GSOC 2022 project proposal

GARVIT JOSHI
Indian Institute of Technology(IIT-BHU), Varanasi
Varanasi, India

Google
Summer of Code

liquid galaxy

350 hr Nerfstudio visualization on Liquid Galaxy

Personal Information

This is Garvit Joshi from Varanasi, India and I am a 2nd year student pursuing my B.Tech in Mining Engineering from IIT BHU. I was introduced to the world of programming and software development in my first year. Since then, I have been very enthusiastic about deep diving into various fields of Computer Science. The areas that capture my interests are Data Structures and Algorithms, Operating Systems, App development, Web Development and computer vision. For most of my programming journey, I have worked primarily with Web-based technologies, my niche and C/C++ programs, and I have recently been diving into App Development as well.

- Github username: Garvit2003
- Linkedin: (23) Garvit Joshi | LinkedIn
- Email: garvit.joshi.min21@itbhu.ac.in
- Phone Number:(+91) 8126696009
- University: Indian Institute of Technology BHU (Varanasi)
- Time-zone: Indian Standard Time (GMT +5:30)
- Address: Varanasi, Uttar Pradesh - 221002, India

- Project Size: 350 hours (Large)

Project Description

The goal of this project is to visualize a web based viewer of Nerfstudio on the Liquid Galaxy platform.

The system will consist of two main components: **a Flutter app** and a **NerfStudio web based viewer**. The Flutter app will allow users to select a 3D model generated using NeRF and send it to the NerfStudio web based viewer. The NerfStudio module will then display the 3D model on the Liquid Galaxy display system.

The NerfStudio web based viewer will be integrated with the Liquid Galaxy platform, which is a multi-screen display system that uses Google Earth as its primary display engine. This will allow users to view and interact with the NeRF models on a large, high-resolution display that is capable of displaying content across multiple screens.

The system will also include a library of pre-made NeRF models that users can select from within the Flutter app. The library will be updated periodically with new models generated using the latest NeRF research.

Overall, the NerfStudio Visualization on Liquid Galaxy project will provide a powerful tool for visualizing and interacting with 3D models generated using NeRF. The system will allow users to explore these models in a way that was not possible before, thanks to the unique capabilities of the Liquid Galaxy display system.

Roadmap

Our target is basically to port the web-based viewer of nerfstudio to the Liquid Galaxy and build the app, the basic steps should be-

1. What is actually nerfstudio and Technology behind nerfstudio.
2. Development of flutter app
3. Setting connection between LG and android app
4. Running nerfstudio on liquid galaxy using our local machine.
5. Connecting nerfstudio running on liquid galaxy with our flutter app
6. Sending and receiving data to nerfstudio for visualization from our flutter app.

    a. Using pre trained NeRF models
        1. What are pre trained NeRF models
        2. Connecting Pre trained NeRF model to our flutter app
        3. Sending the data to Nerfstudio

    b. Using standard image(from Google Earth Engine)
        1. Receiving NeRF image data in app through Google Earth engine(GEE) API
        2. Storing the image data received through Google Earth Engine(GEE) in KML format
        3. Sending stored KML data to liquid galaxy.

Detailed Description of steps of Roadmap

1- What is actually nerfstudio and the technology behind it?

Nerfstudio is a simple API that allows for a simplified end-to-end process of creating, training, and visualizing NeRFs. The Python library Nerfstudio allows us to train our own NeRFs with rudimentary code knowledge and suitable hardware and also supports an interpretable implementation of NeRFs by modularizing each component.

NeRF stands for Neural Radiance Field.

NeRFs (Neural Radiance Fields) are a class of 3D modeling techniques that use deep learning to learn the 3D geometry and appearance of a scene from 2D images or a set of 2D viewpoints.

NeRF, like photogrammetry, is used to create realistic scenes, and it is also ‘designed to generate volumetric representations of a scene, and render high-resolution photorealistic videos of real objects and scenery.’

The processing takes the user's images, and using artificial intelligence, fills any 'gaps' with its presumptions based on the initial image input, then blends them to render a complete scene.

In order to streamline the development and deployment of NeRF research, we propose a modular PyTorch framework, Nerfstudio.This framework includes plug-and-play components for implementing NeRF-based methods, which make it easy for researchers and practitioners to incorporate NeRF into their projects. Additionally, the modular design enables support for extensive real-time visualization tools, streamlined pipelines for importing captured in-the-wild data, and tools for exporting to video, point cloud and mesh representations.

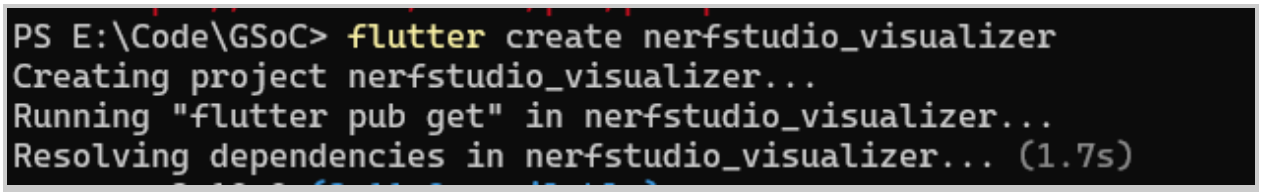
.

