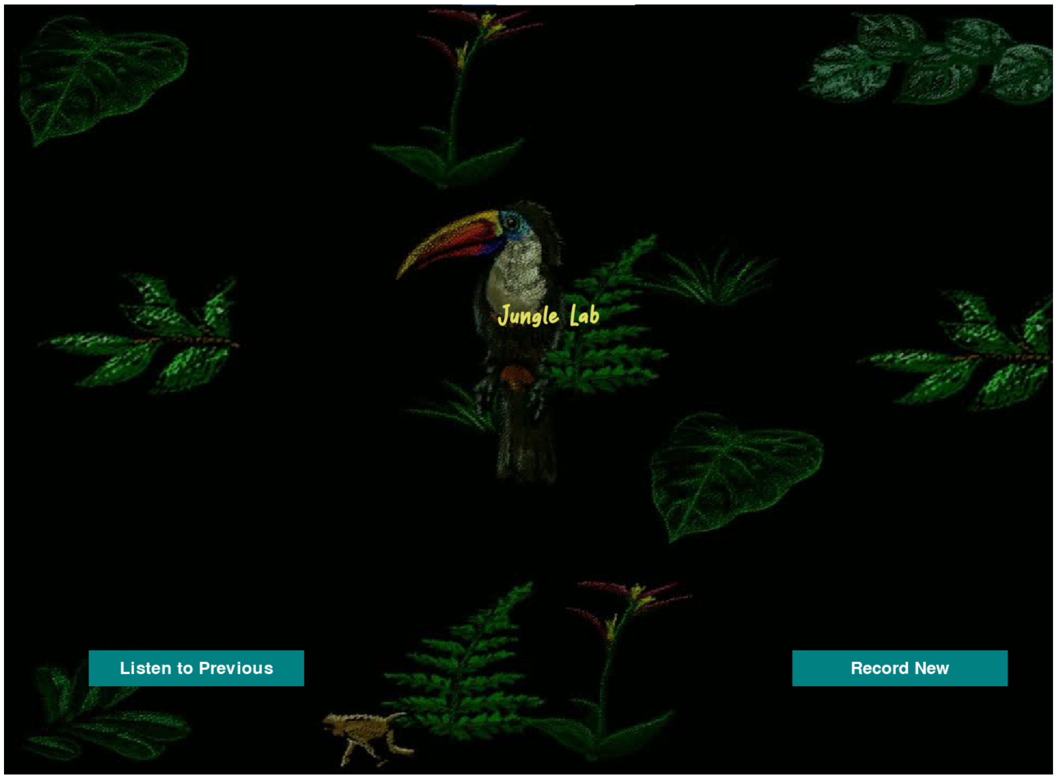
🌴

# The Jungle Lab

| Tags | DnB breakbeats |
|---|---|

## Introduction



The Jungle Lab is an installation piece that allows users to experiment with jungle music/DnB/breakcore in an interactive and non-direct manner. There are two components to this piece, the jungle controller which is a cardboard enclosure featuring sensors and an ESP32, and a host device/RPi running SuperCollider and a PyGame application (pictured above). Approaching users can choose to listen to a previously made 10-second beat sequence, or to record a new one. If they choose to record a new one, they must interact with the device in order to modify the effects on the pre-recorded drum sequence. The audio is not heard immediately however, and can only be listened to once the recording process has finished, mystifying the process behind how the device can be interacted with in order to produce sounds and encouraging the user to experiment multiple times.

This project is inspired by the Jungle music genre and its iconic use of filter sweeps and other effects on drum loops.

## Demo

In the video demo, I record 3 different drum sequences and perform different actions on the box. Notice how the LED blinks when recording is in progress. After each recording, I press the “Listen to Previous” button on the UI to listen to the sequence I just created. Once a recording has been completed, it is impossible to listen to the previous sequences behind it. They are all overwritten.

https://www.youtube.com/watch?v=zphP_vRob-0

## Artistic Principles

The two parameters that affect the recorded drumbeats are the box’s tilt and vibration. Tilting the box downwards and to the right increases the central frequency of a bandpass filter on the drum loop. Tilting the box back towards a neutral position will lower the filter’s frequency. Generating vibrations on the box will also increase the playback rate of the drum loop. These vibrations can be generated via tapping, shaking, or running around while holding the box.

