POLITECNICO DI TORINO

Master's Degree in Cinema and media engineering

Politecnico
di Torino

Master's Degree Thesis

PEDAL project: duplication of real urban areas as 3D virtual environments

Supervisors

Prof. Andrea SANNA

Prof. Vito DE FEO

Candidate

Simona POTENZA

October 2022

"The woods are dark and scary, but the only way out is through"
Todd Chavez, BoJack Horseman

Summary

Serious games today are widely used in many fields, especially in the medical field in order to help patients.
The PEDAL (Personalized Environment for Dementia Assisted Living) project is a VR experience developed for people affected by Alzheimer's disease and frontotemporal dementia. Because of this disease, patients experience short term memory loss and difficulty with mobility. As a result of this, it is hard for them to navigate the real world and their mood is often altered.
The PEDAL project's goal is to help people affected by Alzheimer's disease to face the difficulties that they encounter while navigating in real life. Using a stationary bike and a headset, the patient can navigate through a virtual city and autonomously train both their memory skills and their physical health. In particular, during the experience, the patient has to reach certain points in the environment. Along the path that they have to follow, some information is shown. Doing so, the patient links this information to the route that they have followed. In this way, they can train both their mind and body in total comfort and safety. During the experience, the patient's ECG and other parameters are constantly monitored so as to adapt the experience according to these outputs. Moreover, a virtual assistant, depicted as a robot, helps the patient to fulfill their task.
The project was developed thanks to the help of students from the University of Essex and from the Polytechnic of Turin, guided by professor Vito De Feo. Daniele Bigagli developed the algorithm to define the paths that the users have to follow during the experience, Francesco Penengo developed the system to capture data about the patient during the experience and communicate it to the server and Giacomo Mineo managed the creation of the UI and the development of the virtual assistant.
In order to achieve the best possible result, it is necessary to build a virtual environment that resembles the city in which the patient lives. In this manner, they can perform a transfer of spatial information from the virtual environment to the real one.
Since creating big custom environments can be a time-consuming and expensive task, the aim of this study is to develop an algorithmic method to replicate real

cities as 3D environments. To achieve this result, various algorithms were developed, whose goals were to depict the proper positions and dimensions of streets and buildings, place city components following their actual latitude-longitude coordinates, create buildings depicting real facades and lastly, manage green areas filling them with appropriate assets.

During the years various methods to create realistic urban areas and to replicate real cities were developed. These methods can be classified in three categories: procedural modeling, image-based modeling and inverse procedural modeling.
Procedural modeling consists of a series of grammar-based rules that allow the construction of 3D models. This approach was tested by various researchers, whose methods are capable of creating realistic environments, yet without replicating real urban areas. The rules for such methods can be based on various inputs.
Image-based modeling consists of extracting the city's or the building's features from one or various images. An image-based approach can be used to replicate real cities as virtual environments. Different kinds of images can be used as inputs: topographic maps, aerial and satellite images, street-view photos and more.
Inverse procedural methods consist of extracting features and parameters from an input and create a procedural model to generate such data. Using this approach, it is possible to create very distinctive urban areas, yet without duplicating real ones.
In addition to these studies, a lot of technology was developed to replicate real environments. It is possible to find various plug-ins (Blender GIS and MapsModelsImporter for Blender, Maps SDK for Unity, Placemaker for SketchUp) and pieces of software (ArcGIS SDK and ArcGIS CityEngine).

To achieve the desired result, a series of algorithms were developed together with the Blender OSM add-on and data provided by the Mapillary platform.
Blender OSM is a Blender plug-in that is capable of recreating real environments as simple scenes in Blender, depicting buildings, streets, green areas and waterways, according to real dimensions and positions. The scene imported by Blender OSM is defined by latitude-longitude coordinates. The add-on also provides its code and documentation, which contains a method to perform the conversion from latitude-longitude to Blender coordinates using the transverse Mercator projection.
Mapillary is a platform based on crowd-sourcing, whose goal is to photographically map the entire globe. Thanks to the computer vision technology developed by the Mapillary team, it automatically detects the objects captured in the photos uploaded by the users and calculates their position in latitude-longitude coordinates. Thanks to this platform, it is possible to obtain data about city components, such as traffic signs, traffic lights, street lights, benches and more than 42 different types of objects. For the purpose of this study, Mapillary provided a great dataset containing information about the detected objects' position.

Thanks to the methods provided by Blender OSM and the database provided by Mapillary, it is possible to place city components according to their actual location, performing the conversion from latitude-longitude to Blender coordinates. In addition to this, to achieve a more realistic result, all the traffic signs are algorithmically oriented in opposition to the direction of traffic. This algorithm is based on the geometry of the street to which the traffic sign is closer and on cross products, performed on the vectors that make up the street.
In addition to the placement and orientation of city components, an algorithm to texture buildings was developed.
Moreover, trees can be algorithmically placed on green areas imported by Blender OSM and 3D sidewalks are obtained from the original import provided by the add-on.
The present method was tested replicating a surface that measures approximately $420m^2$ of the city of Colchester (UK), where the PEDAL project is developed.
Because in the PEDAL experience the user has to reach specific points in the environment, a series of points of interest of Colchester were manually modeled and placed in the scene. Other manual changes were applied to obtain a more realistic and visually appealing result.
Using the developed algorithms, over 350 city components were placed according to their actual locations.
Thanks to this method and the applied manual changes a plausible replica of Colchester was obtained.

In this thesis these topics will be accurately analyzed.
In chapter 1 there is an introduction to serious games designed for people affected by Alzheimer's disease and a description of the PEDAL project.
In chapter 2 the previous works and the current technology to replicate real environments is reported.
In chapter 3 the technology used for this study is examined.
In chapter 4 it is possible to find the detailed description of the developed algorithms, together with their pseudo-codes. In addition to this, the building of Colchester is reported.
In chapter 5 there are various comparisons between real environments and the output of this method, both before and after applying manual changes.
In chapter 6 future developments are reported.

List of Contents

List of Figures

List of Algorithms

Acronyms

VR Virtual Reality

PEDAL Personalized Environment for Dementia Assisted Living

AD Alzheimer's disease

MCI Mild cognitive impairment

VE Virtual environment

LoD Level of detail

OSM Open Street Maps

CC City component

TS Traffic sign

OM Original mesh

GI Green item

Chapter 1

Introduction

1.1 The PEDAL project

Serious games are games that do not simply focus on entertainment, but also want to educate the gamer about certain topics regarding education, city planning, finance managing, scientific exploration and more. Today they are widely used in the medical field to both train health workers and help patients.
As reported by Susi et al. [1], there are several uses of these virtual reality (VR) experiences for patients: there are games that focus on physical exercise, education on self-care, distraction therapy, recovery and rehabilitation, memory training and diagnosis of mental illness and mental conditions such as ADHD and PTSD.

The PEDAL project (Personalized Environment for Dementia Assisted Living) is a VR experience designed for patients that suffer from Alzheimer's disease (AD) and frontotemporal dementia.
AD is a neurodegenerative disease that worsens over time. In the first stage it is referred to as mild cognitive impairment (MCI) and the symptoms mainly involve short term memory loss. When the disease worsens, it becomes hard for the patient to be independent as the memory loss turns more severe and language and mobility impediments start to appear. Because of this disease, the patient's mood is often altered and it is hard for them to navigate the real world.

The PEDAL project's goal is to help people with AD to face the difficulties that they encounter while navigating in real life and overcome the obstacles caused by the disease.
Using a stationary bike and a VR headset, the patient can navigate through a virtual city and autonomously train both their memory skills and their physical health.

In particular, the user has to reach certain points located in the virtual environment (VE). Along the path that they follow some information is shown and, using the "memory journey technique", the participant trains their memory linking the shown information to the path. During the experience, the patient's ECG and other parameters are constantly monitored so as to adapt the experience according to these outputs.

1.2 Serious games and Alzheimer's disease

Over the years, many VR experiences that focus on AD have been developed.
Research suggests that it is possible to use this technology, along with other more common tests, to detect the onset of dementia, with particular focus on MCI [2] [3].
Both the researches carried out by Werner et al. [2] and Howett et al. [3] focus on navigational tasks executed by a group of people affected by MCI and a control group. During these tests, the MCI group's results were worse than the control group's one, suggesting that this dissimilarity could be a sign of the onset of MCI and AD.
Usually, the onset of AD is detected by Mini-Mental State Examination (MMSE)[4], Clinical Dementia Rating (CDR) [5], brain scans such as Magnetic Resonance Imaging (MRI) and Positron Emission Tomography (PET), single photon emission computerized tomography (SPECT) and more.
These studies [2] [3] suggest that spatial behavioral tests carried out using VR technology may support the above-mentioned tests to detect the onset of AD.

In addition to these studies, White et al. [6] tried to verify if learning to navigate a VE could benefit navigation in real life. Like the PEDAL setting, the patient had to use a VR headset to navigate the VE.
White et al. report that "The overall results of this case study suggest that people at the early stages of AD can learn to navigate paths in a suitably immersive VR system, and the learned paths may translate to overall real-world spatial navigation skill". It is important to mention that this experience was tested on a singular patient and that the results can not be generalized.

1.3 Virtual-real transfer of spatial knowledge

In order to permit the transfer of spatial learning from a virtual to a real environment, it is necessary to allow the user to move easily and to put them in a realistic VE.

