Landscape Generator： Design Document

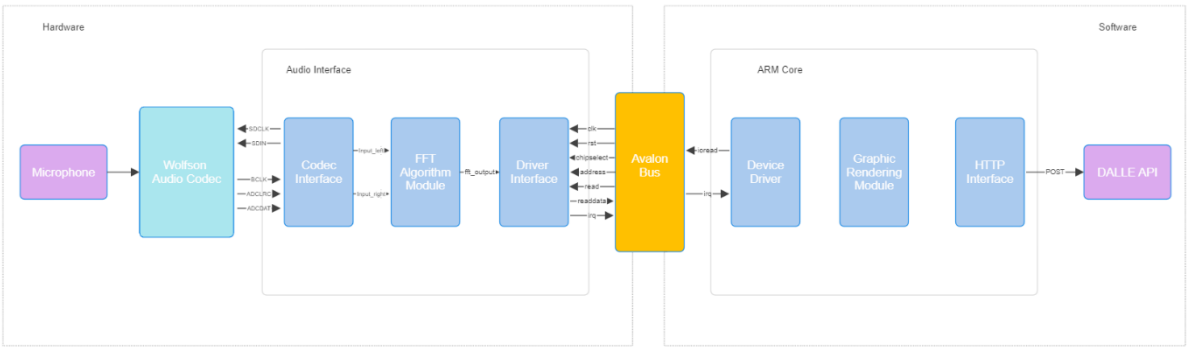
Yuxiao Qu (yq2381) Yucong Li (yl5363) Ning Xia (nx2173) Yimin Yang (yy3352)

1. System Overview

1.1 Design Goal

The Landscape Generator is a project designed to transform audio input captured through a microphone into 3D landscapes, and subsequently into enhanced 2D landscape images through a DALL-E API. The system is implemented using the DE1-SoC development board. The process involves several stages of data transformation from audio signals to 3D landscapes, and finally to rendered 2D images.

1.2 Design Block Diagram



2. Hardware Architecture

2.1 Components

- Microphone: Captures audio input.
- Wolfson Audio Codec (WM8731): Converts the analog signal from the microphone into a digital format.
- Fast Fourier Transform (FFT) Module: Transforms time-domain audio data into frequency-domain data, aiding in landscape generation.

- Avalon Bus: The system interconnect that facilitates communication between modules.
- ARM Core: The main processing unit that manages the software modules and hardware-software interaction.

2.2 Interfaces

- I2C Interface (Codec Interface): Transfers digital audio data between the codec and the FFT module.
- FFT Data Interface: Passes frequency-domain data to the driver interface.
- Driver Interface: Acts as a mediator for data and control signals between the FFT module and the Avalon bus.
- Peripheral Control Registers: Memory-mapped registers used for hardware-software communication, configurable for graphics and audio control.

3. Software Architecture

3.1 Components

- Device Driver: Manages communication with the hardware components and abstracts the functionality of the FFT module.
- Graphic Rendering Module: Converts 3D landscapes into 2D images suitable for display.
- HTTP Interface: Facilitates communication with the DALL-E API to enhance the rendered landscape images.

3.2 Interfaces

- Driver APIs: A set of application programming interfaces that enable software modules to interact with hardware through the device driver.
- Graphics API: An interface for the rendering module to draw 2D images based on 3D data.

- Network API: Manages the HTTP POST requests for communicating with the DALL-E API.

4. Design Modules

4.1 Audio CODEC

A. Overview

The Wolfson Audio WM8731 is a highly integrated low-power audio codec featuring a stereo digital to analog converter (DAC) and an analog to digital converter (ADC). It's designed for portable digital audio applications and offers a wide range of features such as multiple sampling frequencies, on-chip headphone driver, programmable filters, and an I2S interface for digital audio data transfer.

B. Audio Codec Registers

Configured to set up the WM8731 codec's operation mode, sampling rate, and data formats via I2C interface.

