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 escenaris 3D que poden ser llegits en motors gràfics. Aquests objectes se'ls associa informació geogràfica en fitxers JSON que 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 permet generar trajectòries on es 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 you can see how satellite data generates 3D scenarios that can be read in graphic engines. These objects are associated with geographic information in JSON files that allow a simulator developed in Unreal Engine to read them, load textures and position themselves in that position of the physical world using UTM coordinates. This allows you to generate trajectories where samples will be taken with the simulated cameras. The data sources can be of several types emphasizing those of multispectral type, these sequences of images will serve to generate datasets for the training of algorithms of deep learning.

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 resurgence of deep learning as a possible source of data. In this field we want to represent the real world as accurately as possible, thanks to this we can generate unlikely situations necessary for learning. This need is required when you want to represent real environments that may present a danger. For example, situations of risk (earthquakes, floods, fires, etc.), which have an economic impact (agriculture, resources, infrastructure, etc.) or others. (natural environment, people, etc.).

In this project, the information is generated by elevation maps and aerial images of real terrains. This data is used to create a simulated environment that can generate new data sets. This will generate data that can be used in multiple areas of computer learning. In addition, this project can simulate flights, see a geographic area or expand to other fields that use geographic information. This geographic information can be obtained from multiple sources (satellite data, aerial images, unmanned aircraft captures, virtual generated data, etc.), which allows to work with a large amount of external data.

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

2 Objectives

The main objective of this project is to create a tool to download geographic data and transform it into a 3D representation. In order to carry it out the following subobjectives must be completed: visualize the representation in a 3D engine, visualize multispectral data, move through the terrain (know the UTM position in the world), control a virtual vehicle through an external script or keyboard, generate a set of data that can be used in other algorithms, etc. The complete list can be seen in the appendix A.1.

3 Methodology

In this project an Agile methodology is used. This allows us to identify more efficiently the small parts that make up the project and adapt them to the necessary changes. More specifically, a technique called Kanban [2] is used. Kanban consists of organizing the backlog (list of short-term tasks) in cards that will be placed on a board according to the state of the task (To-Do, Doing, Done). For this purpose, Trello [3] is used, a software that allows to see the boards in a web browser, create cards and move them between different lists.
A Gantt chart has been included to manage the project. In this diagram, several tasks and subtasks are taken into account to make a prediction of the work done, what remains to be done and the approximate time each task will take. This allows us to plan the project with a precise use of time.

4 STATE OF THE ART

Currently, there are several solutions for the generation of images in simulated environments. Some of these simulations are used to generate the data used in machine learning algorithms. On the other hand, there are many sources where to take the initial data (geographical and topographic information) for these simulations.
In this field, some of the most important tools are 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 in simulated environments. AirSim incorporates multiple modules that offer the following features (Figure 1 shows an example): car simulation, drone simulation, real drone driver compatibility, recording, depth view, segmentation view, rain effects, lighting control according to the time of day, vehicle control through a python script, etc.

Fig. 1: Snapshot of AirSim simulator.

4.2 Carla SIMULATOR

Carla is a graphic simulator made with Unreal Engine, this simulator has the purpose of generating images in a simulated environment with the maximum realism to generate images. These images are used in neural networks to safely drive an autonomous car. Thanks to the simulated environment it is possible to take into account the unlikely cases that are difficult to generate in the real world (Figure 2 shows a view of a scenario). Carla has the following characteristics: car simulation, in-depth view, segmentation view, traffic simulation, simulation of pedestrians, control of the actors (traffic, pedestrians, cameras, etc.) with a python script, etc.

4.3 LESS

LESS represents a radiation model (Figure 3 shows an example) generated in a three-dimensional object/terrain for different rays. It is generated from a ray tracing technique, this allows to simulate data and images on realistic three-dimensional scenes. This model implements a weighted photon tracking method to simulate the effect of bidirectional multispectral reflectance.

Fig. 2: Carla example scene.

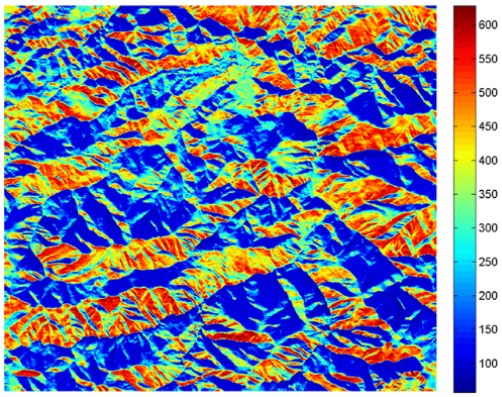


Fig. 3: Example of results of the radiation of a terrain.