2-Development of flutter app

Development of the app will be done by flutter, which is an open-source UI software development kit created by Google. The basic steps are:

1. First we create an app using the following command:

Flutter create app_name



2- Then we analyze the dart code:

```
flutter analyze
```

```
PS E:\Code\GSoC> cd nerfstudio_visualizer
PS E:\Code\GSoC\nerfstudio_visualizer> flutter analyze
Analyzing nerfstudio_visualizer...
No issues found! (ran in 4.0s)
PS E:\Code\GSoC\nerfstudio_visualizer>
```

3-Then we test flutter app:

```
flutter test
```

```
PS E:\Code\GSoC\nerfstudio_visualizer> flutter test
00:10 +1: All tests passed!
```

4- Run the Dart file:

```
flutter run lib/main.dart
```

```
PS E:\Code\GSoC\nerfstudio_visualizer> flutter run lib/main.dart
Multiple devices found:
Windows (desktop) • windows • windows-x64    • Microsoft Windows [Version 10.0.22621.1413]
Chrome (web)      • chrome  • web-javascript • Google Chrome 109.0.5414.120
Edge (web)        • edge    • web-javascript • Microsoft Edge 111.0.1661.54
[1]: Windows (windows)
[2]: Chrome (chrome)
[3]: Edge (edge)
```

5- Downloading the dependencies:

Flutter pub get

```
PS E:\Code\GSoC\nerfstudio_visualizer> flutter pub get
Running "flutter pub get" in nerfstudio_visualizer...
Resolving dependencies...
  async 2.10.0 (2.11.0 available)
  characters 1.2.1 (1.3.0 available)
  collection 1.17.0 (1.17.1 available)
  js 0.6.5 (0.6.7 available)
  matcher 0.12.13 (0.12.15 available)
  material_color_utilities 0.2.0 (0.3.0 available)
  meta 1.8.0 (1.9.1 available)
  path 1.8.2 (1.8.3 available)
  test_api 0.4.16 (0.5.0 available)
Got dependencies!
```

6- Update flutter packages:

flutter pub upgrade

```
PS E:\Code\GSoC\nerfstudio_visualizer> flutter pub upgrade
Resolving dependencies...
  async 2.10.0 (2.11.0 available)
  boolean_selector 2.1.1
  characters 1.2.1 (1.3.0 available)
  clock 1.1.1
  collection 1.17.0 (1.17.1 available)
  cupertino_icons 1.0.5
  fake_async 1.3.1
  flutter 0.0.0 from sdk flutter
  flutter_lints 2.0.1
  flutter_test 0.0.0 from sdk flutter
  js 0.6.5 (0.6.7 available)
  lints 2.0.1
  matcher 0.12.13 (0.12.15 available)
  material_color_utilities 0.2.0 (0.3.0 available)
  meta 1.8.0 (1.9.1 available)
  path 1.8.2 (1.8.3 available)
  sky_engine 0.0.99 from sdk flutter
  source_span 1.9.1
  stack_trace 1.11.0
  stream_channel 2.1.1
  string_scanner 1.2.0
  term_glyph 1.2.1
  test_api 0.4.16 (0.5.0 available)
  vector_math 2.1.4
No dependencies changed.
9 packages have newer versions incompatible with dependency constraints.
Try `flutter pub outdated` for more information.
```

After this we will start to build the UI of our app and the needed features like buttons, info, etc and then connect it to our Liquid Galaxy rig by sending KML data which is explained in the next section.

After all this we will build the apk of our app.

7-Building the flutter application in the directory we want:

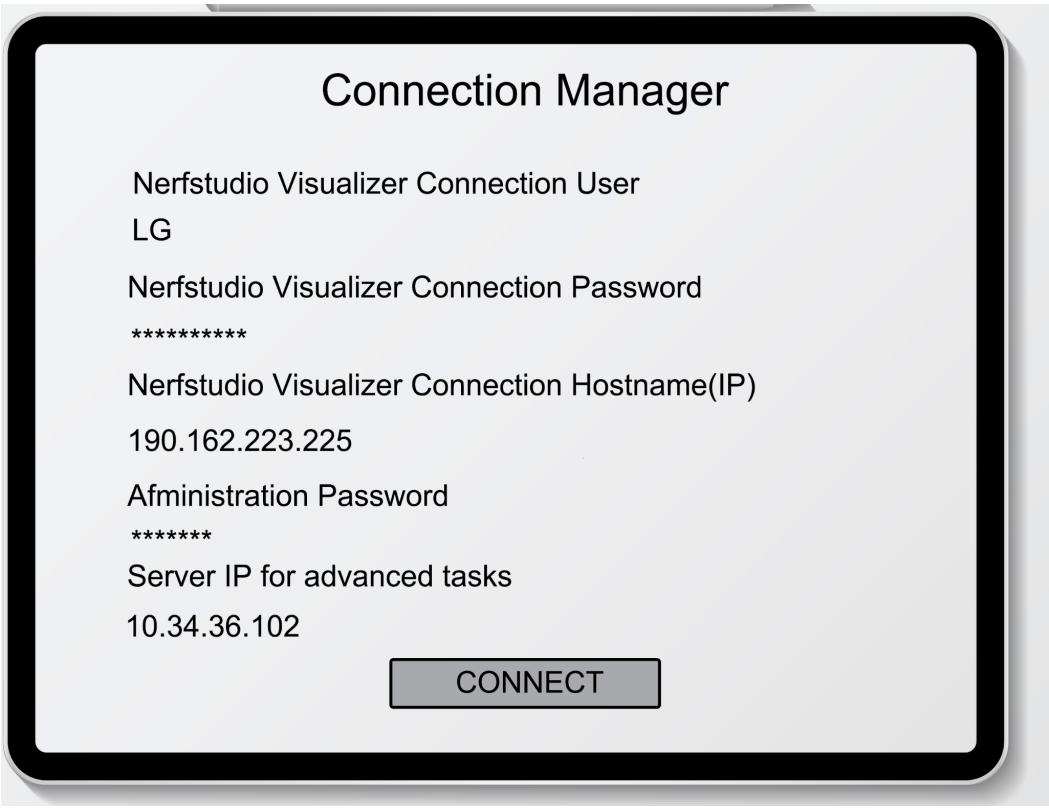
Flutter build

```
PS E:\Code\GSoC\nerfstudio_visualizer> flutter build
Build an executable app or install bundle.

Global options:
-h, --help                  Print this usage information.
-v, --verbose               Noisy logging, including all shell commands executed.
                            If used with "--help", shows hidden options. If used with
                            "flutter doctor", shows additional diagnostic information. (Use
-d, --device-id             Target device id or name (prefixes allowed).
    --version               Reports the version of this tool.
    --suppress-analytics    Suppress analytics reporting when this command runs.

Usage: flutter build <subcommand> [arguments]
-h, --help    Print this usage information.

Available subcommands:
  aar         Build a repository containing an AAR and a POM file.
  apk         Build an Android APK file from your app.
  appbundle   Build an Android App Bundle file from your app.
  bundle      Build the Flutter assets directory from your app.
  web         Build a web application bundle.
  windows     Build a Windows desktop application.

Run "flutter help" to see global options.
```

