

GEOspectralSimulator

Kevin Martín Fernández

Resum– En aquest projecte es pot veure com a partir de dades de satèl·lit és generat objectes 3D reproduïbles en motors gràfics, als quals se li 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 en les quals mitjançant les càmeres incorporades a un vehicle es podran generar datasets.

In this project can see how from satellite data are generated 3d objects, It is possible to reproduce in a graphical engine.

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

Abstract– In this project it can be seen how 3d objects are generated from satellite 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, Satelite, Multiespectre, Unreal Engine

1 INTRODUCTION

The simulation of real worlds for the generation of synthetic images is a field in which 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 its image datasets, which can be used in multiple areas of computer learning. Also, this project can simulate flying, see a geographical area or expand in other environments that use geographic information. That geographical information can be provided from multiple sources, which allows to work with a lot of external data.

2 OBJECTIVES

The goal of this project is give a tools that allows to download satellite data, transform satellite data to 3D data, generate data that is can be reproduced in a 3D engine, adapt multispectral data to 3D models (textures), navigate on the world trough on the simulator, control a vehicle from a external data and generate datasets can be used in other algorithms.

3 METHODOLOGY

In this project an Agile methodology is used, this allows to identify more efficiently the little parts that compose the project, and to adapt to changes needed along the way. More specifically, a technique called Kanban[2] is used, which consists in organizing the backlog (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.

3.1 Gant Diagram

To manage the project a Gant diagram has been included done with excel. In this diagram several tasks and subtasks are contemplated 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, a diverse number of applications exist for the generation of images in simulated environments with the goal of generating data used in machine learning algorithms.

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

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

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 1): 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.

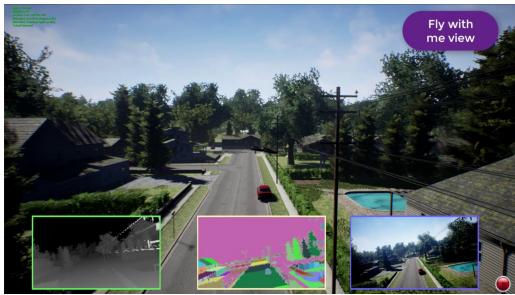


Fig. 1: AirSim simulator

4.2 Carla SIMULATOR

The 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.

4.3 LESS

LESS is a model of the radiation (You can see an example, in the figure 2) 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.

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 hyperspectral from visible trough of the infrared thermal region of the electromagnetic spectral. You can see an example generated by DIRSIG on the figure 3.

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 visual-

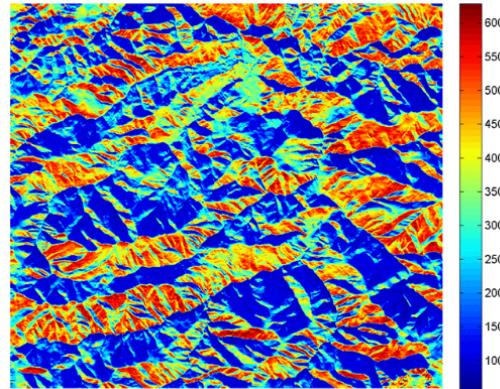


Fig. 2: Example of results of the radiation of a terrain



Fig. 3: Frame from the Tacoma port

ize geospatial 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 deciding 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 through 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 module to see the AirSim information.
- **Unreal Engine (C++):** Unreal version, includes a module to see the AirSim information.

5.2 Carla SIMULATOR

Carla SIMULATOR is composed of multiple modules as you can see in the figure 4 written in multiple languages. The modules of Carla are:

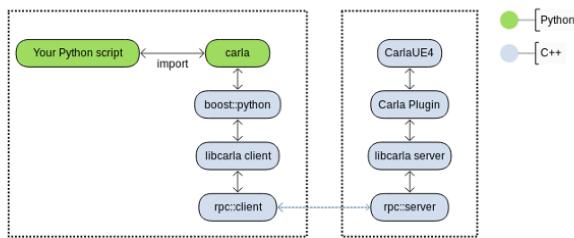


Fig. 4: Relation between the modules of Carla

- **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 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 5, 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.