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, multispectral or hyperspectral images from the visible to the infrared region of the electromagnetic spectre. 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 Google project, dedicated to provide the necessary tools to analyse and visualize geospatial data. It is intended for academic studies, non-profit insti-

tutions, companies and governments. The main features of the Google Earth engine are:

- Work with datasets from different satellites such as LANDSAT, MODIS, SENTINEL, etc.
- IDE to manipulate data, an extensive API that allows you to combine images from different spectral bands, see them on a world map, export data to Google Drive and more.

5 Structure of project

To determine the structure of the project, the different alternatives that are seen in the section 4 are studied. Specifically, the structure of AirSim and Carla will be analysed to decide the structure of this 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 through RPC, it also includes code to manipulate 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:

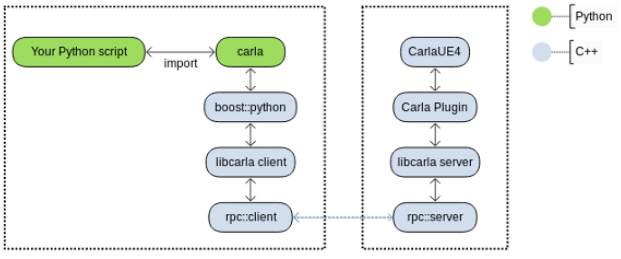


Fig. 5: Relation between the modules of Carla.

- **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 to send orders to the Carla module that works like a server, this API it is useful to make your own scripts.

5.3 The structure chosen

We have analysed two projects with similar characteristics to decide the main structure. This scheme can be seen in the figure 6. A single structure has been chosen because the other projects are based on the creation of Unreal Engine assets. This factor is contrary to the purpose of this project, in which we want to load land at runtime with data which is external to the engine.

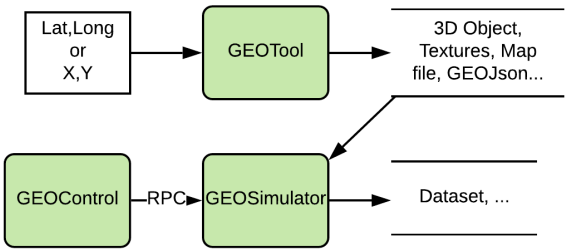


Fig. 6: Structure of GEOSpectralSimulator.

The project consists of the following modules:

- **GEOTool (Python)**: The objective of GEOTool is to provide the necessary tools to obtain and transform the geographical representation in 3D representation. Base data is obtained from various sources such as: standard web services, files, etc. This module downloads and generates the necessary files to be played on a graphic engine.
- **ScriptAPI (Python)**: This module provides the necessary code to read CSV files that determine the behaviour of our vehicle. In order to do this, it's necessary to manage time, files and the connection to the RPC client. The files control aspects such as: the position the place where you are looking from (vehicle or cameras), the possibility of obtaining an image to generate data sets and which texture you want to use.
- **Unreal Engine Simulator (C++)**: Simulator in Unreal Engine that includes the following: A library to implement the RPC server, the necessary classes to read the files generated by GEOTool and manage them, the generation of images, textures management, vehicle management and cameras, etc.

6 Module GEOTool

In this section you can see how the GEOTool module works and what the implemented functionality is. More specifically you can see: the origin of the data, how to configure GEOTool, how the data is transformed to be read in graphics engines such as Unreal, Unity, OpenGL, etc. Finally, the generation performance will be analysed from wavefront object files (`.obj`).

6.1 Obtaining data

To obtain geographic data, it has been decided that the communication will use the standards proposed by the

Open Geospatial Consortium [10]: Web Map Service (WMS) [11] to obtain orthophotos and Web coverage service (WCS) [12] to obtain maps of elevations. These services are sent as parameters to the region that you want to download. In order to carry it out, HTTP requests are used. This process is used by various institutions that follow the standard, in this case tests have been done with the "Institut Cartogràfic i Geològic de Catalunya" [13].