Our app has been build and now we will set a connection between our app and Liquid galaxy rig.

3-Setting up Connection between LG and flutter app:

To setup the connection between Liquid Galaxy and our application I will use the code base of “SimpleCMS” and make a fork of all connection setup code from the repository. In our application I will create a separate section for making connections(as seen in fig below).
Under this user will find the field for username,password and IP for connection. After hitting the “Connect” button the connection is set up successfully.



4- Running Nerfstudio on Liquid Galaxy using our local machine

The steps to run NerfStudio on Liquid Galaxy are:

1. We have to first install nerfstudio on our local machine by following these instructions.
2. Connecting our local machine to the Liquid Galaxy system.

```
import paramiko

# Set up SSH client
ssh = paramiko.SSHClient()
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())

# Connect to Liquid Galaxy
ssh.connect(hostname='liquid-galaxy-hostname', username='your-username', password='your-password')

# Execute command on Liquid Galaxy
stdin, stdout, stderr = ssh.exec_command('ls')

# Print results
print(stdout.read().decode())
```

In this code, the paramiko library is used to create an SSH connection to the Liquid Galaxy. The hostname parameter specifies the hostname or IP address of the Liquid Galaxy system, while the username and password parameters specify the login credentials for the remote system.

Once the SSH connection is established, you can execute commands on the Liquid Galaxy using the exec_command method of the SSH client object.

3. Once our machine is connected to the Liquid Galaxy, launch NerfStudio and create a new project or open an existing one.

4. Configure the display settings within NerfStudio to utilize the multiple screens of the Liquid Galaxy. This will typically involve selecting a custom display resolution and specifying the number and layout of screens.

```
import os

# Launch NerfStudio
os.system("/path/to/nerfstudio")

# Configure display settings
os.system("DISPLAY=:0.0 /path/to/nerfstudio --display=liquid-galaxy --screens=6 --layout=horizontal")
```

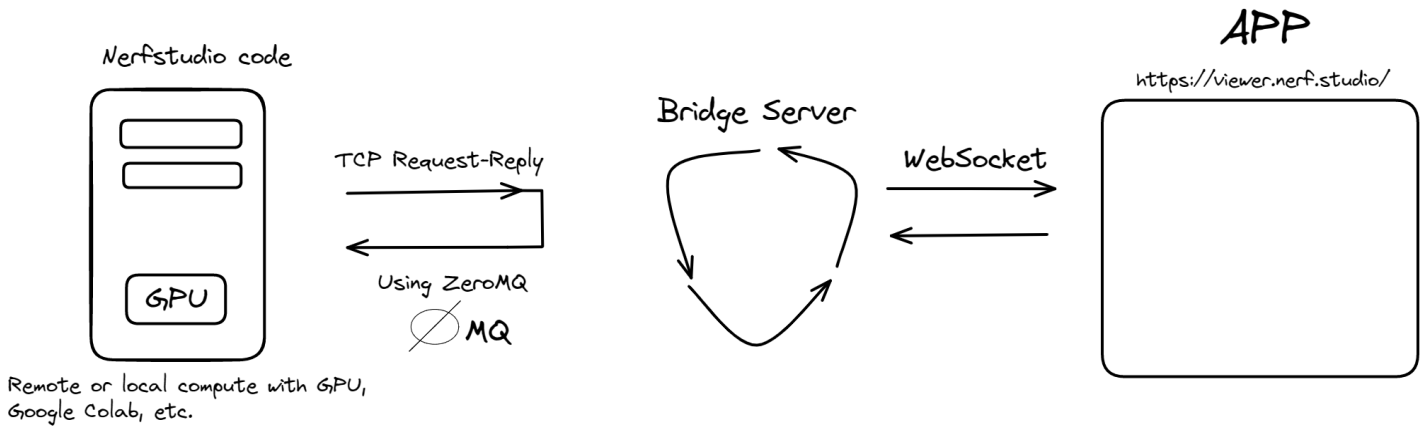
In this code, the os.system function is used to execute command-line commands that launch NerfStudio and configure the display settings. The DISPLAY=:0.0 parameter specifies the X11 display server that should be used to render the application, and the --display=liquid-galaxy --screens=6 --layout=horizontal parameters specify the Liquid Galaxy display configuration to be used. Depending on your specific setup, we may need to adjust these parameters or use different commands to achieve the desired result.

5. Once the display settings are configured, we should be able to see the NerfStudio application running on the multiple screens of the Liquid Galaxy. We can interact with the application using a mouse or other input device, and create and visualize 3D models as normal.
6. If necessary, we can also configure the networking settings within NerfStudio to enable collaboration and sharing of models with other users on the Liquid Galaxy network.Overall, the process of running NerfStudio on Liquid Galaxy involves configuring the display and networking settings within the software to take advantage of the unique features of the platform.