### Why Wireless

This project takes advantage of wireless technologies as the box is completely enclosed and able to freely move around. One way of generating vibrations on the piezo sensor would be to hold the box in one's hand and walk/run around. Having the box completely closed off adds to the mystery behind the box’s purpose. It is purposefully very plain and only contains an on/off switch and a status LED to show the current state of the program.

### The Unseen Effect and Experimentation

What makes this design interesting is that users of this installation will not know how the parameters of the sound are mapped to interaction with the box. When they interact with the box in a recording session, they cannot hear the effects of their action in real-time. Only after the recording has finished will they be able to listen back. This is my implementation of the “unseen effect”. This encourages users to experiment with multiple recording sessions, trying different physical actions with the box to see what, how, and if any, effects are changed.

## Design/Implementation

### SuperCollider

The SuperCollider portion of the project controls the creation and application of effects on the drum sequences. There is a single Synth that can play portions of a sampled amen break in a loop. The start of the loop, bandpass filter frequency, and playback rate of the sample are all parameterized. Instead of routing the audio out to an audio bus, the audio is placed into a buffer called `~recordBuffer`

```
b = Buffer.read(s, ~currentDir +/+ "samples/amen.wav");
SynthDef(\amen, {| out = 0, srate = 1, filterfreq = 500, startPos = 0 |
    var audioOut;
    audioOut = BPF.ar(
      PlayBuf.ar(
        2, b, srate, startPos: startPos, loop: 1),
      filterfreq);

    RecordBuf.ar(audioOut,
      ~recordBuffer
    );
    // plays audio out in real-time, use for debugging but NOT for prod
    // as that defeats the whole purpose
    // Out.ar(0, audioOut);
  }).add();
```

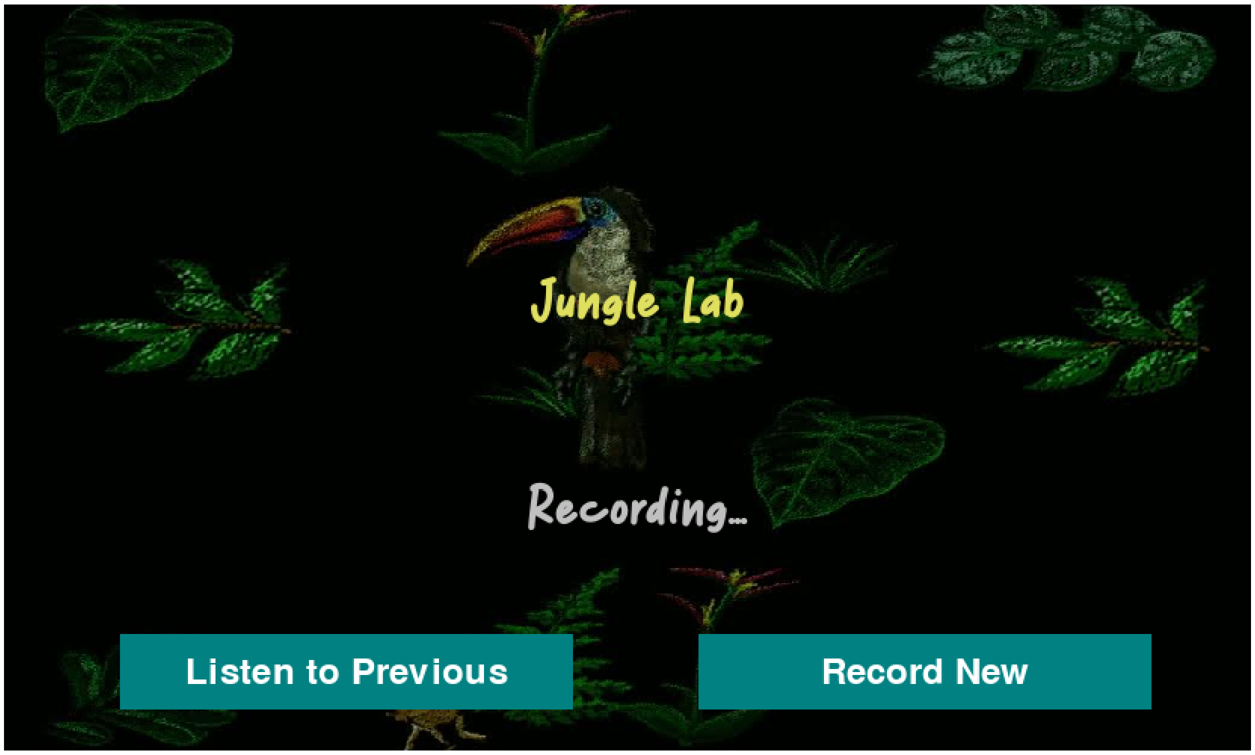
Then I define OSC listener functions to manipulate the drum loop synth and handle the recording/writing to disc process as needed. The ESP32 sends messages to the `/amen_synth` topic to update the synth while the PyGame controller sends messages to `/start_record` and `/stop_record` when needed.

```
OSCFunc({ |msg, time, addr, recvPort|
    // srate, filter
    ~amenSynth.set(\srate, msg[1], \filterfreq, msg[2]);
  }, '/amen_synth');


  OSCFunc({ |msg, time, addr, recvPort|
    "started".postln;
    ~amenSynth = Synth(\amen, [\startPos, b.numFrames.rand]);
  }, '/start_record');



  OSCFunc({ |msg, time, addr, recvPort|
    "stopped".postln;
    ~amenSynth.free;
    ~recordBuffer.write("/home/student334/Documents/school/cpsc334/module3/task1/produced_sound.wav", "WAVE");
  }, '/stop_record');
```