6.1.1 Configuration file

In the configuration file (Written in JSON) we can determine: what area you want to get, what web services you want to request, etc. This file will be received by GEOTool as a command line parameter. In this file you can configure the following parameters (We can see an example with real data in the appendix A.2):

- **Type**: Format of the coordinates. It can be `latlng` or `xy`, 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 determines the sizes that we want to download (The maximum size depends on the webservice). It is composed of:
  - Bbox: Bbox are the dimensions in pixels of the geographical area, this parameter is affected by "cellsize".
  - Texture: Texture is a resolution in pixels request to the texture image.
- **chunks**: The fragments that you want to download. The application calculates the displacement and generates n pieces of width×height defined by BBox. It contains two parameters (height and width) to determine the number of pieces.
- **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.
- **folderWcs**: Folder to save the files.
- **Outputwcs**: Base name of the output files.
- **Formatwcs**: Format of the file generated by heightmaps. Available: `raw, obj` (Object 3D).
- **Wmsrequests**: Array with each request that will be for obtaining textures, each item is 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

In this section we explain the different options to generate files that can be read in graphics engines. It has been decided to investigate the formats: RAW and OBJ.

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 it decided to add the generation of 3d objects in the standard format, generating its objects as you can see in section 6.2.2.

6.2.2 Generation of mesh 3D

To import the terrain in the graphic engine, it is decided to generate a 3D mesh in format `.obj` (format compatible with most 3D editor, graphic engine, etc.). This format offers freedom to control the distance between the vertices, the vertices where the texture is applied and the normal vectors (important for the application of lighting algorithms).

Because the looping is slow, all operations are performed with the NumPy library. NumPy takes advantage of the parallelism of the computer allowing to increase the speed in the calculation of matrices. The problem has been adapted to the operations of matrix type, you can see the code in the appendix **??**.

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

- **Vertex**: They are in every vertex of the world. The vertices are identified by the index as they are inserted (1, 2, 3, ..., H × W). Each point is multiplied by a K (K represents the distance between two vertices according to the distance indicated by 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 it is necessary to 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 it finally normalizes 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 7 applying the next equation for each vertex where V corresponds to the vertices and F corresponds to the faces on the figure 7:

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

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

- **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, that is, for each square of the mesh two triangular faces are generated. It is important to generate the faces in the correct order, writing the vertices in an order opposite to the hands of the clock, in this way the graphic engine determines that the normal face will point upwards when showing the 3D mesh correctly.

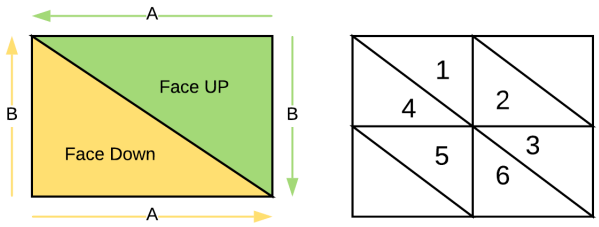


Fig. 7: Pattern for calculating the normal vector on the faces (left) and vertex (right).

Once the generation process has been completed, the application will generate a mesh that can be opened in any editor. You can see an example of open mesh in the MeshLabs application in the figure 8.

6.2.3 GEOJson associated at the terrain

To locate the terrain generated in third-party software that works with geospatial data, a GEOJson file is included. This file indicates which is the geographical zone that has been downloaded. An example of this can be seen in the appendix A.4.

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.

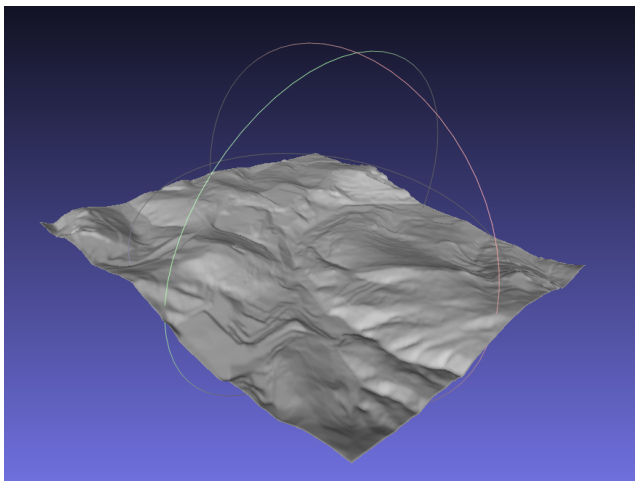


Fig. 8: Terrain mesh visualized in software MeshLab.

On the figure 9 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.

