GEOSpectralSimulator: Multispectral terrain generator and viewer

Kevin Martín Fernández

Resum– En aquest projecte es pot veure com a partir de dades de satèl·lit es generen objectes 3D reproduïbles en motors gràfics, als quals se'ls associa informació geogràfica en fitxers JSON. Aquests fitxers permeten a un simulador desenvolupat en Unreal Engine llegir-los, carregar les textures i situar-se en aquella posició del món físic utilitzant coordenades UTM, això permetrà generar trajectòries on és prendran mostres amb les càmeres simulades. Les fonts de dades poden ser de diversos tipus destacant aquelles de tipus multispectral, aquestes seqüències d'imatges serviran per generar datasets per l'entrenament d'algorismes de deep learning.

Paraules clau– Simulador, Terreny, Satèl·lit, Multispectral, Unreal Engine

Abstract– In this project it can be seen how 3d objects are generated from satelite data, and how those objects are reproduced in a graphical engine. The 3d objects have graphical data associated on JSON files, and these files allow the simulator (developed in unreal engine) to read the data, load the textures and set the location on the real world (in format UTM). All this allows to generate datasets from cameras located on the vehicle.

Keywords– Simulator, Terrain, Satellite, Multispectral, Unreal Engine

1 Introduction

The simulation of real worlds for the generation of synthetic images is a field of high impact with the new revival of deep learning as a possible source of data. We want to represent the real world as accurately as possible, to generate synthetic information that represents real environments that can present a danger (for the natural environment, people, etc) or a high economic cost.

On this project the informations is acquired using elevation maps and aerial images of real terrains that it is used to create a simulated environment that generates new datasets, which can be used in multiple areas of computer learning. Also, this project can simulate flights, see a geographical area or expand in other environments that use geographic information. That geographical information can be provided from multiple sources (satellite data, aerial images, drone captures, virtual generated data, etc), which allows to work with a lot of external data.

2 Objectives

The goal of this project is to create a tool that allows to download geoghrapical data or transform this data to 3D data, generate data that can be reproduced in a 3D engine, adapt multispectral data to 3D models (textures), navigate the world with a simulator, control a virtual vehicle through an external script or keyboard and generate datasets that can be used in other algorithms.

• E-mail de contacte: kevinmf94@gmail.com
• Menció realitzada: Enginyeria de Computació
• Treball tutoritzat per: Felipe Lumbreras Ruíz
• Curs 2018/19

3 Methodology

In this project an Agile methodology is used, this allows us to identify more efficiently the little parts that compose the project, and to adapt them to changes needed. More specifically, a technique called Kanban [2] is used, which consists in organizing the backlog (list of tasks of short duration) on cards that will be placed on a board according to the point of the life cycle where the tasks currently are. For this purpose Trello [3] is used, a software that allows to see the boards on a web browser, create cards and move them between different lists.
To manage the project a Gantt diagram has been included. In this diagram several tasks and subtasks are taken into account in order to make a prediction of the work done and left to do and the approximate time each task will take, which will allow us to plan the project with an accurate use of time.

4 State of the art

Currently there are several solutions, for the generation of images in simulated environments with the goal of generating data used in machine learning algorithms and also a large amount of source from which to extract the data for our proposal.

In this scope one of the most important is AirSim [4] developed by Microsoft and Carla SIMULATOR [5] developed by "Centre de Visio per Computador (CVC)", and others, like LESS [6], DIRSIG [7] or Google Earth Engine [8] for more specific scopes.

4.1 AirSim

AirSim is a graphic simulator made with Unreal Engine, this simulator has the purpose of generating synthetic images on fake environments, it incorporates multiple modules that offer the following functionalities (You can see an example in the figure 2): simulation of cars, simulation of drones, compatibility with real drone controllers, recording, depth view, segmentation view, rain effects, control of illumination according to the daily hours, control of vehicles through a python script, etc.

Fig. 1: AirSim simulator

4.2 Carla SIMULATOR

Carla is a graphical simulator made on the Unreal Engine, this simulator has the finality of generating images on a fake environment with maximum realism to generate images that they can use on neural networks to drive an autonomous car safely, taking into account the unlikely cases that are difficult to generate in the real world. Its environment has the next functionalities: simulation of cars, depth view, segmentation view, simulation of traffic, simulation of pedestrians, control of the actors (traffic, pedestrian, cameras, etc) with a python script, etc.

Fig. 2: Example Carla.

4.3 LESS

LESS is a model of the radiation (You can see an example, in the figure 3) generated on a three-dimensional object/terrain for different rays, generating from a technique of ray-tracing, which is able to simulate data and images over realistic three-dimensional scenes. This model implements a method of follow weighted photons for simulating the effect of multi-spectral bidirectional reflectance.

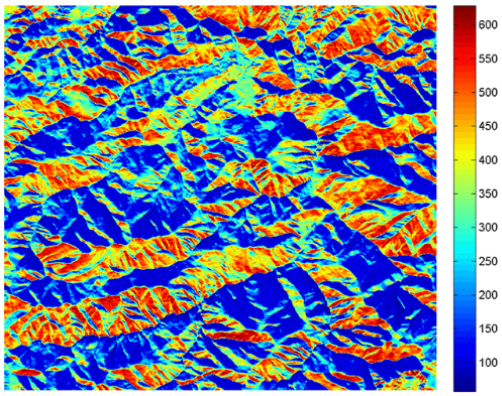


Fig. 3: Example of results of the radiation of a terrain.

4.4 DIRSIG

The model of Digital Imaging and Remote Sensing Image Generation (DIRSIG) is a model of generation of synthetic images developed by the lab of Digital Imaging and Remote Sensing of the Rochester Institute of Technology. The model can produce one band images, multispectral or hiperspectral from visible trough of the infrared thermal region of the electromagnetic spectral. You can see an example generated by DIRSIG on the figure 4.

Fig. 4: Frame from the Tacoma port.

4.5 Google Earth Engine

Google Earth Engine is a project by Google, dedicated to offer the necessary tools to be able to analyse and visualize geospacial data, intended for an academic studies, non-profit institutions, companies and governments. The principal features of the Google Earth Engine are:

- We can work with datasets from different satellites such as LANDSAT, MODIS, SENTINEL, etc.
- Incorporate a work environment to manipulate the data and wide API that allows us to combine images from different spectral, see it on a world map, export data to Google Drive and more.

5 Structure of project

In order to determine the project structure, it is study the diverse alternatives viewed in section 4. The structure of AirSim and Carla will be analyzed with the objective of de-

ciding the ownership structure and the external libraries to incorporate them into the project.