REGISTER	B 15	B 14	B 13	B 12	B 11	B 10	B 9	B8	B7	B6	B5	B4	B3	B2	B1	B0
R0 (00h)	0	0	0	0	0	0	0	LRIN BOTH	LIN MUTE	0	0	LINVOL				
R1 (02h)	0	0	0	0	0	0	1	RLIN BOTH	RIN MUTE	0	0	RINVOL				
R2 (04h)	0	0	0	0	0	1	0	LRHP BOTH	LZCEN	LHPVOL						
R3 (06h)	0	0	0	0	0	1	1	RLHP BOTH	RZCEN	RHPVOL						
R4 (08h)	0	0	0	0	1	0	0	0	SIDEATT		SIDETONE	DAC SEL	BY PASS	INSEL	MUTE MIC	MIC BOOST
R5 (0Ah)	0	0	0	0	1	0	1	0	0	0	0	HPOR	DAC MU	DEEMPH		ADC HPD
R6 (0Ch)	0	0	0	0	1	1	0	0	PWR OFF	CLK OUTPD	OSCPD	OUTPD	DACPD	ADCPD	MICPD	LINEINPD
R7 (0Eh)	0	0	0	0	1	1	1	0	BCLK INV	MS	LR SWAP	LRP	IWL		FORMAT	
R8 (10h)	0	0	0	1	0	0	0	0	CLKO DIV2	CLKI DIV2	SR				BOSR	USB/ NORM
R9 (12h)	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	ACTIVE
R15(1Eh)	0	0	0	1	1	1	1	RESET								
	ADDRESS							DATA								

Figure 94-I2C programmable registers

(Register Map Description References: http://www-ug.eecg.utoronto.ca/desl/manuals/Wolfson_audio.pdf Table 30 Page 47)

4.2 FFT Implementation

A. Introduction

In the project, the audio data is input through a microphone. The data is fed into FFT in hardware to be processed. Based on the processed data, a colored spectrogram representation plot is generated as the foundation for the 3D landscape generation in Python.

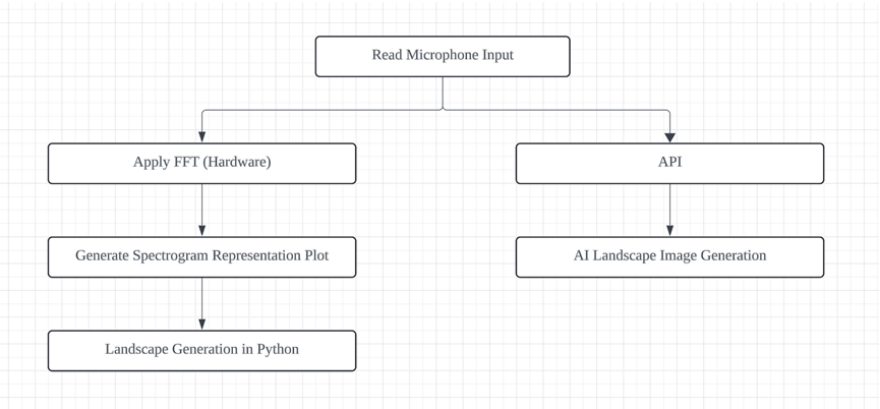


Figure: Work flow for processing microphone input data with FFT

Compared with implementing FFT in software, FFT in hardware can have a better performance because it can execute tasks in parallel, increasing the computing efficiency. However, the algorithms for FFT in hardware is more complex. Generally, Cooley-Tukey Radix-2 FFT is the most widely used algorithm for FFT in hardware implementation.

