ENGDUINO EXAMPLES: THEREMIN

The theremin is a weird and wonderful electronic instrument that requires no physical contact. Have a listen to a program about them at: http://www.bbc.co.uk/programmes/b0076nqv

The theremin was invented in 1920 by Léon Theremin, an early Russian electronic engineer. It is played by moving one’s hands near two antennas – the first controls the volume of the output and the second the pitch.

For those that are musical it is worth knowing that the Theremin inspired Robert Moog to invent the synthesiser, so, although it’s a little-used instrument, it has had a powerful effect on the history of music.

THE PROJECT

In this project, we will use the light sensor to control the frequency of a tone. We could use a second Engduino to control the volume, but that requires a bit more hardware and it’s the principles that matter here.

You will need an Engduino and a little piezo-electric speaker. Plug the speaker into the Engduino’s expansion bus – to the connections marked IO_0 (third from top) and GND (bottom).

The project itself falls into two parts: making a tone at a given frequency and controlling that frequency using the light sensor.

PRODUCING A TONE

A tone is a pure sine wave. A sine wave of 440Hz is the A above middle C, commonly used for tuning.

Mixtures of different sine waves are what gives us the complex sounds we hear from musical instruments. Pure tones on their own sound artificial and electronic because they don’t happen in nature.
Unfortunately, our Engduino can’t even generate a sine wave – it is a digital system so can only generate square waves (0s and 1s). For our purposes, this doesn’t matter much – in fact a square wave can be represented by a series of mixed sine waves so it is not quite the same as a pure tone, but it’s close enough for us.

**WRITING OUR OWN CODE TO MAKE A TONE.**

If you’ve connected the speaker up as suggested, you have connected it to digital pin D5. To make a very boring sound, we can switch the output from 0 to 1 and back again at a known frequency. For example:

```c
int speaker = 5;                // Speaker connected to digital pin 5

void setup()
{
    pinMode(speaker, OUTPUT);     // sets the digital pin as output
}

void loop()
{
    digitalWrite(speaker, HIGH);  // sets the speaker pin high
    delay(1);                     // waits for a 1ms
    digitalWrite(speaker, LOW);   // sets the speaker pin low
    delay(1);                     // waits for a 1ms
}
```

In setup, we set the digital pin to be an output. Then we loop setting it high, waiting for 1ms, setting it low, waiting for 1ms, and continuing. What frequency note would you expect this to be? See what happens when you change the delay values.

**AN ALTERNATIVE**

Fortunately, someone has done the hard work and written a library for us that will generate tones. The library uses a feature built in to the microprocessor called pulse width modulation (PWM) – in our case this just allows us to generate tones more accurately than we could do ourselves.

There are two calls – one to make a tone, or make it for a given amount of time; and the second to stop a tone being played:

```c
tone(speaker, frequency)
tone(speaker, frequency, duration)

noTone(speaker)
```

Write a second sketch to try it – using the frequency you thought the previous tone was at to check whether you’re right. See if you can make it play a tune. HINT: the frequencies of notes are given at the bottom of the page at [http://arduino.cc/en/Tutorial/tone](http://arduino.cc/en/Tutorial/tone).
GETTING INPUT FROM THE LIGHT SENSOR

You should work through the light sensor worksheet.

Print the light values to the screen. Even if keep everything as constant as possible – don’t move the Engduino, sit still, etc., is the value a constant? If we use this value to change the sound, what kind of thing do you think we’ll hear?

RESCALING THE INPUT

The input from the light sensor ranges, in theory, from 0-1023. In practice it will rarely reach 1023 and, if you’re just using the illumination in the room, the maximum in practice might only be, say, 700 (you’ll need to measure this). Likewise, the minimum might not be zero when it’s fully covered – try it.

Choose a range of frequencies you’d like your theremin to cover. Let’s say that’s 50Hz to 5000Hz.

So, in my case, I would want the following:

<table>
<thead>
<tr>
<th>Input from light sensor</th>
<th>Output frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>700</td>
</tr>
<tr>
<td>50</td>
<td>5000</td>
</tr>
</tbody>
</table>

See if you can work out how a way of calculating the output frequency for a given value of light input.