5.1 AirSim

AirSim is composed of multiple modules written in various languages, as can be seen below:

- **AirLib (C++)**: Module for Unreal Engine that provides the base classes to communicate through the RPC protocol and control the simulated vehicles.
- **DroneServer (C++)**: Server to receive orders from RPC client.
- **DroneShell (C++)**: Shell client to send orders to the Server.
- **PythonClient (Python)**: Client to send orders trough RPC, also includes code to the manipulation of images.
- **SGM (C++)**: Code to manipulate images and generate the segmentation view.
- **Unity (C# i C++)**: Unity version, includes a modules to see the AirSim information.
- **Unreal Engine (C++)**: Unreal version, includes a modules to see the AirSim information.

5.2 Carla SIMULATOR

Carla SIMULATOR is composed of multiple modules as you can see in the figure 5 written in multiple languages. The modules of Carla are:

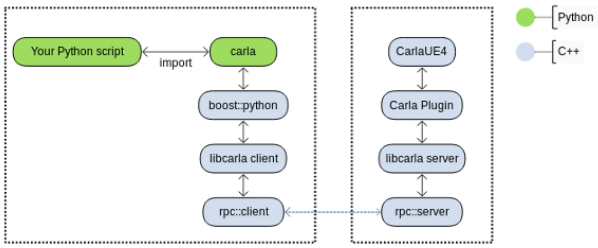


Fig. 5: Relation between the modules of Carla.

- **LibCarla (C++)**: The main library of Carla, is in charge of the logic of the simulation.
- **Unreal (C++)**: Graphic engine with the Carla plug-in, this includes all the functionalities added to Unreal.
- **PythonAPI (Python)**: The API allows sending orders to the Carla module that works like server, this API it is useful to make own scripts.

5.3 The structure chosen

In this section multiple projects with similar characteristics have been analyzed to decide the structure that can be seen in figure 6, in which some modules from other open source projects will be used. Its make this decision due to the fact that the other projects are based on the creation of terrain predefined on Unreal Editor, the opposite to the finality of this project in which its make terrain automatically provided by real data.

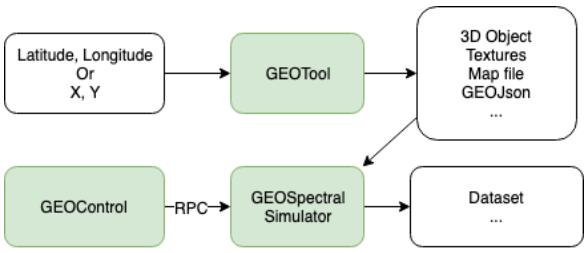


Fig. 6: Structure of GEOSpectralSimulator.

The project is consisted of these modules:

- **GEOTool (Python)**: This module has the goal of providing the needed tools for obtaining and adapting the data from some standards for geographical data webservice with the objective of can import this data on any graphic motor, it can generate diverse layers of data for the downloaded terrains as textures (RGB, infrared, etc.), geojson, etc.
- **ScriptAPI (Python)**: This module provides the interface to control the simulator trough scripts and trajectories files, allow it to control the vehicle, cameras, generate images for the dataset, etc. Also can implements extra functionality as generation of noise to the vehicle and more.
- **Unreal Engine Simulator (C++)**: Based on the Unreal Engine includes the plugins to implement an RPC Server, interface necessary to read files generated by GEOTool and the generation of images.

6 Module GEOTool

In this section it can seen how the GEOTool module works and what is the functionality implemented, more concretely can see the origin of data, the configuration file, how to prepare data for graphical engines like as Unreal, Unity, OpenGL, etc. Finally the performance of the generation from wavefront object files (.obj) will be analysed.

6.1 Obtaining data

With the goal of getting geoghrapical data it has been decided that the communication will use standards proposed by Open Geospatial Consortium [10]: Web Map Service [11] in charge of making image data available we take an orthophoto from a geographical region selected and Web Coverage Service [11] and we get data concerning the elevation of terrains in a concrete geographical region. In order to get that data HTTP requests are made to the web services offered by multiple institutions that follows the standards chosen, and we'll make tests with the Institut Cartogràfic I Geològic de Catalunya [13].

6.1.1 Configuration file

For determining which data will wish be obtained and where webservices the app accept by command line parameter a file with the configuration in JSON format. This allows determining the properties of the data they want to request as I can see on the appendix A.1. On this file can configure the next parameters:

- **Type**: It refers to the type of coordinates that we will pass, can be latlong or x-y, in the first case it will make the corresponding transformation to the UTM format (x-y).
- **Coordinates**: Coordinates on which we want to make the request in the format indicated in the type field. If "xy" is selected, the attributes x,y will be defined, and in case of choosing "latlong" we will define the lat, long attributes.
- **Dimensions**: The dimension field can choose the dimensions that you want to request in pixels, it is composed by:
  - Bbox: Bbox is the dimensions in pixels of geographical area, this is used by all requests to identify the area in a unique way.
  - Texture: Texture is a resolution in pixels request to the texture image.
- **chunks**: The fragments that want to be downloaded, the application calculates the displacement and generates n pieces of width×height.
  - width.
  - height.
- **cellsize**: Size of each pixel in meters.
- **meshStep**: The number indicates the quantity of points that wish to skip on the mesh (Default: 1). More information on the section 6.4.
- **Wcsurl**: The URL to the web service that will give us the height data.
- **Outputwcs**: Base name of the output files.
- **Formatwcs**: Format of file generated by heightmaps. Available: `raw, obj` (Object 3D).
- **Wmsrequests**: Array with each request that will be for obtaining textures, each item are composed by:
  - Url: URL to the webservice WMS.
  - Layers: The layers we want to get from these webservice.
  - Name: The name of texture, used by the generated files.
  - Format: Output format. Available: `jpg`.

6.2 Generation of terrains from a elevations maps

This section explains the different ways to generate terrain data that can be interpreted by multiple graphical engines. More specifically, this section shows the raw format and wavefront object.

6.2.1 Format RAW

The RAW format is a format based on values of 16 bits, where the value of sea level is 128. All the heights are saved in binary format and put in a file. This format is accepted by the Unreal and Unity land builder, but has dimension limitations; You must meet several specified conditions in the Unreal Editor, this causes you to lose control of the generated mesh, the texture coordinates do not match and the texture that is applied to the mesh will have to adapt. Motivation by which is decide to add the generation of 3d objects in the standard format, generating own objects as you can see in section 6.2.2.

6.2.2 Generation of mesh 3D

To import land in the graphics engine, it decides to generate a 3D mesh in `obj` format compatible with any 3D editor, graphics engine, etc. This device gives the freedom to control the distance between the vertex, where the texture is applied and which is the normal vector of each vertex to correctly apply the algorithms of illumination.

As the treatment with loops is slow, all the operations are made with the library NumPy take advantage of efficiency incorporates this library with the calculus of matrices. The problem has been adapted to operations of matricial type, you can see the code in the annex A.2.