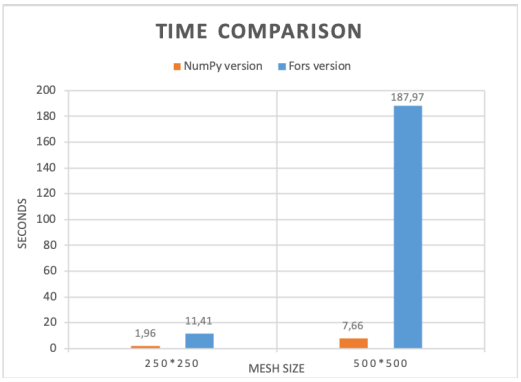


Fig. 9: Graph with the time of generation of a mesh according to size.

6.4 Quality

This section shows the results of reducing the quality of the land, both qualitatively (visual difference) and quantitatively (size in bytes). To make this reduction, ”meshStep” size jumps are made, established in the JSON configuration file. The tests have been carried out with powers of 2 (1,2, 4, 8, 16, ...) in the own matrix of points and using the terrain of size 300×300 pixels.

As you can see in the figure 10 by reducing the number of points, the size in bytes is reduced exponentially, which causes the file to load faster in the Unreal environment. As you can see in the figure 11, the loss of quality is visible. If you take the mountain as a reference, you can see the loss of definition on 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 perceptible. At 4, you may notice a little, and finally at levels 8 and 16, you notice a noticeable loss of quality.

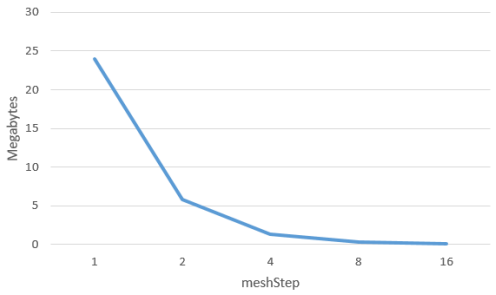


Fig. 10: Graph with the occupied size in bytes varying the parameter "meshStep" for a graph of size 300×300.

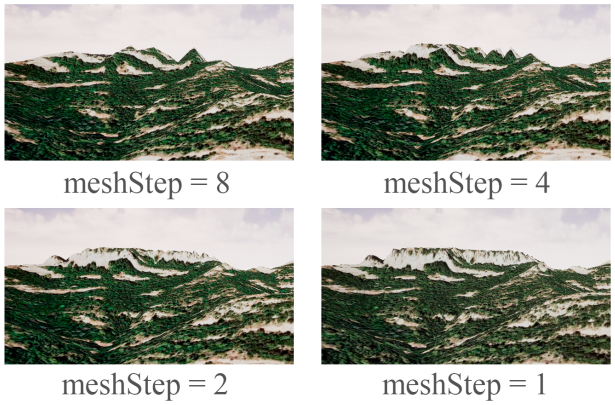


Fig. 11: Visual comparison of the effects produced by the level quality.

6.5 Map file

GEOTool generates a file to be able to locate and load the scenario (map file). The map file refers to all the files downloaded with GEOTool (chunks). These parts contain basic information: location in the real world, size, size of each pixel, the textures we want to load, etc. The file follows a format so that it can be generated without using GEOTool and then loaded into the GEOSimulator module adding information of any origin (respecting the formats of the files). You can see an example in the appendix A.5.

7 Module: Simulator

In this section the multiple parts of which the GEOSimulator module is composed are explained. The objective of this module is to provide a tool that simulates a trip with a generic vehicle on a terrain, obtaining images of several cameras on board of the vehicle. One way to do this is to control the vehicle using the RPC protocol. In this case the RPC server is implemented in the simulator as can be seen in the section 7.1.1 and it is controlled with an RPC client implemented in the GEOControl module as it can be seen in the following section 8.

7.1 Visualization and control of the vehicle

The objective of this part is to provide the interface with the ability to move a generic vehicle in the simulated world. In the vehicle you can configure several cameras, this way you can see the same area in multiple perspectives. As an example, you can see the zenith view in the figure 12.