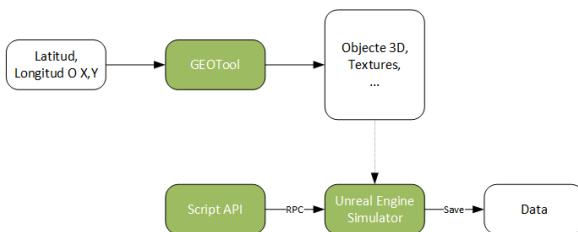


Fig. 5: Structure of GEOSpectralSimulator

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 web-service 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 through 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 be 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 geographical 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 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 heights. 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 Bytes, 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.

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

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

- **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 6 where we follow the equation 1 for the calculation of the normal face. \vec{A} , it makes cross product $\vec{C} = \vec{B} * \vec{A}$ 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 7 applying the next equation for each vertex where V correspond 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, 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.

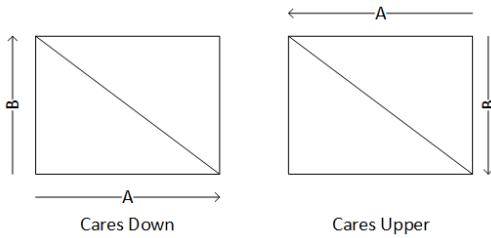


Fig. 6: Pattern for the calculation of normal on the faces

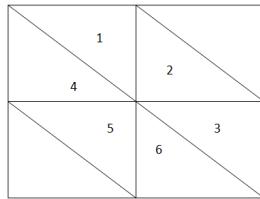


Fig. 7: Pattern for calculation of normal at a vertex

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 8 where it is open using the MeshLabs software.

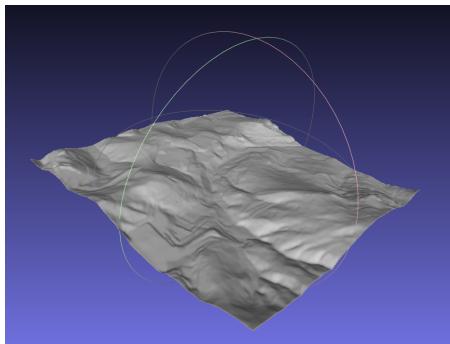


Fig. 8: Terrain mesh visualized in software MeshLab

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 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.

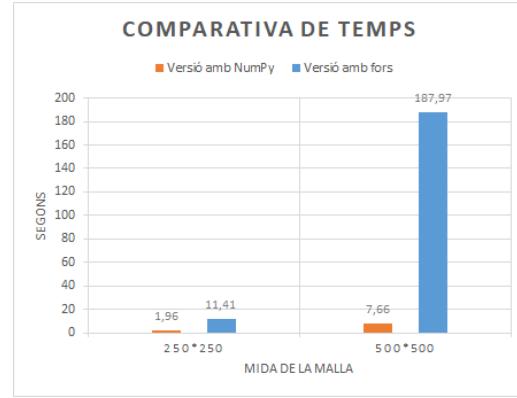


Fig. 9: Graph with the time of generation of a mesh according to size

6.4 Quality

This section shows the results of reducing the quality of the 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 300x300 pixels.

As it can be seen in the figure 10 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 11 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

It is generated a file for the after reading of the map on the simulator. The map file is associated to the chunks downloaded in request, contains the basic information from the GEOTool module: location on real world, size, pixels size, textures that will be load, etc. This file can be modified to modify locations, add textures (It is required that all chunks has the texture with same name in opposite case can't will load), etc. In the appendix A.4 can be seen an example.