The DFT is a linear transformation from a given input vector x (i.e., a sampling sequence of the signal) to an output vector X (the spectrum, whose elements are all complex), which is given by the Equation:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j(2\pi \frac{nk}{N})}$$

N is the size of the vectors x and X, which can be denoted as w, is called by the “roots-of-unity” or twiddle factors. But in practical use, if the number of samples is very large, it is hard to calculate the equation above. So one of the methods to solve the scaling problem is to split the summation into the addition of two subparts. For example, with the decimation in time (DIT) split, we can convert the original equation into the equation below as follows.

$$X[k] = \sum_{n=0}^{N/2-1} x[2n]\omega^{2nk} + \omega^{k} \sum_{n=0}^{N/2-1} x[2k+1]\omega^{2nk}$$

The core operation of FFT is completed by butterfly operator. Butterfly operation describes a type of calculation involves recursive two-point transforms. Multiple two-point transforms

make up the complete transform. A signal flow graph for 8 point FFT is showed in the Figure below.

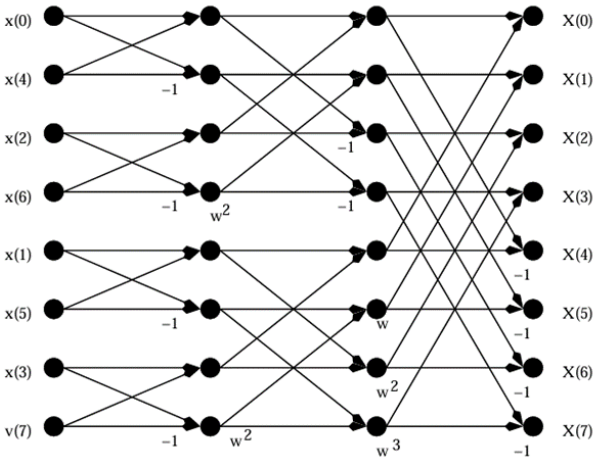


Figure: Signal flow graph for 8 point FFT

B. FFT Implementation on DE1-SoC

Compared with implementing FFT in software, hardware FFT implementation allows for parallel activity, extensive use of integer arithmetic and the limited resources in FPGA.

The figure below is an example of the diagram for an FFT in hardware with signed integer input data. The full part of the FFT requires 4 parts: an address generator, a “butterfly” operator to do the complex multiply/add, memory blocks and twiddle factor generator.

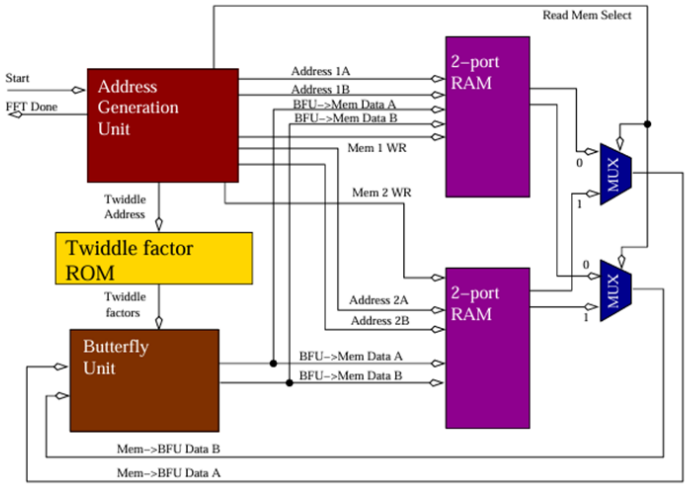


Figure: Hardware diagram for an FFT in hardware with signed integer input data

Therefore, the hardware part for FFT contains components as follow：

(1) Address Generation Unit (AGU)

Address Generation Unit (AGU) generates the addresses for reading and writing the memory contents to and from the Butterfly Processing Unit. It also generates the control signals for reading and writing data with the 2-port RAMs.

(2) Butterfly Unit (BFU)

BFU performs a special 2-point FFT based on the data transferred by the AGU. It follows the butterfly algorithms mentioned in the section above. As is shown in the figure below, A and B are the inputs from the previous level. A' and B' are the outputs of the butterfly operation.

