Raspberry Pi Activity: Paper Piano

To the prof: This activity is intended to span **two 75**-minute classes. Suggested lesson breakpoints are included.
Module included: There is an included module called "waveform_vis.py" that helps students visualize their generated samples. It is recommended to give students access to this module and explain how it is used. Directions on usage are found in comments at the bottom of the module itself. This can be used for students to know if they generate the waveforms correctly by matching it to the figures in the pdf. Profs can also use this to help with grading.

In this activity, you will implement a simple piano using nothing but pieces of paper and pencil lead as the keys! Well, almost. Instead of using push-button switches (as in previous activities), you will use capacitive touch switches. These switches work by using your body's capacitance. **Capacitance** is the ability of something to store an electrical charge (which your body can do!). When you touch the switch, the capacitance is increased – and the switch is triggered. The electrical signal will then be amplified by a transistor so that the RPi can detect it as a switch press on an input pin. Transistors will be discussed later in this activity.

For this activity, you will need the following items:

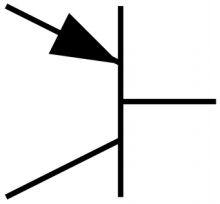
- Raspberry Pi B v3 with power adapter;
- LCD touchscreen;
- Keyboard and mouse;
- USB-powered speakers;
- Breadboard;
- GPIO interface board with ribbon cable;
- LEDs, resistors, switches, and jumper wires provided in your kit;
- Transistors;
- Paper and pencil (#2 works well) for the piano keys; and
- Scotch tape to tape jumper wires to the paper piano keys (provided to you).

Regarding the electronic components, you will need the following:

- 1x RGB LED;
- 2x push-button switches;
- 4x S8550 PNP transistors;
- 24x jumper wires.

The activity is broken down into three parts. In the first part, you will just implement a single piano key that will play a middle C note. In the second part, you will extend the piano to include four keys so that four notes can be played (C, E, G, and B). In the final part, you will add the functionality to record notes played – and play them back!

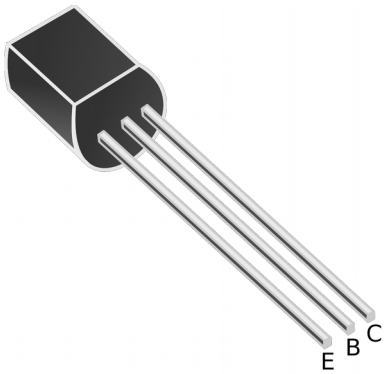
Part one: a single piano key
For this part of the activity, you will implement a circuit that includes a new component that has not yet been used in an activity. In a circuit diagram, its symbol looks like this:



This is the symbol for a PNP transistor. A **transistor** is an electronic device that is used to amplify and/or switch electrical signals. It has three terminals: a base, an emitter, and a collector. Voltage or current applied to one pair of terminals changes the current through the other pair. Today's electronic devices are fundamentally based on transistors.

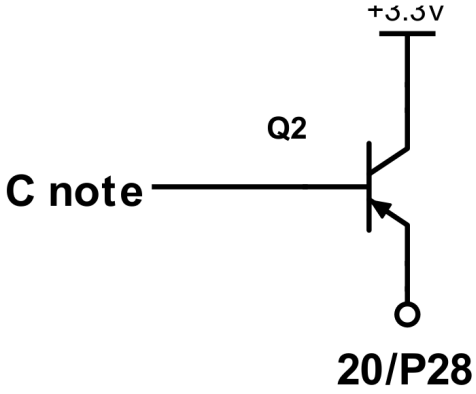
A thorough discussion of transistors is beyond the scope of this activity. However, the way that the transistors are used in this activity is simple enough. The base will be connected to a piece of paper with some graphite on it (the middle C note), the collector will be connected to +3.3V, and the emitter will be connected to an input pin on the RPi. When the base is not touched, the circuit is open. That means that electric current cannot pass from the collector to the emitter. In order to turn on the transistor, a little bit of power must be supplied to the base. Our body contains a little bit of electric current (via capacitance) which is enough to turn the transistor on. When the note (and by consequence the base) is touched, this little amount of current will be greatly amplified and turn on the transistor, making it act like a switch.

The transistor that you will use in this activity is an S8550 in what is called a TO-92 package. It looks like this (note that the flat side is to the front, and this orients the legs so that the emitter, base, and collector are easy to identify):

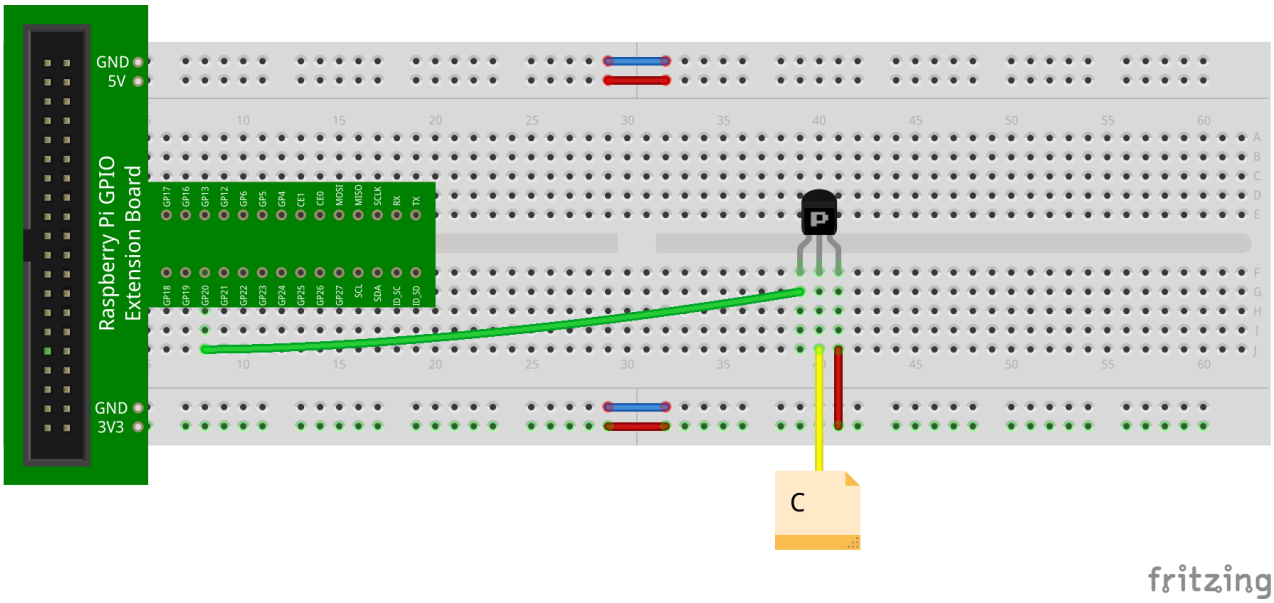


Note the labels for the transistor's legs in the image above. When the flat side of the transistor is facing the front, the emitter (labeled E) is the left leg, the base (labeled B) is the center leg, and the collector (labeled C) is the right leg. When inserting the transistor into the center of the breadboard, make sure that each leg is in a separate column.

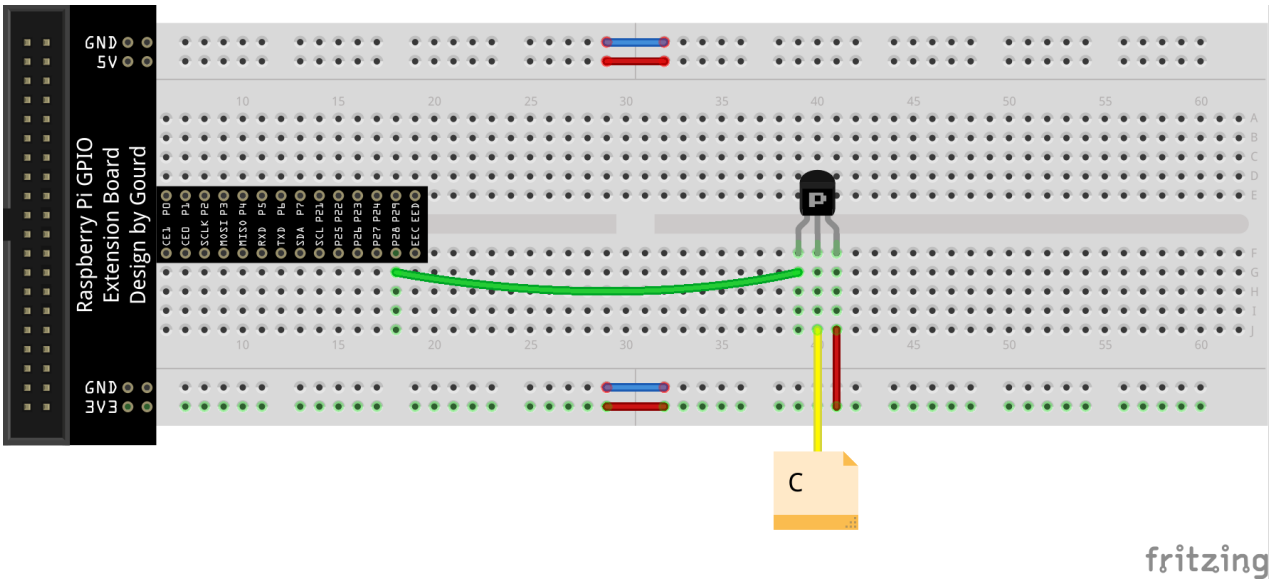
Implement the following circuit:



Here's one way to layout this circuit:



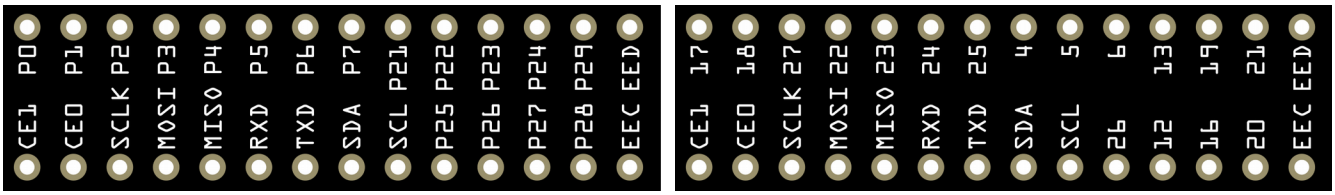
If you have the black GPIO interface board, layout the circuit as follows instead:



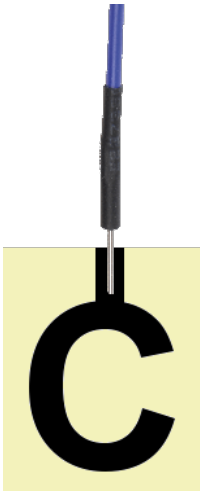
Recall that there are actually **three** different pin numbering schemes in use with GPIO pins on the RPi: (1) the **physical** pin order on the RPi; (2) the numbering assigned by the manufacturer of the **Broadcom** chip on the RPi; and (3) an older numbering assigned by an early RPi user who developed a library called **wiringPi**. Here's the cross-reference table shown in an earlier activity:

BCM	wPi	Name	Physical		Name	wPi	BCM
		3V3	1	2	5V		
2	8	SDA.1	3	4	5V		
3	9	SCL.1	5	6	GND		
4	7	GPIO.7	7	8	TXD	15	14
		GND	9	10	RXD	16	15
17	0	GPIO.0	11	12	GPIO.1	1	18
27	2	GPIO.2	13	14	GND		
22	3	GPIO.3	15	16	GPIO.4	4	23
		3V3	17	18	GPIO.5	5	24
10	12	MOSI	19	20	GND		
9	13	MISO	21	22	GPIO.6	6	25
11	14	SCLK	23	24	CE0	10	8
		GND	25	26	CE1	11	7
0	30	SDA.0	27	28	SCL.0	31	1
5	21	GPIO.21	29	30	GND		
6	22	GPIO.22	31	32	GPIO.26	26	12
13	23	GPIO.23	33	34	GND		
19	24	GPIO.24	35	36	GPIO.27	27	16
26	25	GPIO.25	37	38	GPIO.28	28	20
		GND	39	40	GPIO.29	29	21

If you have the green GPIO interface, you won't have to refer to the table since the RPi uses the BCM pin numbering scheme (which the green GPIO interface also uses). If you have the black GPIO interface, the following comparison of the GPIO interface boards labeled with both pin numbering schemes (shown in an earlier activity) will help:



In the layout diagram above, the emitter of the transistor is connected to **GP20** (which refers to BCM pin **20** on the RPi and **P28** on the black GPIO interface). The collector is connected to +3.3V, and the base is connected to a piece of paper with some graphite on it. Here's an example of how a jumper wire can be connected (with some Scotch tape) to the graphite:



Make sure that the collector is connected to 3.3V (and not 5V)!

Next, let's take a look at some Python source code that will implement the simple paper piano that plays a single middle C note (for now). The first step is to import the required libraries:

```
import RPi.GPIO as GPIO
from time import sleep
import pygame
from array import array
```

The GPIO library will be used to support GPIO on the RPi. The sleep function will be used to temporarily "sleep" while waiting for piano keys to be pressed and released. The pygame library will be used to generate and play the notes. Finally, the array library will be used to create arrays of note samples. In Python, arrays are just like lists, except that the type of data stored in them is constrained. In the case of this activity, we will store a note's samples in 16-bit signed numbers. A thorough discussion of how the notes are actually generated is beyond the scope of this activity. A few details will be provided later in the activity; however, you are encouraged to research on your own the technical aspects of how the pygame library can generate notes.

Next, we specify several constants and a Note class (which inherits from pygame's mixer's Sound class):

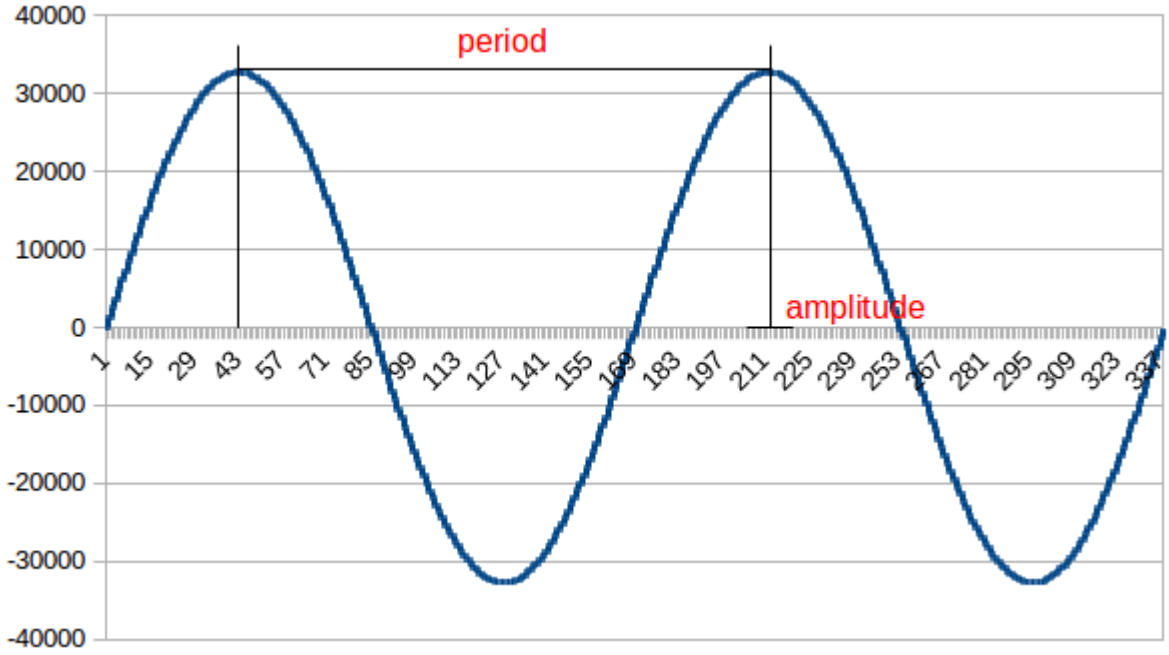
```
MIXER_FREQ = 44100
```

```
MIXER_SIZE = -16
MIXER_CHANS = 1
MIXER_BUFF = 1024

# the note generator class
class Note(pygame.mixer.Sound):
     # note that volume ranges from 0.0 to 1.0
     def __init__(self, frequency, volume):
          self.frequency = frequency
          # initialize the note using an array of samples
          pygame.mixer.Sound.__init__(self, \
               buffer=self.build_samples())
          self.set_volume(volume)

     # builds an array of samples for the current note
     def build_samples(self):
          # calculate the period and amplitude of the note's wave
          period = int(round(MIXER_FREQ / self.frequency))
          amplitude = 2 ** (abs(MIXER_SIZE) - 1) - 1
          # initialize the note's samples (using an array of
          # signed 16-bit "shorts")
          samples = array("h", [0] * period)

          # generate the note's samples
          for t in range(period):
               if (t < period / 2):
                    samples[t] = amplitude
               else:
                    samples[t] = -amplitude

          return samples
```

The constants setup the mixer to sample 44,100 times per second. The minimum and maximum values for each sample is represented by a 16-bit signed number (i.e., -32,768 to 32,767). The mixer will have a single channel (mono) and a buffer of 1KB.

This forms the core of generating the notes to play. In short, a note is initialized with a frequency and volume. The frequency of a middle C, for example, is 261.6 Hz. Frequency is the number of cycles per second and controls a note's pitch. To make sure that we hear the notes well, we will blast them at full volume (1.0). Start with your speakers set to low and increase the volume as needed. If the speakers are too loud, the sound will sometimes be distorted and appear to skip.

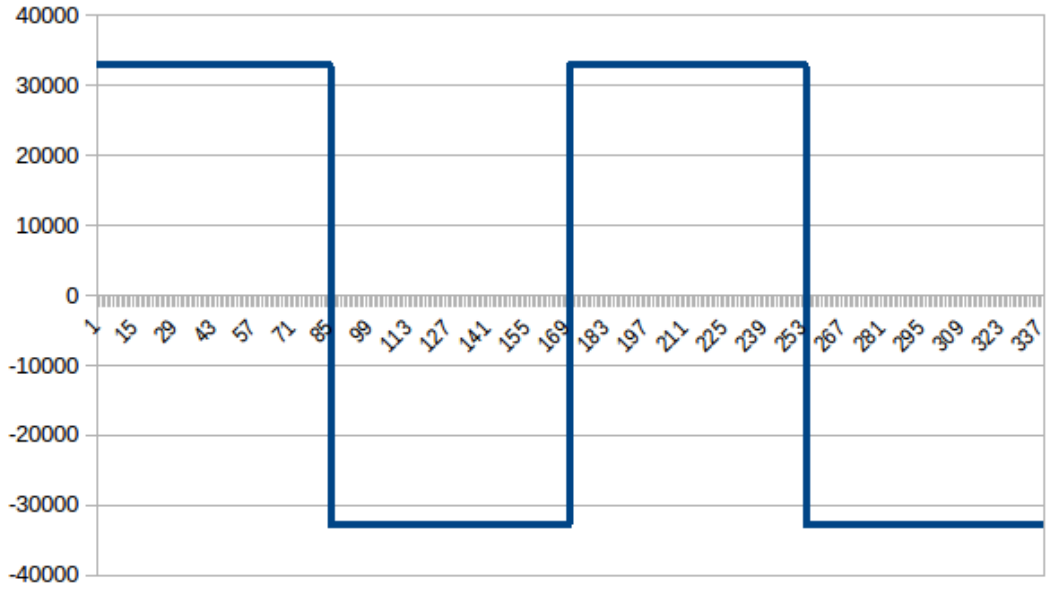
Each note will need samples for the mixer to generate and play them. The samples are generated from the mixer's sampling frequency (set at 44.1 KHz). The sampling frequency (or sample rate) is the number of samples per second in some sound. 44.1 KHz is considered CD quality. Notes are just waves with a period (the length from one peak to the next), and an amplitude (the height from the center line to the peak):



Formally, a period is just the number of seconds per cycle. In our program, we will tweak this meaning slightly to the number of samples within the sampling rate (of 44.1 KHz). For the middle C (at a frequency of 261.6 Hz), for example, the number of samples that make up a single period is 169 (well, 168.58 – but we'll round up). The number of samples in a period is calculated as the ratio of the sampling frequency to the note's frequency. We want to generate a number of samples equal to the period of the note's wave. The mixer will then play this over and over again, for the duration of the note. The amplitude of a wave affects how loud the note is.