In fact as reported by Larrue et al. [7], a full body-based information treadmill with rotation could allow the user to better transfer the navigation information

learnt in a VE to real life. In this study, the VE was a reproduction of Bordeaux and the participant had to learn a path using the VR experience and replicate it in real life. The proposed environment was handmade and consisted of a series of simple geometries with real facades applied as textures. The best results (in terms of number of mistakes and time needed to fulfill the navigation in real file) were obtained by participants that used a treadmill with head rotation, compared to the ones that used a joystick or a treadmill without head rotation.
"These results indicated that visual flow alone is not sufficient to be effective in spatial learning and that both rotational and translational vestibular information are required to obtain the best performances in this virtual-real transfer-based wayfinding task" Larrue et al. report.

In addition to this, Wallet et al. [8] pointed out that visual fidelity could help to transfer spatial knowledge from a virtual to a real environment. In this case, the focus was not on the VE's realism, but on its fidelity to a real urban area.
As per Larrue et al. [7], the VE was a reproduction of Bordeaux and the user's spatial learning was evaluated by various tasks. Two versions of the VE were proposed: a textured one and one without textures. In both cases, the buildings were depicted using simple geometries.
The results of this study show how a more accurate VE (in terms of fidelity to the real one) allows a better virtual–real transfer of spatial knowledge.

To obtain better outcomes with the PEDAL experience, it is necessary to create VEs that are as true as possible to real ones and navigate them easily.
Because of this, a custom made stationary bike has been designed to allow the participant to navigate through the VE in total comfort.

Figure 1.1: Test of the PEDAL experience performed by the members of the team

1.4 Goals

One of the main focuses of the PEDAL project is to produce custom made virtual cities that resemble actual urban areas in which the patients live. In this way, each patient can travel their own city using the PEDAL experience and navigate it more easily in real life, recalling the training done using the VR game.

This paper focuses on the work done to replicate real cities in a VE using the available technology, together with an algorithm that allows the developer to easily add other city components (CCs), such as traffic signs, street lights and more, based on their actual location in real life.
The duplicated areas properly depict the streets', buildings' and CCs' positions and dimensions.
Buildings and streets are built using the Blender OSM add-on. Thanks to this plug-in's code and the data provided by the Mapillary platform about other CCs' positions, it was possible to place other CCs according to their actual location.
Beside this major goal, other algorithms were developed to place trees on green areas, depict sidewalks and texture the buildings provided by Blender OSM.
All the work was done using the Blender software, programming in Python and then the VE was exported to Unity where the PEDAL project is developed. For the purpose of testing this algorithm, a replica of the city of Colchester (UK) was created.

Chapter 2

Background

2.1 Related work

Realistic cities are widely used today in several media: they can be found in movies, video games, art performances, serious games, training projects and more.
These scenes can either replicate real places or just be inspired by them.
The process needed to create real cities requires a large amount of geospatial data and the result often poorly represents the actual environments [1].
Moreover, producing large settings with a high level of detail by hand is a long process, which implies great costs for companies and customization is often needed for simulators and serious games in order to achieve the best possible result, which implies extra work.

A lot of research was carried out in order to create these settings in the fastest way possible, using either procedural, image-based or inverse procedural methods. Quite often, these methods create cities inspired by real cities but do not replicate them.
Creating the streets' pattern and the buildings is usually the main focus of these studies.

2.1.1 Procedural modeling

Procedural modeling consists of a series of grammar-based rules that allow the construction of 3D models. It can be used to produce any kind of 3D model; in this case it is mainly used to create cities' layouts or buildings.
All the studies encountered that use this type of approach do not replicate real cities, but create plausible urban areas.

Park et al. [9] proposed a model to create realistic cities in real time and use them in virtual reality experiences for seniors.

Initially, a building is added to the scene, its dimensions are chosen randomly. The streets' pattern develops around this building and it is a grid pattern [10], so as to simplify the algorithm and reduce motion sickness. Next, other buildings are added according to the city structure and trees are added randomly.
The added buildings are obtained using assets provided by Unity and combining them together.

Talton et al. [11] used a probabilistic method to procedurally create trees geometry, buildings and city structures.
In this system, reversible jump Markov chain Monte Carlo is used, a generalization of the Markov chain Monte Carlo model.
The initial input to produce buildings is an original LoD4 [12] building, modeled and textured manually. Meshes can be classified based on their level of detail in a range from 0 to 4, whereas 0 corresponds to an almost absence of details and 4 represents a really detailed object (Figure 2.1).

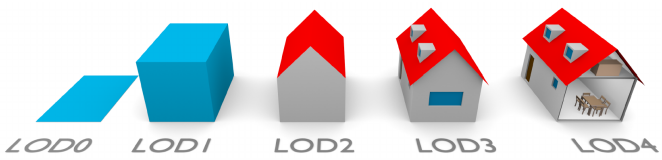


Figure 2.1: LoD geometries by [12]

Using the probabilistic method developed, a series of shrink and split steps are applied to generate other geometries.
In order to create the city pattern, an incremental growth method is used: a series of rectangular buildings are arranged, so as to create defined blocks; the farther from the city center, the smaller the added buildings.
In order to develop trees' geometry, a budget-based algorithm is used, meaning that "each tree starts with a budget of branches that is depleted during growth" [11]. This approach could be an alternative to L-systems to create trees, even though it is not based on biological growth.

Lipp et al.'s [13] procedural method focuses on streets' layout and its ability to adapt to manual changes. The whole process is both procedural and manual to "allow us to hand-craft every aspect of a city" [13]. This method uses layers to easily change the streets' pattern. The researchers developed procedural rules that rearrange the city pattern in case of changes. The scene is then exported to the CityEngine software, where buildings are created and placed.

2.1.2 Image-based modeling

Image-based modeling consists of extracting the city's or the building's features from one or various images. The images used in this process can be satellite views, street-view photos or LiDAR surveys. LiDAR technology uses beams of light to obtain data about the observed objects' dimensions and distances [14].

Shen et al. [15] proposed an image-based model that creates a realistic environment in both streets' structure and buildings. This generator's input consists of a map image that shows the streets' pattern. The algorithm is then capable of building these streets and defining groups that make up blocks. Knowing the blocks' dimensions, the algorithm proceeds to place various buildings with fixed dimensions.
The roads are defined by length, width, center coordinates and rotation angle, which means that it works better with grid patterns.
The placed buildings are textured using real images taken from the CMP (Center for Machine Perception) dataset [16], which consists of more than 600 photos of buildings, processed to obtain usable facade textures. Unfortunately, in these pictures other city components are captured, like street lights and people (Figure 2.2).

Figure 2.2: Textured buildings provided by Shen et al. [15]

Most of the other image-based research focuses mainly on building modeling. Vezhnevets and colleagues' [17] work focuses on buildings viewed from a person's perspective; because of this roofs' geometry is not considered in the process. Differently from the algorithm proposed by Shen et al.[15], the input image consists of a street-view photo. The image is processed in order to fix any tilt and rotation, so as to obtain the building's facade in front of the camera view.
Thanks to the segmentation algorithm, the sky and the actual building are identified, which allows the algorithm to understand the building's height. Thanks to these data it is possible to define a LoD1 building and apply the processed input photo

as texture.
Unfortunately in this case as well, the buildings' textures may depict other objects such as cars, people and more (Figure 2.3). Moreover, the actual construction process needs to be carried out partially manually.

Figure 2.3: Textured buildings provided by Vezhnevets et al. [17]

Tsai et al.[18] proposed another image-based modeling method that focuses on building duplication from actual cities. The process is map based and uses topographic maps, aerial and satellite images.
The algorithm extracts the data to create LoD1 buildings from the topographic maps, without information about the roof geometry. In order to get this missing information, stereo aerial views are used.
As mentioned by the researchers, LiDAR surveys were taken into consideration as input since they provide more information about both buildings' height and structure (including the roof shapes), but they produce discrete point clouds, which are hard to manage.
In order to texture the assets, other images and videos are used, which are specifically processed in order to recognize occlusions and delete them (Figure 2.4).

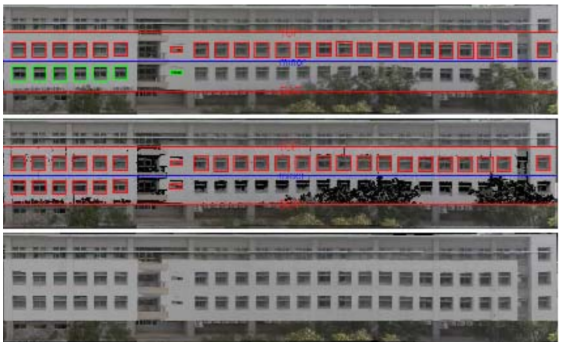


Figure 2.4: Occlusion removal provided by Tsai et al. [18]

Lukas Beer's method's [19] goal is to recreate real buildings respecting their true locations and dimensions. To achieve this, both orthographic images and conditional generative adversarial networks (GANs) are used.
As reported by Salimans et al. themselves, "Generative adversarial networks are a class of methods for learning generative models based on game theory" [20]. GANs' goal is to generate new data from an original given set, trying to replicate the original features.
Despite respecting the actual dimensions and positions, this research produces not textured LoD1 buildings, making the scene not look realistic.