5- Connecting nerfstudio running on liquid galaxy with our flutter app:

❖ Nerfstudio viewer Overview-

The viewer is built using ThreeJS and packaged into a ReactJS application. This client viewer application will connect via a websocket to a server running on our Liquid Galaxy machine. The following figure helps to illustrate how our viewer framework works:



❖ System design-

Connection between nerfstudio and the Bridge Server-

In the center, we have the Bridge Server, which facilitates the connection between nerfstudio code and our flutter app. This server runs on the same machine that you are using nerfstudio and Liquid Galaxy. It has a TCP Request/Reply (REQ/REP) connection that nerfstudio can connect to with the Viewer object (left). We use ZeroMQ, an open-sourced messaging library, to implement this lightweight TCP connection. The Viewer class can send commands to the Bridge Server and receive replies. The Bridge Server will either dispatch commands to our App via a websocket or it will return information stored in the Bridge Server state.

Connection between the Bridge Server and the Flutter App-

The connection between the Bridge Server and the Client App works with WebSockets.

> **WebSocket connection** - The WebSocket is used by the Bridge Server to dispatch commands coming from the nerfstudio TCP connection. Commands can be used for drawing primitives, for setting the transform of objects, for setting various properties, and more.

❖ Installing and running locally-

1- Running the Client App

```
PS C:\Users\HP> cd nerfstudio/viewer/app
```

- Install npm (to install yarn) and yarn

```
PS C:\Users\HP> sudo apt-get install npm
```

```
PS C:\Users\HP> npm install --global yarn
```

- Install nvm and set the node version Install nvm with instructions.

```
PS C:\Users\HP> nvm install 17.8.0
```

- Now running node --version in the shell should print “v17.8.0”. Install package.json dependencies and start the client viewer app

The local web server runs on port 4000 by default, so when ns-train is running, you can connect to the viewer locally at http://localhost:4000/?websocket_url=ws://localhost:7007

2- Running the Bridge Server Manually

The viewer bridge server runs on the same machine that you use for training. The training code will connect to this server with a lightweight TCP connection using ZMQ. The training job will launch the viewer bridge server if you specify --viewer.launch-bridge-server in the terminal. Otherwise, you can launch the bridge server manually with the following script.

6- Sending data to Nerfstudio for visualization from our flutter app to get visualization-

There are two ways types of data which we can send to Nerfstudio for visualization:

6.1- Using pre trained NeRF models

In this method we send pre trained models to nerfstudio for visualization only.

6.1.1- What are pre trained NeRF model

Pre-made NeRFs, also known as pre-trained NeRF models, are pre-built neural networks that have been trained on large datasets of 3D scenes and objects to represent their radiance fields. These pre-made NeRFs can be used by researchers, developers, and artists to generate high-quality 3D images and animations with minimal effort.

Some examples of pre-made NeRFs that are available to the public include:

1. NeRF-W: This pre-trained NeRF model is available on GitHub and is trained on the DTU dataset, which contains over 60 high-resolution 3D scans of real-world scenes.
2. Nerfies: This is a pre-trained NeRF model designed for use with mobile devices and is available on GitHub
3. COLMAP-NeRF: This pre-trained NeRF model is trained on the ScanNet dataset, which contains over 2.5 million RGB-D images of indoor scenes.
4. DTU-NeRF: This pre-trained NeRF model is trained on the DTU dataset and is available on GitHub.
5. iNeRF: This is a pre-trained NeRF model that is designed to work with incomplete or noisy input data, such as partial 3D scans or point clouds.

6.1.2- Connecting Pre trained NeRF model to our flutter app

To connect a pre-trained NeRF model to a Flutter app, you can use a package like tflite_flutter to load the model into our app.

Here are some general steps we can follow to connect a pre-trained NeRF model to our Flutter app:

1. Converting our pre-trained NeRF model to a format that can be loaded by tflite_flutter. This typically involves exporting the model in TensorFlow Lite format (.tflite) or Open Neural Network Exchange format (.onnx) and optimizing the model for inference.

2. Add the tflite_flutter package to your Flutter project by adding tflite_flutter as a dependency in your pubspec.yaml file.
3. Load the pre-trained NeRF model into your Flutter app using the Interpreter.fromAsset method of the Interpreter class. This method loads a model from an asset file bundled with your app.
4. Use the Interpreter.run method of the Interpreter class to make predictions with the NeRF model. This method takes an input tensor (typically a 3D point in space) and returns an output tensor (typically a color and opacity value).

```dart
import 'package:flutter/material.dart';
import 'package:tflite_flutter/tflite_flutter.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatefulWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  _MyAppState createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
  late Interpreter _interpreter;

  @override
  void initState() {
    super.initState();

    // Load the pre-trained NeRF model from an asset
    loadModel().then((_) {
      print('NeRF model loaded');
    });
  }

  Future<void> loadModel() async {
    try {
      _interpreter = await Interpreter.fromAsset('assets/nerf_model.tflite');
      print('Interpreter created');
    } catch (e) {
      print('Error loading model: $e');
    }
  }
```

```
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'NeRF App',
      home: Scaffold(
        appBar: AppBar(
          title: const Text('NeRF App'),
        ),
        body: Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
              ElevatedButton(
                onPressed: () {
                  // Make a prediction with the NeRF model
                  makePrediction();
                },
                child: const Text('Make Prediction'),
              ),
            ],
          ),
        ),
      ),
    );
  }

  void makePrediction() {
    // Create an input tensor with 3D point data
    var input = List.filled(3, 0.0).reshape([1, 3]);

    // Run the NeRF model with the input tensor
    var output = List.filled(4, 0.0).reshape([1, 4]);
    _interpreter.run(input, output);

    // Print the output tensor to the console
    print('NeRF model prediction: $output');
  }
}
```

In this example, we load the pre-trained NeRF model from an asset file named nerf_model.tflite in the initState method using the Interpreter.fromAsset method. When the model is loaded successfully, we print a message to the console.