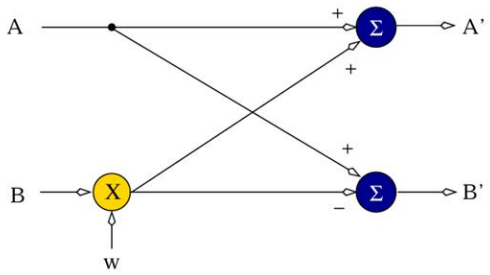


Figure: Simple description of the BFU operation

(3) Data Memeory

The input data and the output results for FFT operations should be stored in a RAM. The datapath diagram for memory is showed in the figure below.

a) *LoadDataWrite : control the writing of new data to the memory;*

b) *MemoryBankEn: enable writes to the memory block;;*

c) *Data_real_in / Data_imag_in*: Real/Imaginary part of the input data

d) *MemoryBankSelect*: select signal for controlling the specific memory block to transfer data to BFU;

e) *LoadDaraAddr / ReadDataAddr*: Addresses for data writing and reading to and from G and H;

f) *X_real / X_imag*: Real/imaginary part of the data to be written into A block;

g) *Y_real / Y_imag*: Real/imaginary l part of the data to be written into B block;

h) *G_real / G_imag / H_real / H_imag*: the real/imaginary part of the output data from Block A and B.

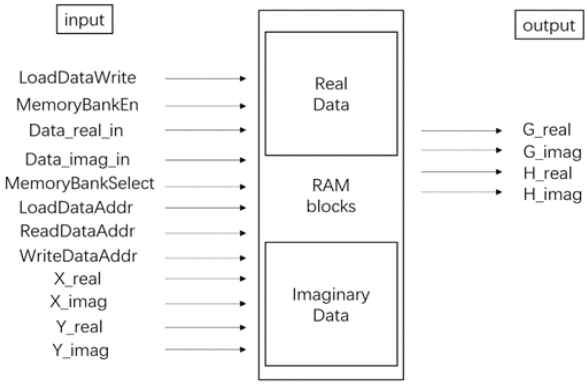


Figure: Datapath diagram for memory

(4) Twiddle Factor Generator

Due to the limited computation and memory resources on the board, the twiddle factor can be achieved through a look-up table.

(StudyMaterial: Slade, George. (2013). The Fast Fourier Transform in Hardware: A Tutorial Based on an FPGA Implementation.)

C. FFT Simulation in Python

We simulated the procedure of reading an audio file and processing the audio data with inbuilt FFT function in Python to prove the feasibility of the scheme and it can also serve as the reference result to verify the FFT implementation in the Verilog.

In the audio processing simulation in Python, we use libraries and modules for audio processing (*'librosa', 'scipy'*), audio visualization (*'matplotlib'*), and audio playback (*'IPython.display'*). First, we load the audio file and create an audio player widget using *IPython.display.Audio*, allowing us to play the audio directly in the Python environment. *librosa.load()* function returns the audio waveform information of the input signal and the sampling rate, which specifies the number of samples per second used to represent the audio.

```
import librosa
import librosa.display
import scipy as sp
import IPython.display as ipd
import matplotlib.pyplot as plt
import numpy as np

# load audio file in the player
audio_path = "audio/violin_c.wav"
ipd.Audio(audio_path)
# load audio file
signal, sr = librosa.load(audio_path)
```

Second, with the *sp.fft* function, the Fourier transform of the audio signal (ft) is computed. And then we calculate the magnitude spectrum by taking the absolute magnitude, and create a corresponding frequency axis using *np.linspace()*.

```
# load audio file
signal, sr = librosa.load(audio_path)

# plot waveform
plt.figure(figsize=(18, 8))
plt.plot(signal, alpha=0.5)
plt.title('Waveform')
plt.xlabel('Sample')
plt.ylabel('Amplitude')
plt.show()
```