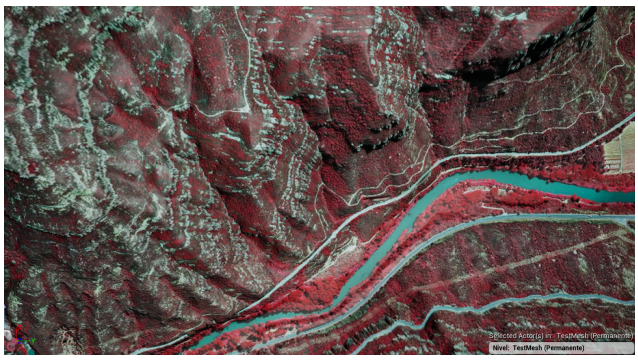


Fig. 12: Infrared view of Montserrat in zenith position.

7.1.1 Remote control

In order to control the vehicle remotely, a communication using the RPC protocol (Remote Procedure Call) is implemented. More specifically, it is by using the Rpclib [15] library for C++. This server is implemented in the GEOSimulator module. The administration of the RPC server is done through an "AVehiclePawn" class. This class can be inherited so that we can add more specific functionality (vehicles of different types, specific behaviours, etc.). The following actions are implemented: initialize / stop the RPC server, change the position of the vehicle, change the rotation of the vehicle or cameras (implemented with Look At), request images of one of the cameras that will be generated, etc.
The assignment of the functions is done in the function `void BindFunctions(rpc::server* server)` that the server instance receives. This function can be overwritten to add more functionality (To see how to do it see the appendix A.6). 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, from 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 chooses a camera and configures in what direction the camera looks 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 the image are going to be saved 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

To implement the sky, the native Unreal module is used, where a sphere is generated in which the sky is represented. It has multiple parameters to configure the sky in multiple ways, some of the things that can be done are: determine the position of the sun, the lighting of the stars, the amount of clouds, the speed of the clouds, etc.

This allows to generate synthetic images at different times of the day with different lighting to generate more complex data sets. On the other hand, if you want to get the maximum realism you have to use textures without shadows causing Unreal to generate them, otherwise it will create an undesired effect. An example of the different hours of the day can be seen in the figure 13.

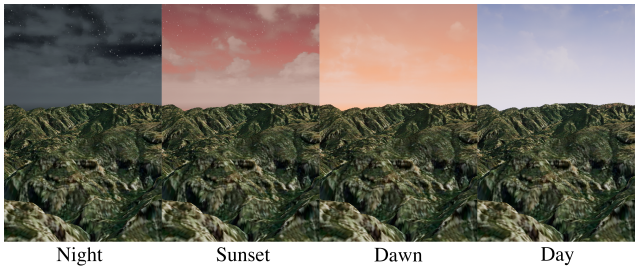


Fig. 13: Example of differents sky.

7.2 Terrain

The terrain is loaded through the code executed at runtime, for it loads the `obj` file and is processed using the class UProceduralMesh [16], this class allows us to add the geometrical information to the simulator. To load the meshes, it must be considered that Unreal works with a different system of coordinates, in which the y axis is inverted with respect to the UTM model. Due to this, it is necessary to make the necessary changes.

This mesh is loaded in the world and it is represented by the class "MapChunk", this class is assigned a dynamic material that allows to change between the loaded textures. The texture can be any image prepared to be used as a texture in Unreal, giving the possibility to see RGB, infrared, multispectral bands, indices generated by multispectral data, etc.

7.3 Results of visualization of terrains

In this section you can see the results obtained from visualizing 3D generated by the GEOTool module in the GEOSimulator module. To generate these results, images from different sources have been used, more specifically, the RGB and infrared views are obtained from the "Institut Cartogràfic de Catalunya", the multispectral data is obtained from the Sentinel 2 [18] satellite through the EO Browser [17] and depth data are generated by applying a shader to the camera.

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. The first three bands B02, B03 and B04 correspond to the bands of colours, the band B05 corresponds to the fast change of the reflectance that is produced in the vegetation in a range near the 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 15 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 to 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 in lakes or reservoirs.