Try it – print the original and scaled values to the screen using `Serial.print`.

CONNECTING THE TWO

We now need to connect the light sensor to the speaker.

Use the rescaled value you’ve just generated to set the tone on the speaker. How does it sound?

Awful, in all probability. And there are two problems. The first is that your hand has to get very very close to the light sensor to get low tones and the second is that it just sounds awful. Let’s fix the second.

PLEASE STOP, YOU’RE HURTING MY EARS

The problem with the tone that never seems to stand still comes from the fact that the light sensor value doesn’t stand still – it is noisy, partly because the lights flicker, albeit faster than you can see. So we need to filter the value – to smooth it out over time.

To do that, we might make use of some maths. We could read the sensor and make an average out of the last \( n \) readings, where \( n \) is a number you choose. This is called a simple moving average and, mathematically, it’s represented by the formula:

\[
out_i = \frac{i_{n_i} + i_{n_i-1} + i_{n_i-2} + i_{n_i-3} + \cdots + i_{n_i-n+1}}{n}
\]
Try to implement this. To do so, you’ll need to know what an array is.

---

**ASIDE: ARRAYS**

An array is a data structure used in just about all programming languages. In our case, it’s a numbered collection of things of the same type with a fixed maximum size. In essence it’s a set of boxes into which we can put values – each box has a number, starting at 0 and going up.

Say we needed to keep the last 3 values of a variable, then we create an array of length 3. Let’s say these values are floating point values rather than integers and we want the name of the array to be ‘buffer’. We declare the array immediately after the #include directives in the following way:

```c
#define BUFFERSIZE 3
float buffer[BUFFERSIZE];
```

This creates an array of 3 boxes (labelled 0, 1, and 2) in which we can store floating point numbers. But, before we can use this for our purposes, we need to know where to put the latest sensor reading. For this we need another variable that will tell us which of those spaces should contain the value we have just read from the sensor (i.e. in box 0, 1 or 2) – we’ll call this the index variable. We will calculate the average of all the readings stored in buffer and, if we keep moving the index value on, we can make sure that we always have the last three values from the sensor in the array. After some time, let’s imagine we have to deal with reading 27. Before we put it in, the buffer should contain readings 24, 25 and 26:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>26</td>
<td>24</td>
</tr>
</tbody>
</table>

After we put in 27, we want it to contain:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
</tbody>
</table>

When we get reading 28, this gets put into box 0...

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>26</td>
<td>27</td>
</tr>
</tbody>
</table>

And so on. We just need to organise that the index variable always points to the next box into which we should put a reading. To achieve this, add the following after the above two lines – it creates a variable ‘index’ and sets its value to be 0 (the first location in buffer).

```c
int index = 0;
```

Now let’s look at how to use this in calculating a simple moving average. In the loop() section of your code, you will need something that looks a bit like the following:
void loop()
{
    ...
    int sensorValue;
    int rescaledValue;
    float movingAverage;
    ...
    Get the light sensor value

    buffer[index] = sensorValue;
    index = index + 1;
    if (index > BUFFERSIZE)
    {
        index = 0;
    }
    movingAverage = 0;
    for (int i = 0; i < BUFFERSIZE; i++)
    {
        movingAverage = movingAverage + buffer[i];
        movingAverage = movingAverage / BUFFERSIZE;
    }
    ...
    Rescale the moving average value to the range we want
    ...
    and play the tone
    delay(1);
}

Give this a go....

BACK AGAIN

The longer the buffer we choose, the smoother the value will be, but the longer the instrument
takes to respond to a change. Welcome to engineering.

Experiment with this. Why does the instrument sound odd for a short while after you reset it and
what could we do to change that?

RESCALING THE INPUT REVISITED

One of the reasons that a simple scaling doesn’t work so well – much of the change happens close to
the light sensor – is that the input is not linear in the distance your hand is from the sensor. Take a
tape measure and work out what the (averaged) light sensor readings are for each distance. Plot this
as a curve.

Can you figure out a way of changing your rescaling so that we can make changes (even a bit) further
away than at present?