2.1.3 Inverse procedural modeling

Inverse procedural methods consist of extracting features and parameters from an input and create a procedural model to generate such data [21].

Demir Ilke proposed a specific method [22] to recreate real environments from various types of data such as disorganized architectural meshes, building point clouds and textured urban areas. These inputs are then segmented so as to distinguish the various features (e.g. roof, sky, floor, ...). Knowing these groups, it is possible to define a hierarchy and create a procedural model to produce other assets inspired by the original one.

The most complete model encountered was developed by Kim et al.[23]. This study produces a city strongly inspired by a real one, using a single street-view image as input.

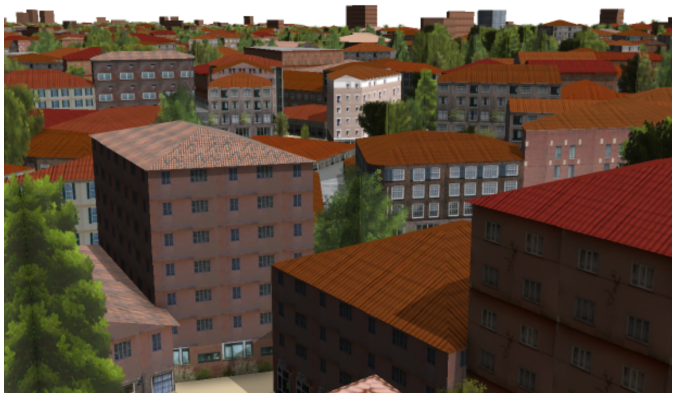


Figure 2.5: Example of output by Kim et al.[23]

Using machine learning, the system understands where to place the city, what components to add and what textures to use. In order to understand the city's appearance, it analyzes the roads' pattern, the buildings' density and their typology.

Using GANs and the data extracted from the image, it creates the ground and a height map. It also uses convolutional neural networks (CNNs) to define the components' spatial information. CNN is a class of neural networks used in deep learning to process visual information [24].
In order to depict the buildings as real as possible, the researchers used the architectonic likelihood concept proposed by Bellotti et al.[25], meaning that there are certain features that, when depicted, allow the viewer to perceive the environment as belonging to a certain geographical area.
According to the analysis done on the input image, it is possible to understand the type of textures and features needed to respect the architectonic likelihood principle.
In this way, a very realistic and distinctive city is produced, yet without being a replica of an actual city (Figure 2.5).

Modeling type	Advantages	Disadvantages
Procedural modeling	Useful to algorithmically model assets and to arrange them. Rules based on different inputs.	Does not replicate cities. No solution for texturing.
Image-based modeling	Useful to replicate real cities. Possibility to use images as textures. Different possible inputs.	Outputs usually lacks definition, in both geometry and textures.
Inverse procedural modeling	Different possible inputs. Plausible environments.	Does not replicate real cities.

Table 2.1: Advantages and disadvantages of the various analyzed modeling types

2.2 Current technology

During the process of building real cities, the present technology developed for this purpose was analyzed.
In this section, the various software and plug-ins that were taken into consideration will be analyzed and each one's positive and negative aspects will be described, so

as to explain why these particular technologies were rejected in the development of the project. In addition, this analysis shows the reader the state of the art of available technology to build real cities.

During this research the focus was on three main aspects to respect in order to obtain a plausible 3D copy of real cities: depict the proper positions and dimensions of streets and buildings, place city components following their actual latitude-longitude coordinates and lastly create buildings depicting real facades. From this research, it was highlighted that there is no way to obtain LoD4 models that recreate the actual buildings, since it would imply recreating every single one by hand as no building is like another.
Due to this, it was chosen to produce buildings as simple LoD1 meshes textured with real facades. The developing team already had experience using the Blender software and the PEDAL project is developed in Unity, therefore, this research focused on these software's add-ons, yet without omitting other software's plug-ins.

The first plug-ins that will be analyzed are developed for Blender.

2.2.1 MapsModelsImporter

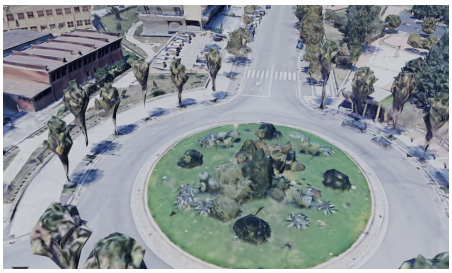
The MapsModelsImporter is an add-on built by Élie Michel with a GPL-3.0 license. This plug-in is based on Google Maps data only. The goal of this add-on is to recreate in Blender the 3D satellite view provided by Google Maps.
In order to use this add-on, it is necessary to use the Google Chrome browser and another software called RenderDoc, a free license debugger that allows single-frame capture.

The process required to use this add-on is really intricate. In the beginning, it is necessary to modify Google Chrome's properties to set it up to be used by RenderDoc. The user has to navigate to Google Maps using the mentioned browser, choose the area that they want to recreate in Blender and proceed to capture that specific frame with RenderDoc. This software then generates a .rdc file that is then imported in Blender, recreating the 3D environment.
The models obtained in Blender are textured and are an exact copy of the 3D satellite view provided by Google Maps.

The add-on displays various problems, both in the results and the ease of the importing process.
Looking at the 3D satellite view in the original Google Maps search, it is already possible to see that the 3D models are not characterized by a clean and defined geometry (Figure 2.6a); due to this, the assets generated in Blender are characterized by a very bad topology and a very low level of definition (Figure 2.7b): many features are not even depicted in a proper way (Figure 2.8b).

Furthermore, the imported models are all joined together, which does not allow

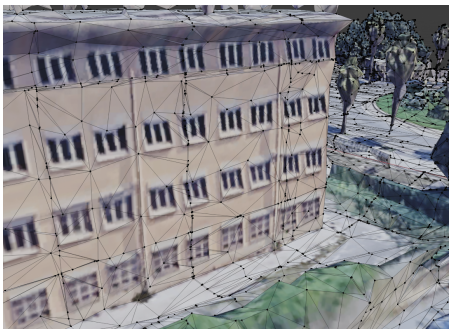
(a) Selected area as seen in Google Maps 3D view



(b) Selected area imported in Blender using the add-on

Figure 2.6: Selected area in both Google Maps 3D view and Blender impoted using the Blender MapsModelsImporter add-on

(a) A building imported using the add-on

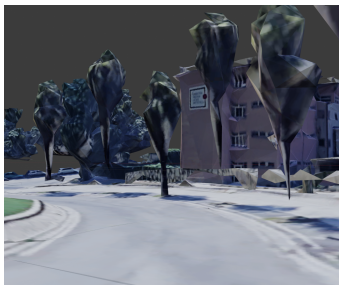


(b) Topology of the building shown in 2.7a

Figure 2.7: Detail of texture and topology of a building created using the Blender MapsModelsImporter add-on

to customize the environment without impacting it all.
In addition to these flaws, using this add-on turned out to be very complex and susceptible to mistakes.
On a positive note, the 3D models generated by the plug-in are already textured, making this one the only add-on among all the ones taken into consideration to recreate the environment with the actual textures and buildings facades. Although, just as the 3D definition, unfortunately, textures as well are little defined (Figure 2.7a). Considering that in the PEDAL experience the user has a close look at the surroundings, this plug-in does not generate a suitable environment.
The general result and the above-mentioned problems are really hard to fix, making this add-on not the ideal technology for this project. Despite this, the MapsModelsImporter plug-in can be a valid option for aerial views.

(a) Selected area as seen in Google Maps Street View



(b) View from 2.8a depicted using the MapsModelsImporter add-on

Figure 2.8: Selected area in both Google Maps Street View and Blender imported using the Blender MapsModelsImporter add-on

2.2.2 Blender GIS

The Blender GIS (Geographical Information System) add-on is developed by several Blender users and has a GPU-3.0 license.
This plug-in is really similar to the one that has been selected for the development of this project (Blender OSM, section 3.3) and presents both advantages and disadvantages compared to that one. This add-on allows users to use data from different providers: Google, Open street maps (OSM), Bing and ESRI.

Compared to the MapsModelsImporter plug-in, the importing method is clearer and does not involve other programs, as the whole process is carried out in Blender. In order to create a 3D scene based on geographical data, the user firstly has to select a provider. Once it is selected, an interactive map shows up in the Blender Viewport, its appearance varies according to the chosen provider. The user can easily search for the place that they want to recreate and then define the specific rectangular area that they want to import.
At this point, the add-on is able to create a plane that portrays the ground and its actual altitude changes; choosing different providers at the beginning of the process reflects on the aspect of the ground that is built. Afterwards, the plug-in uses the data provided by OSM to add 3D buildings, highways, landuse (a generic land area), leisures, natural areas, railways and waterways.

The conversion from latitude-longitude coordinates to Blender coordinates is obtained using the web Mercator projection [26], a version of the Mercator map projection [27] widely used in web mapping applications (described in section 3.3.1). The provided environment is really simple (Figure 2.9): all created assets are planes but the buildings, which are LoD1 replicas of actual buildings.