For the generation of objects four types of objects must be defined:

- **Vertex**: Are each point in the world. Vertex are referenced in the index according to the elevation grid (1, 2, 3,..., H×W). Each point is multiplied by a K (K represents the distance between two vertex according to the distance that indicate the elevation map obtained).
- **Vertex of texture**: Vertex with two components x and y compressed between 0 and 1 that indicates the correspondence between the points of the texture and location in the mesh. This property is calculated with the equations 1 and 2.

$$u = f(column) = column/(width - 1) \quad (1)$$

$$v = f(row) = 1 - (row/(height - 1)) \quad (2)$$

- **Normal of vertex**: Vectors indicate the direction in which the light is reflected for each vertex of the object. In order to calculate these normals is necessary, calculate the normal for each face, though these are not included in the final file because the engines generate it by default.
  - Generation of normal faces: In order to generate the normals of a face once you know the relation between the faces, we follow the pattern you can see in the figure 7 where we follow the equation 1 for the calculation of the normal face $\vec{A}$, it makes cross product $\vec{C} = \vec{B} \times \vec{A}$ and finally is normalize the vector $\vec{Normal} = \frac{\vec{C}}{|\vec{C}|}$.
  - Generation of the normal for the vertex: In order to generate the vector normal for each vertex, we use the structure that can see in figure 8 applying the next equation for each vertex where V correspond to the vertices and F corresponds to the faces on the figure 8:

$$\vec{NormalV} = \vec{F_1} + \vec{F_2} + \vec{F_3} + \vec{F_4} + \vec{F_5} + \vec{F_6} \quad (3)$$

$$\vec{NormalV} = \frac{\vec{NormalV}}{|\ \vec{NormalV}\ |} \quad (4)$$

- **Faces**: In this step it is determined which is the union of vertices that will be used to generate the different faces of the mesh, in this implementation it is decided to make a triangulation, in other words, that is for each square of the own mesh two triangular faces will be generated. It is important generate the faces in correct order, writing the vertices following an opposite clockwise order, in this way the graphic engine determines that the normal face will point up by displaying the 3D mesh correctly.

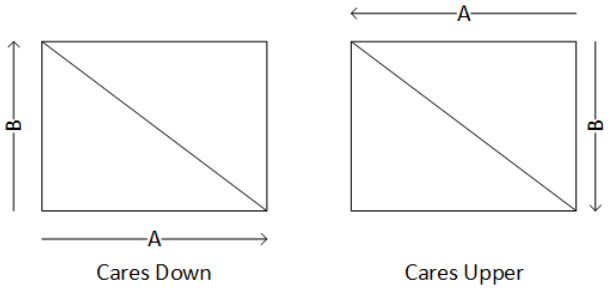


Fig. 7: Pattern for the calculation of normal on the faces.

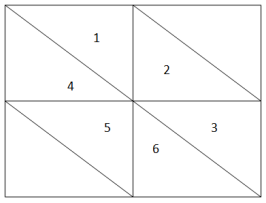


Fig. 8: Pattern for calculation of normal at a vertex.

Once the generation process has been completed, the application will generate a mesh that can be opened in any editor. As it can be seen in the figure 9 where it is open using the MeshLabs software.

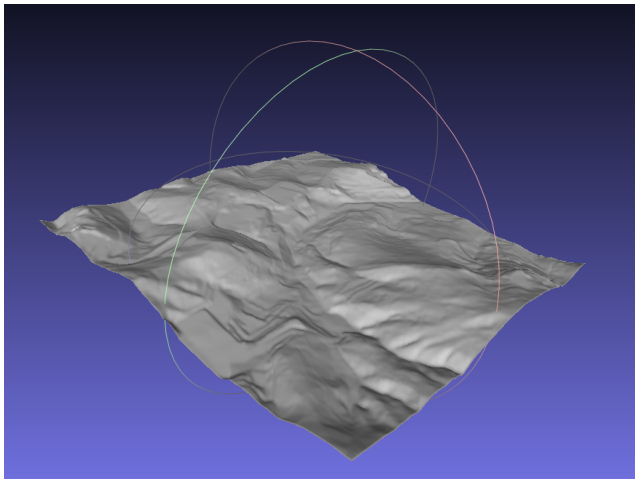


Fig. 9: Terrain mesh visualized in software MeshLab.

6.2.3 GEOJson associated at the terrain

In order to localize the terrain generated in other software that's working with geospatial data or in within the Unreal simulator itself, a GEOJson file it is included. This file indicates which belongs to the files generated. An example of this can be seen in the appendix A.3.

6.3 Time of generation

In this section the two versions implemented are analyzed to see the difference in generation time. This is an important point for future implementations in which the goal might be to implement a viewer in real time, where new data loads as the user moves on the world.
On the figure 10 the differences in time between the version using for loops and the version using the NumPy library can be seen. The implementation using the library NumPy is 24 times faster for a size of 500×500, this is due to the fact that NumPy is optimized for parallel processing of vectorial data.

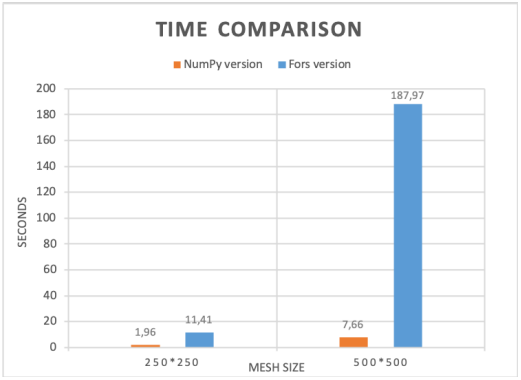


Fig. 10: Graph with the time of generation of a mesh according to size.

6.4 Quality

This section shows the results of reducing the quality of the terrain, both qualitatively (visual difference) and quantitatively (size in bytes). In order to make this reduction skips of size "meshStep" are made, set in the JSON configuration file. Test with powers of 2 (2, 4, 8, 16,...) are made on the matrix of points itself, using the terrain of size 300×300 pixels.

As it can be seen in the figure 11 when reducing the amount of points, the size on the disk is exponentially reduced, making the file load faster on the Unreal environment. As it can be seen in the figure 12 the loss of quality is visible, taking as a reference the mountain in the background it can be seen the loss of definition at the top of the mountain. It can be considered that visually, when "meshStep" is configured as 1 or 2, the loss of quality is not noticeable. At 4, it can be noticed slightly, and finally at levels 8 and 16 a noticeable loss in quality is noticed. This can be seen in the peak of the mountain, which has been simplified, as opposed to when it is configured with a value of 1, where it shows the mountain with more definition.

6.5 Map file

A file which will be read later on the simulator is generated. The map's file is divided in a set of smaller parts, down-

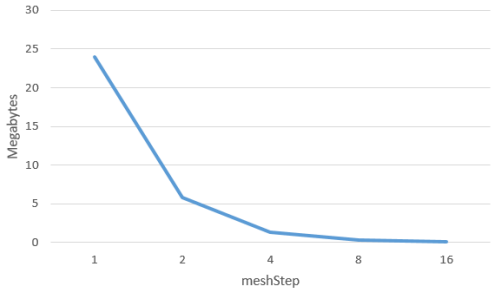


Fig. 11: Graphic with the size occupied on hard disk varying the "meshStep" parameter for a terrain of size 300×300.