The last view of the figure 15 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:

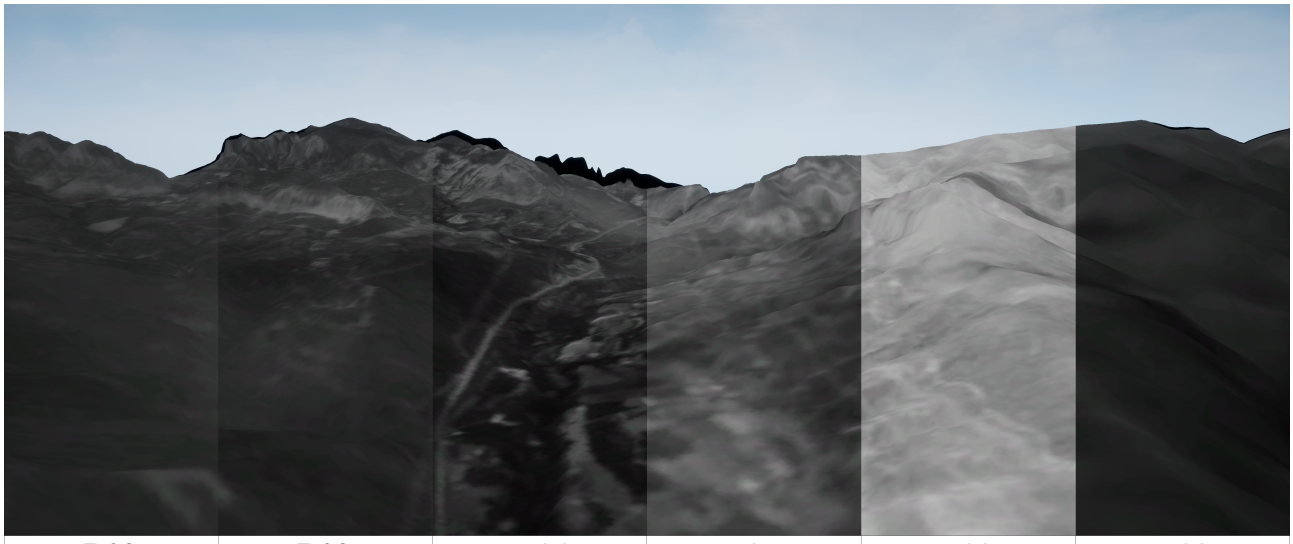
- A,W,S,D: Left, Front, Back, Right movement of the vehicle.
- Alt, SpaceBar: Down and up the vehicle.
- Q,E: Left and right rotation of vehicle.
- Y: Starts an RPC Server
- U: Stops an RPC Server.
- H: Show/Hide the visualization of vehicle.
- K: Save image in the simulator folder from the second camera.

8 Scripting module

This section contains an explanation of the main points of the scripting module. Some of the functionalities of this module are: read trajectory files, read control files of cameras, manage time, request images, etc.

8.1 Paths

The scripting module allows you to make trips, read trajectories created in a real world (adapting this to the correct format) or generate synthetic trajectories. In this way, a trajectory that has been previously made can be recreated in the simulator, it can be reproduced as many times as desired. This allows us to carry out multiple tests with different visualizations of the land passing through the same points. For this, finally, some files are generated in CSV format that are explained in the section 8.1.1.



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

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

8.1.1 File of paths

Now it's explained how to configure 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 indicate to the simulator the positions of the vehicle and where both the vehicle and the cameras should be viewed (in UTM coordinates).
The file which controls the vehicle is composed by the next fields:

- Time: Time in milliseconds from start of the script.
- 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 which controls the cameras is 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.7.

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 first's tests, noise was generated through code, but that way the tests can't be reproduced multiple times. Due to that, 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, drones or synthetic which guarantees that the tests can be reproduced multiple times.