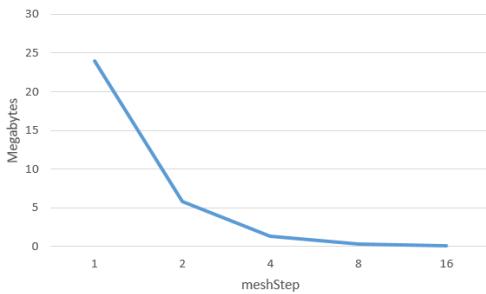


Fig. 10: Graphic with the size occupied on hard disk varying the "meshStep" parameter for a terrain of size 300x300.

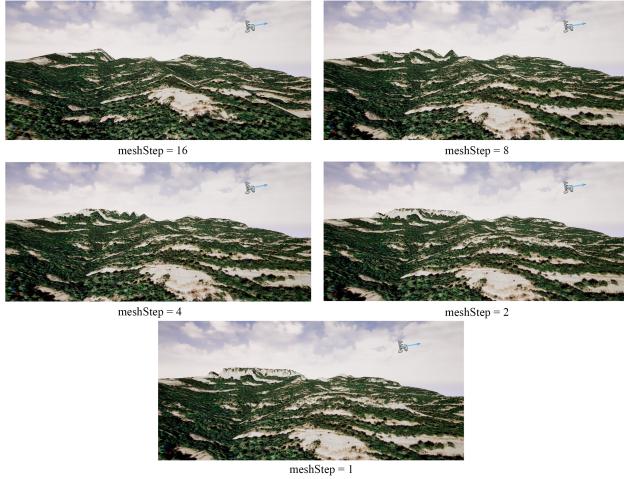


Fig. 11: Visual comparison of the effects produced by the level quality

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 controlled 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 12.

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)

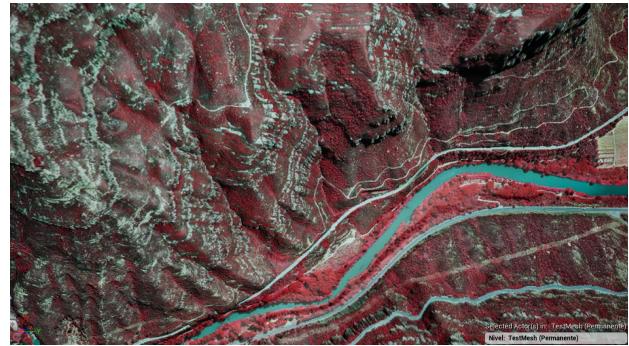


Fig. 12: Infrared view of Montserrat in zenith position

hiclePawn) 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: Initialize the server by pressing the Y key and stop it with the U key.
- 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.

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 ???. 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): It allows us to indicate which camera and where we want to save the image.

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 different hours of the day can be seen in figure 13.

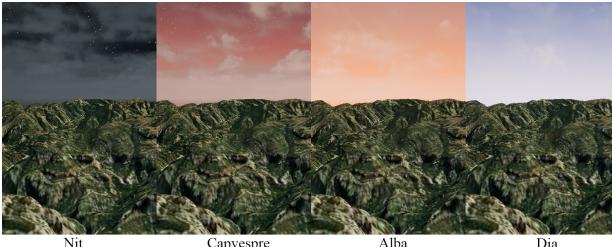


Fig. 13: Example of sky

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 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 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 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 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:

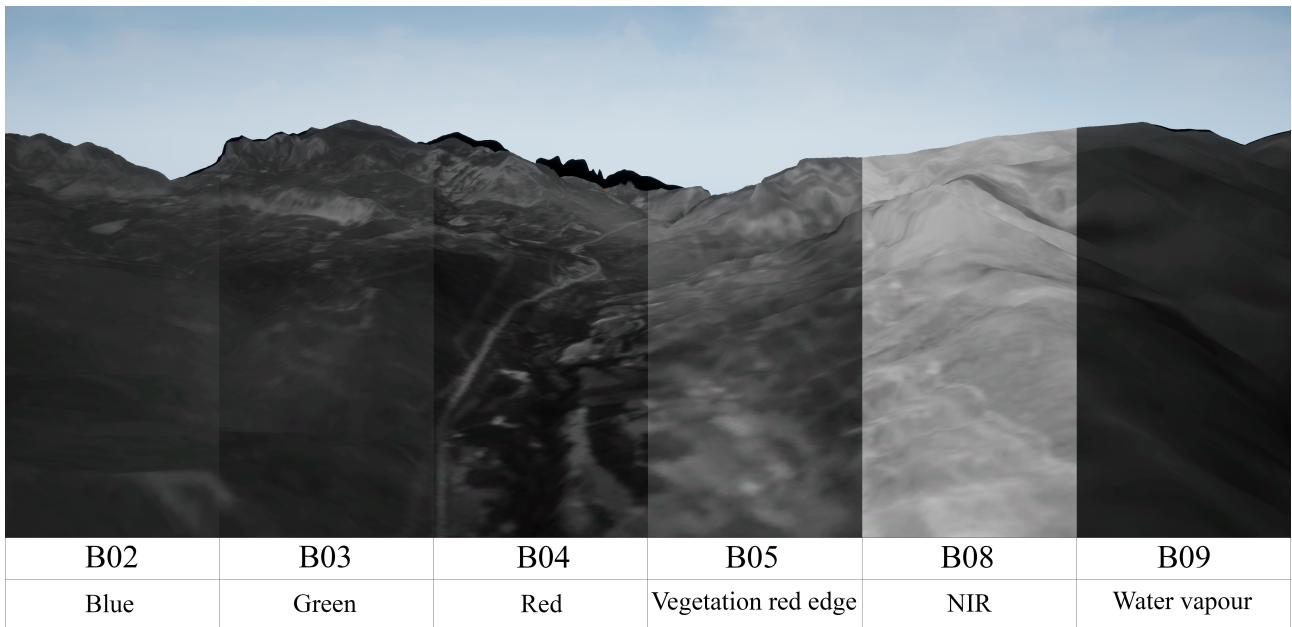


Fig. 14: View of "Sales de Pallars" in six spectral bands differents.