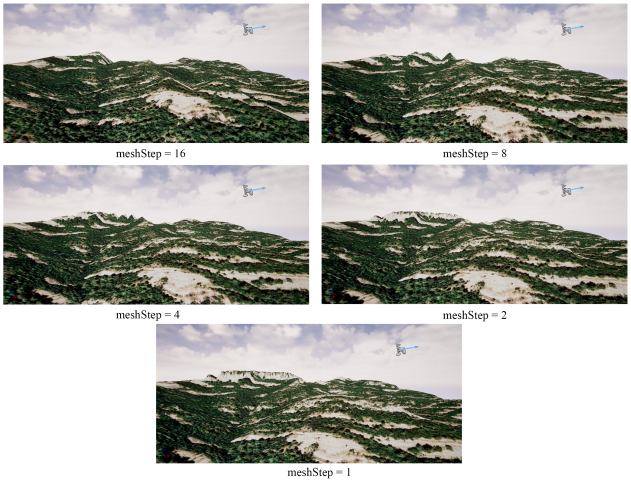


Fig. 12: Visual comparison of the effects produced by the level quality.

loaded in a request done through GEOTool. These parts contain basic information: location in the real world, size, size of each pixel, textures we want to load, etc. The file can be modified manually to modify locations, add textures that have not been acquired through GEOTool (it's necessary that the portions of terrain have a texture configured with the same name, otherwise it won't load), etc. An example can be seen in the appendix A.4 .

7 Module: Simulator

In this section the multiple parts of which the module of simulation it is composed are explained. The objective of this module is to make an available a tool that allows to simulate a travel with a generic vehicle above a terrain, getting images from multiple cameras on board of the vehicle. A way to make this is providing the interface control of the vehicle with an external script sending commands with the RPC Protocol, in this case, it is implemented on the server as it can be seen in section 7.1.1 and controllled with a client implemented in a module as it can be seen in the section 8.

7.1 Visualization and control of the vehicle

The goal of this part is to provide the interface for moving a generic vehicle in the simulated world, this vehicle is configured with several cameras that are displayed on the screen and that can generate images of what they see. Using this, the possibility of viewing the same area in multiple perspectives is added. As an example, you can see the zenith view in the figure 13.

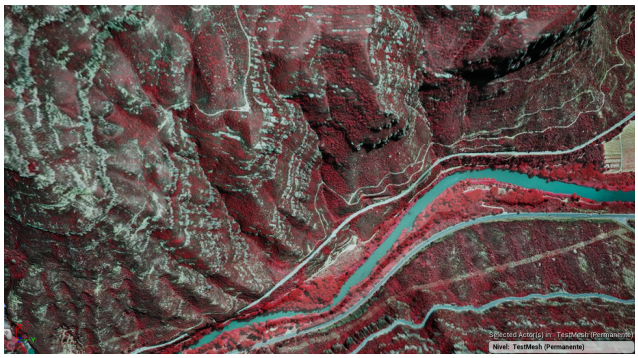


Fig. 13: Infrared view of Montserrat in zenith position.

7.1.1 Remote control

In order to control the vehicle remotely, a communication using the RPC protocol (Remote Procedure Call) it is implemented, more specifically, it is using the Rpclib [15] library for C++. This server is implemented on the Unreal module, the RPC server management is done through a class (AVehiclePawn) that can be inherited so that it basically offers the interface necessary to control a vehicle. The following actions are implemented: initialize/stop the RPC server, change the position of the vehicle, change rotation of the vehicle or cameras (Implemented with LookAt), ask for images from one of the cameras to be generated, etc.
The assignment of the functionalities is done in the function `void BindFunctions(rpc::server* server)` which receives the instance of the server, this function can be overridden to later be added to legacy classes with more functionality as it can be seen in the annex A.5. By default the AVehiclePawn class allows us to call the following RPC functions:

- `setLocation(double x, double y, double z)`: Send the location on which one place the vehicle.
- `setLookAt(double x, double y, double z)`: Send a vector with the position in the world you want to look at, so that at which point the simulator should keep the camera's eyes.
- `setLocationAndLookAt(double x, double y, double z, double lx, double ly, double lz)`: The function that implements the two previous calls in a single call.
- `setCameraLookAt(int cameraId, double x, double y, double z)`: A function that choose a camera and configure in what direction the camera look at.
- `getImage(int idCamera, std::string path, std::string channel)`: Generates images using the following information: the camera that will generate the image, the place where to save the image and the texture from which you want to obtain the image. All this is done synchronously, to

guarantee that the image is generated in the correct position.

7.1.2 Sky visualization

In order to implement the sky, the native module of Unreal is used, where a sphere is generated on which the sky is rendered. This sky has multiple parameters for configuring the sky in multiple ways, some of the things that can be done are: determining the sun position, the lighting of stars, the quantity of clouds, the velocity of the clouds, etc.

This allows to generate synthetic images in different moments of the day with a different lighting, to generate more complex datasets. On the other hand, it is applied without shadows because the shadows are generated by Unreal, if these shadows are generated by the 2 bands, they overlap generating an undesirable strange effect. An example of differents hours of the day can be seen in figure 15.

7.2 Terrain

The terrain is loaded through code executed at runtime obtaining the obj file and this file is processed to generate a UProceduralMesh [16] that contains the mesh generated with the api of this class to generate meshes in runtime. To generate this mesh it must be considered that Unreal works with a different system of coordinates, it is inverted respective to the UTM model on the y axis. Due to this, it is necessary to make the necessary changes.

This mesh is loaded on the world and it is represented by the class “MapChunk”, to which a dynamic material that contains the loaded textures for generate the different visualizations is assigned. The texture can be any image prepared to use as a texture on Unreal, giving the possibility of seeing RGB, infrared, multispectral bands, indexes generated by the multispectral data, etc.

7.3 Results of visualization of terrains

In this section can see the results obtained from the 3D terrains generated by the module GEOTool and apply multiple visualizations from origins as textures or shaders obtained from different sources. More specifically, the RGB and infrared views are obtained from ”Institut Cartogràfic de Catalunya”, the multispectral data is obtained from the browser EO Browser [17] and the depth data is generated applying a shader to the scene.

You can see in Figure 14 how it has been applied to a terrain that corresponds to ”Sales de Pallars”, different multispectral bands obtained from the satellite Sentinel 2 [18]. The first three bands B02, B03 and B04 correspond to the bands of colours, the band B05 corresponds to the fast change of the reluctance that is produced in the vegetation in a range near of infrared, the band B08 obtains NIR [19] data and the B09 is able to detect water vapours.
With these bands composed data from multiple spectrums that can be applied to multiple scopes is obtained. In figure 16 some examples can be seen, between them the indexes are:

- NVDI [20]: Based on the combination of the bands (B08 - B04)/(B08 + B04), index used to see the farming state.
- Moisture index [21]: Based on the combination of the bands (B8A - B11)/(B8A + B11), indicates the proportion of precipitation that is needed so that satisfy the needs of vegetation.
- NDWI [22]: Based on the combination of the bands (B03 - B08)/(B03 + B08), the index is used to determine the hydrique stress of the vegetation, saturation of the humidity in the land or limit the mass of water a lakes or reservoirs.