The build method returns a Scaffold widget with a single ElevatedButton. When the button is pressed, the makePrediction method is called. This method creates an input tensor with 3D point data and an output tensor with room for a color and opacity value. We use the Interpreter.run method to make a prediction with the NeRF model and store the result in the output tensor. Finally, we print the output tensor to the console.

6.1.3- Sending the data to Nerfstudio

We have received a output tensor in output and this needs to be send to nerfstudio running on liquid galaxy.

To send the data received from a pre-trained NeRF model to NerfStudio running on a Liquid Galaxy in KML format, you can use the KML API provided by Liquid Galaxy.

```python
import requests

# Load the pre-trained NeRF model
# ... code to load model ...

# Create an input tensor with 3D point data
# ... code to create input tensor ...

# Run the NeRF model with the input tensor
# ... code to run model and get output tensor ...

# Convert output tensor to a list of KML placemarks
placemarks = []
for i in range(len(point_data)):
    color = '#%02x%02x%02x' % tuple(output[0][0][i]*255)
    opacity = str(output[0][1][i])
    coordinates = ','.join(str(x) for x in point_data[i])
    placemarks.append(f'<Placemark><name>{i}</name><Point><coordinates>{coordinates}</coordinates>
</Point><Style><IconStyle><color>{color+opacity}</color></IconStyle></Style></Placemark>')

# Construct the KML file with the placemarks
kml = f'<kml xmlns="http://www.opengis.net/kml/2.2"><Document>{"".join(placemarks)}</Document></kml>'

# Send the KML file to the Liquid Galaxy
try:
    response = requests.post(
        'http://<galaxy_ip_address>/cgi-bin/mapserv?map=lgsvl/default.map&',
        data=kml.encode('utf-8'),
        headers={'Content-Type': 'application/vnd.google-earth.kml+xml'}
    )
    print('Liquid Galaxy response: ', response.text)
except Exception as e:
    print('Error sending data to Liquid Galaxy: ', e)
```

In this example, we use the pre-trained NeRF model to create an input tensor, run the model, and convert the output tensor to a list of KML placemarks.

We then construct a KML file with the placemarks and send it to the Liquid Galaxy using an HTTP POST request to the mapserv CGI script.

6.2- Using Standard images

This method includes sending the image data to nerfstudio and then nerfstudio will create, train and visualize nerfs on its own.

6.2.1- Receiving NeRF image data in app through Google Earth engine(GEE) API:

Google Earth Engine (GEE) is a cloud-based platform for processing and analyzing satellite imagery and other geospatial data. It includes a vast amount of Google Earth image data.

Some key features of GEE:

- Access to a vast archive of satellite imagery, including Landsat, Sentinel-2, MODIS, and more.
- Ability to process and analyze the imagery using JavaScript or Python.
- Integration with other Google products, such as Google Drive and Google Maps.

To connect Google Earth Engine (GEE) API with our app, we will need to obtain an API key and set up authentication. Here are the general steps to follow:

- First we make a Google Cloud Platform (GCP) project: To use GEE, we will need to create a GCP project and enable the Earth Engine API.
- After creating the GCP project, we will obtain an API key that will allow our app to access the Earth Engine API.

- In order to access the Earth Engine API, we have to authenticate our app. There are two types of authentication: OAuth 2.0 and service account authentication. For an app, service account authentication is usually the best option.So we will use service account authentication.
- Once we have set up authentication, we can connect to the Earth Engine API using the client libraries provided by Google. These client libraries are available for several programming languages, including JavaScript, Python, and Java. We will be using the python library.

Connecting a Flutter app to the Google Earth Engine API using the client libraries provided by Google, this involves a few steps:

1. First, we need to add the googleapis and googleapis_auth packages to our project by adding the following lines to our pubspec.yaml file:

```
dependencies:
  googleapis: ^2.7.0
  googleapis_auth: ^3.0.0
```

2. Importing the necessary packages and classes:

```
import 'package:googleapis/earthengine/v1.dart' as ee;
import 'package:googleapis_auth/auth_io.dart';
import 'package:http/http.dart' as http;
```

3. Authenticate with the Earth Engine API using our Earth Engine account credentials:

```
await ee.init();
await ee.authenticate();
```

4. Using the client libraries to access the Earth Engine API from our Flutter app:

```
var landsat8 = ee.ImageCollection('LANDSAT/LC08/C01/T1_TOA')
    .filterDate('2022-01-01', '2022-03-31')
    .filterBounds(ee.Geometry.Point(-122.2627, 37.8739))
    .select(['B4', 'B3', 'B2'])
    .sort('CLOUD_COVER');

var image = ee.Image(landsat8.first());
var thumbnailUrl = await image.getThumbUrl({'dimensions': '256x256'});
```

In this example, we first initialize and authenticate with the Earth Engine API using the ee.init() and ee.authenticate() functions. We then use the ee.ImageCollection constructor to filter the Landsat 8 image collection based on a start and end date, a bounding box, and a selection of spectral bands. We then use the ee.ImageCollection.first() method to get the first image in the filtered collection and create a thumbnail URL for the image using the ee.Image.getThumbUrl() method.

After connecting to the Earth Engine API, we can use this library in our app to retrieve and process satellite imagery and other geospatial data from Earth Engine API.