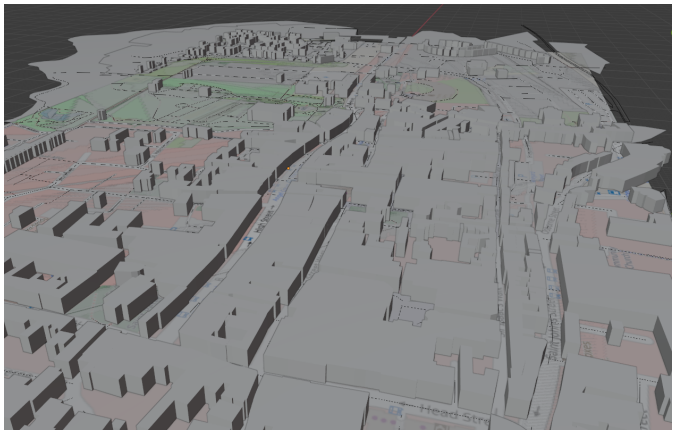


Figure 2.9: The city of Colchester (UK) imported in Blender using the Blender GIS add-on and OSM as provider

Despite this bare scene, the add-on produces a valid foundation as the objects' topology is clean and the provided locations and dimensions are very accurate compared to the real environment.
Nonetheless, this plug-in was rejected due to the lack of documentation that impede using its code for the purpose of this project.

2.2.3 Maps SDK for Unity

The Maps SDK (software development kit) is a plug-in developed for Unity. It is the only valid add-on that was found for Unity.
As for the Blender GIS, an area is selected and a reproduction of that area is created with planes as ground and streets and LoD1 models as buildings. Compared to the previously reported plug-ins, the Maps SDK links the streets assets to their actual names.

The add-on allows developers to use various textures on the buildings, but it does not provide diversity, as the same texture is applied to all the buildings, without discriminating between the ground floor and upper floors. Because of this, even the ground floor of the buildings could be textured with windows, making it not look realistic.

This SDK was discarded because of various reasons besides the above-mentioned one.
Like Blender GIS, the documentation is really little and it would have been necessary to move to a modeling software in order to fill the city with other assets.
On top of these reasons, the Maps SDK was deprecated as of October 2021.

2.2.4 Placemaker

Placemaker is a Sketchup extension. Although not being familiar with this software, it was tested to check its potentiality, but even this one was discarded.
Like the other evaluated plug-ins, the primary outcome consists of a simple environment made of planes and LoD1 meshes.

The assets are created without texture, but it is possible to use Placemaker Tour to apply them directly from Google Street View. Placemaker Tour allows developers to navigate the same latitude-longitude coordinates simultaneously in the produced 3D environment and in Google Street View. Furthermore, it lets them take screenshots of the Google Street View images and place them directly on the 3D buildings faces in the Sketchup software. Consequently, the developer has to take screenshots of the images focusing on the single buildings' facades, which automatically apply on the 3D buildings.
In spite of seeming a good way to obtain textures that properly picture the real ones, the buildings present everything captured in front of the building in the Google Street View images, such as cars, street lights, people and so on.

Moreover, this plug-in does not provide its code, therefore placing assets in the city following its coordinate conversion would have been a really hard obstacle to overcome.

2.2.5 ArcGIS SDK and ArcGIS CityEngine

ArcGIS is a family of various software programs focused on geographic information developed by Esri (Environment Systems Research Institute).
The ArcGIS SDK is available for game engines, such as Unity and Unreal engine.

The outcome produced by this plug-in is really similar to the MapsModelsImporter Blender add-on: the scene replicates the 3D satellite view of the area with textures applied. Because of this, the geometry looks really unclean and the scene is not suitable for the purpose of this project.
Moreover, the SDK does not provide its code, so it would have been really hard to place other assets in the scene without knowing the coordinate conversion algorithm used by the add-on. Along with the ArcGIS SDK, Esri developed the ArcGIS CityEngine which is a software that allows the developers to procedurally create streets and buildings.
It is also possible to recreate real environments from actual geospatial information. This generates a terrain on top of which LoD0 representations of the buildings are placed. It is possible to replace these buildings with others, procedurally or manually modeled in CityEngine.
Although it respects the actual positions and dimensions of the buildings, it is not true to other city components' information, such as trees and traffic signs and neither to real buildings' appearance.

During this research a lot of technology that depicts the proper positions and dimensions of streets and buildings was found. However, among the ones mentioned so far, only the Blender GIS add-on provides the algorithm to carry out the latitude-longitude to Blender coordinates conversion, but it does not provide enough documentation to use its code.
Because one of this study's goals is to add city assets using their actual latitude-longitude positions, this would have been a really hard obstacle to overcome.

Another great obstacle in replicating real urban areas is creating the buildings. As said, this research's goal is to replicate real facades on LoD1 geometries.
The biggest database of building fronts available is, without any doubt, the one made out of Google Street View imagery. Unfortunately, however, Google is really strict with the use of Google Street View photos, authorizing it only in embedded uses ("You may not screenshot Street View imagery or remove it from embedded sources for any purpose").
In addition to this, it is important to point out that, even with a permission to use Google Street View imagery, the photos would need a massive post production process to delete superficial objects captured in the photos.
As seen, this process was carried out by various researchers [16] [17], but none of those studies could delete all superficial captured information.

Chapter 3

Used technologies

The focus of this chapter will be on programs used to replicate real urban areas.

3.1 Blender

Blender is a 3D computer-graphics software developed by the Blender Foundation and released under the GNU General Public License.
It provides many tools that make it a really flexible program. Like other 3D graphics applications, it can be used to carry out all the tasks expected in a 3D graphics pipeline, such as modeling, texturing, rigging, animating, rendering and so on, so as to produce both static and animated scenes.

Using Blender, it is possible to develop add-ons and custom tools using Python language, a versatile open programming language, largely used in various fields.
The Blender Foundation provides a large documentation, that allows users to be able to program and properly use the Blender API to create custom tools. Moreover, it provides its own modules, that can be imported and that give access to Blender's data, classes and functions. These modules give access to the objects' features, so as to process them as needed.

The software has a great community of users, that provides reliable resources and knowledge about the program and all its uses, both artistic and technical.
Moreover, there are various add-ons developed for countless needs, many of which developed by the users themselves.

Since the team's goal was to develop a method that places and orients various CCs, and already had experience using this software, Blender was a perfectly suitable program for this work.

The software provides different views that focus on the task that is carried out, one for each of the major steps expected in the 3D graphics pipeline (e.g. texturing, animating, ...). The main view is the 3D Viewport where the user can create

objects and manage them.
In the 3D Viewport there is a 3D Cartesian coordinate system with the z-axis representing the height. This space's origin corresponds to the (0,0,0) coordinates. Each created model (or mesh) is added to this 3D workspace and has its own position, rotation and scale, expressed referring to the 3D coordinates system.
An object is characterized by a geometry made out of vertices, edges and faces. The user can easily operate on these features in order to modify the object's geometry and shape it as they prefer.

Most of the work was developed in Blender, version 2.92.

3.2 Mapillary

Mapillary is a web-mapping platform based on crowdsourcing. The goal of this platform is to photographically map the entire globe. As mentioned, the service relies on people's contribution, meaning that the photos provided by Mapillary are taken by users from all around the world. The uploaded images have a Creative Commons 4.0 Attribution-ShareAlike (CC-BY-SA) license and can be taken by various devices: smartphones, 360° cameras, dashcams and so on.
Once an user uploads a photo, Mapillary blurs any sensitive captured data, like cars' licenses and people's faces and creates a flow of images (if possible), so as to recreate a view similar to the one provided by Google Street View.

Moreover, thanks to the computer vision technology developed by the Mapillary team, it automatically detects the CCs captured in the uploaded photos. The method is based on semantic segmentation, an algorithm trained to detect and assign a category label to every pixel in the image.
Furthermore, combining at least two images of the same place, it produces a 3D recreation of the photographed places and triangulates the location of the captured objects in the analyzed photos. In this way, it can estimate the items' location in latitude-longitude coordinates.
CCs detected by Mapillary are classified in two categories: points and traffic signs. It is able to detect 42 types of points, which are:

- Temporary Barrier
- Crosswalk - Plain
- Driveway
- Lane Marking - Arrow (Left)
- Lane Marking - Arrow (Right)
- Lane Marking - Arrow (Split Left or Straight)
- Lane Marking - Arrow (Split Right or Straight)
- Lane Marking - Arrow (Straight)
- Lane Marking - Crosswalk
- Lane Marking - Give Way (Row)

- Lane Marking - Give Way (Single)
- Lane Marking - Other
- Lane Marking - Stop Line
- Lane Marking - Symbol (Bicycle)
- Lane Marking - Text
- Banner
- Bench
- Bike Rack
- Catch Basin
- CCTV Camera
- Fire Hydrant
- Junction Box
- Mailbox
- Manhole
- Parking Meter
- Phone Booth
- Signage - Advertisement
- Signage - Information
- Signage - Store
- Street Light
- Pole
- Traffic Sign Frame
- Utility Pole
- Traffic Cone
- Traffic Light - Cyclists
- Traffic Light - General (Horizontal)
- Traffic Light - General (Single)
- Traffic Light - General (Upright)
- Traffic Light - Other
- Traffic Light - Pedestrians
- Trash Can
- Water Valve

Moreover, Mapillary detects more than 1500 different traffic signs, categorized as regulatory, information, warning and complementary. Thanks to these technologies, Mapillary provides reliable and up-to-date information, reporting also temporary objects such as temporary barriers and cones, typical of momentary working sites.