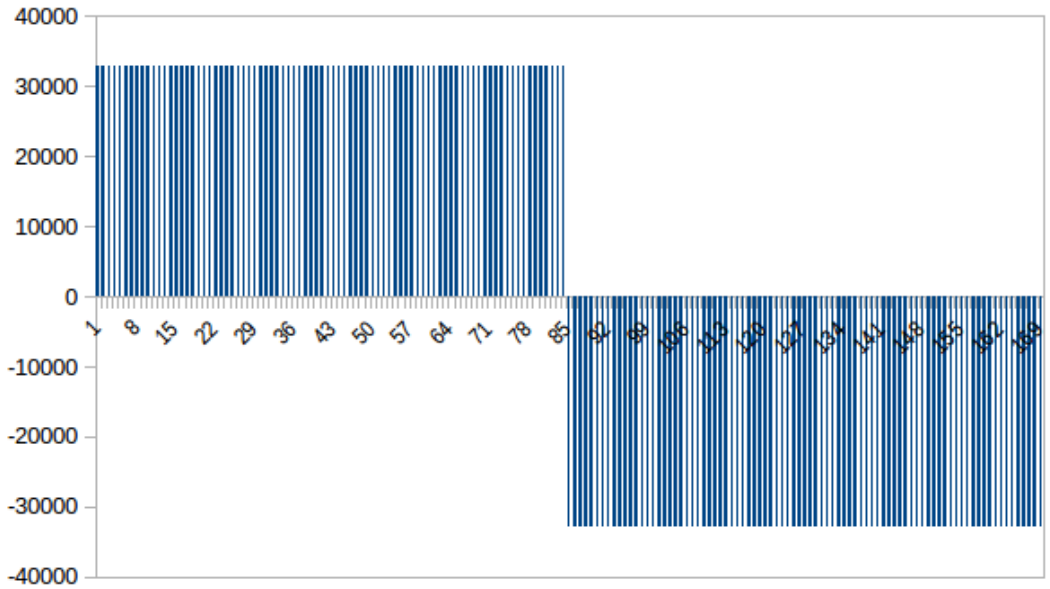
It is not possible to represent a continuous sinusoidal wave in a computing machine (since it is discrete!). Therefore, we have to approximate without creating too much of a loss. One modification allows the note to keep most of its characteristics, albeit perhaps a bit more harsh and bright. This square version of the wave (called a square wave) results in a maximum amplitude for half of the period, and the negative of the maximum amplitude for the other half of the period.

The following shows the square wave approximation of the sinusoidal wave shown above:



Note how much sharper it is. In the end, however, the result is a sound that is near the intended one. In terms of pitch (which is affected by the wave's frequency), it is exact. The sound is actually a bit louder, since more points along the wave are at the frequency limit (-32,768 to 32,767). This limit is due to the size of the mixer (set at 16-bits signed).

For this activity, we will create a square wave and fill the samples with half of the maximum amplitude and half of the negative of the maximum amplitude as follows:



The array of samples is initialized in the following statement:

```
samples = array("h", [0] * period)
```

In Python, arrays are just objects that are defined in a class. The first parameter specified when instantiating an array is the type of data that will be stored in it. There are many different types, and `h` restricts the array to 16-bit signed numbers. The second parameter is just a list of zeros, the number of which is determined by the period of the note. A middle C with a frequency of 261.6 Hz will have a period of 169 samples with a sample rate of 44.1 KHz. That is, a middle C takes up 169 samples per period in the 44.1 K samples per second.

Next, we'll define two functions that do the work of waiting for notes to be pressed and released. In addition, we initialize the pygame library, setup GPIO, and create the middle C note:

```
# waits until a note is pressed
def wait_for_note_start():
     while (not GPIO.input(key)):
          sleep(0.01)

# waits until a note is released
def wait_for_note_stop():
     while (GPIO.input(key)):
          sleep(0.1)

# preset mixer initialization arguments: frequency (44.1K), size
# (16 bits signed), channels (mono), and buffer size (1KB)
# then, initialize the pygame library
pygame.mixer.pre_init(MIXER_FREQ, MIXER_SIZE, MIXER_CHANS,\
     MIXER_BUFF)
pygame.init()

# use the Broadcom pin mode
GPIO.setmode(GPIO.BCM)

# setup the pin and frequency for a C note
key = 20
freq = 261.6

# setup the input pin
GPIO.setup(key, GPIO.IN, GPIO.PUD_DOWN)

# create the actual C note
note = Note(freq, 1)
```

The `wait_for_note_start` function sleeps until the note is pressed. The `wait_for_note_stop` function does the opposite: it sleeps until the note is released. As mentioned earlier, P28=BCM 20 is used to detect the note press and release. The frequency for a middle C (261.6 Hz) is specified, and the note itself is generated as an instance of the Note class.

Lastly, the main part of the program:

```
# the main part of the program
print "Welcome to Paper Piano!"
print "Press Ctrl+C to exit..."

# detect when Ctrl+C is pressed so that we can reset the GPIO
# pins
try:
     while (True):
          # play a note when pressed...until released
          wait_for_note_start()
          note.play(-1)
          wait_for_note_stop()
          note.stop()
except KeyboardInterrupt:
     # reset the GPIO pins
     GPIO.cleanup()
```

It's fairly simple, actually. Also note that it is just a driver. Until the user presses Ctrl+C, the program waits for a note to be pressed, plays the note, waits for the note to be released, and stops playing the note. The -1 argument passed to the note's `play` function means that it will play indefinitely (well, at least until its `stop` function is called).

Implement the circuit and program above, then test it! For completeness, here is the program in its entirety:

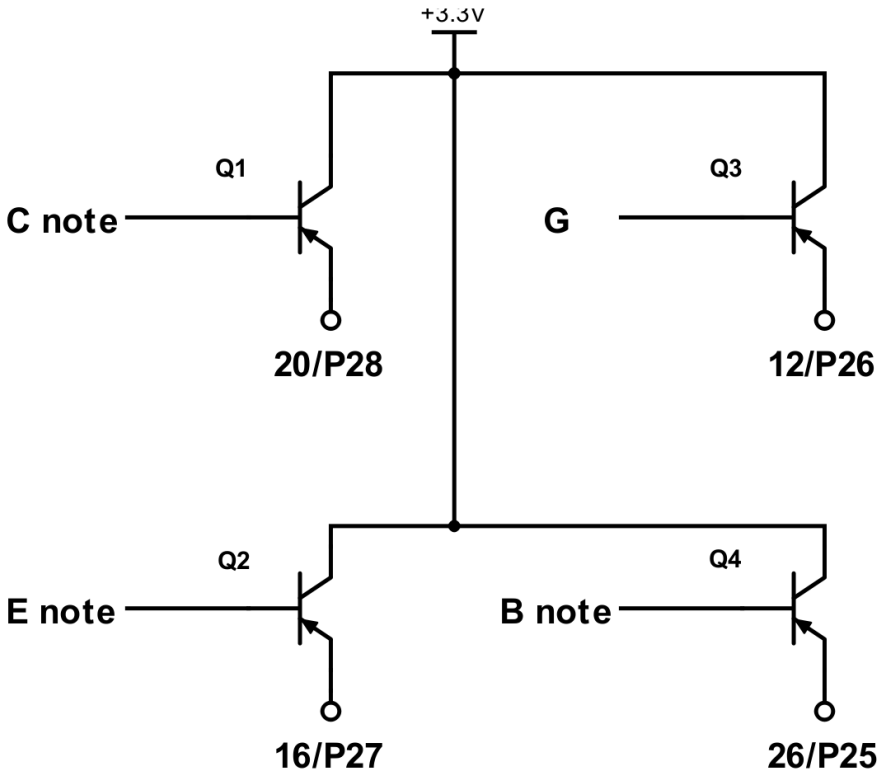
```
########################################
# Name:
# Date:
# Description: Paper piano (v1).
########################################
import RPi.GPIO as GPIO
from time import sleep
import pygame
from array import array

MIXER_FREQ = 44100
MIXER_SIZE = -16
MIXER_CHANS = 1
MIXER_BUFF = 1024

# the note generator class
class Note(pygame.mixer.Sound):
     # note that volume ranges from 0.0 to 1.0
     def __init__(self, frequency, volume):
          self.frequency = frequency
          # initialize the note using an array of samples
          pygame.mixer.Sound.__init__(self,\
```

```
                buffer=self.build_samples())
            self.set_volume(volume)

        # builds an array of samples for the current note
        def build_samples(self):
            # calculate the period and amplitude of the note's wave
            period = int(round(MIXER_FREQ / self.frequency))
            amplitude = 2 ** (abs(MIXER_SIZE) - 1) - 1
            # initialize the note's samples (using an array of
            # signed 16-bit "shorts")
            samples = array("h", [0] * period)

            # generate the note's samples
            for t in range(period):
                if (t < period / 2):
                    samples[t] = amplitude
                else:
                    samples[t] = -amplitude

            return samples

# waits until a note is pressed
def wait_for_note_start():
    while (not GPIO.input(key)):
        sleep(0.01)

# waits until a note is released
def wait_for_note_stop():
    while (GPIO.input(key)):
        sleep(0.1)

# preset mixer initialization arguments: frequency (44.1K), size
# (16 bits signed), channels (mono), and buffer size (1KB)
# then, initialize the pygame library
pygame.mixer.pre_init(MIXER_FREQ, MIXER_SIZE, MIXER_CHANS,\
    MIXER_BUFF)
pygame.init()

# use the Broadcom pin mode
GPIO.setmode(GPIO.BCM)

# setup the pin and frequency for a C note
key = 20
freq = 261.6

# setup the input pin
GPIO.setup(key, GPIO.IN, GPIO.PUD_DOWN)
```

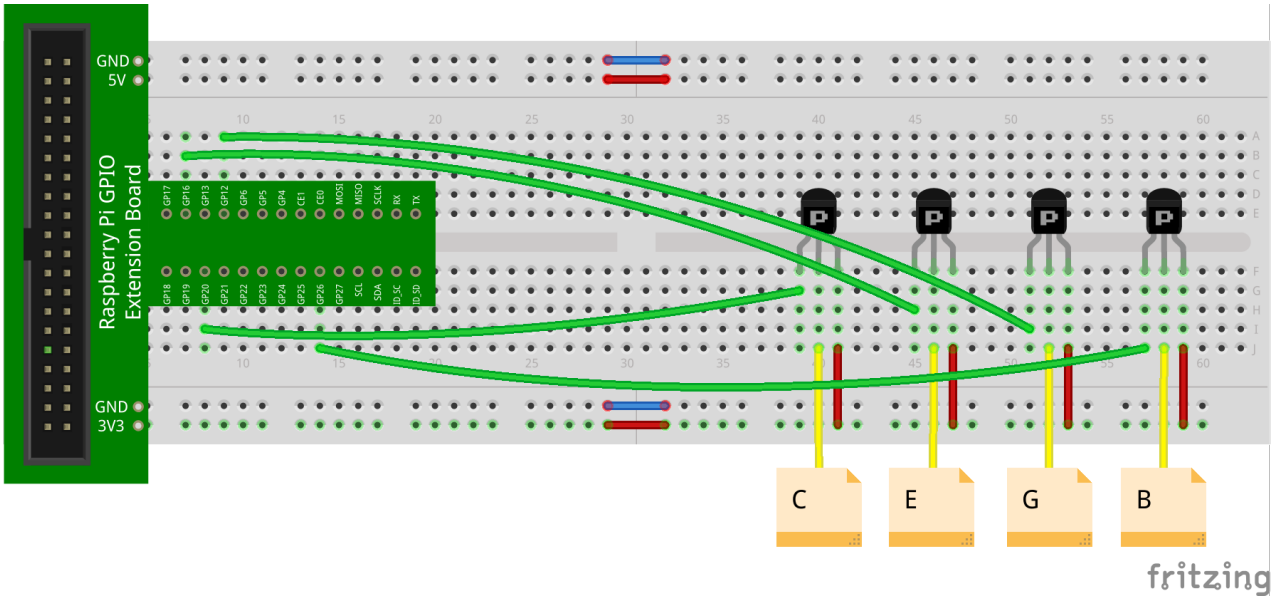
```
# create the actual C note
note = Note(freq, 1)

# the main part of the program
print "Welcome to Paper Piano!"
print "Press Ctrl+C to exit..."

# detect when Ctrl+C is pressed so that we can reset the GPIO
# pins
try:
     while (True):
          # play a note when pressed...until released
          wait_for_note_start()
          note.play(-1)
          wait_for_note_stop()
          note.stop()
except KeyboardInterrupt:
     # reset the GPIO pins
     GPIO.cleanup()
```