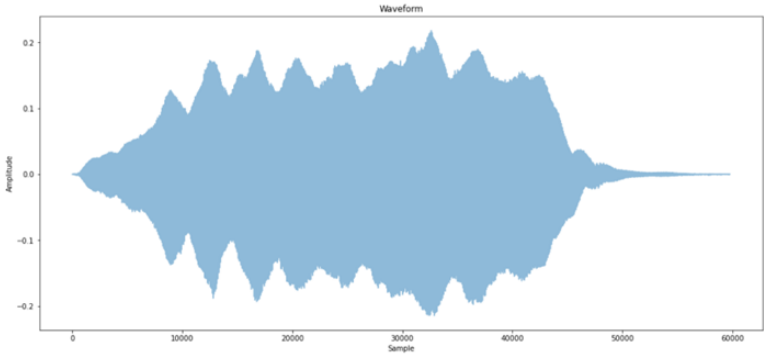


Figure: Result for read audio file in Python

Third, the magnitude spectrum of the audio signal is plotted, showing how the signal is distributed across different frequencies.

```
# derive spectrum using FT
ft = sp.fft.fft(signal)
magnitude = np.absolute(ft)
frequency = np.linspace(0, sr, len(magnitude))

# plot spectrum
plt.figure(figsize=(18, 8))
plt.plot(frequency[:5000], magnitude[:5000]) # magnitude spectrum
plt.xlabel("Frequency (Hz)")
plt.ylabel("Magnitude")
plt.show()
```

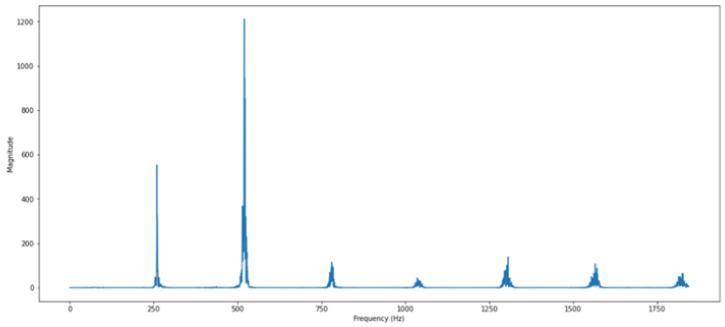


Figure : Result for spectrum plot with FFT in Python

Fourth, in order to generate the 3D landscape image in Python, a colored spectrogram should be generated according to the FFT result. In this part, the spectrogram function involves processes including windowing the input signal, computing the FFT of each window, and stacking the resulting spectra to form the spectrogram. In this example, the window duration is set to 20ms, the window overlap is set to 50%, and the max frequency is set to 8000.

```
    def spectrogram(samples, sample_rate, stride_ms = 10.0,
                                window_ms = 20.0, max_freq = None, eps = 1e-14):

          stride_size = int(0.001 * sample_rate * stride_ms)
          window_size = int(0.001 * sample_rate * window_ms)

          # Extract strided windows
          truncate_size = (len(samples) - window_size) % stride_size
          samples = samples[:len(samples) - truncate_size]
          nshape = (window_size, (len(samples) - window_size) // stride_size +
1)
          nstrides = (samples.strides[0], samples.strides[0] * stride_size)
          windows = np.lib.stride_tricks.as_strided(samples,
                                              shape = nshape, strides =
nstrides)

        assert np.all(windows[:, 1] == samples[stride_size:(stride_size +
window_size)])

          # Window weighting, squared Fast Fourier Transform (fft), scaling
          weighting = np.hanning(window_size)[:, None]

        fft = np.fft.rfft(windows * weighting, axis=0)
          fft = np.absolute(fft)
          fft = fft**2

        scale = np.sum(weighting**2) * sample_rate
          fft[1:-1, :] *= (2.0 / scale)
          fft[(0, -1), :] /= scale

        # Prepare fft frequency list
          freqs = float(sample_rate) / window_size * np.arange(fft.shape[0])

        # Compute spectrogram feature
          ind = np.where(freqs <= max_freq)[0][-1] + 1
          specgram = np.log(fft[:ind, :] + eps)
          return specgram, freqs[:ind]

    # Plot the specgram
    specgram, freqs = spectrogram(signal,sr, max_freq=8000)
    t = librosa.samples_to_time(samples, sr=sr)
    plt.figure(figsize=(10, 6))
    plt.imshow(specgram, aspect='auto', origin='lower', extent=[t.min(),
t.max(), freqs.min(), freqs.max()])
    plt.colorbar(label='Intensity (dB)')
    plt.xlabel('Time (s)')
    plt.ylabel('Frequency (Hz)')
    plt.title('Spectrogram')
    plt.show()
```