The platform also allows users to download data about CCs' features using a JSON file. For each object, information on *geometry* and *properties* is shown. *Geometry* pieces of data are the object's type (point or traffic sign) and its geographical coordinates; *properties* are the object's ID, its name and a "first seen at" and a "last seen at" value (Figure 3.1).

The information provided by Mapillary was used to obtain a considerable database about CCs' locations.
Despite being based on crowdsourcing and susceptible to people's contribution, the amount of photos provided by users makes Mapillary a solid resource.

```
{
    "geometry": {
        "type": "Point",
        "coordinates": [
            0.8787667751312256,
            51.90371210093929
        ]
    },
    "type": "Feature",
    "properties": {
        "first_seen_at": 1453547086445,
        "id": 315216086910456,
        "last_seen_at": 1453547086445,
        "value": "object--street-light"
    }
},
```


Figure 3.1: Example of point item's information in the JSON file

3.3 Blender OSM

The Blender OSM add-on is developed by Prochitecture. There are two versions of this plug-in: one that relies on donations that does not provide textures, and one at a fixed cost that provides automatic texturing. For this project the first-mentioned version of the add-on was selected, as the texturing algorithm does not discern floor plans from upper plans, producing an unrealistic result for this study's purpose. Despite this, the automatically textured environment can be a great option for aerial views, despite not being true to real facades.
The plug-in is released with a GPL license and both versions give the user access to its code.

As for other analyzed plug-ins, the process of importing data is really straightforward. Once the add-on is installed, a new panel appears in the Blender 3D Viewport, allowing the user to select the area that they want to import. In order to select a surface, it is necessary to state four latitude-longitude coordinates, so as to obtain a rectangular area to import.
In order to select the area, Blender automatically opens a browser tab showing a global view based on OpenStreetMaps data. The user can easily search for the interested area and then select a rectangle section, choosing the zoom level as well. After the user selects this area, the latitude-longitude coordinates are shown and they can easily be pasted to Blender.
Using this information, it is possible to firstly import the ground of the selected area, which is depicted by a plane whose geometry shows the actual altitude changes.

Successively, the user has to import other data such as streets, buildings,

waterways, railways and more. These assets are replicated using mainly planes and LoD1 geometries, and they all depict the real dimensions and positions.
Differently from the Blender GIS add-on, the imported items' names are descriptive, making the objects more easily manageable by code.

Despite producing a very simple environment, the result is a great starting point to create real cities, especially because the developer provides a great documentation, thanks to which it is possible to easily use the code made to perform the latitude-longitude to Blender coordinates conversion.

3.3.1 The transverse Mercator projection

Contrary to the Blender GIS add-on that uses the web Mercator projection, Blender OSM uses the transverse Mercator projection.
They are both adaptations of the original Mercator projection, a conformal cylindrical map projection assembled during the XVI century. It is defined conformal as angles formed on Earth are preserved in the map representation.
As per definition, this is a cylindrical projection in which the cylinder axis has the same direction as the polar axis and the line of tangency corresponds to the equator, therefore meridians and parallels are always perpendicular to each other. Despite being the most used projection, it defects in representing the proper dimensions of lands; in fact the areas further from the equator result bigger compared to the ones closer to it.

The transverse Mercator projection uses a cylindrical projection as well, but in this case the cylinder axis is parallel to the equator and the line of tangency can be any meridian. Because of this, this type of projection can properly represent any point of the globe.

On the other hand, the web Mercator projection is a light variation of the original Mercator projection with adaptation, so as to obtain better results representing the extremes of the globe and large areas.

3.4 Unity

Unity is a game engine that is used to develop the PEDAL project. It presents an intuitive interface that allows the creation of games and VR experiences supported by numerous devices such as mobiles, consoles, desktop and more. Moreover, it permits the development of both 2D and 3D products.

This software supports C#as the main programming language, a very versatile object-oriented language.
The objects in Unity are called GameObjects and each script has to be attached to a GameObject in order to run.

Like Blender, Unity provides a 3D space with a 3D Cartesian coordinate system but, unlike Blender, the height information is linked to the y-axis, not the z-axis. The user can easily add GameObjects in the 3D space and each one of them has its own position, rotation and scale information, expressed in relation to the 3D Cartesian coordinates system. GameObjects can be primitive meshes (e.g. cube, capsule, cylinder, ...) or imported models. Unity supports different 3D models file types (e.g. .fbx and .obj files).
There are various components that can be applied to GameObjects, one of the most important ones being the Mesh Collider option that adds a collider to the mesh.

Once a model is imported, it is possible to make a Prefab out of it, a reusable object made out of a 3D model complete with all its components and features. Prefabs allow the developers to easily apply changes to all the duplicates placed in the 3D space, as an alternation applied to the original Prefab reflects on all of them.

Despite being the principal software used for the PEDAL project, it is not so central for the purpose of this research and it was mainly used to import the VE created in Blender.

Chapter 4

Design and development

4.1 Aims

The purpose of this work is to quickly build 3D VEs that replicate real urban areas, so that the PEDAL users can virtually navigate through the city where they live. To properly depict real cities it is necessary to create assets that properly depict dimensions and locations of real buildings, streets, CCs and trees.
In this way a plausible replica of true environments is obtained.

4.2 Blender OSM initial scene

Firstly a LoD1 replica of the urban area is imported into Blender using the Blender OSM add-on.

4.2.1 Ground

Thanks to this plug-in, it is possible to firstly import into Blender the ground of the selected area, which is depicted by a plane whose geometry shows the actual altitude changes.
For the purpose of the PEDAL project, it is not necessary to depict the altitude changes, therefore the ground is portrayed as a flat and not textured plane.
Successively other assets are imported, such as buildings, streets, green areas, waterways, railways and more. These assets are replicated using simple geometries, and they all depict the real objects' dimensions and positions (Figure 4.1).

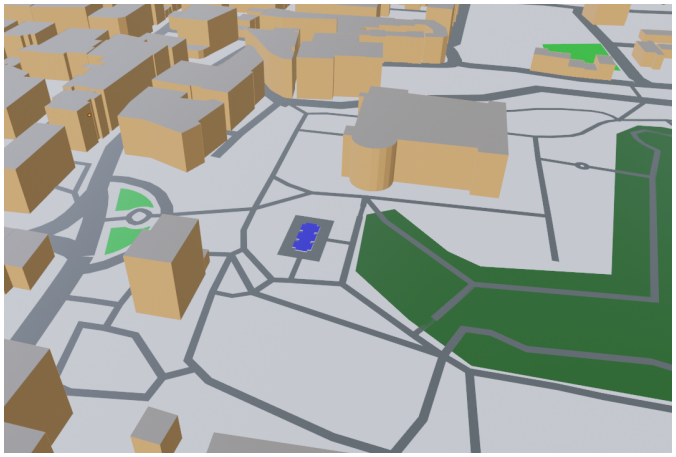
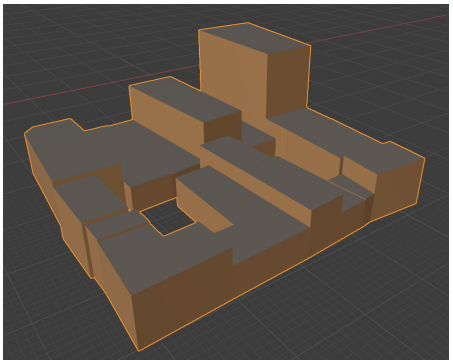


Figure 4.1: Example of initial import provided by Blender OSM

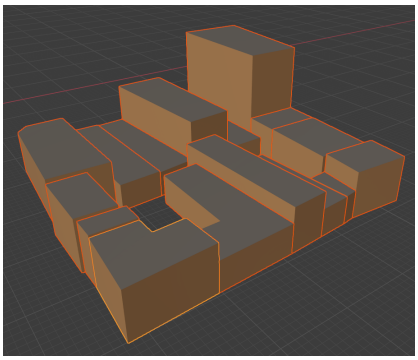
4.2.2 Buildings

Because the actual altitude changes are not depicted, all the buildings are arranged at the same height. Each building is replicated using a LoD1 geometry, it is not textured and properly depicts the real one's dimension and position.

All the buildings are joined together in a single item; the various buildings are not merged together, but they are individual meshes simply joined in the same item (Figure 4.2a). Because of this, it is possible to separate the various structures using the separate tool provided by Blender, specifically applying the "separate by loose parts" command. In this way, a single mesh for each building is obtained, making it more easily manageable (Figure 4.2b).
These meshes are then moved to a designated collection named "Buildings".



(a) Buildings imported as a single item



(b) Buildings divided using the "separate by loose part" tool

Figure 4.2: Buildings initially imported as a single item and then separated

4.2.3 Streets

The meshes that replicate roads are divided in various categories:

- footway
- cycleway
- steps
- pedestrian road
- residential
- service road
- tracks
- primary
- secondary
- tertiary
- unclassified

Each category is made of a set of curves and refers to a profile type that mimics a plane. This object is applied to the corresponding mesh, shaping it into a continuous road.
The various profile types differ in terms of width; in this way the add-on depicts both wide and narrow roads (Figure 4.3).