Part two: adding more piano keys

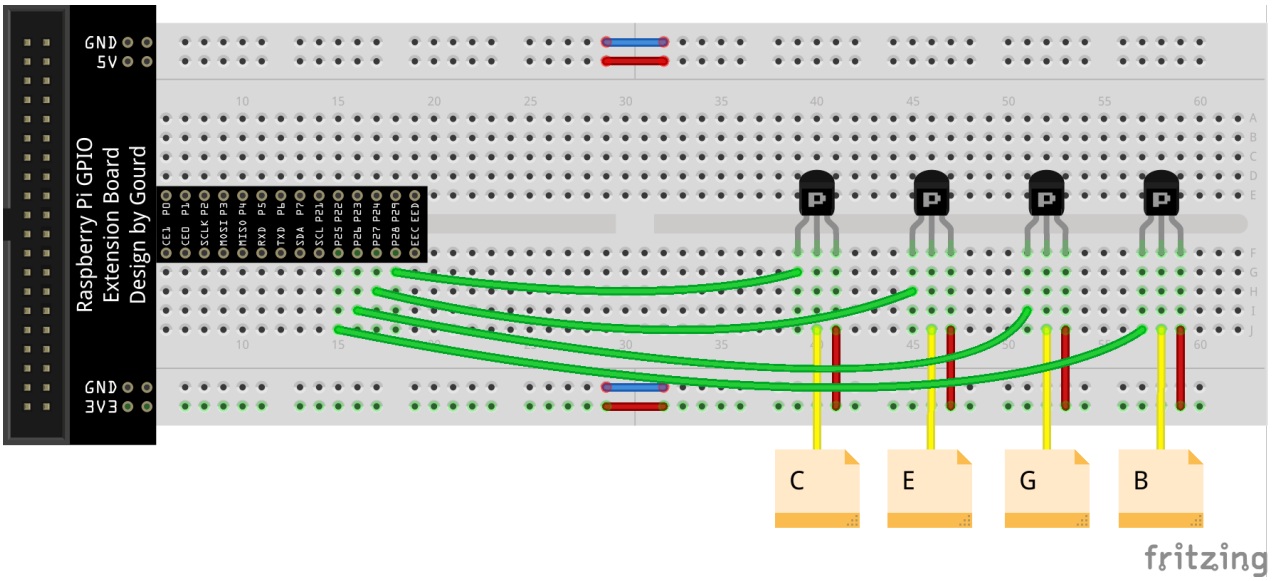
In this part of the activity, we will add three more piano keys (E, G, and B). The process is exactly the same; however, we will slightly modify the code to handle the extra notes. Here's the circuit:



Here's one way to layout this circuit:



If you have the black GPIO interface board, layout the circuit as follows instead:



Note that only three transistors have been added (and their respective connections to GPIO pins, +3.3V, and jumper wires connected to pieces of paper with some graphite on them). **It is recommended to use the longest jumper wires to connect the base of the transistors to the pieces of paper.** This makes playing the piano a bit easier. The emitter of the transistor used for the E note is connected to GP16 (P27). Similarly, GP12 (P26) is used for the G note, and GP26 (P25) is used for the B note.

Again, make sure that the collector of each transistor is connected to 3.3V (and not 5V)!

Once you have implemented the additions to the circuit, you will need to change the code to handle the extra notes. First, the extra GPIO pins and the note frequencies are added:

Replace:

```
# setup the pin and frequency for a C note
key = 20
freq = 261.6
```

With:

```
# setup the pins and frequencies for the notes (C, E, G, B)
keys = [ 20, 16, 12, 26 ]
freqs = [ 261.6, 329.6, 392.0, 493.9 ]
notes = []
```

Note that we are using lists to represent the GPIO pins, note frequencies, and the generated notes themselves.

Next, we change the setup of the GPIO pins:

Replace:

```
# setup the input pin
GPIO.setup(key, GPIO.IN, GPIO.PUD_DOWN)
```

With:

```
# setup the input pins
GPIO.setup(keys, GPIO.IN, GPIO.PUD_DOWN)
```

Then, we create the new notes as instances of the Note class:

Replace:

```
# create the actual C note
note = Note(freq, 1)
```

With:

```
# create the actual notes
for n in range(len(freqs)):
    notes.append(Note(freqs[n], 1))
```

Next, we slightly modify the main part of the program so that the note that is pressed is saved, and it's release can be properly detected:

Replace:

```
# detect when Ctrl+C is pressed so that we can reset the GPIO
# pins
try:
    while (True):
        # play a note when pressed...until released
        wait_for_note_start()
        note.play(-1)
        wait_for_note_stop()
        note.stop()
```

```
except KeyboardInterrupt:
```

With:

```
# detect when Ctrl+C is pressed so that we can reset the GPIO
# pins
try:
     while (True):
          # play a note when pressed...until released
          key = wait_for_note_start()
          notes[key].play(-1)
          wait_for_note_stop(keys[key])
          notes[key].stop()
except KeyboardInterrupt:
```

Finally, we must also modify the `wait_for_note_start` and `wait_for_note_stop` functions as follows, so that every note press and release can be detected:

Replace:

```
# waits until a note is pressed
def wait_for_note_start():
     while (not GPIO.input(key)):
          sleep(0.01)

# waits until a note is released
def wait_for_note_stop():
     while (GPIO.input(key)):
          sleep(0.1)
```

With:

```
# waits until a note is pressed
def wait_for_note_start():
     while (True):
          for key in range(len(keys)):
               if (GPIO.input(keys[key])):
                    return key
          sleep(0.01)

# waits until a note is released
def wait_for_note_stop(key):
     while (GPIO.input(key)):
          sleep(0.1)
```

Implement the changes specified above and make sure that your paper piano can play all four notes properly.

For reference, here is the entire modified source code:

```
########################################
# Name:
```

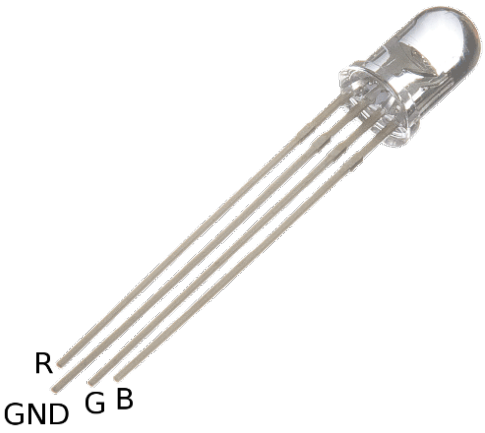
```
# Date:
# Description: Paper piano (v2).
#########################################
import RPi.GPIO as GPIO
from time import sleep
import pygame
from array import array

MIXER_FREQ = 44100
MIXER_SIZE = -16
MIXER_CHANS = 1
MIXER_BUFF = 1024

# the note generator class
class Note(pygame.mixer.Sound):
     # note that volume ranges from 0.0 to 1.0
     def __init__(self, frequency, volume):
          self.frequency = frequency
          # initialize the note using an array of samples
          pygame.mixer.Sound.__init__(self,\
               buffer=self.build_samples())
          self.set_volume(volume)

     # builds an array of samples for the current note
     def build_samples(self):
          # calculate the period and amplitude of the note's wave
          period = int(round(MIXER_FREQ / self.frequency))
          amplitude = 2 ** (abs(MIXER_SIZE) - 1) - 1
          # initialize the note's samples (using an array of
          # signed 16-bit "shorts")
          samples = array("h", [0] * period)

          # generate the note's samples
          for t in range(period):
               if (t < period / 2):
                    samples[t] = amplitude
               else:
                    samples[t] = -amplitude

          return samples

# waits until a note is pressed
def wait_for_note_start():
     while (True):
          for key in range(len(keys)):
               if (GPIO.input(keys[key])):
                    return key
          sleep(0.01)
```

```
# waits until a note is released
def wait_for_note_stop(key):
     while (GPIO.input(key)):
          sleep(0.1)

# preset mixer initialization arguments: frequency (44.1K), size
(16 bits signed), channels (mono), and buffer size (1KB)
# then, initialize the pygame library
pygame.mixer.pre_init(MIXER_FREQ, MIXER_SIZE, MIXER_CHANS,\
     MIXER_BUFF)
pygame.init()

# use the Broadcom pin mode
GPIO.setmode(GPIO.BCM)

# setup the pins and frequencies for the notes (C, E, G, B)
keys = [ 20, 16, 12, 26 ]
freqs = [ 261.6, 329.6, 392.0, 493.9 ]
notes = []

# setup the input pins
GPIO.setup(keys, GPIO.IN, GPIO.PUD_DOWN)

# create the actual notes
for n in range(len(freqs)):
     notes.append(Note(freqs[n], 1))

# the main part of the program
print "Welcome to Paper Piano!"
print "Press Ctrl+C to exit..."

# detect when Ctrl+C is pressed so that we can reset the GPIO
# pins
try:
     while (True):
          # play a note when pressed...until released
          key = wait_for_note_start()
          notes[key].play(-1)
          wait_for_note_stop(keys[key])
          notes[key].stop()
except KeyboardInterrupt:
     # reset the GPIO pins
     GPIO.cleanup()
```