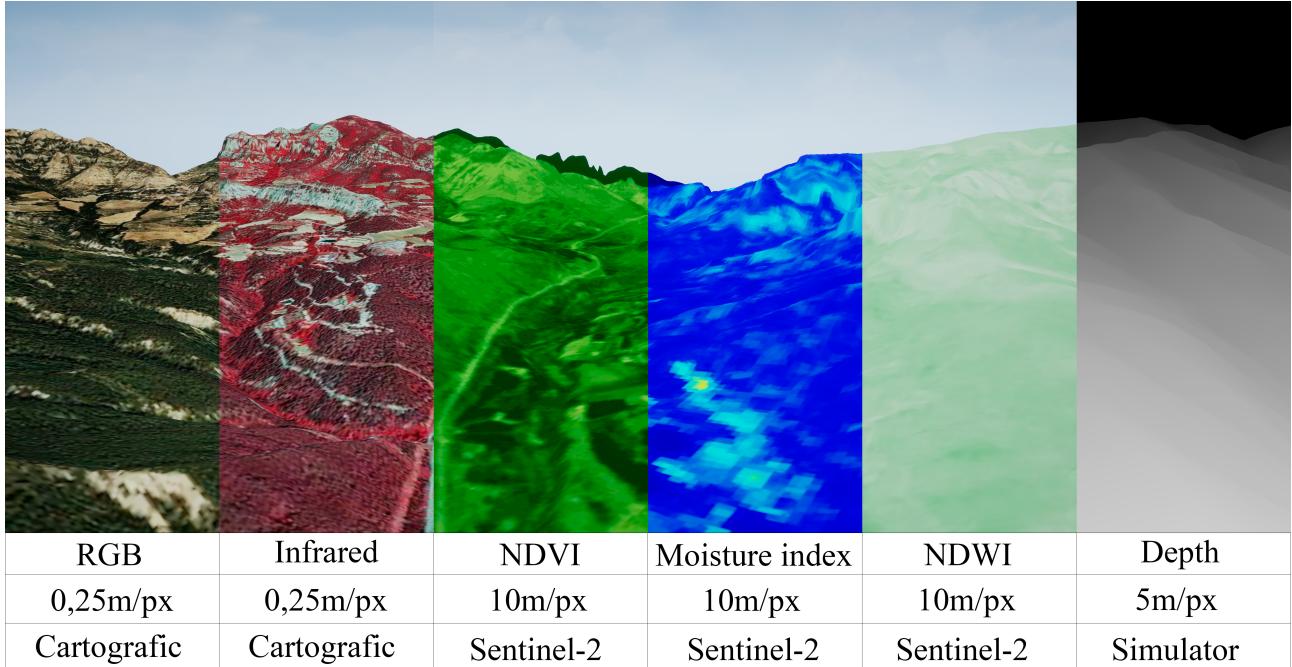


Fig. 15: View of "Sales de Pallars" in RGB, infrared, NDVI, moisture, NDWI and depth.

- Time: Time in milliseconds from start of the script.
- x: Position X in UTM format.
- y: Position Y in UTM format.
- z: Position Z in UTM format.
- LookX: Position X to which we look in UTM format.
- LookY: Position Y to which we look in UTM format.
- LookZ: Position 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: Position X to which we look in UTM format.
- LookY: Position Y to which we look in UTM format.
- LookZ: Position 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.

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

8.2 Noise simulation

As vehicles have unexpected movements due to the wind and the condition of the asphalt among others, a generator of noise is implemented and applied to the trajectory you want to follow, this way images with noise and unexpected movements are generated. This noise is implemented through a Gaussian noise in which the point at which you want to move is used as the center and the sigma is small (between 0 and 1), which generates a light movement.

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 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.

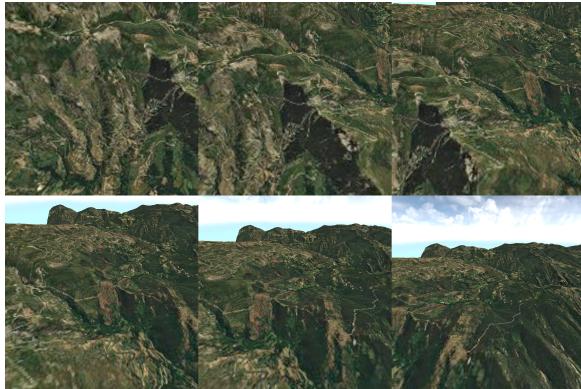


Fig. 16: Set of images generated by GEOControl

9 FULL WORK FLOW

This section contains an example of full work flow it can be made with this modules. This allow know how to be used and can be seen new applications that use as base this project.

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 treated data is transformed into 3D information that can be interpreted by graphical engines like 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 environment 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 Satellites like the Sentinel 2, show how this information is represented in the simulated environment, which uses 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 environment, 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 condition 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. And last, I would like to thank my friend Javier García Cantero, for helping me by revising the English version of this paper.

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 msgpack-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/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 diferenciada 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", "geometry": {"type": "Polygon", "coordinates": [[[100, 100], [200, 100], [200, 200], [100, 200], [100, 100]]]}}, {"type": "Feature", "geometry": {"type": "Point", "coordinates": [150, 150]}}, {"type": "Feature", "geometry": {"type": "LineString", "coordinates": [[100, 100], [200, 150], [300, 200], [200, 250], [100, 200], [100, 100]]]}]}
```

```

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	camerald	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