The last view of the figure 16 the depth or proximity can be seen using algorithms of 3D reconstruction with monocular stereo, prevention of collisions, algorithm of multiview stereo, etc.

7.4 How to interact with the simulator

In order to interact with the simulator the following keys have been defined:

- A,W,S,D: Left, Front, Back, Right movement of the vehicle.
- Alt, SpaceBar: Down and up the vehicle.
- Q,E: Left and right rotation of vehicle.
- Y: Starts an RPC Server
- U: Stops an RPC Server.
- H: Show/Hide the visualization of vehicle.
- K: Save image in the simulator folder from the second camera.

8 SCRIPTING MODULE

This section contains the explanation of the main points of the scripting module with which the simulator is controlled, in a way that allos to reproduce travels, read trajectories from files, generate images, generate noise to the trajectories, etc.

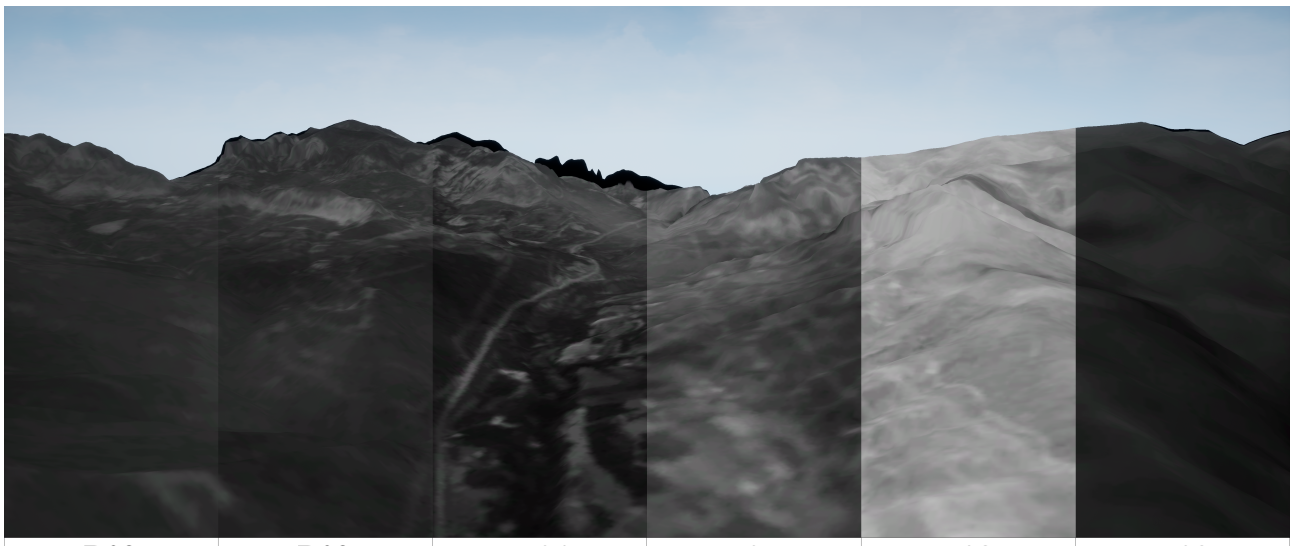
8.1 Paths

The scripting module allows to make travels, read trajectories created in a real world (adapt this to the right format) or generate synthetic this way can recreate on the simulator a trajectory done previously were able reproduce so many times, this allows to do multiple tests with different visualizations of the terrain. For this finally its generate a file that explained in the section 8.1.1.

8.1.1 File of paths

This section explains how to form the path file and the file to place the cameras. The file is in CSV format and can be read from the GEOControl module to tell the simulator the positions of the vehicle and where both the vehicle and the cameras should look, in coordinates of the world.
The file controls the vehicle is composed by the next fields:

- Time: Time in milliseconds from start of the script.



B02	B03	B04	B05	B08	B09
Blue	Green	Red	Vegetation red edge	NIR	Water vapour

Fig. 14: View of ”Sales de Pallars” in six spectral bands differents.

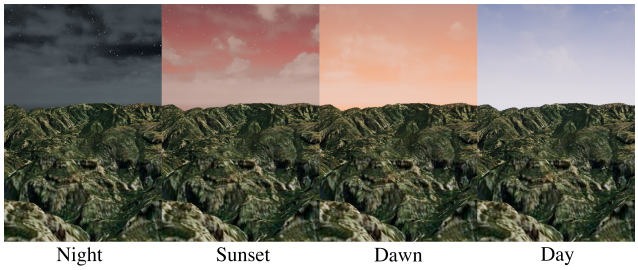


Fig. 15: Example of differents sky.

- x, y, z: Position X, Y, Z in UTM format.
- LookX, LookY, LookZ: Position X, Y, Z to which we look in UTM format.

The file controls the cameras are in another file composed of the fields:

- Time: Time in milliseconds from start of the script.
- cameraId: The Id of the camera that want to modify.
- LookX, LookY, LookZ: Position X, Y, Z to which we look in UTM format.
- GetImage: Boolean (0 or 1) that indicates if you want to generate an image from this camera in this instant of time.
- Channel: Textures of which we want to obtain images separated by |.

An example from the CSV file generated with Excel can be seen in the appendix A.6.

8.2 Noise simulation

Vehicles experience sudden movements due to wind, asphalt conditions and other factors. To simulate this, noise has been introduced in the trajectory of the tests. During the firsts tests, noise was generated through code, but that way the tests can't be reproduced multiple times, that strategy was dismissed. After that, the noise was introduced directly in the CSV files pertaining to the trajectory. This option is better because it allows to work with external files produced by real vehicles and drones, which guarantees that the tests can be reproduced multiple times.