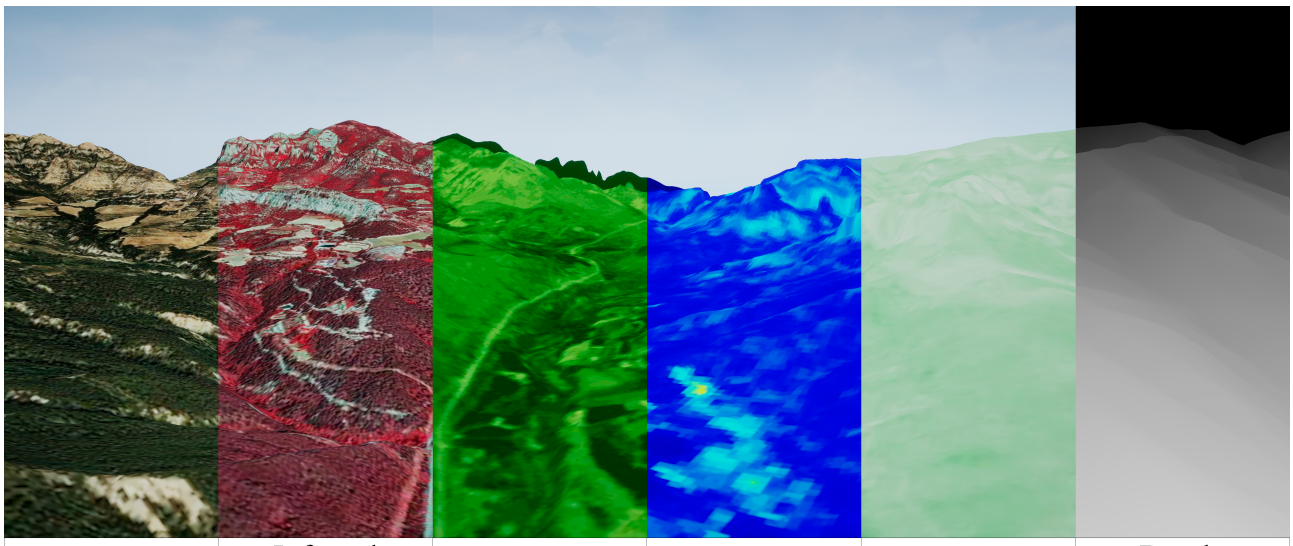
8.3 Generation of Datasets

One of the objectives of this project is to offer the possibility of generating data sets of synthetic images generated by the simulator. In order to generate data sets, the “GetImage” field is used, as seen in the section **??**, which generates an image each time it finds this field in 1.
In the figure 16 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 section shows an example of the workflow that can be achieved with the modules generated in this project. This will show you how to use them and will serve to think about new applications that can be developed using this project as a base.

First, the GeoTool module is used to obtain data: For this we will obtain the coordinates of the area we want to obtain, we can use applications such as Google Maps. In this case it is used to download the area of the “Santa Margarita Volcano”, the approximate coordinates (lat 42.130596, longitude 2.521956) are selected to obtain the area around it. Once chosen, the JSON is configured, which can be seen in the section 6.1.1. Once the process is completed, the module is called with the following command: `./GEOTool.sh config.json` (the requirements to execute this application can be seen in the appendix A.8). Once the process is finished, a series of files are generated, as can be seen in the figure 17.



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

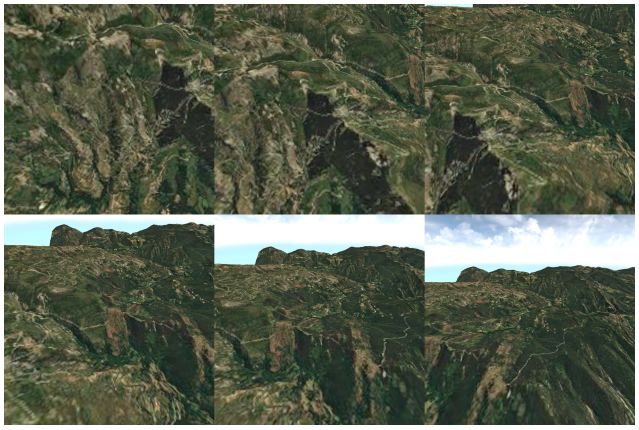


Fig. 16: Set of images generated by GEOControl.

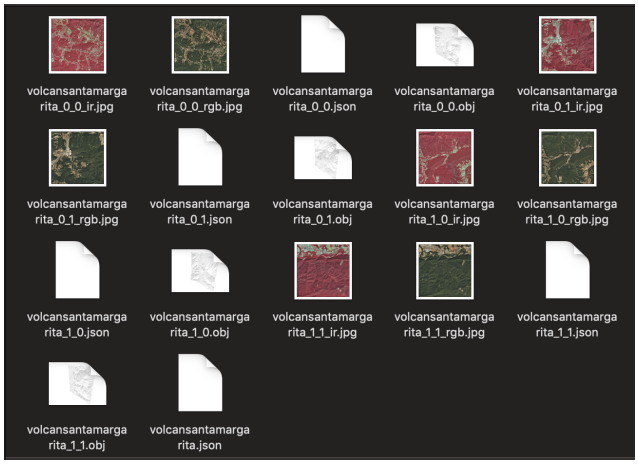


Fig. 17: Files generated by GEOTool.

Once the map is generated, the files will be added to the folder “Maps” which is located next to the simulator’s executable. The simulator will run with the following command `./GEOSimulator.sh -folderMap="Maps/volcan/" -map="volcan.json"`. Once the simulator has been initialized, it can be seen in the interface located in the upper left corner: the position in the world where the vehicle is located and the status of the RCP server, an example can be seen in the figure 18.