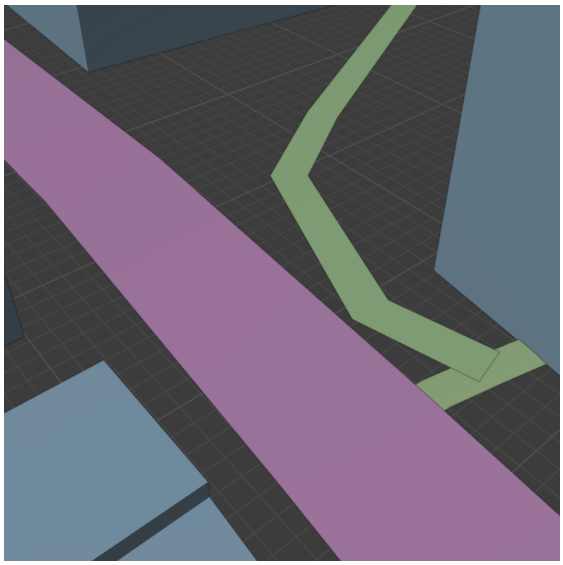


Figure 4.3: Example of tertiary (pink) and footway streets (green)

4.2.4 Other assets provided by Blender OSM

Other assets are mainly landuses, green areas and waterways and they are all depicted using planes that replicate the geometry of the real area (Figure 4.4).
As for the buildings, they are joined in various items according to their category.

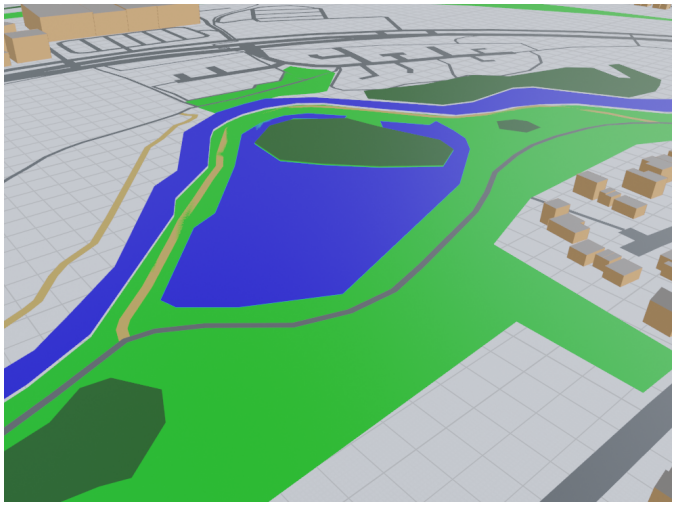


Figure 4.4: Example of waterways and green areas

4.2.5 Conversion from geographical coordinates to Blender coordinates

The Blender OSM add-on provides a TransverseMercator class, whose methods carry out the conversion from latitude-longitude to Blender coordinates and vice versa.
These methods are based on the transverse Mercator projection as explained in section 3.3.1.
Once an area is imported into Blender, the origin's coordinates of the Blender global coordinate system are referred to as *lat* and *long*.
In order to use the methods to carry out the coordinates conversion, it is necessary to create a new instance of the TransverseMercator class, providing the scene's latitude and longitude (above mentioned *lat* and *long*).
After the TransverseMercator class is instanced, it is possible to use the conversion methods and place endless objects in the scene, providing their latitude and longitude coordinates.

4.3 City components' placement

In order to place CCs in the VE, it is necessary to obtain data about the geographical coordinates from Mapillary, which provides this information, as mentioned in section 3.2.
Using the Mapillary platform, it is possible to select the same area that is imported into Blender using the Blender OSM add-on. In this way, the CCs are shown on

the Mapillary map (Figure 4.5) and it is possible to export the data from such selection using a JSON file. The JSON file structure is explained in section 3.2.

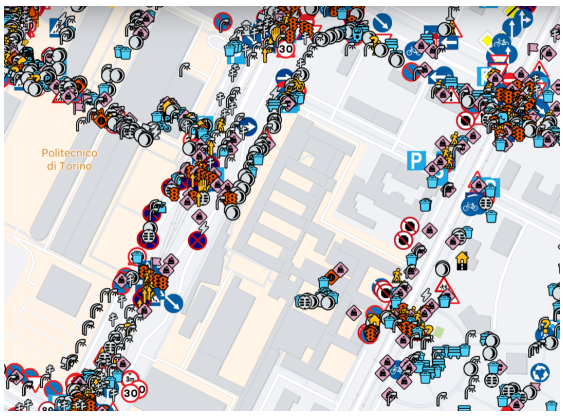


Figure 4.5: CCs detected by Mapillary around the Polytechnic of Turin

Filtering the JSON file by name values, it is possible to determine what objects to model in order to place them in the VE.
For each distinct name, an original mesh (OM) is modeled and named according to the original name value.
Once an OM is modeled for each distinct object, it is possible to duplicate it and place its replicas in the scene.
Because there might be some items hard to portrait (e.g billboards, objects placed at the side of buildings like cameras, generic text signs, ...), a list of the objects' names that do not need to be duplicated is defined.

For each object in the Mapillary JSON file (except the ones in the above-mentioned list), theTransverseMercator's method that converts from geographical coordinates to Blender coordinates is applied. Depending on the item's name, a duplicate of the OM is placed in the scene. In this way, CCs are placed in the VE according to their actual position in real life.
CCs categorized by Mapillary as points are moved to a dedicated collection, while each traffic sign (TS) is sorted into one of four different collections: one for each quadrant of the XY plane.
This algorithm's pseudo-code is described in Algorithm 1.

Placed CCs are simple duplicates of OMs, therefore their rotation is the same for all the placed copies. To reach a more realistic environment, an algorithm to rotate TSs was developed. The goal of the algorithm is to rotate the TSs so as to place them opposite to the direction of traffic.

Algorithm 1 Algorithm that places CCs according to their actual location

```
1: if CC's name not in list of names to avoid then
2:     create new istance of TransverseMercator class
3:     convert from latitude-longitude to Blender coordinates
4:     duplicate corresponding OM
5:     place new object in scene
6:     if CC is a point then
7:         place CC in points' collection
8:     else if CC is a traffic sign then
9:         sort object in collection according to its position
10:    end if
11: end if
```

4.3.1 Streets' set up

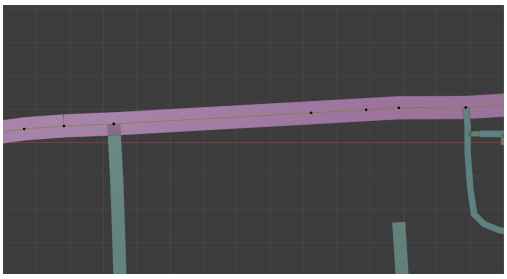
As mentioned, the streets created by the Blender OSM add-on are curves shaped as planes. Because of this, it is not possible to know the direction of traffic beforehand. Therefore, TSs' rotation is determined knowing the side of the road to which the object is closer. In order to do so, it is necessary to set the streets' geometry appropriately.

The curves that make up the street objects are joined in a single mesh according to their category, therefore the street objects are not made up of a continuous distinct curve, but rather from several curves joined in the same object (Figure 4.6).

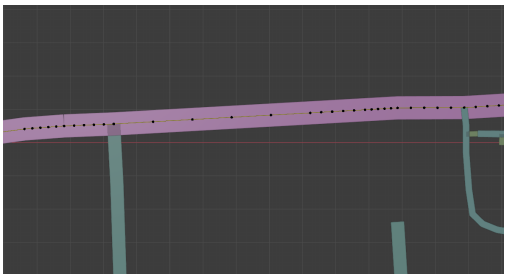
Figure 4.6: Highlighted service road object, made out of different curves

For the purpose of this algorithm, it is necessary to determine the single curve closest to the TS and the vertex closest to the TS.

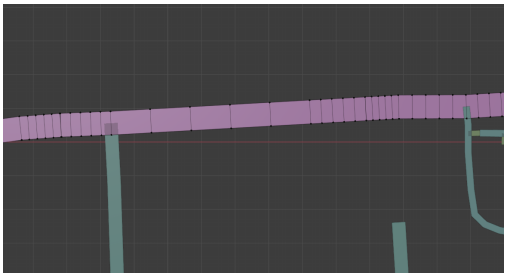
Firstly, the curves that make up the street objects are subdivided multiple times (Figure 4.7b), so as to obtain a greater amount of vertices for a more precise result. Secondly, the curves are converted into meshes; in this way the streets turn from simple paths to planes as shown in figure (Figure 4.7c). Then the "separate by loose parts" tool is applied: in this way, the various pieces of plane that make up each street object are divided into independent street meshes (Figure 4.7d).
In case the street is too long (e.g. its number of faces is greater than 5), it is further divided in smaller pieces for a better result (Figure 4.7e).