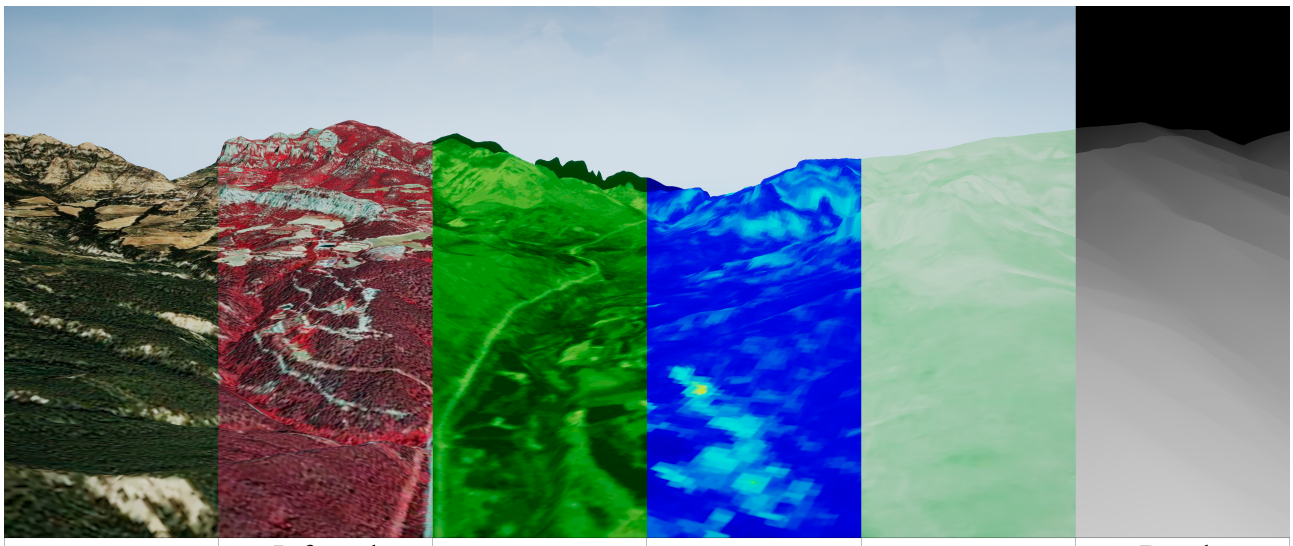
8.3 Generation of Datasets

One of the objectives of this project is giving the possibility of generating datasets of synthetic images satellite data, in order to be able generate datasets the field “GetImage” is used, as seen in section 8.1.1, which generates an image every time it finds this field.
In the figure 17 there is a little example with a few images of a tour on which the camera its fixed with a concrete point that it is left behind with the passage of time.

9 Full workflow to generate a dataset

This sections shows an example of the workflow that can be achieved with the modules generated on this project. This will show how to use them and some examples of new applications that can be developed using this project as a base.

First, the GeoTool module is used to obtain data, with the goal of obtaining the coordinates of the area which we want to obtain. For this, we can use apps like Google Maps, which is used in this case to download the zone of “Volcan de Santa Margarita”. Approximate coordinates are chosen (lat 42.130596, long 2.521956) to obtain the sorrounding area. Once chosen, the JSON that can be seen in section 6.1.1 is configured. Once the process is completed, the module is called with the following command : `./GEOTool.sh config.json` (The requirements to run this application can be seen in the annex A.7). Once the process is over, it generates a series of files, as it can be seen in figure 18.



RGB	Infrared	NDVI	Moisture index	NDWI	Depth
0,25m/px	0,25m/px	10m/px	10m/px	10m/px	5m/px
Cartografic	Cartografic	Sentinel-2	Sentinel-2	Sentinel-2	Simulator

Fig. 16: View of ”Sales de Pallars” in RGB, infrared, NDVI, moisture, NDWI and depth.

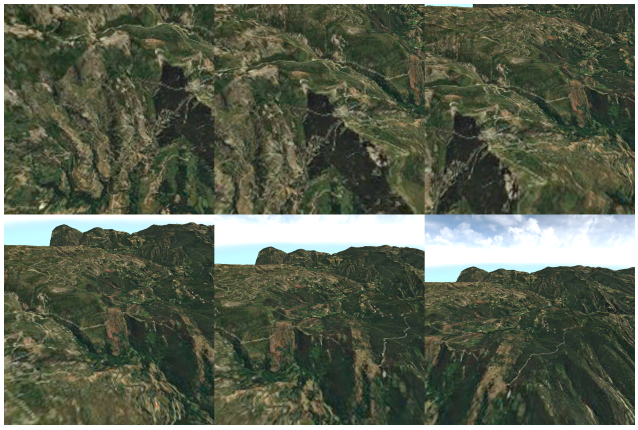


Fig. 17: Set of images generated by GEOControl.

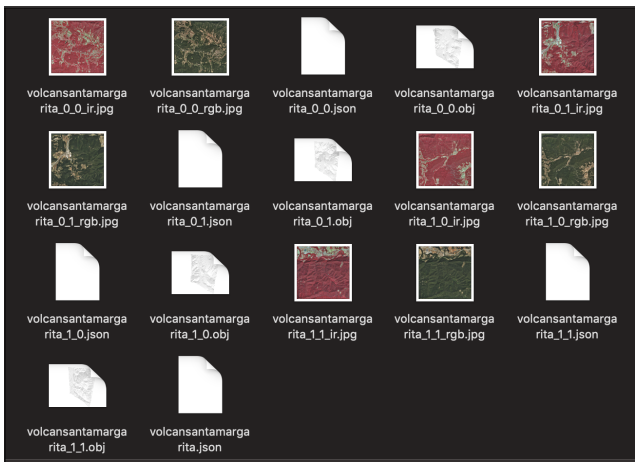


Fig. 18: Files generated by GEOTool.

Once the map is generated, the files will be added to the “Maps” folder that’s next to the executable generated by Unreal, and the simulator will be run with the following command `./GEOSimulator.sh -folderMap="Maps/volcan/" -map="volcan.json"`. Once the simulator has initialized, the interface can be seen in the upper left corner, which shows the position in the world where the vehicle is located and the state of the RCP server. An example can be seen in figure 19.
The RPC server can be initialized with the Y key, and once this is done, the module that controls the vehicle and requests the generation of images will be invoked.

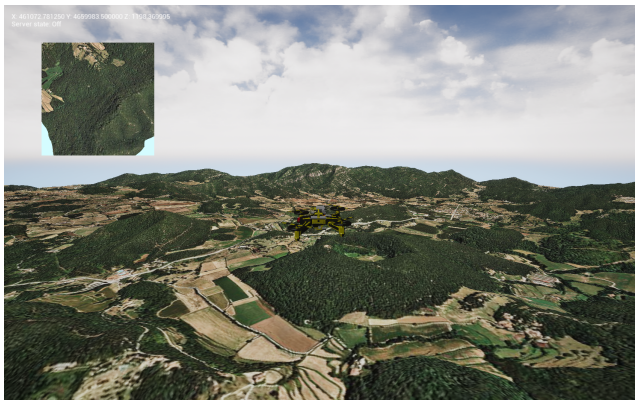


Fig. 19: Simulator view.

With the files containing the trajectories already created as indicated in section 8.1.1, the GEOControl module can be invoked with the following command `./GEOControl.sh filevehicle.csv filecamera.csv`, which executes and generates the dataset in the outputs folder. An example can be seen in figure 17.