➔ LESSON BREAK – LESSON BREAK – LESSON BREAK – LESSON BREAK ➔

Part three: recording and playback

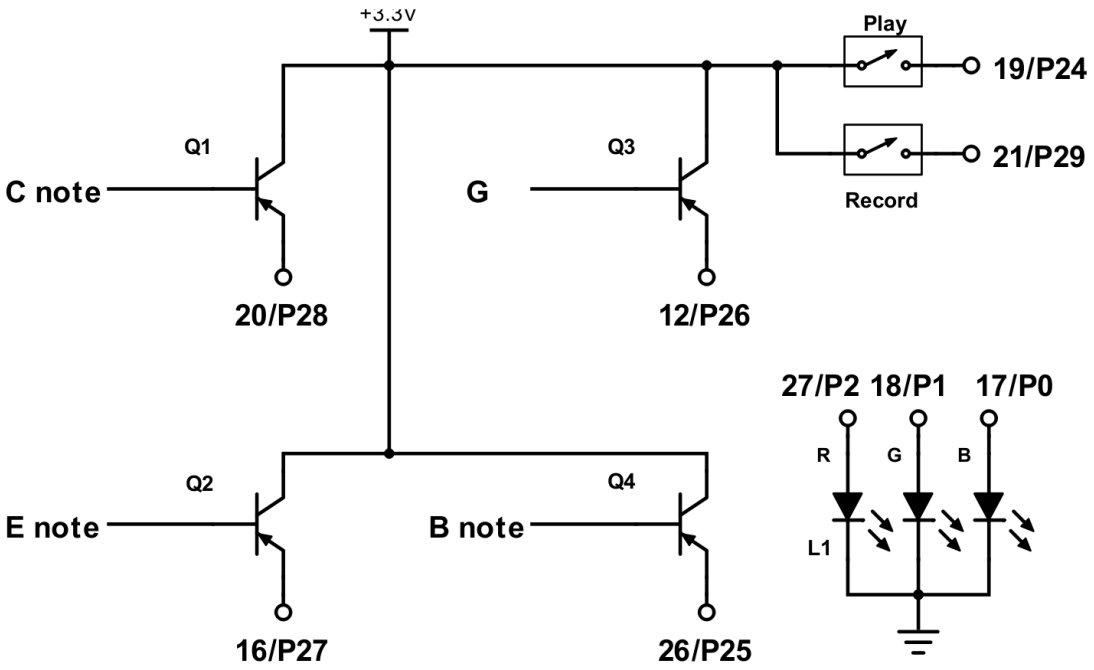
In this part of the activity, you will implement a record and playback feature. Two pushbuttons will be added (one for recording and one for playback). In addition, a RGB LED will be used to provide cues when the paper piano is recording (red LED) and playing back (green LED). Although two separate LEDs could be used for this, using the RGB LED will introduce you to the RGB LED. Let's take a closer look at the RGB LED:



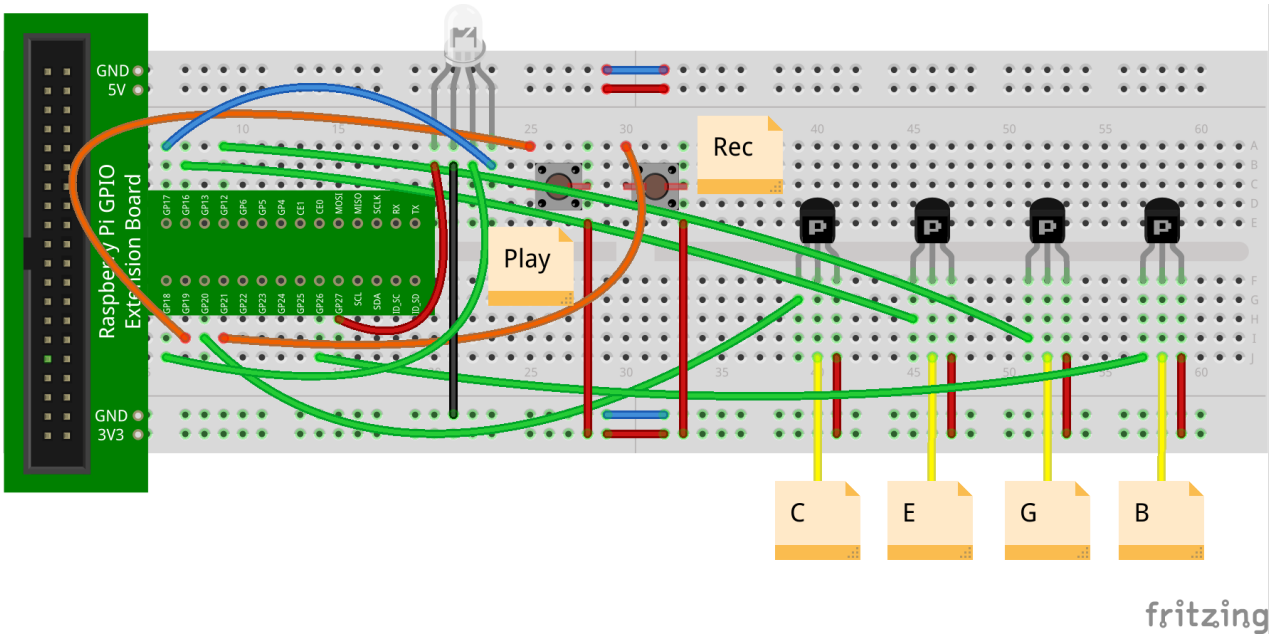
If oriented such that the longest leg is the second-from-the-left, then the legs in order from left-to-right are: red, ground, green, and blue. RGB LEDs that share a common ground are called common cathode RGB LEDs. This makes sense, since the ground leg of the LED is the cathode. As with the transistor, make sure that each leg of the LED is in a different column when inserting it into the breadboard. As expected, the ground leg is connected to ground. The three other (color) legs are connected to output pins on the RPi.

You will also note that there is no resistor on the RGB LED. Although one could be put in series with the ground leg, it really isn't needed since the RPi will only be supplying 3.3V. This is fine when turning on any of the color legs. You may notice that the red color LED is a bit dim when compared to green and blue. Although the activity uses the red LED to indicate that the paper piano is recording, you may instead use blue if desired. Green will be used to indicate playback.

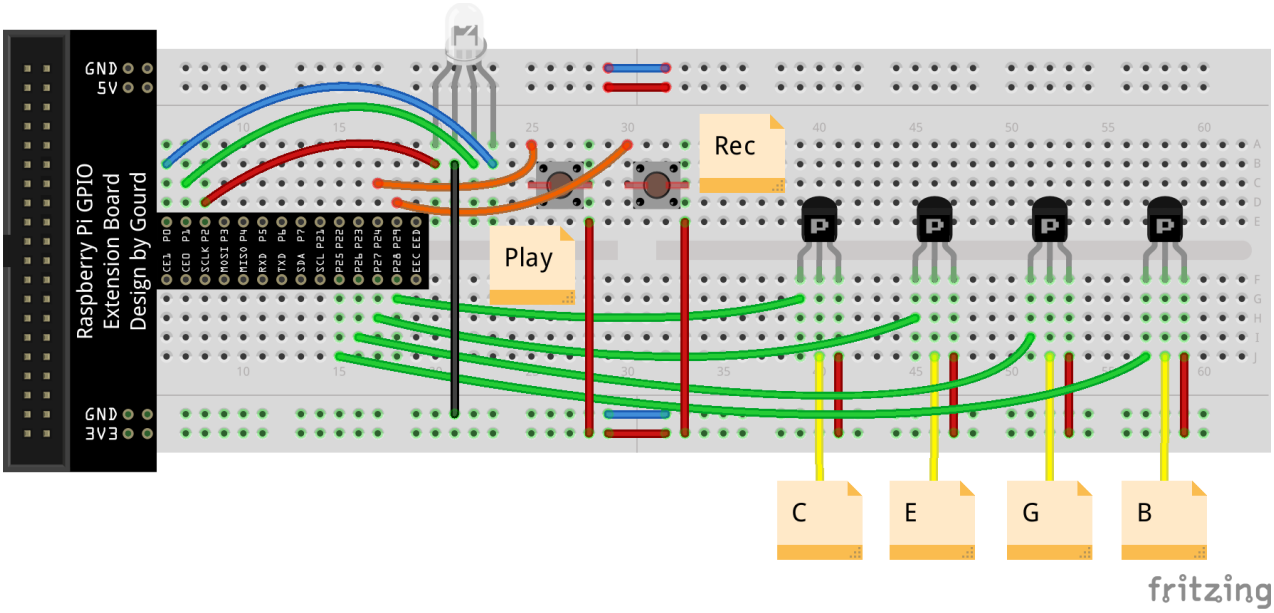
Here's the updated circuit:



Here's one way to layout this circuit:



If you have the black GPIO interface board, layout the circuit as follows instead:



Note that the right portion of the circuit with the transistors is the same as in the previous part of the activity. Also note that the play and record labels on the diagram simply serve to identify them. That is, there is no need to make labels with graphite on them since we are using pushbutton switches (and not capacitive touch switches) for record and playback functionality.

For reference, the play button is wired to GP19 (P24), and the record button is wired to GP21 (P29). The pushbuttons will also be wired to +3.3V; therefore, the input pins on the RPi will be configured so that they are pulled down (to ground) by default. The RGB LED colors are wired as follows: red to GP27 (P2), green to GP18 (P1), and blue to GP17 (P0).

The source code will need pretty major changes. First, detecting input now includes the pushbutton switches for recording and playback. Of course, these trigger the RGB LED as well. In addition, we need to support recording as the user plays the paper piano. This means that both the notes played and their duration (i.e., how long the notes were held down) must be stored. In addition, the duration of any silence in between the notes played must also be recorded! Playback of the recorded "song" must be able to play the notes and their duration, and any silence in between the notes.

We will structure the behavior of the recording and playback functionality so that the record button starts and stops recording. That is, if the program is not currently in a recording state, pressing the record button will put it in that state and turn on the red color LED. Pressing the record button again will toggle the recording state (to off) and turn off the LED. Pressing the play button will turn on the green color LED and play the recorded song. If the program is in a recording state, the play button will also stop recording and trigger the recording state (to off), turn the green color LED on, and play the song. That is, the play button will also stop recording if the program is in the recording state.