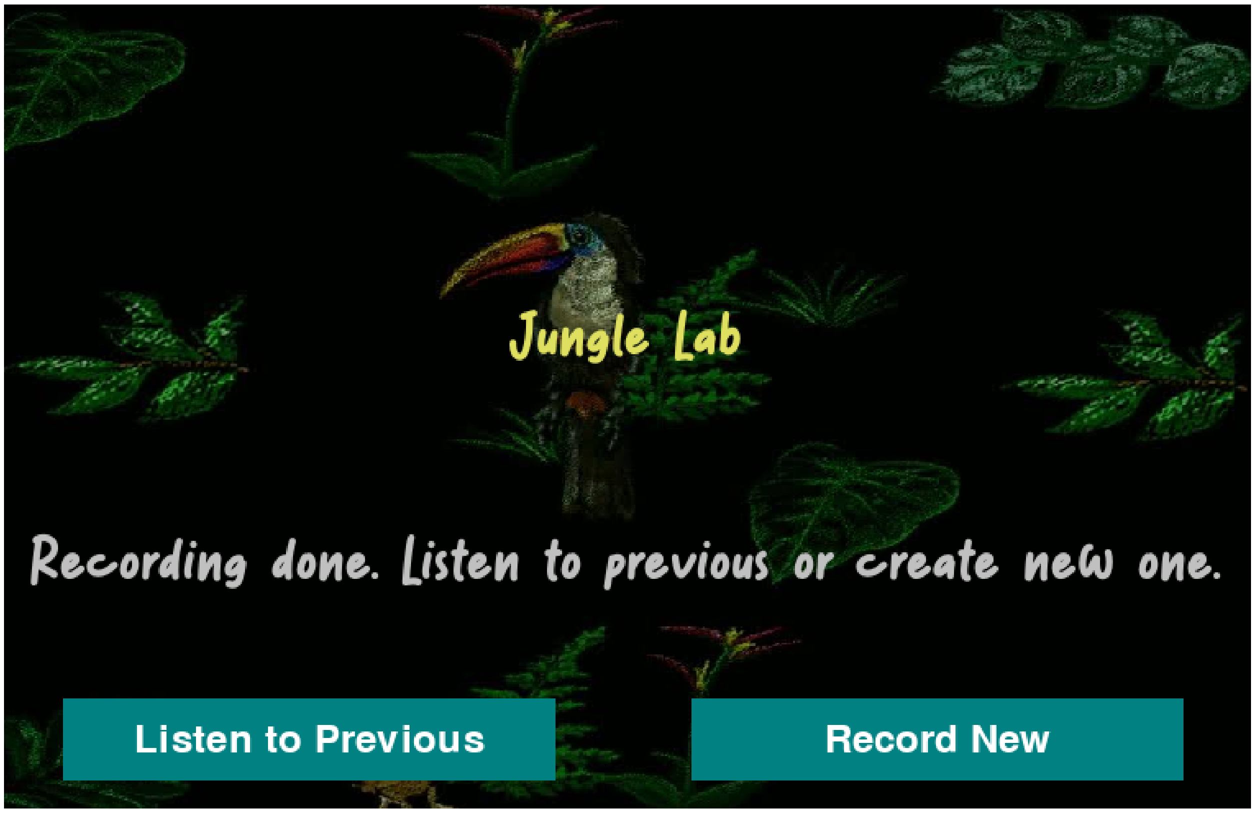
## PyGame Interface

The PyGame interface is used to coordinate the recording/playback of the installation. Users can choose to listen to the previously created drum sequence or to record a new one. This interface communicates with both the ESP32 and SuperCollider to coordinate the recording process. When a user presses the “Record New” button, the interface sends a message to the ESP32 to put it in recording mode and a message to SuperCollider to start the synth. After 10 seconds has passed in recording mode, the ESP32 sends a done message back to the interface. Once the done message is received, the UI updates accordingly and the interface sends a message to SuperCollider to stop the recording and save the recorded sequence onto disc.