10 Conclusions

This project shows how to obtain textures and elevation maps originated from satelites. This information can be

manipulated to generate indexes, new information generated through neural networks, etc. The treated data is transformed into 3D information that can be interpreted by graphical engines line Unreal Engine. Aspects like the time needed to generate the 3D model are used to see the effect of libraries like NumPy during the manipulation of matrixes, accelerating the process thanks to parallelism. The effect can be seen both in the faster times and in the quality of the meshes with the reduction of the number of points used to generate the 3D model used to render models of big sizes.
The section pertaining to the simulation module shows the information in a 3D enviroment generated with Unreal Engine. We can add cameras to a simulated vehicle and see different perspectives of the same information. To do this, another module has been created, which sends commands to a RPC server implemented in Unreal Engine, in which we can move, choose where the vehicle looks in the real world, where cameras point to and obtain the images captured by the cameras. The tests, which were performed with data from Satelites like the Sentinel 2, show how this information is represented in the simulated enviroment, which use has each type of data and what possibilities the simulator offers when seeing this information.
The section pertainten to the scripting shows how to generate trajectories that can be reproduced multiple times in the simulated enviroment, which allows to make several journeys with different data, or the same journey observing this data from multiple perspectives, extracting this information in the form of datasets. These trajectories can control multiple parameters of the vehicle and in which point and from which cameras the images are obtained. In the real world, the vehicle experiences unexpected movements due to several factors, like wind and the conditon of the asphalt. To simulate this, a filter that generates noise has been added, to induce random movements to the vehicle and the camera.

ACKNOWLEDGMENTS

First of all, I want to thank my tutor Felip Lumbreras for the time he invested in this project, the help provided when consulting experts and all the ideas provided during meetings. Secondly, I would like to thank Marc Garcia for guiding me in the use of Unreal Engine and helping me complete this project successfully. Thirdly, I would like to thank my friend Javier García Cantero, for helping me by revising the English version of this paper. And the last, i would like to thanks the project BOSSS TIN2017-89723-P.

REFERENCES

[1] Agile software development https://en.wikipedia.org/wiki/Agile_software_development [19/02/2019]

[2] Kanban https://www.iebschool.com/blog/metodologia-kanban-agile-scrum/ [19/02/2019]

[3] Trello - https://trello.com/ [19/02/2019]

[4] AirSim - https://github.com/Microsoft/AirSim [19/02/2019]

[5] Carla SIMULATOR - http://carla.org [19/02/2019]

[6] LESS - http://lessrt.org/ [11/04/2019]

[7] Digital Imaging and Remote Sensing Image Generation - http://dirsig.org/ [11/04/2019]

[8] Google Earth Engine - https://earthengine.google.com/ [20/05/2019]

[9] Unreal Engine - https://www.unrealengine.com/en-US/what-is-unreal-engine-4 [09/03/2019]

[10] Open Geospatial Consortium (OGC) - http://www.opengeospatial.org/ [08/04/2019]

[11] Web Map Service (WMS) - https://www.opengeospatial.org/standards/wms [08/04/2019]

[12] Web Coverage Service (WCS) - https://www.opengeospatial.org/standards/wcs [08/04/2019]

[13] Institut Cartogràfic i Geològic de Catalunya (ICGC) - http://www.icgc.cat/ca/ [08/04/2019]

[14] Wavefront .obj file - https://en.wikipedia.org/wiki/Wavefront_.obj_file [09/04/2019]

[15] RPC Lib - Modern mgspack-rpc for C++ - http://rpclib.net/ [10/04/2019]

[16] Procedural Mesh Component - https://wiki.unrealengine.com/Procedural_Mesh_Component_in_C%2B%2B:Getting_Started [21/05/2019]

[17] EO Browser - https://apps.sentinel-hub.com/eo-browser/ [25/05/2019]

[18] Sentinel 2 - https://www.esa.int/esl/ESA_in_your_country/Spain/SENTINEL_2 [25/05/2019]

[19] Tecnología NIR, sus Usos y Aplicaciones - https://www.engormix.com/balanceados/articulos/tecnologia-nir-sus-usos-t32534.htm [25/05/2019]

[20] El NDVI o Índice de vegetación de diferencia normalizada - https://geoinnova.org/blog-territorio/ndvi-indice-vegetacion/ [25/05/2019]

[21] Moisture index - http://glossary.ametsoc.org/wiki/Moisture_index [25/05/2019]

[22] Cálculo del índice NDWI - http://www.gisandbeers.com/calculo-del-indice-ndwi-diferencial-de-agua-normalizado/ [25/05/2019]

APPENDIX

A.1 JSON example for configure GEOTool

```
{
  "type": "xy",
  "coordinates" : {
    "x": 326737.23,
  "y": 4676971.49,
  "zone": "31T"
  },
  "dimensions": {
    "bbox":{
      "height": 1500,
      "width": 1500
    },
    "texture":{
      "height": 1500,
      "width": 1500
    }
  },
  "chunks": {
    "width": 3,
    "height": 3
  },
  "cellsize": 15,
  "meshStep": 4,
  "wcsUrl": "http://geoserveis.icc.cat/
     icc_mdt/wcs/service",
  "outputWcs": "example9x9big/pallars15
     ",
  "formatWcs": "obj",
  "wmsRequests": [
    {
      "url": "http://geoserveis.icc.cat
         /icc_ortohistorica/wms/service
         ",
      "layers": "orto25c2016",
          "name": "rgb",
      "format": "jpg"
    },
    {
      "url": "http://geoserveis.icc.cat
         /icc_ortohistorica/wms/service
         ",
      "layers": "ortoi25c2016",
          "name": "ir",
      "format": "jpg"
    }
  ]
}
```