Let's take a look at the code. First, to implement timing (e.g., to record the duration of notes and silences), we import the time function from the time library:

Replace:

```
from time import sleep
```

With:

```
from time import sleep, time
```

The next change occurs in the `wait_for_note_start` function. We need to be able to detect when the play and record buttons are triggered, in addition to the notes:

Replace:

```
def wait_for_note_start():
    while (True):
        for key in range(len(keys)):
            if (GPIO.input(keys[key])):
                return key
        sleep(0.01)
```

With:

```
def wait_for_note_start():
    while (True):
        # first, check for notes
        for key in range(len(keys)):
            if (GPIO.input(keys[key])):
                return key
        # next, check for the play button
        if (GPIO.input(play)):
            # debounce the switch
            while (GPIO.input(play)):
                sleep(0.01)
            return "play"
        # finally, check for the record button
        if (GPIO.input(record)):
            # debounce the switch
            while (GPIO.input(record)):
                sleep(0.01)
            return "record"
        sleep(0.01)
```

The strategy is to first check if a note is being played. If so, its key is returned. If not, we then check if the play button is being pressed. If so, we debounce it by waiting until it is no longer pressed; then, we return the string "play." If the play button is not being pressed, then we check if the record button is being pressed. If so, it is handled similarly to the play button; however, the returned string is "record." If no button or note is being pressed, the program sleeps for a short while and checks again.

Next, we add the setup for the new GPIO pins (both for recording and playback, and for the RGB LED):

Replace:

```
# setup the pins and frequencies for the notes (C, E, G, B)
keys = [ 20, 16, 12, 26 ]
freqs = [ 261.6, 329.6, 392.0, 493.9 ]
notes = []
```

```
# setup the input pins
GPIO.setup(keys, GPIO.IN, GPIO.PUD_DOWN)

# create the actual notes
for n in range(len(freqs)):
     notes.append(Note(freqs[n], 1))
```

With:

```
# setup the pins and frequencies for the notes (C, E, G, B)
keys = [ 20, 16, 12, 26 ]
freqs = [ 261.6, 329.6, 392.0, 493.9 ]
notes = []

# setup the button pins
play = 19
record = 21

# setup the LED pins
red = 27
green = 18
blue = 17 # if red is too dim, use blue

# setup the input pins
GPIO.setup(keys, GPIO.IN, GPIO.PUD_DOWN)
GPIO.setup(play, GPIO.IN, GPIO.PUD_DOWN)
GPIO.setup(record, GPIO.IN, GPIO.PUD_DOWN)

# setup the output pins
GPIO.setup(red, GPIO.OUT)
GPIO.setup(green, GPIO.OUT)
GPIO.setup(blue, GPIO.OUT)

# create the actual notes
for n in range(len(freqs)):
     notes.append(Note(freqs[n], 1))
```

Note that no actual code was replaced or removed (i.e., only new statements were added). First, we add the input pin definitions for the record and play buttons. Then, we add the output pin definitions for the RGB LED. Lastly, we add the GPIO setup code for the new input and output pins.

Then, we initialize a list that will contain the recorded song. We also initialize the recording state to false. This is done before the actual start of the main part of the program:

Replace:

```
# the main part of the program
print "Welcome to Paper Piano!"
print "Press Ctrl+C to exit..."
```

With:

```
# begin in a non-recording state and initialize the song
recording = False
song = []

# the main part of the program
print "Welcome to Paper Piano!"
print "Press Ctrl+C to exit..."
```

Each element in the song list will contain both the note played (or a silence), along with its duration. These will both be stored as a list. So, the song list is a list of lists. Here's an example of one that contains the note C for 3.5s, a silence for 1.2s, and the note G for 2.45s:

```
[ [ "C", 3.5 ], [ "SILENCE", 1.2 ], [ "G", 2.45 ] ]
```

Next, we modify the loop in the main part of the program:

Replace;

```
try:
    while (True):
        # play a note when pressed...until released
        key = wait_for_note_start()
        notes[key].play(-1)
        wait_for_note_stop(keys[key])
        notes[key].stop()
except KeyboardInterrupt:
```

With:

```
try:
    while (True):
        # start a timer
        start = time()
        # play a note when pressed...until released (also
        # detect play/record)
        key = wait_for_note_start()
        # note the duration of the silence
        duration = time() - start
        # if recording, append the duration of the silence
        if (recording):
            song.append(["SILENCE", duration])
        # if the record button was pressed
        if (key == "record"):
            # if not previously recording, reset the song
            if (not recording):
                song = []
            # note the recording state and turn on the red LED
            recording = not recording
            GPIO.output(red, recording)
        # if the play button was pressed
        elif (key == "play"):
            # if recording, stop
```

```
                if (recording):
                    recording = False
                    GPIO.output(red, False)
                # turn on the green LED
                GPIO.output(green, True)
                # play the song
                play_song()
                GPIO.output(green, False)
            # otherwise, a piano key was pressed
            else:
                # start the timer and play the note
                start = time()
                notes[key].play(-1)
                wait_for_note_stop(keys[key])
                notes[key].stop()
                # once the note is released, stop the timer
                duration = time() - start
                # if recording, append the note and its duration
                if (recording):
                    song.append([key, duration])
    except KeyboardInterrupt:
```

There's a lot going on here. First, we start a timer to record any silence while waiting for a button to be pushed or note to be pressed. Once any input is received, the duration of the silence is stored. If the program is in a recording state, the silence and its duration is added to the song. If not, it is ignored.

Next, we determine what button was pressed or note was pushed. If the record button was pressed, we then check if the program is not in a recording state. If that is the case, then the song is initialized (or possibly reinitialized if a song already existed). The recording state is then toggled, and the red color LED is turned on. The loop then iterates again, recording the silence and waiting for a button to be pushed or a note to be pressed.

If the play button is pressed, we first check if the program is in a recording state. If so, the recording state is toggled, and the red color LED is turned off. Next, the green color LED is turned on, and the song is played (via a `play_song` function). Once the song has played, the green color LED is turned off. The loop then iterates again, recording the silence and waiting for a button to be pushed or a note to be pressed.

Lastly, the else part of the main if statement detects if a note is pressed. If so, the a timer is started to record the length of the note as it is played. The note is then played while the program waits for it to be released. Once this happens, the note stops playing, and its duration is stored. Lastly, if the program is in a recording state, the note and its duration are added to the song. The loop then iterates again, recording the silence and waiting for a button to be pushed or a note to be pressed. Of course, the program stops when the user presses Ctrl+C.

The last addition to the program is the `play_song` function. It can be placed below the `wait_for_note_stop` function:

```
# plays a recorded song
def play_song():
     # each element in the song list is a list composed of two
     # parts: a note (or silence) and a duration
     for part in song:
          note, duration = part
          # if it's a silence, delay for its duration
          if (note == "SILENCE"):
               sleep(duration)
          # otherwise, play the note for its duration
          else:
               notes[note].play(-1)
               sleep(duration)
               notes[note].stop()
```

Since the song list is initialized (and modified) in the main part of the program, it can be directly accessed in the `play_song` function. The function first iterates through each list in the song list. Remember that the song list contains lists! Each of the "sublists" contains either a note or a silence, and its duration.

The function breaks each part of the song into its note (or silence) and duration. If the part is a silence (i.e., it contains the string "SILENCE"), the program sleeps for its duration. Otherwise, the part contains a note that is played for its duration, and then stopped.

Once the entire song has been played, control is transferred back to the main part of the program, at which point the green color LED is turned off, and the program waits for another button to be pressed or note to be pushed.

For completeness, the entire modified source code is listed here:

```
########################################
# Name:
# Date:
# Description: Paper piano (v3).
########################################
import RPi.GPIO as GPIO
from time import sleep, time
import pygame
from array import array

MIXER_FREQ = 44100
MIXER_SIZE = -16
MIXER_CHANS = 1
MIXER_BUFF = 1024

# the note generator class
class Note(pygame.mixer.Sound):
     # note that volume ranges from 0.0 to 1.0
```

```
     def __init__(self, frequency, volume):
          self.frequency = frequency
          # initialize the note using an array of samples
          pygame.mixer.Sound.__init__(self,\
               buffer=self.build_samples())
          self.set_volume(volume)

     # builds an array of samples for the current note
     def build_samples(self):
          # calculate the period and amplitude of the note's wave
          period = int(round(MIXER_FREQ / self.frequency))
          amplitude = 2 ** (abs(MIXER_SIZE) - 1) - 1
          # initialize the note's samples (using an array of
          # signed 16-bit "shorts")
          samples = array("h", [0] * period)

          # generate the note's samples
          for t in range(period):
               if (t < period / 2):
                    samples[t] = amplitude
               else:
                    samples[t] = -amplitude

          return samples

# waits until a note is pressed
def wait_for_note_start():
     while (True):
          # first, check for notes
          for key in range(len(keys)):
               if (GPIO.input(keys[key])):
                    return key
          # next, check for the play button
          if (GPIO.input(play)):
               # debounce the switch
               while (GPIO.input(play)):
                    sleep(0.01)
               return "play"
          # finally, check for the record button
          if (GPIO.input(record)):
               # debounce the switch
               while (GPIO.input(record)):
                    sleep(0.01)
               return "record"
          sleep(0.01)

# waits until a note is released
def wait_for_note_stop(key):
```

```
     while (GPIO.input(key)):
          sleep(0.1)

# plays a recorded song
def play_song():
     # each element in the song list is a list composed of two
     # parts: a note (or silence) and a duration
     for part in song:
          note, duration = part
          # if it's a silence, delay for its duration
          if (note == "SILENCE"):
               sleep(duration)
          # otherwise, play the note for its duration
          else:
               notes[note].play(-1)
               sleep(duration)
               notes[note].stop()

# preset mixer initialization arguments: frequency (44.1K), size
# (16 bits signed), channels (mono), and buffer size (1KB)
# then, initialize the pygame library
pygame.mixer.pre_init(MIXER_FREQ, MIXER_SIZE, MIXER_CHANS,\
     MIXER_BUFF)
pygame.init()

# use the Broadcom pin mode
GPIO.setmode(GPIO.BCM)

# setup the pins and frequencies for the notes (C, E, G, B)
keys = [ 20, 16, 12, 26 ]
freqs = [ 261.6, 329.6, 392.0, 493.9 ]
notes = []

# setup the button pins
play = 19
record = 21

# setup the LED pins
red = 27
green = 18
blue = 17 # if red is too dim, use blue

# setup the input pins
GPIO.setup(keys, GPIO.IN, GPIO.PUD_DOWN)
GPIO.setup(play, GPIO.IN, GPIO.PUD_DOWN)
GPIO.setup(record, GPIO.IN, GPIO.PUD_DOWN)

# setup the output pins
```

```
GPIO.setup(red, GPIO.OUT)
GPIO.setup(green, GPIO.OUT)
GPIO.setup(blue, GPIO.OUT)

# create the actual notes
for n in range(len(freqs)):
     notes.append(Note(freqs[n], 1))

# begin in a non-recording state and initialize the song
recording = False
song = []

# the main part of the program
print "Welcome to Paper Piano!"
print "Press Ctrl+C to exit..."

# detect when Ctrl+C is pressed so that we can reset the GPIO
# pins
try:
     while (True):
          # start a timer
          start = time()
          # play a note when pressed...until released (also
          # detect play/record)
          key = wait_for_note_start()
          # note the duration of the silence
          duration = time() - start
          # if recording, append the duration of the silence
          if (recording):
               song.append(["SILENCE", duration])
          # if the record button was pressed
          if (key == "record"):
               # if not previously recording, reset the song
               if (not recording):
                    song = []
               # note the recording state and turn on the red LED
               recording = not recording
               GPIO.output(red, recording)
          # if the play button was pressed
          elif (key == "play"):
               # if recording, stop
               if (recording):
                    recording = False
               # turn on the green LED
               GPIO.output(red, False)
               GPIO.output(green, True)
               # play the song
               play_song()
```

```
                    GPIO.output(green, False)
               # otherwise, a piano key was pressed
               else:
                    # start the timer and play the note
                    start = time()
                    notes[key].play(-1)
                    wait_for_note_stop(keys[key])
                    notes[key].stop()
                    # once the note is released, stop the timer
                    duration = time() - start
                    # if recording, append the note and its duration
                    if (recording):
                         song.append([key, duration])
     except KeyboardInterrupt:
          # reset the GPIO pins
          GPIO.cleanup()
```

Make sure to test the program with several runs of different behaviors. Try recording a few songs, some that don't have notes, some with different silences and durations, and so on.