**Recording process started:**



**Recording process ended:**



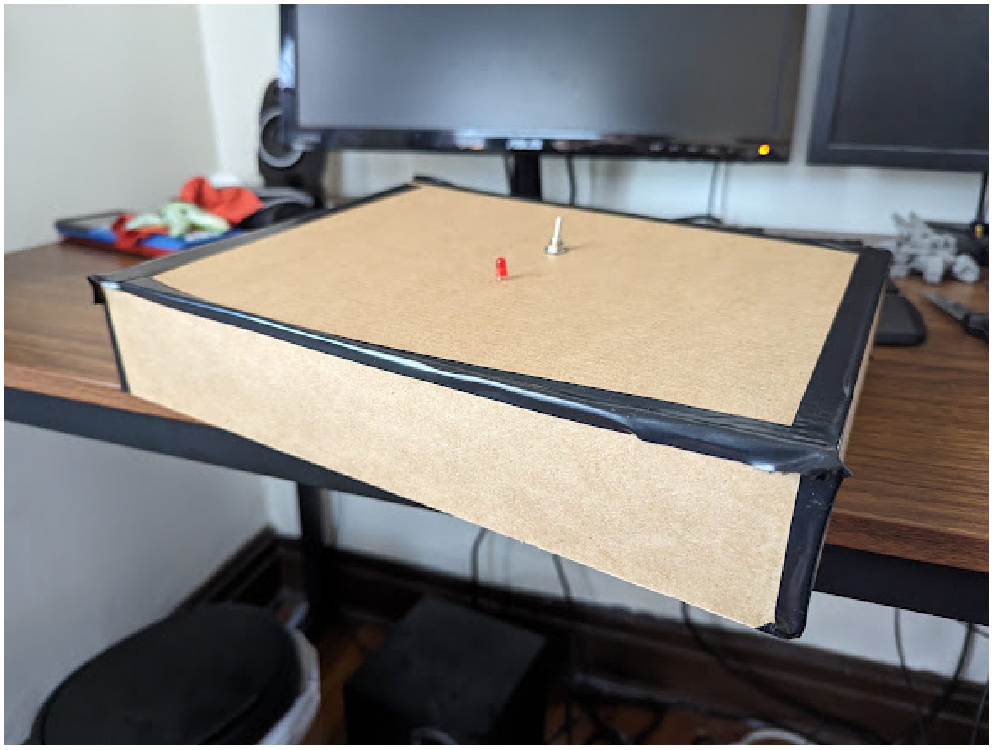
## Communication

I used UDP throughout the entire communication stack. If I were to redesign this project though, I would definitely spend time to make the state switching messages be sent in TCP instead to ensure that a stable connection is established between the ESP32 and the Python program. UDP works great when dropped packets every now and then are ok (updating the Synth with new parameters from the sensor for example). UDP is not great when you want to send a single message that is extremely important (a program state change for example). These should be done in TCP for the future.