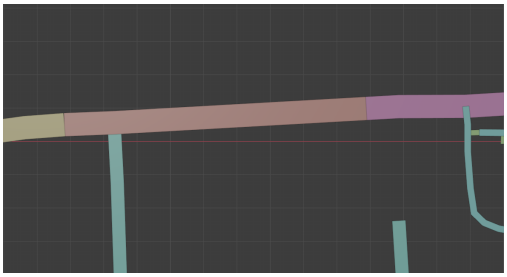
(a) Initial topology of the street



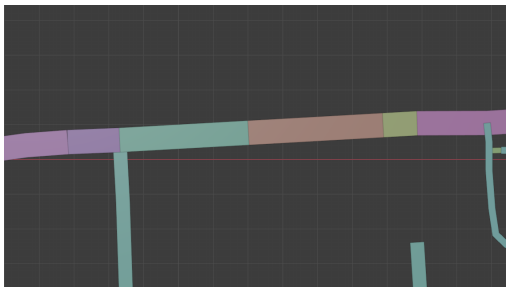
(b) Topology after subdividing the curve



(c) Topology after converting the curve to mesh



(d) Street object divided by "loose parts"



(e) Street meshes further divided because of excessive length

Figure 4.7: Example of iteration on a tertiary road

For each obtained street mesh, the origin is set to the center of its geometry,

otherwise it would be set in the origin of the coordinate system, making the algorithm not functioning.
Each street mesh is then sorted into one of five different collections according to its position: a collection for each quadrant of the XY plane and one for meshes close to the axes. In order to determine if a street mesh is close to the axes or not, a threshold is set up, the greater the threshold, the greater the amount of objects in the axes collection.
The algorithm's pseudo-code is described in Algorithm 2.

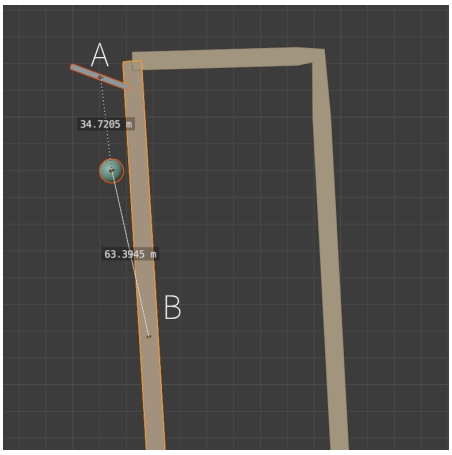
Algorithm 2 Algorithm to properly set up streets in order to rotate TSs

```
for each street object: do
    sudivide curve multiple times
    convert curve to mesh
    divide mesh by "loose parts"
end for
while number of mesh faces > 5 do
    further divide mesh in smaller objects
end while
for each new street mesh: do
    place object's origin to geometry
    if street mesh's position < threshold coordinates then
        sort object in axis collection
    else
        sort object in collection according to the quadrant its origin is located in
    end if
end for
```

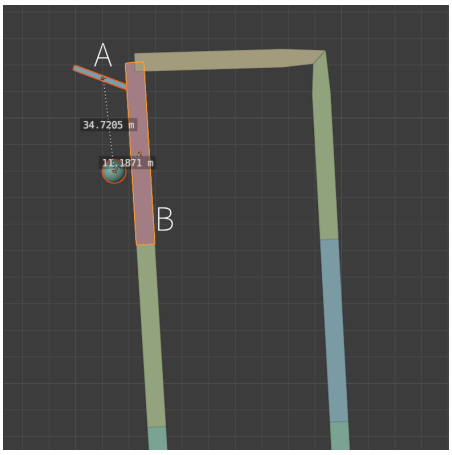
4.3.2 Closest streets and closest vertex search

At this point, it is possible to find the street and the vertex closest to the TS.

In the beginning, the algorithm checks in which quadrant of the XY plane the TS is located, then it proceeds to calculate the distance between the TS's origin and the origin of each street object present in the collection of the same quadrant. Because this search is processed by calculating the distance between origins, smaller street meshes provide better results as their origin actually reflects the object's geometry.
In figure 4.8a it is possible to see how the origin of a bigger street would not be reliable for this evaluation, while it provides a better result when divided into smaller meshes (Figure 4.8b). In the provided example, the algorithm would evaluate the TS closer to street A, while actually being closer to street B.



(a) Wrong placement of long streets' origins



(b) Right placement of long streets' origins

Figure 4.8: Example of placement of street meshes' origins

This iteration is carried out for each street in the collection, then the information about the three closest objects is saved in a list and the vertex closest to the TS is searched among these three items.
To achieve this, the KDTree python module is used which aims to perform spatial searches. For each one of the three closest meshes, the KDTree search is applied to identify the closest vertex among them.
This is another reason why the original street curves are subdivided in the beginning (subsection 4.3.1), as a greater amount of vertices implies a better result in the closest vertex search.

In case the TS's position is lower than a certain threshold, this search is performed on streets in the same quadrant in which the TS belongs and in the street axes collection as well.

The algorithm's pseudo-code is described in Algorithm 3.

Algorithm 3 Algorithm to identify the closest street and vertex

```
1: for each traffic sign: do
2:     check traffic sign's quadrant
3:     call closestMesh(street collection from the same quadrant, traffic sign)
4:     if traffic sign position < threshold then
5:         call closestMesh(axis collection, traffic sign)
6:     end if
7:     sort list of distances in ascending order
8:     limit list of distances to the first 3 closest streets
9:     for each street in list of distances: do
10:        identify street's vertex closest to traffic sign
11:    end for
12:    determine closest vertex among the considered ones
13: end for
14: function closestMesh(street collection, traffic sign)
15:    for each street object in street collection: do
16:        calculate distance between street object and traffic sign
17:        save distance in list of distances
18:        return list of distances
19:    end for
20: end function
```

4.3.3 Traffic signs' orientation

Once the closest vertex is identified, it is possible to determine the TSs' orientation, so as to place them in opposition to the direction of traffic.

The identification of the direction of traffic differs whereas the TS is located closer to a vertex in the middle of the street mesh or that is located at the end of the street mesh.
This distinction is necessary as the street setting provides a great number of street meshes and therefore it is very likely that the TS is close to the end of a mesh, whilst not actually being at the end of an actual road.
In both cases, the search relies on the number of vertices neighboring the closest vertex.

In order to identify these vertices the BMesh module is used. This module provides access to blenders bmesh data structure, in particular geometry connectivity data. For each vertex it is possible to know its index and its global and local position.

Case 1: closest vertex in the middle of the street mesh

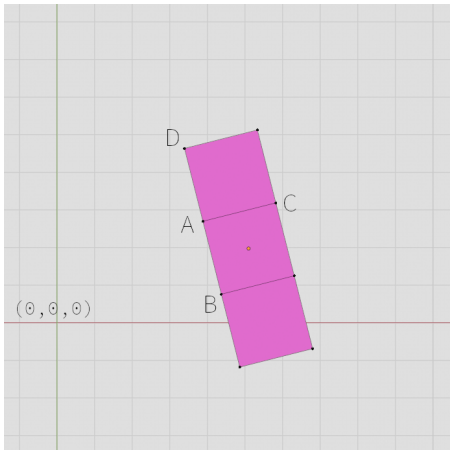


Figure 4.9: Example of case 1

In this case, shown in figure 4.9, the closest vertex, named A, has three neighbor vertices (B, C, D). Vectors $\overrightarrow{AB}$ and $\overrightarrow{AD}$ share the same direction and opposite magnitude, while $\overrightarrow{AC}$ is ideally perpendicular to both $\overrightarrow{AB}$ and $\overrightarrow{AD}$. All the vectors lie in the XY plane.

Starting from the closest vertex identified, the initial goal is to determine $\overrightarrow{AC}$. All the vertices linked to A are identified thanks to the BMesh module and $\overrightarrow{AB}$, $\overrightarrow{AC}$ and $\overrightarrow{AD}$ are calculated. Among these three vectors, only the $\overrightarrow{AC}$ vector forms

an ideally 90° angle with the other vectors.
Consequently, such search is carried out calculating the angles between the various vectors. For each vector, a counter is initialized, if its amount reaches the value of two it means that the vector taken into consideration forms two 90° angles with the other vectors and, therefore it is $\overrightarrow{AC}$.

It is very likely that the angle formed between $\overrightarrow{AC}$ and the other vectors is not precisely 90°, consequently, the search is carried out using a threshold close to such value (e.g. in order to identify $\overrightarrow{AC}$ the two calculated angles need to be both lower than 150°).

The algorithm's pseudo-code is described in Algorithm 4.

Algorithm 4 Algorithm to identify $\overrightarrow{AC}$ in case 1

```
identify vertices neighboring A
calculate AB , AC and AD
for each vector: do
    set counter=0
    calculate angle between vector and other vectors
    if angle = 90° then
        counter = counter + 1
    end if
    if counter = 2 then
        examined vector is AC
    end if
end for
```

Case 2: closest vertex at the end of the street mesh

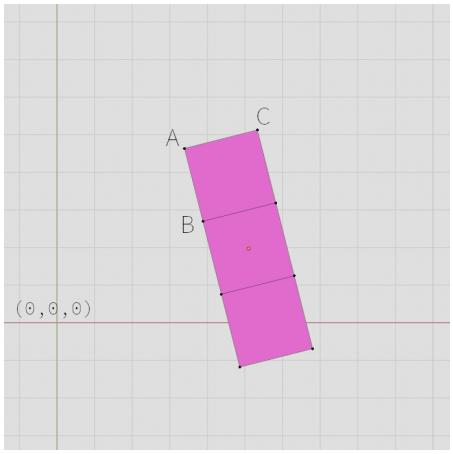


Figure 4.10: Example of case 2

In this case, shown in figure 4.10, the closest vertex, named A, has only two neighbor vertices (B, C). The two vectors $\overrightarrow{AB}$ and $\overrightarrow{AC}$ lie in the XY plane and are ideally perpendicular to each other.
As a result of this, it is not possible to apply the algorithm used in case 1, as there are not enough angles to rely on.
Therefore, the search focuses on neighbor vertices to identify $\overrightarrow{AC}$.
In order to do so, the number of neighboring vertices is calculated for both B and C. The C vertex is the only one that, as A, has only two linked vertices; knowing both A and C, the $\overrightarrow{AC}$ vector is then calculated.