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.

With the files containing the trajectories already created as indicated in the section 8.1.1, the GEOControl module can be invoked with the following command `./GEOControl.sh filevehicle.csv filecamera.csv` (the requirements to run this application can be seen in the appendix A.9). This module will execute and generate the data set in the folder of “output”. You can see an example in the figure 16.

10 Conclusions

This project shows how to obtain textures and elevation maps originated from satellites, this information can be manipulated to generate indexes, new information generated through neural networks, etc. The processed data is transformed into 3D information that can be interpreted by graphics engines such as Unreal Engine. It has taken into account aspects such as the time needed to generate the 3D model, which have been analysed to see the effect of libraries like NumPy during matrix manipulation, accelerating the process thanks to parallelism. You get faster times thanks to the reduction of points to generate 3D models, this reduction is useful for rendering large models.

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 an RPC server implemented in Unreal Engine, in which we

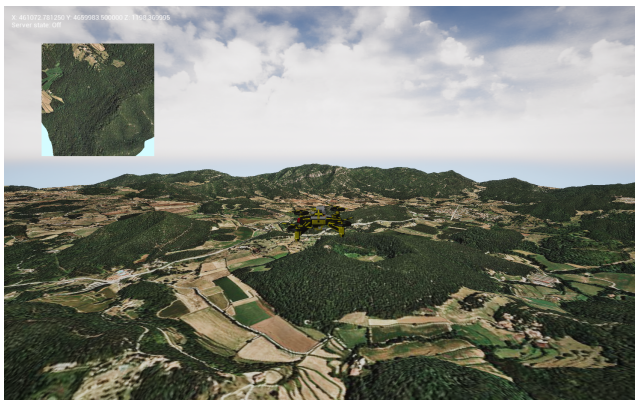


Fig. 18: Simulator view.

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 Satellites like the Sentinel 2, show how this information is represented in the simulated environment, which use has each type of data and what possibilities the simulator offers when seeing this information.
The section pertaining 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 friends Javier García Cantero and Jordi Miranda for helping me by revising the English version of this paper. And lastly, 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 List of objectives

1. Analyze.
2. Define a structure.
3. Developing.
   - 3.1. Develop module GEOTool.
     - 3.1.1. Develop configuration file.
     - 3.1.2. Develop the transformation to RAW and OBJ.
     - 3.1.3. Develop reduction of quality.
     - 3.1.4. Develop map file.
   - 3.2. Develop module GEOSimulator
     - 3.2.1. Develop vehicle logic
     - 3.2.2. Develop RPC Server
     - 3.2.3. Develop the image acquisition
     - 3.2.4. Develop movement of cameras
   - 3.3. Develop module GEOControl
     - 3.3.1. Develop clock for all scripts
     - 3.3.2. Develop csv files.
4. Test
5. Keep record

A.2 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.3 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.4 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.5 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.6 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.7 Example of CSV file to control the vehicle and cameras

	1	2	3	4	5	6	7
1	# Time	x	y	z	LookX	LookY	LookZ
2	0.0	461183.0	4660069.0	1100.0	461193.0	4660079.0	1100.0
3	16.0	461190.65265265264	4660075.0560560056	1100.0	461200.652652652564	4660085.056056056	1100.0
4	32.0	461198.3053053053	4660081.112112112	1100.0	461208.305305305 3	4660091.112112112	1100.0
5	48.0	461205.95795795793	4660087.1681681685	1100.0	461215.957957957 93	4660097.1681681685	1100.0
6	64.0	461213.61061061063	4660093.2242242225	1100.0	461223.610610610 63	4660103.224224225	1100.0
7	80.0	461221.2632632633	4660099.28028028	1100.0	461231.263263263 3	4660109.28028028	1100.0

CSV file to control a vehicle

	1	2	3	4	5	6	7
1	# Time	cameraId	LookX	LookY	LookZ	GetImage	Channel
2	0	1	461183	4660069	0	1	rgb|ir
3	16	1	461183	4660069	0	0	rgb|ir
4	32	1	461183	4660069	0	0	rgb|ir
5	48	1	461183	4660069	0	0	rgb|ir
6	64	1	461183	4660069	0	0	rgb|ir

CSV file to control cameras

A.8 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.9 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".