6.2.2- Storing the image data received through Google Earth Engine(GEE) in KML format

The images received from google earth engine can be stored locally. I plan to use the path_provider plugin to store our data in a KML file. We are also required to read, write or update the stored KML data in the local, which we can implement as follows:

1. Create a reference to the file location

```
Future<File> get _localFile async {
final path = await _localPath;
return File('$path/counter.kml');
}
```

2. Write data to the file

```
Future<File> writeCounter(int counter) async {
                final file = await _localFile;
                // Write the file
                return file.writeAsString('$counter');
}
```

3. Read data from the file

```
Future<int> readCounter() async {
            try {
                final file = await _localFile;
                // Read the file
                final contents = await file.readAsString();
                return int.parse(contents);
                }
            catch (e) {
                        // If encountering an error, return 0
                        return 0;
                    }
}
```

4. Save KML file in the downloads directory.

```
import 'dart:io';
// ignore: import_of_legacy_library_into_null_safe
import 'package:path_provider/path_provider.dart';
class KMLGenerator {
      static generateKML(data, filename) async {
      try {
          final downloadsDirectory =
            await DownloadsPathProvider.downloadsDirector
          var savePath = downloadsDirectory.path;
          final file = File("$savePath/$filename.kml");
          await file.writeAsString(data);
          return Future.value(file);
      }
      catch (e) {
                print(e);
                return Future.error(e);
      }
    }
```

6.2.3- Sending stored KML data to liquid galaxy

1- To send KML data from our flutter app to liquid galaxy rig we need to first set up a web socket connection between the app and the liquid galaxy.

```
import 'package:web_socket_channel/io.dart';
```

```
final channel = IOWebSocketChannel.connect('ws://liquid-galaxy-url:port');
```

2- Convert your KML data to a string format that can be sent over the WebSocket connection.

```
final kmlData = 'KMLGenerator';
```

```
final kmlString = kmlData.toString();
```

3- Send the KML data string over the WebSocket connection to the Liquid Galaxy rig.

```
channel.sink.add(kmlString);
```

Code for step 1st, 2nd and 3rd.

```dart
import 'package:web_socket_channel/io.dart';

final channel = IOWebSocketChannel.connect('ws://liquid-galaxy-url:port');

#  Convert KML data to string format
final kmlData = 'KMLGenerator';
final kmlString = kmlData.toString();

#  Send KML data string over WebSocket connection
channel.sink.add(kmlString);
```

4- In 4th step, we use a script to receive the KML data string and convert it back to a KML file format.

5- Load the KML file into the Google Earth application on the Liquid Galaxy rig.

Code for step 4th and 5th.

```
import websocket
import os

def on_message(ws, message):
    kml_file = open('kml_data.kml', 'w')
    kml_file.write(message)
    kml_file.close()

    os.system('/opt/google/earth/free/google-earth-free kml_data.kml')

websocket.enableTrace(True)
ws = websocket.WebSocketApp('ws://flutter-app-url:port', on_message=on_message)
ws.run_forever()
```

After this we can visualize the 3D version of the area in Nerfstudio running on liquid galaxy.

A breakdown of the approach can be illustrated as follows:

1. First we develop our flutter app and set up LG on our system.

2. Then we set up a connection between the flutter app and LG.

3. After this we will run Nerfstudio on LG using a local machine and connect it to our flutter app.

Till this point we have installed and connected LG, flutter app, local machine and Nerfstudio with each other. Now we only need to send data to nerfstudio.

4. To send data we have two choices

   1) Pre trained nerfs

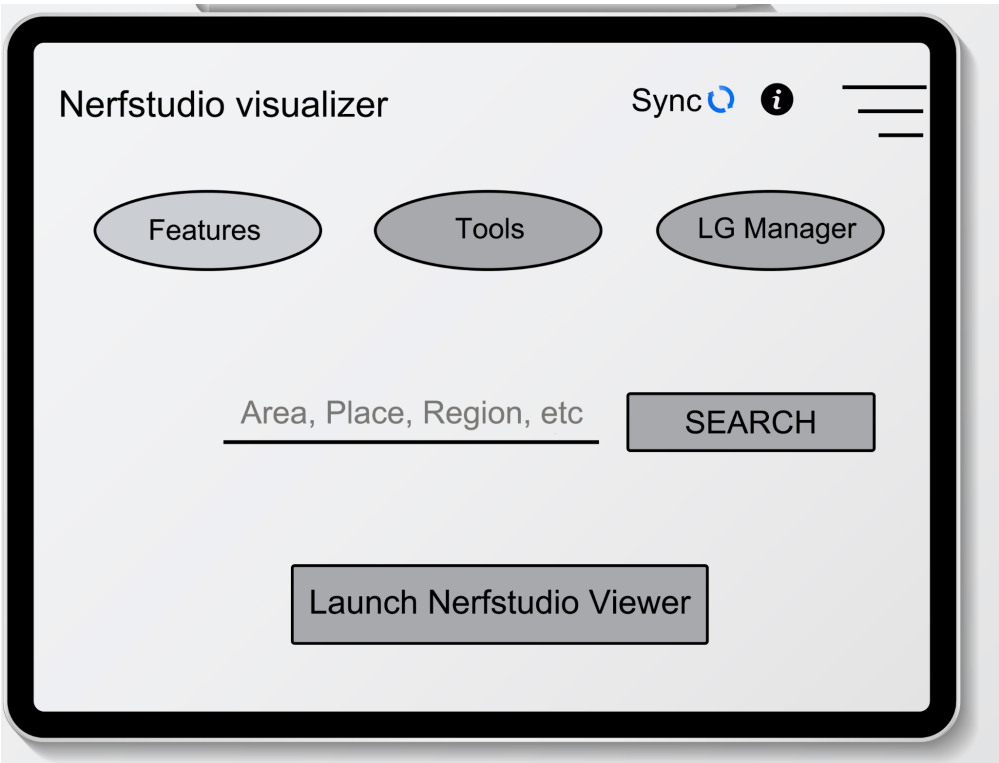
   2) standard images

   We then send data of our choice in KML format to nerfstudio for visualization.