A.2 Code for generate a 3D mesh from a heights file

```
def generate_obj(values, k=5):

    h, w = values.shape

    ij = np.meshgrid(np.arange(h), np.
        arange(w), indexing='ij')
    i = ij[0].reshape(h*w)
    j = ij[1].reshape(h*w)
    values = values.reshape(h*w)

    # Generate list of vertex
    vertex = np.zeros((h*w, 3))
    vertex[:, 0] = i*k
    vertex[:, 1] = j*k
    vertex[:, 2] = values

    # Generate faces
    mask = np.logical_and(i < (h-1), j
        < (w-1))
    uFaces = np.zeros((h*w, 3), dtype=
        np.int32)
    indexVertex = np.arange(h*w)+1

    uFaces[mask, 0] = indexVertex[mask]
    uFaces[mask, 1] = indexVertex[mask]
         + w
    uFaces[mask, 2] = indexVertex[mask]
         + w + 1
    uFaces = uFaces[mask]

    dFaces = np.zeros((h * w, 3), dtype
        =np.int32)
    dFaces[mask, 0] = indexVertex[mask]
         + w + 1
    dFaces[mask, 1] = indexVertex[mask]
         + 1
    dFaces[mask, 2] = indexVertex[mask]
    dFaces = dFaces[mask]

    faces = np.concatenate((uFaces,
        dFaces), 0)

    # Generate UV Map
    uvMap = np.zeros((h*w, 2))
    uvMap[:, 0] = j / (w - 1)
    uvMap[:, 1] = 1 - (i / (h - 1))

    # Generate normal faces upperFaces
    #mask = np.logical_and(i > 0, j < h
        -1)
    aUpper = (np.roll(vertex, h, axis
        =0) - vertex)
    bUpper = (np.roll(vertex, -1, axis
        =0) - vertex)

    normalUpperFaces = np.cross(bUpper,
         aUpper)
    module = np.linalg.norm(
        normalUpperFaces, axis=1)
    normalUpperFaces[:, 0] =
        normalUpperFaces[:, 0] / module
    normalUpperFaces[:, 1] =
        normalUpperFaces[:, 1] / module
    normalUpperFaces[:, 2] =
        normalUpperFaces[:, 2] / module

    # Generate normal faces downFaces
    #mask = np.logical_and(i < w-1, j >
         0)
    aDown = (np.roll(vertex, -h, axis
```

```
    =0) - vertex)
bDown = (np.roll(vertex, 1, axis=0)
    - vertex)

normalDownFaces = np.cross(bDown,
    aDown)
module = np.linalg.norm(
    normalDownFaces, axis=1)
normalDownFaces[:, 0] =
    normalDownFaces[:, 0] / module
normalDownFaces[:, 1] =
    normalDownFaces[:, 1] / module
normalDownFaces[:, 2] =
    normalDownFaces[:, 2] / module

# Generate vertex normals
normalVertex = np.zeros((h*w, 3))
mask = np.logical_and(np.
    logical_and(i > 0, j > 0), np.
    logical_and(i < w-1, j < h-1))

normalVertex = np.roll(
    normalUpperFaces, 1, axis=0)
normalVertex += normalUpperFaces
normalVertex += np.roll(
    normalUpperFaces, -h, axis=0)
normalVertex += normalDownFaces
normalVertex += np.roll(
    normalDownFaces, h, axis=0)
normalVertex += np.roll(
    normalDownFaces, -1, axis=0)

module = np.linalg.norm(
    normalVertex, axis=1)
valid_modules = module != 0
normalVertex[valid_modules, 0] =
    normalVertex[valid_modules, 0] /
     module[valid_modules]
normalVertex[valid_modules, 1] =
    normalVertex[valid_modules, 1] /
     module[valid_modules]
normalVertex[valid_modules, 2] =
    normalVertex[valid_modules, 2] /
     module[valid_modules]

normalVertex[np.logical_not(mask),
    0] = 0
normalVertex[np.logical_not(mask),
    1] = 0
normalVertex[np.logical_not(mask),
    2] = 1

return vertex, faces, uvMap,
    normalVertex
```

A.3 GEOJson example

```
{"type": "FeatureCollection", "name": "
    example9x9q3/outputwcs00", "crs": {"
    type": "name", "properties": {"name
    ": "urn:ogc:def:crs:EPSG:23031"}}, "
    features": [{"type": "Feature", "
    properties": {}, "geometry": {"type
    ": "Polygon", "coordinates":
    [[[412394.118618708,
    4612149.208618707],
    [412394.118618708,
    4613649.2313812915],
    [413894.1413812921,
    4613649.2313812915],
    [413894.1413812921,
    4612149.208618707],
    [412394.118618708,
    4612149.208618707]]}}]}
```

A.4 Map file generated by GEOTool

Example of JSON with 2 chunks.

```
[
    {
        "file": "pallars15_0_0.obj",
        "size": {
            "x": 1000,
            "y": 1000
        },
        "cellsize": 15,
        "chunkpos": {
            "x": 0,
            "y": 0
        },
        "x": 326737.1161870794,
        "y": 4676971.3761870805,
        "textures": [
            {
                "name": "rgb",
                "file": "
                    pallars15_0_0_rgb.
                    jpg"
            },
            {
                "name": "ir",
                "file": "
                    pallars15_0_0_ir.jpg
                    "
            }
        ]
    },
    {
        "file": "pallars15_0_1.obj",
        "size": {
            "x": 1000,
            "y": 1000
        },
        "cellsize": 15,
        "chunkpos": {
            "x": 0,
            "y": 1
        },
        "x": 326737.11618707946,
        "y": 4691971.37618708,
        "textures": [
            {
                "name": "rgb",
```

```
            "file": "
                pallars15_0_1_rgb.
                jpg"
        },
        {
            "name": "ir",
            "file": "
                pallars15_0_1_ir.jpg
                "
        }
    ]
  }
]
```

A.5 Example code for understand the RPC functionality

```
.h:
  virtual void BindFunctions(rpc::
      server* server) override;

.cpp:
void MyClass::BindFunctions(rpc::server
    * server)
{
  Super::BindFunctions(server);

  server->bind("nameOfFunction", [
      context_params](Variables...) {
    //MyCode
  });
}
```

A.6 Example of CSV file to control the vehicle and cameras

Time	x	y	z	LookX	LookY	LookZ
0	10000	2000	3000	10000	2100	3000
50	10000	2020	3000	10000	2120	3000
100	10000	2040	3000	10000	2140	3000
150	10000	2060	3000	10000	2160	3000
200	10000	2080	3000	10000	2180	3000
250	10000	2100	3000	10000	2200	3000
300	10000	2120	3000	10000	2220	3000
350	10000	2140	3000	10000	2240	3000
400	10000	2160	3000	10000	2260	3000
450	10000	2180	3000	10000	2280	3000
500	10000	2200	3000	10000	2300	3000
550	10000	2220	3000	10000	2320	3000

CSV file to control a vehicle

Time	cameraId	LookX	LookY	LookZ	GetImage
0	1	10000	2000	0	1
50	1	10000	2020	0	0
100	1	10000	2040	0	0
150	1	10000	2060	0	0
200	1	10000	2080	0	0
250	1	10000	2100	0	0
300	1	10000	2120	0	0
350	1	10000	2140	0	0
400	1	10000	2160	0	0
450	1	10000	2180	0	0
500	1	10000	2200	0	1
550	1	10000	2220	0	0
600	1	10000	2240	0	0
650	1	10000	2260	0	0
700	1	10000	2280	0	0
750	1	10000	2300	0	0
800	1	10000	2320	0	0
850	1	10000	2340	0	0
900	1	10000	2360	0	0
950	1	10000	2380	0	0
1000	1	10000	2400	0	1

CSV file to control cameras

A.7 Requirements to execute GEOTool

In order to use this application correctly we will have to have installed the Anaconda package (Tested with version 4.6.11 and Python 3.7.3) to which the 'utm' library will be installed with the command "pip install utm".

A.8 Requirements to execute GEOControl

In order to use this application correctly we will have to have installed the Anaconda package (Tested with version 4.6.11 and Python 3.7.3) to which the "mprpc" library will be installed with the command "pip install mprpc".