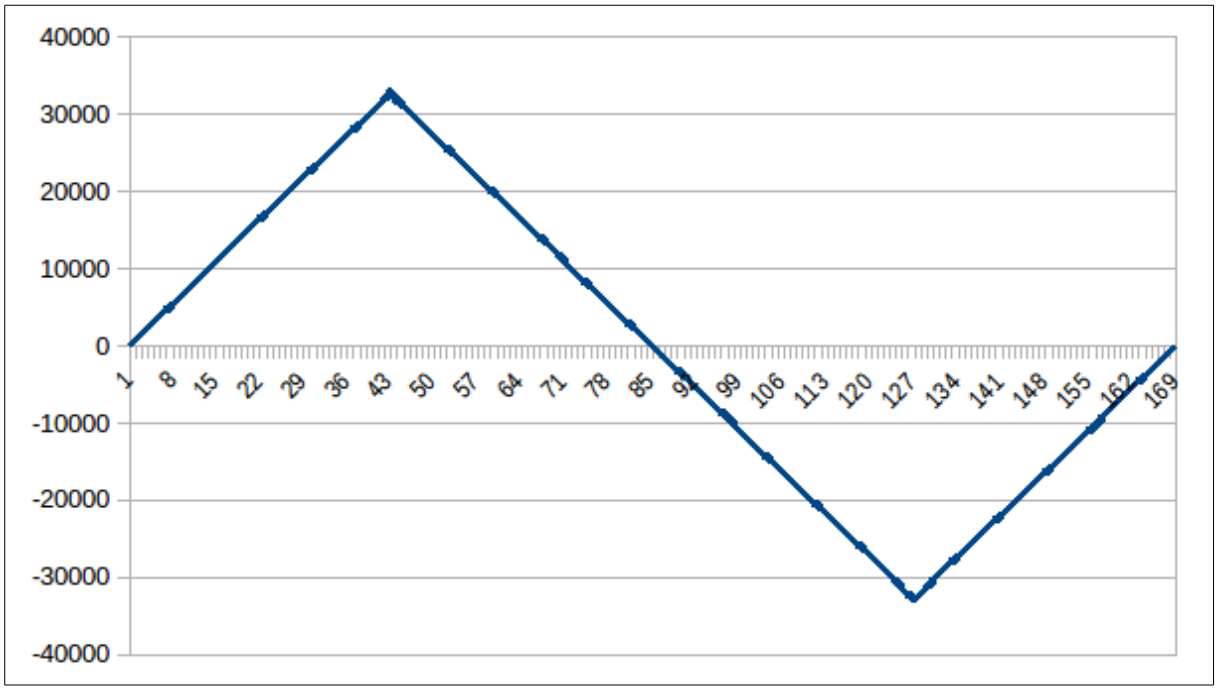
Homework: Paper Piano

An interesting tweak to this activity is to change the shape of the wave for each note as generated by the mixer. That is, keep the frequency and sample rate constant, but change the amplitudes across the samples so that the shape of the wave changes. As noted earlier, the current program generates square waves. This is done in the `build_samples` function of the Note class:

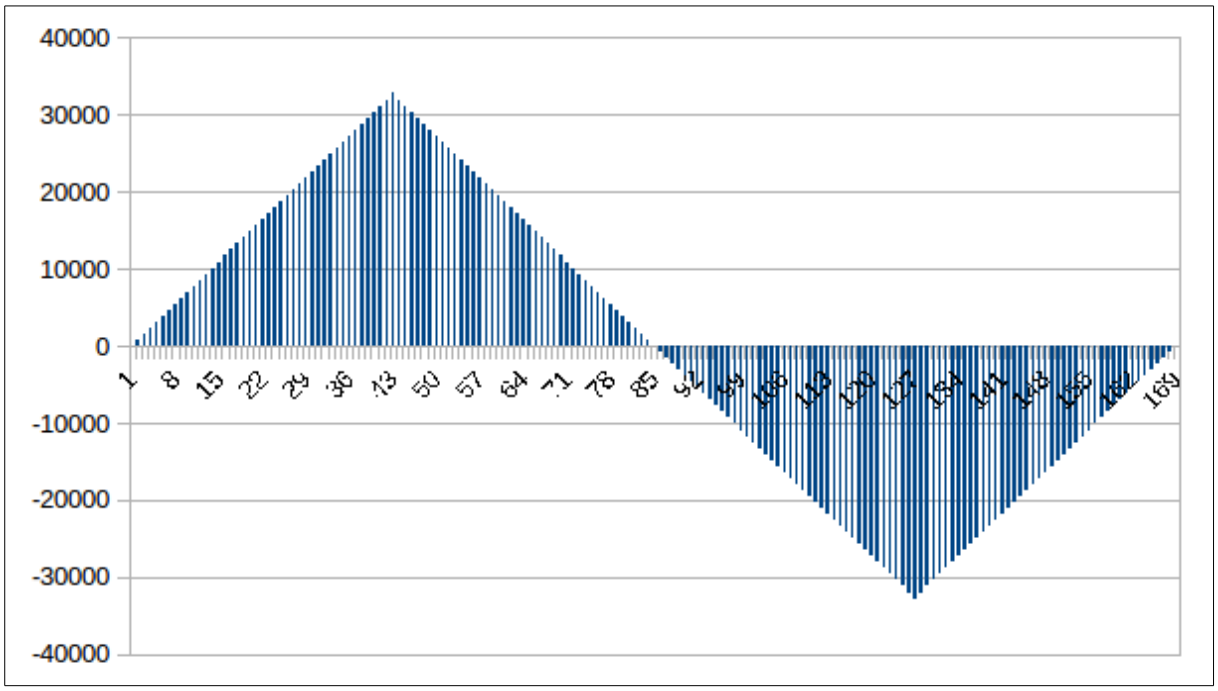
```
     # generate the note's samples
     for t in range(period):
          if (t < period / 2):
               samples[t] = amplitude
          else:
               samples[t] = -amplitude
```

Note that half of the samples are at the maximum amplitude, and half are at the negative of the maximum amplitude.

There are other kinds of waves (or wave approximations) that can be generated, however. Consider a modification that forms a **triangle wave**:



We could, instead, fill the array of samples so that it forms a triangle wave as follows:

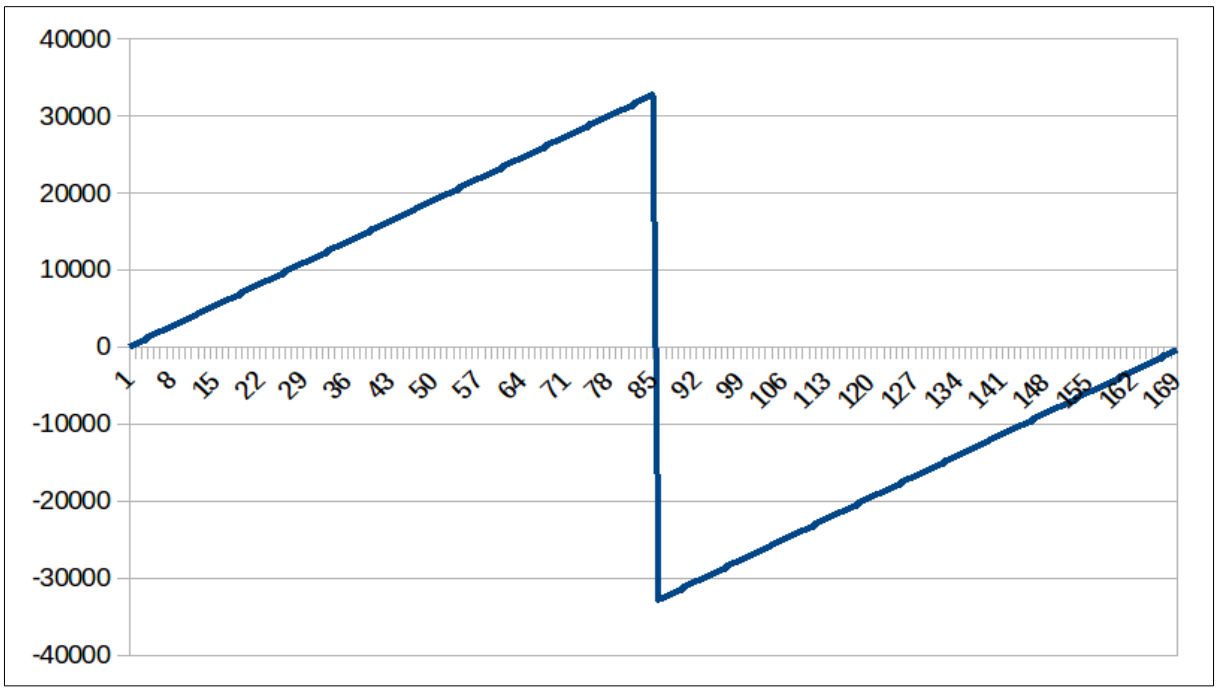


The modification to the source code snippet shown above is fairly simple. Algorithmically, we simply need to increase the magnitude of each sample from 0 to 32,767 (the maximum amplitude, also the maximum value for 16-bit signed numbers) until we reach one fourth of the total number of samples. For a middle C at 261.6 Hz with 169 total samples in a period within a 44.1 KHz sample rate, that's about 42 samples. The increase (or delta) at each sample can easily be calculated. Then, decrease the magnitude of each sample from 32,767 to -32,768 (the smallest value for 16-bit signed numbers)