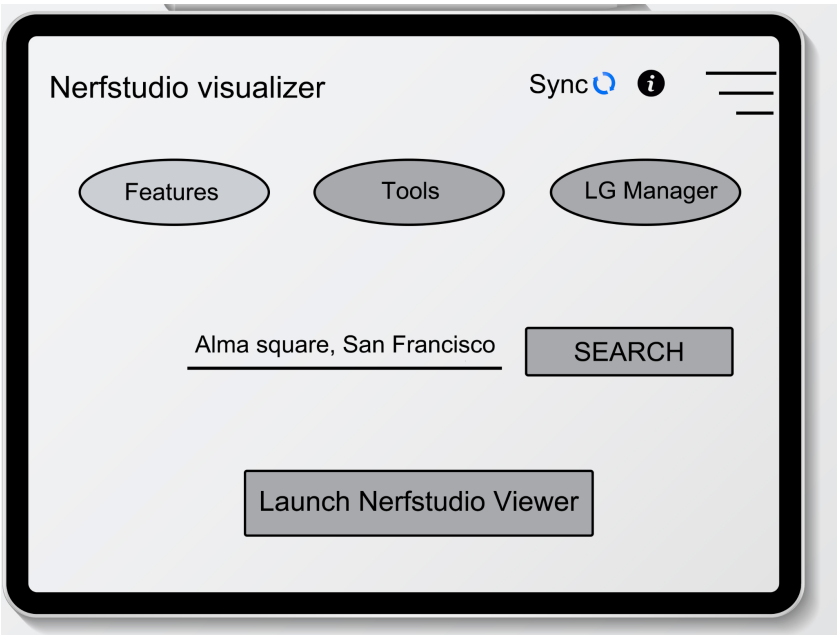
Mockup screens:

1- Mockup screens for Flutter App:

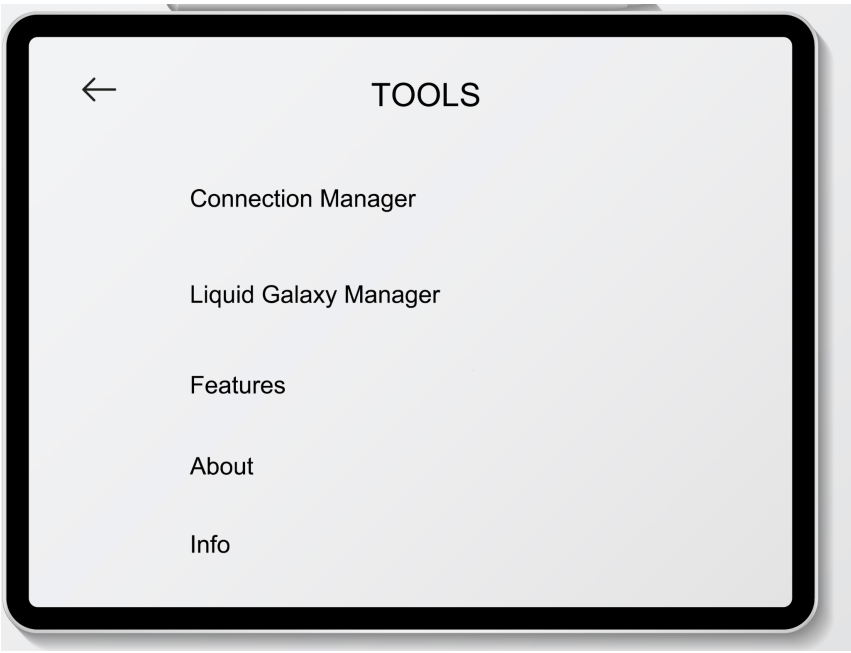
The first screen consists of all the collected data segregated that we can select and hit the Launch Nerfstudio viewer button to load them.



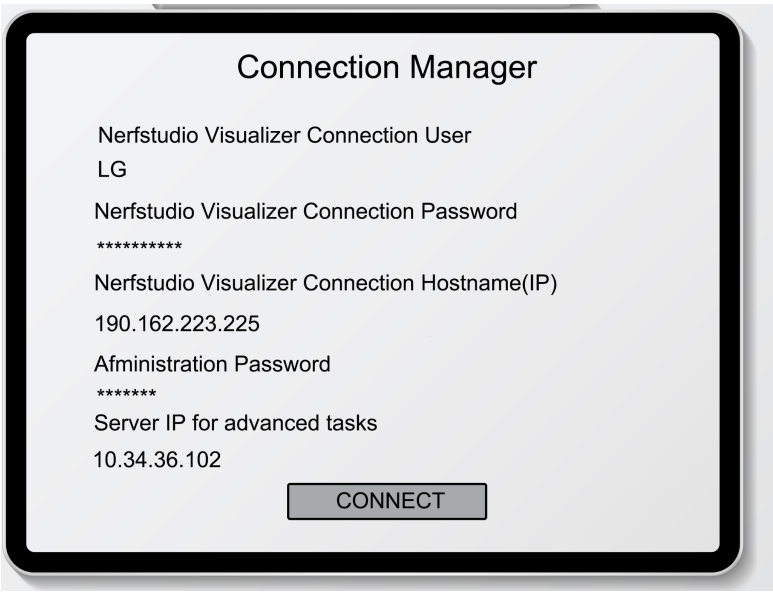
The second screen consists of searching Alma Square, San Francisco which is then visualized in LG in second mockup screen of LG rig.