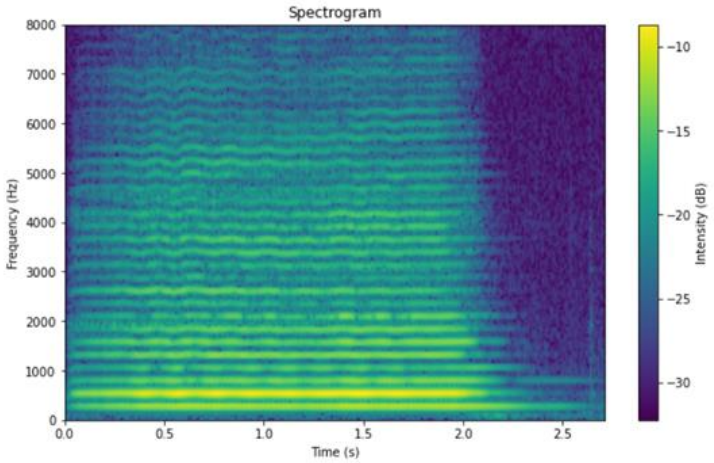


Figure : Result for colored spectrogram generation in Python

***(Reference: https://gist.github.com/kartikgill/a1a6efdac872a9e66d528b93eb049f74*; https://github.com/musikalkemist/AudioSignalProcessingForML/tree/master/14-%20Extracting%20the%20Discrete%20Fourier%20Transform)**

4.3 Generate 3D Landscape model and render 2D images

A. Design Flow

1) Coloring 3D in python for testing

→ implemented in C (Converting HSV color values to RGB)

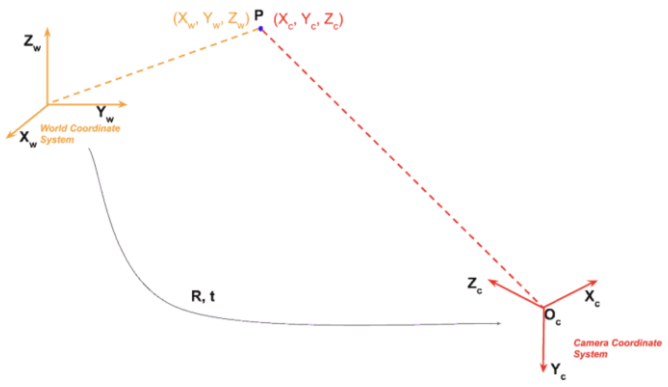
2) Render 3D object to 2D screen

→ Using Numpy: Camera coordinate system, perspective mapping matrix, triangularly displayed

(Reference:https://people.ece.cornell.edu/land/courses/ece5760/FinalProjects/s2022/jjw254_lmm343_ag988/ece5760-gh-pages/index.html)

B. Algorithms:

(1) Camera coordinate system



(Study Material: https://learnopencv.com/geometry-of-image-formation/
Study Material: https://pytorch3d.org/docs/cameras)

a) ***cam_coord()*** : a function takes camera position(VEC3), target point(VEC3), up vector(VEC3) inputs. calculate the view vector and normalize it. return the cross product.

   Insights: The view vector is defined as the unit vector pointing from the camera position to the target point. A perpendicular unit vector is generated by taking the cross product of the up vector and the view vector that was just calculated, then normalized. Finally, the coordinate system is completed by taking the cross product of the two perpendicular vectors.

b) ***cam_view_matrix()*** : a function takes camera position(VEC3) and output from cam_coord function. returns a camera view matrix.

