green, blue);
  translate(width/2, height/2, 0);
  i = i + speed;
  if (i > 180) {
    i = 0.0;
  }
  rotateY(radians(i));
  noFill();
  box(500);
}
```

Pizza Cat

The pizzaCat() animation demonstrates bringing in image files to a Processing script. Here, pizza and cat emojis bounce around the screen at different speeds.

Its background color is affected by the NeoSliders. The green NeoSlider is affecting how many cat emojis are on the screen. The blue NeoSlider is affecting how many pizza emojis are on the screen. The VL53L4CD is affecting the speed of the pizza and cat emojis' bounce.

```cpp
void pizzaCat() {
  background(red, green, blue);
  float meow = map(green, 0, 255, 32, 16);
  float pie = map(blue, 0, 255, 15, 0);
  float speed = map(flight, 0, 45, 0, 25);
  for (int e = 16; e < meow; e++) {
    if (last_speed != speed) {
      moveX[e] = x_speeds[e] + speed;
      moveY[e] = y_speeds[e] + speed;
    } else {
      moveX[e] = moveX[e];
      moveY[e] = moveY[e];
    }
    x_dir[e] += moveX[e];
    if (x_dir[e] &lt; 0 || x_dir[e] &gt; width) {
      moveX[e] *= -1;
    }
    y_dir[e] += moveY[e];
    if (y_dir[e] &lt; 0 || y_dir[e] &gt; height) {
      moveY[e] *= -1;
    }
    box(100); //Draw the pizza and cat emojis
  }
}
```
Dancing Triangles

The **dancingTriangles()** animation randomly generates a hollow 2D triangle with random coordinates in the middle of the screen. Its background color is affected by the NeoSliders.

The VL53L4CD is affecting the speed at which the triangles are being generated. The value of **int speed** is mapped to create a delay range of **25** to **100**. The closer you are to the sensor, the slower the triangles will generate.
int speed = int(map(flight, 0, 45, 25, 100));

background(red, green, blue);
strokeWeight(30);

for (int w = 800; w < 1000; w++) {
  for (int h = 1100; h < 1500; h++) {
    x1 = int(random(h));
    y1 = int(random(w));
    x2 = int(random(h));
    y2 = int(random(w));
    x3 = int(random(h));
    y3 = int(random(w));
  }
}

noFill();
triangle(x1, y1, x2, y2, x3, y3);
delay(speed);

---

**Constructing the Mounting Plate**

A template is available to construct a mounting plate for this project. The files are available for download below.

- mountingBoard_v1.f3z
- mountingBoard_v1.dxf
- mountingBoard_v1.svg
- mountingBoardWithOutlines_v1.dxf
- mountingBoardWithOutlines_v1.svg
- vidSynthMountingBoard.stl
Template Ideas

There are a few different ways that you can use this template.
You can print or cut it out of paper and use it as a guide to drill mounting holes on your favorite material (cardboard, acrylic, wood, etc).
You can use it for laser cutting
If your 3D printer is large enough, you can 3D print the .STL file
Assembly

This project does not require any soldering, making assembly a straightforward experience.

Cut the NeoSlider's I2C Address Jumpers

The NeoSliders work by sending data over I2C with the adafruit_seesaw CircuitPython library. Since this project uses three of the NeoSliders, you will need to adjust the I2C addresses on two of them by cutting the address jumpers.

For more information on the NeoSlider’s Address Jumpers, please reference the Primary Learn Guide
The default I2C address for the NeoSlider is 0x30. You will leave one NeoSlider on this address, leaving the address jumpers uncut.

The second NeoSlider's I2C address will be set to 0x31 by cutting address jumper A0.

The third NeoSlider's I2C address will be set to 0x32 by cutting address jumper A1.

Connect the STEMMA Boards

All of the STEMMA boards will be connected to each other using QT to QT cables.
Connect the 0x30 NeoSlider's bottom socket to the 0x31 NeoSlider's bottom socket.

Connect the 0x31 NeoSlider's top socket to the 0x32 NeoSlider's top socket.
Connect the 0x32 NeoSlider's bottom socket to the rotary encoder's left socket.

Connect the rotary encoder's right socket to the VL53L4CD's left socket.

Connect to the Raspberry Pi

Using a QT to female socket cable, connect the STEMMA boards to the Raspberry Pi.

Connect the QT cable's female socket wires to the Raspberry Pi's GPIO pins.

Red cable (VIN) connects to physical pin 1
Black cable (GND) connects to physical pin 6
Blue cable (SDA) connects to physical pin 3
Yellow cable (SCL) connects to physical pin 5
Plug the QT to female socket cable into the VL53L4CD's right socket.

Mount the Components

Using M2.5 screws and stand-offs, mount the components to the mounting board made out of your chosen material.
Attach M2.5 stand-offs to the mounting board with M2.5 screws.
Attach the NeoSliders to the stand-offs with screws. The order going left to right is 0x30, 0x31 and 0x32.

0x30 and 0x31's QT cable should be on the top. 0x31 and 0x32's QT cable should be on the bottom.
Attach the rotary encoder to its stand-offs with screws next to the 0x32 NeoSlider.

Attach the VL53L4CD to its stand-offs with screws. Run its QT cables around the stand-offs so that the cabling is hidden under the VL53L4CD and Raspberry Pi.

Attach the Raspberry Pi to its stand-offs with screws next to the rotary encoder and VL53L4CD.

Assembly Complete!

Now you're ready to get video synthesizing!
Usage

The rotary encoder lets you switch between the four different animations.
Use the three NeoSliders to affect the color of the background for all of the animations using RGB values. Each slider has a mapped value range of 0 to 255 and affects the red, green or blue value. The NeoSliders' NeoPixels will be the same color as the animation's background color.

The VL53L4CD affects different parameters for each animation.

- For Circles, it affects the size of the circles.
- For Cube, it affects the speed of the spinning cube.
- For Dancing Triangles, it affects the rate of new triangles being generated.
- For Pizza Cat, it affects the speed of the cat and pizza emojis.

When speed is affected, the speed increases the closer you get to the sensor. When size is affected, the size increases the closer you get to the sensor.

Going Further

You can customize this project by creating your own animations or adjusting the provided animations to suit your needs. Processing has a lot of documentation and examples to use for inspiration.

Additionally you could try using different sensors, such as an accelerometer or a light sensor, to control the animations.