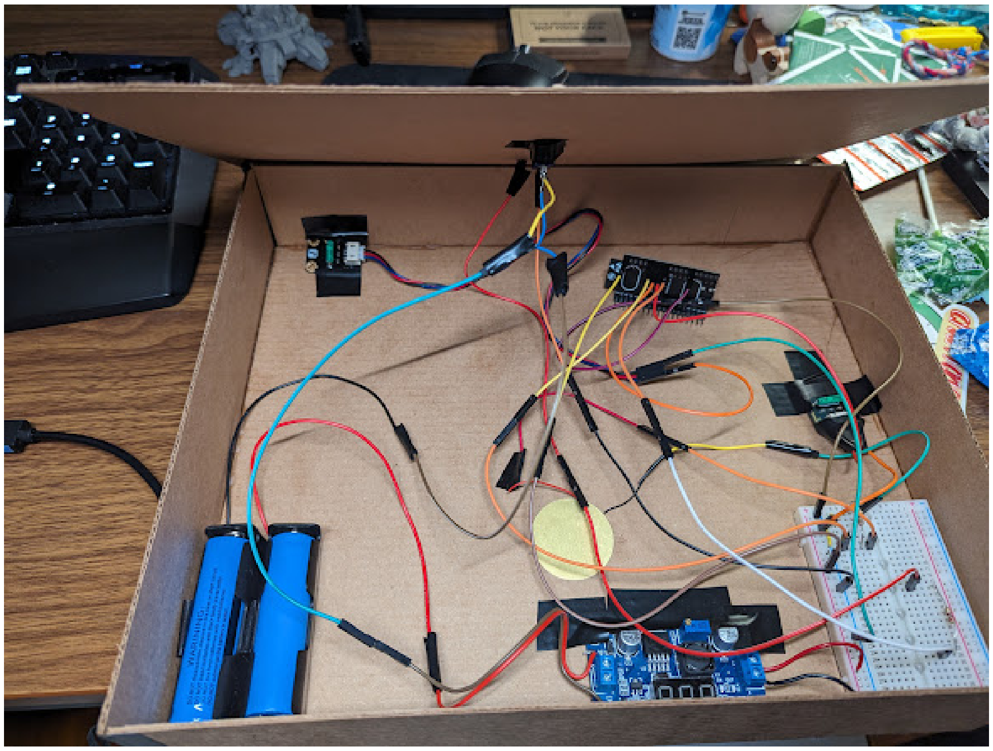
```
graph TD
  PyGame -->|Record New  -> start recording mode| ESP32
  PyGame -->|Record New -> Create amen synth in SC server| SuperCollider
  ESP32 -->|Recording Done| PyGame
  ESP32 -->|Recording Mode -> send sensor data to drum sample synth| SuperCollider
  PyGame -->|Recording Done -> save buffer to disc| SuperCollider
```

# Enclosure/Electronics

One of the principles behind this project is the theme of uncertainty and experimentation. To aid in those themes, I wanted my enclosure to be very plain and seemingly featureless. The only visible objects are an on/off switch to control power to the device, and a LED that will blink in recording mode and shine consistently otherwise.



The inside contains all the wiring and sensors needed for this project. I used 2 3.7v Li-ion batteries connected in serial to provide ~8v which are then regulated down to 3.3v with a buck converter. The switch is connected in between the batteries and the buck converter to act as an on/off switch for the entire system.



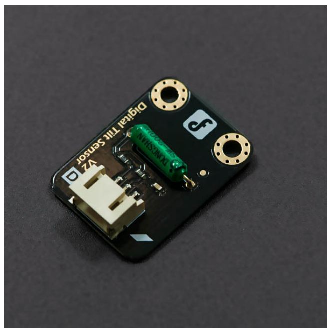
The piezo sensor is attached loosely inside the box and allowed to float around. This allows the motion from movement with the box to be easily measured.

## DIY Sensor

My DIY sensor is a composite sensor of 2 digital tilt sensors provided by DFRobot. Normally, these sensors will only return 1 if they are tilted downwards in a specific direction. By using two sensors and placing them perpendicular to each other though, I can combine the values from both sensors and detect if the box is being tilted downwards and to the right. This motion will increase the frequency of the bandpass filter on the drum loop. I originally wanted to use 3 tilt sensors, with the third one being placed in a way that would allow me to detect when the box was being pushed downwards and *to the left,* decreasing the frequency of the filter. However, the third sensor I found was broken and I could not find anymore in AKW. I therefore had to compromise with 2 and have the "neutral" state of **not** being tilted downwards and to the right as a state of decreasing the frequency of the filter. This still satisfies the criteria of a composite sensor though.

## Sensor Troubles

The tilt sensors use a 3-pin JST connector as shown below:



Only one of the sensors had a cable attached to it. I could not find any other connector, and the space is too small to fit in 3 protocables. My solution was to therefore only use 2 protocables and skip the middle (5v) line. This line isn't needed as since the tilt sensor is basically just a digital switch at heart, I can use the internal pullup resistor in the ESP32 and only connect the sensor to GND and a GPIO pin with the pinmode set to `INPUT_PULLUP`.

## Run on Boot

The SuperCollider code is written in a way that it can be called from the terminal and run on boot. A bash script was created that can run both the PyGame and SuperCollider programs. One just has to turn on the host device and flip the switch of the enclosure on.22

# Code

https://github.com/dsmaugy/cpsc334-module3/tree/main/task1