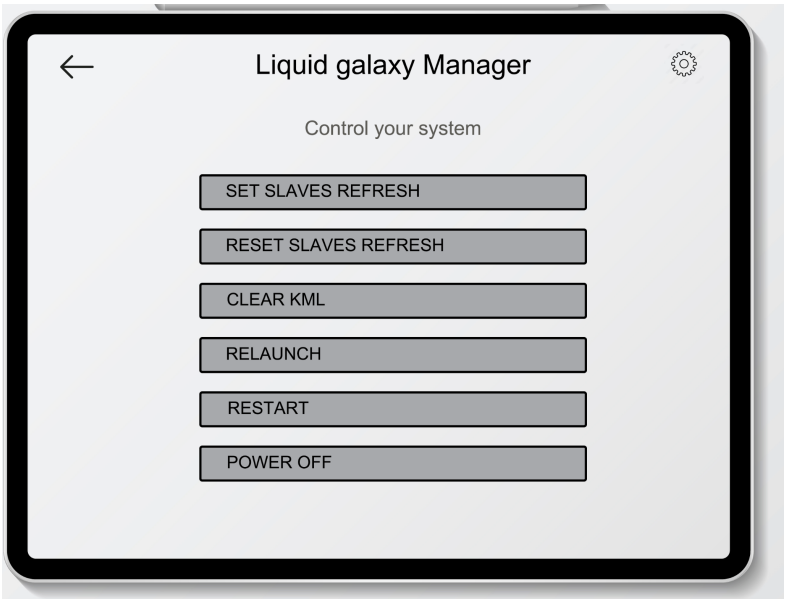
The third screen shows the various tools of Nerfstudio visualizer.



The fourth screen shows the Connection manager needed to connect to the Master machine of the Liquid Galaxy.



The fifth screen shows the Nerfstudio tasks, such as set slave refresh, reset slave refresh, clean KML, relaunch, restart, power off buttons, that we can access from the hamburger menu.



2- Mockup Screens for Liquid Galaxy

To view Liquid Galaxy mockups click here. Since our Liquid Galaxy mockup screens contain video/gif files which cannot run in PDF format so I have used google documents for this task.

Use Cases

1-Archeologists & Cartographers

Nerfstudio on Liquid Galaxy can be used to create and display 3D models of geographic features, such as mountains, buildings, or other landmarks, within a Google Earth or Google Maps environment. Undiscovered parts of the world can be mapped out from the land to the ocean. This could provide a more detailed and realistic representation of the world for users exploring these environments on a Liquid Galaxy display.

2-Engineering, Architecture & Design

Engineers and designers can use nerfstudio to form models to help them to fit parts to an existing item, but also to reverse-engineer entire structures. Visualization, site planning, and the monitoring of construction are aided by NerfStudio for architects.Nerfstudio can also be used to create and display 3D

models of proposed architectural designs or urban planning projects within a Google Earth or Google Maps environment. This could enable stakeholders and decision-makers to better visualize and understand the impact of these projects on the surrounding environment and community.

Deliverables

1- Published app on Play store.

2- Nerfstudio web based viewer visualized on liquid galaxy.

3- Full documentation for installing and using this Project on github.

Linked Technologies

1. Liquid Galaxy
2. Nerfstudio
3. Flutter
4. Dart
5. Pre trained NeRF libraries such as NeRF-W, Nerfies, COLMAP-NeRF, DTU-NeRF, iNeRF, etc.
6. Google Earth Engine(GEE)
7. Google Maps API
8. ZeroMQ
9. ThreeJS
10. ReactJS

Timeline

Previous to GSoC **(April 5-May 3)**

I will continue to contribute towards liquid galaxy and will work to refurbish the old apps of the community.

Community Bonding Period **(May 4- May 28)**

Week 1 (May 4 - May 10): I plan to thoroughly understand the open-source code available to create and send KML data from the flutter app to the Liquid Galaxy rig.

Week 2 (May 11- May 17): I will try to learn more about nerfstudio and read its documentation of ThreeJS to get more understanding of the project.

Week 3(May 18- May 25) to May 28: I plan to learn more about Google Earth Engine(GEE) API and its features.

First Working Period **(May 29 - July 9)**

Week 4 & 5(May 29-June 12): I will begin with basic UI of flutter app from scratch and connection settings.

Week 5 & 6(June 13-June 26): I will complete building the app in this period. And also setup LG on our system and connect it with our app.

Week 7 & 8(June 27-July 9): I Plan to display nerfstudio into our Liquid Galaxy rig during this period .

Second Working Period **(July 10 - August 20)**

Week 9 & 10(July 10- July 23): I plan to connect nerfstudio running in LG with our flutter app.

Week 11 & 12(July 24- August 6): In this period I will connect our app with both (1) pre-trained NeRF models and (2) Google earth engine api(standard images) , to send data to nerfstudio.

Week 13 & 14(August 7- August 20): In this period i will finish any remaining task of visualizing and connection settings in the project.

Third Working Period **(August 21- August 28)**

I will Debug any errors and will make sure that there are not any issues with the code and the project runs smoothly.

Closing and FInalizing **(August 28- September 4)**

Submit complete work including documentation of Nerfstudio installation to mentor. And Publish the app on play store.

Note of Thanks

I would like to thank Andreu Ibáñez, Víctor Sánchez and Yash Raj Bharti for actively clearing my doubts regarding the project and their guidance. I'd also like to thank the Liquid Galaxy community for fostering such a great atmosphere during their regular meetings, which are frequently the best part of the day.

Post GSoC

I will continue contributing to Liquid Galaxy even after GSoC and actively participate in code review and discussion in community channel and hopefully become a mentor for GSoC 2024. Additionally, I'm willing to work on more new projects that could benefit the community.

Conclusion

I will always be willing to help and update the project in the near future. Also I will always be ready for debugging code since Flutter is in the active development stage. So for any changes in their codebase that might reflect an error in our App, I'll be happy to modify and update them.