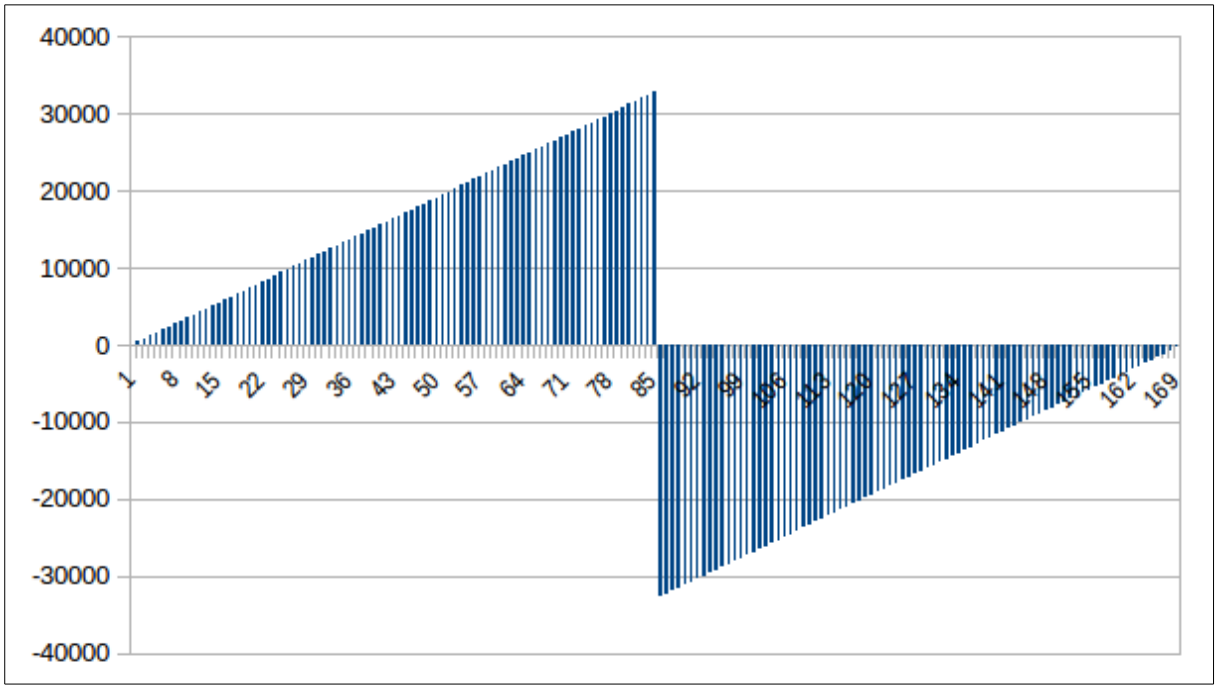
through the next half of the samples (i.e., through three quarters of the total number of samples – about 85 samples). Finally, increase the magnitude of each sample from -32,768 to 0 until we reach the end of the total number of samples.

What does such a sound wave sound like?

What about a wave that looks jagged (like saw teeth). Such a wave is called a **sawtooth wave**:

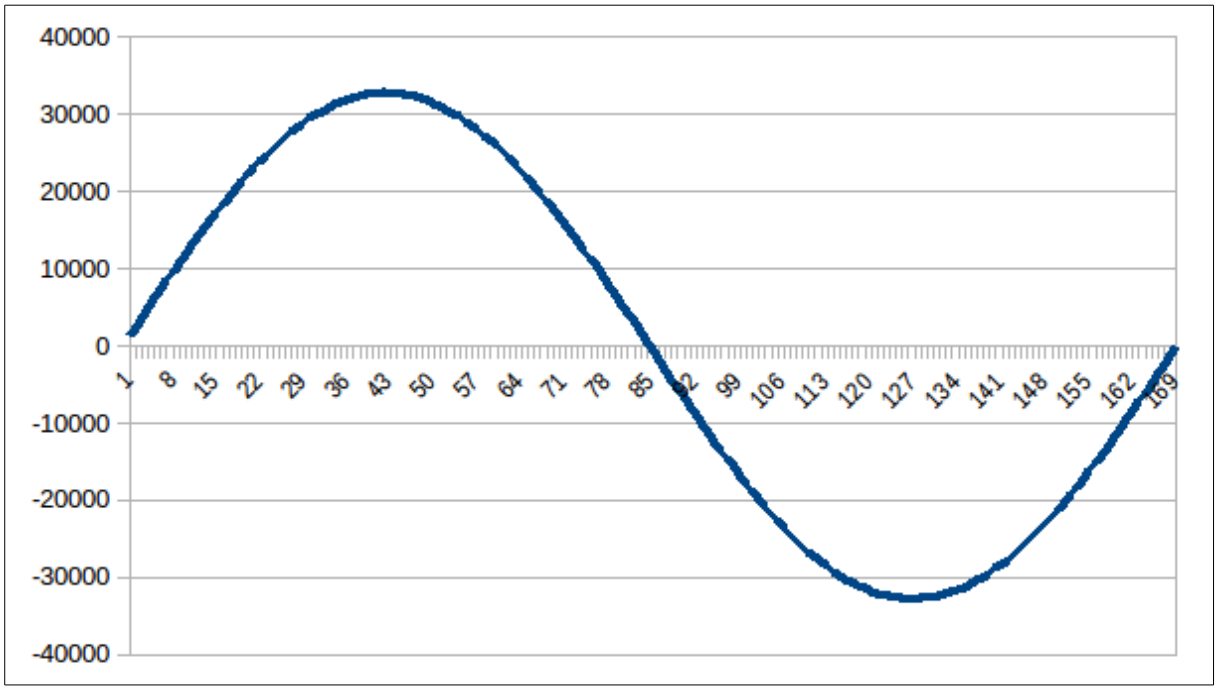


Similarly, we could fill the array of samples so that it forms a sawtooth wave as follows:

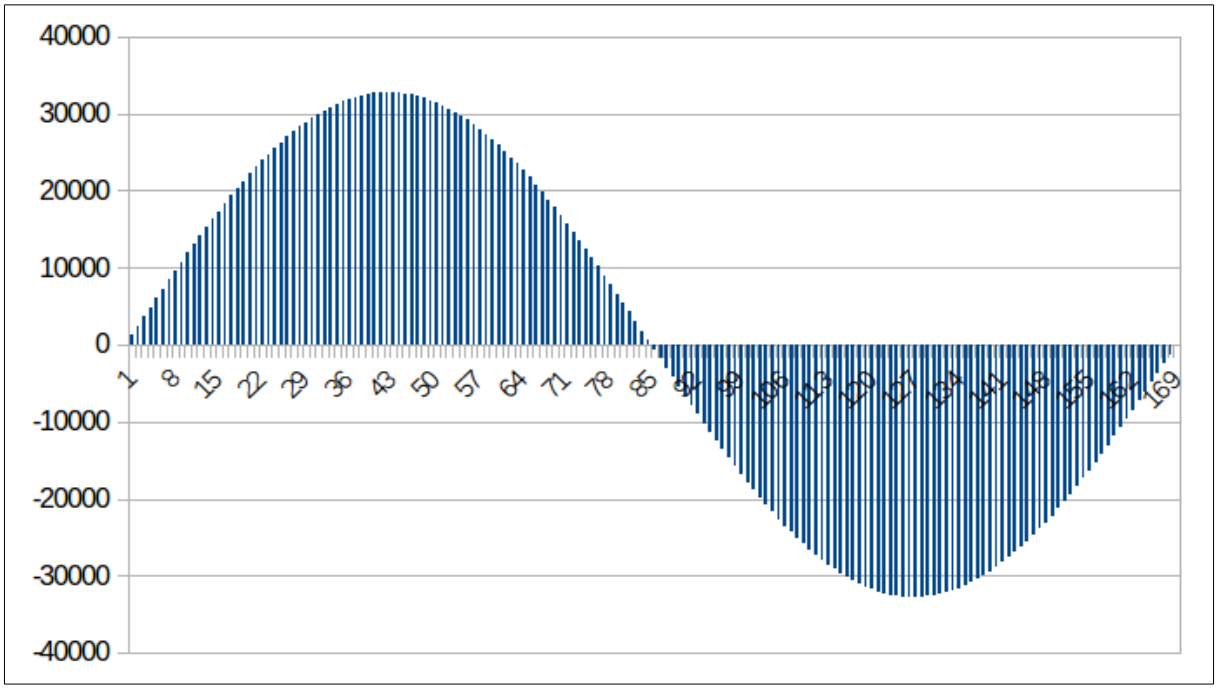


The modification to the snippet of code for this type of wave is also fairly simple. How different does this type of wave sound as compared to the others?

Finally, perhaps the most difficult set of samples to generate are those that represent the **sinusoidal wave** as accurately as possible:



Again, we could fill the array of samples so that it forms a sinusoidal wave as follows:



How different does this type of wave sound as compared to the others? Although this is possible by modifying the snippet of code above, it will require more changes than the other types of waves. In fact, various trigonometric functions will be necessary.

Your task is to modify the Paper Piano to implement **at least two** of the types of waves shown above (triangular, sawtooth, and sinusoidal) **in addition to the existing square wave**. **Implementing all three may result in bonus points!**

The behavior of the Paper Piano will change slightly to accommodate the changes specified. Each of the note inputs must be set to the same note (and same frequency). Therefore, the piano no longer plays C, E, G, and B. Instead, it plays a single note **(let's agree to use a middle C at 261.6 Hz)**; however, it does so using different wave forms: pressing one of the keys plays a middle C using a square wave, pressing another key plays a middle C using another wave form of your choosing (e.g., sawtooth), and so on. If you implement two of the additional wave forms, then your piano will only require three keys. Implementing all three will require all four keys.

Efficient solutions to adding the various wave forms will include a modification of the Note class to accept a wave form type upon instantiation. Sample building will then depend on type of the wave form for the current note object.

Although you may remove the play and record buttons (and their functionality), this is not necessary. In fact, it may be interesting to record and playback the differences in the wave forms. **If you choose to remove the play and record buttons, it will be much easier if you begin with the source code specified in part two of this activity.**

Include some comments in your source code that briefly discuss the differences in how the different wave forms for a middle C sound to you.

You may have the option to work in **groups** (pending prof approval). It is suggested that groups contain at least one confident Python coder. Make sure to put an appropriate header at the top of your program (with the names of everyone in your group, if allowed and applicable) and to appropriately comment your source code as necessary. **Only submit your source code (i.e., a single `.py` file).**