   Insights: The camera view matrix is created by matrix multiplying a translation 4x4 matrix, which is a diagonal matrix of 1's except for the last row whose first three entries are the camera's position, with a rotation 4x4 matrix, whose entries are the vectors that comprise the camera coordinate system and a [0, 0, 0, 1] column vector at the end.

c) ***trans_matrix()*** : a function takes a perspective mapping matrix(pre-defined 4x4) and a camera view matrix which is the output from cam_view_matrix. perform matrix multiplication and return a camera view matrix.

   Insights: Next, a perspective mapping matrix is defined. Multiplying by this matrix, shown below, transforms the view volume (in our case defined with n as 4, f as 10, and h as 0.7) into a cube. The camera view matrix is a matrix multiplied with the perspective matrix to form a final transformation matrix that is applied to all of the vertices in the object.

d) ***matrix_multi()*** : a helper function takes matrix input and returns the matrix multiplication result as output matrix.

e) ***pers_div()*** (optional): a function to do perspective division to make rendering more realistic.

   Insights: divide the x, y and z components of the position vectors by the vector's homogeneous w component

(Reference:https://learnopengl.com/Getting-started/Coordinate-Systems#:~:text=Once%20all%20the%20vertices%20are,coordinates%20to%203D%20normalized%20device)

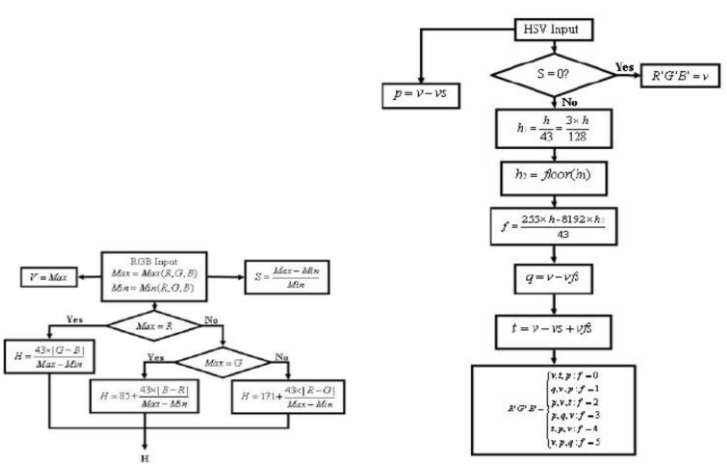
(2) VGA rendering system

a) ***get_v()*** : a helper function that generates a list of vertices corresponding to each triangular face of the landscape.

b) ***v_list()*** : a helper function that defines a 2d list of vertex indices corresponding to each face.

Insights: To display the mountain, triangular faces are used. A 2d list of vertex indices corresponding to each face is defined using a helper function. This list of faces is iterated through, and each vertex is plotted.

c) ***gsv_rgb()*** : a helper function takes HSV color input, return 16-bit RGB color output



Flow Chart for RGB to HSV and HSV to RGB conversion

d) ***draw_vga()*** : the main function to perform vga rendering.

(Study Material: http://www.brackeen.com/vga/)

4.4 API Access

The system uses an HTTP Interface to make POST requests to the DALL-E API. This requires the system to construct an HTTP request that includes the 2D image data along with any necessary authentication credentials and parameters that specify how the image should be processed.

```
from openai import OpenAI
client = OpenAI()
```

```
response = client.images.generate(
  model = "dall-e-3",
  prompt="generate a landscape based on input",
  size="1024 x 1024",
  quality="standard",
  n=1,
)
Image_url = response.data[0].url
```

5. Design Budget

Audio Memory Budge - Audio Data Input	
Time (s)	3
Fs (kHz)	8
Memory (bit)	23998 * 16 = 383968

Image Memory Budge		
	FFT Output Image	Generated Image with API
Size (bits)	Sample_rate = 8k	1024 * 1024
# of Image	1	1
Memory (bit)	577536	25165824
Total Memory	25743360	