The algorithm's pseudo-code is described in Algorithm 5.

Algorithm 5 Algorithm to identify $\overrightarrow{AC}$ in case 2

```
identify vertices neighboring A
for each vertex neighboring A do
    identify amount of neighbor vertices
    if amount of neighbor vertices = 2 then
        considered vertex is C
    end if
end for
calculate AC
```

At this point, for both case 1 and 2, the cross product between $\overrightarrow{AC}$ and the $\hat{z}$ unit vector is carried out. In case of right-hand traffic, the result of such cross

product is opposite to the direction of traffic and corresponds to $\overrightarrow{AB}$, otherwise, in case of left-hand traffic its opposite has to be calculated and in case 1 corresponds to $\overrightarrow{AD}$.
This value is used to calculate the angle between itself and the $\hat{x}$ axis. Such angle is then applied to the TS's rotation that, therefore, orients itself opposite to the direction of traffic.

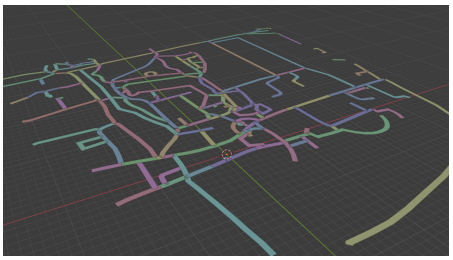
Algorithm 6 Algorithm orient traffic signs considering a right-hand traffic system

1: calculate $\overrightarrow{AC} \wedge \hat{z}$
2: calculate angle between result value and $\hat{x}$
3: set calculated angle as traffic sign's rotation value

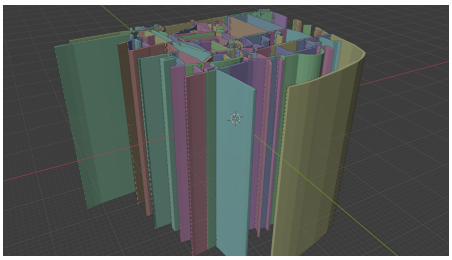
4.4 Sidewalks management

Once the CCs are placed and TSs are appropriately rotated, the focus moves on to the VE in order to improve its appearance.
As imported by Blender OSM, the meshes that depict the streets are mainly flat, producing an unrealistic result. Therefore, 3D sidewalks and roads need to be built.

To do so, the original streets' structure is used: the curves that make up the street objects are converted into meshes and divided using the "separate by loose parts" tool (Figure 4.11a). Differently from the setting used for the CCs placement, it is not necessary to subdivide these meshes. These streets are then extruded, obtaining the result shown in figure 4.11b.



(a) Original street geometries

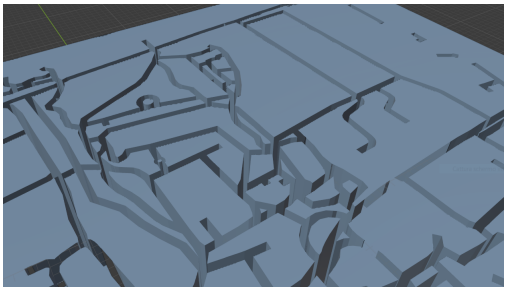


(b) Extruded streets

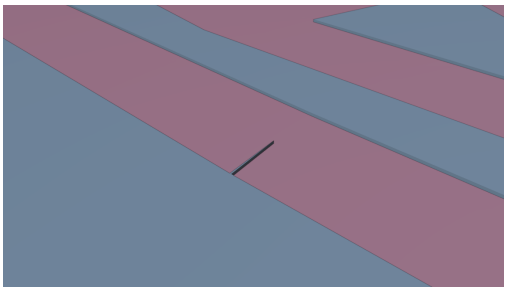
Figure 4.11: Set up of street meshes to depict 3D sidewalks

The ground is then replaced with a cube and, algorithmically, a boolean modifier is applied to the cube using each street mesh as value.
The boolean modifier alters the cube's original geometry, cutting out the wanted mesh from it. In this way, in the original cube various empty spaces are created, where the streets were originally placed (Figure 4.12a).
This cube can therefore represent the sidewalks and all the ground features different from roads (e.g. green areas, parks, ...) using appropriate textures.
Because originally the streets were not a continuous curve, there might be some artifacts to fix manually, as shown in figure 4.12b.
About 10cm lower than the sidewalk level a plane is placed, portaing the road (Figure 4.12c).

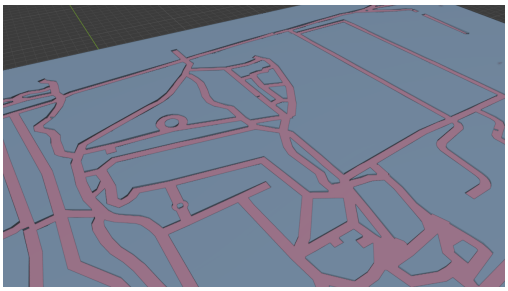
Both these meshes are appropriately textured, so as to obtain the result shown in figure 4.13. In this way, there is a difference in height between sidewalks and roads, making the result more realistic.
In order to depict the green areas that are not imported by the add-on, it is possible to select the cube's involved faces and texture them with suitable textures.



(a) Result of boolean iteration on cube



(b) Example of artifact



(c) Result using a plane to depict the road

Figure 4.12: Sidewalks depicted using a cube (blue) and roads depicted using a plane (pink)

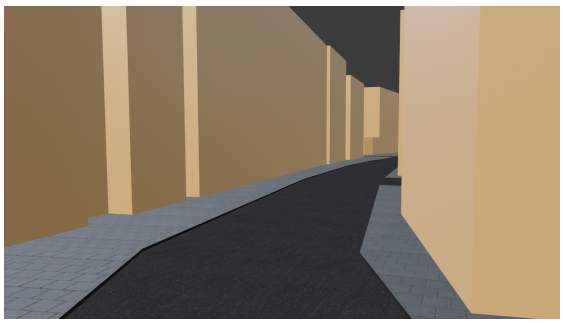
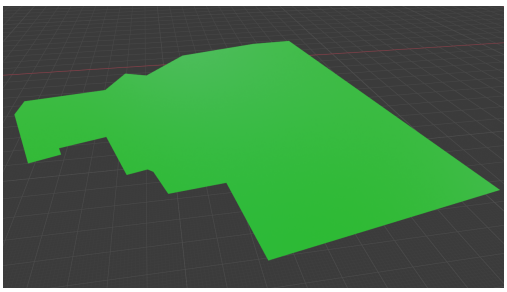


Figure 4.13: Example of textured sidewalks and roads

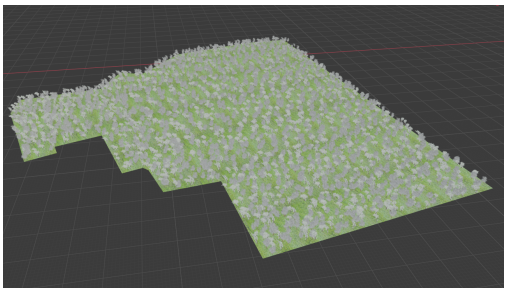
4.5 Green areas management

In the original scene created by Blender OSM, green areas are depicted (Figure 4.14a). In order to manage them, it is necessary to separate the various meshes that make up the vegetation item provided by the add-on.
Once done this, these areas are moved to a designated collection and are filled with green items (GIs) (e.g. trees, bushes and grass). To achieve such a result, an algorithm to manage green areas was developed.
In particular, the algorithm selects the green areas and applies a particle system to each one of them. This is a hair particle system that renders the hair as a collection of meshes, this collection contains previously modeled GIs.
The number of GIs that need to be placed on each green area depends on the extension of its surface.
Assuming the average area needed for a GI, the surface of the green area is divided by such value so as to obtain the number of GIs to place on the area using the particle system.
At this point, a grass texture is applied to the surface and the result shown in figure is obtained (Figures 4.14b and 4.14c).

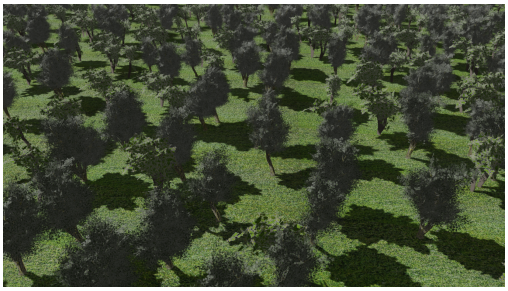
The algorithm's pseudo-code is described in Algorithm 7.



(a) Initial appearance of a green area



(b) Textured green area with particle system applied (4160 GIs placed)



(c) Close-up of the result

Figure 4.14: Output of the algorithm applied to a $0{,}2km^2$ area

Algorithm 7 Algorithm to fill green areas

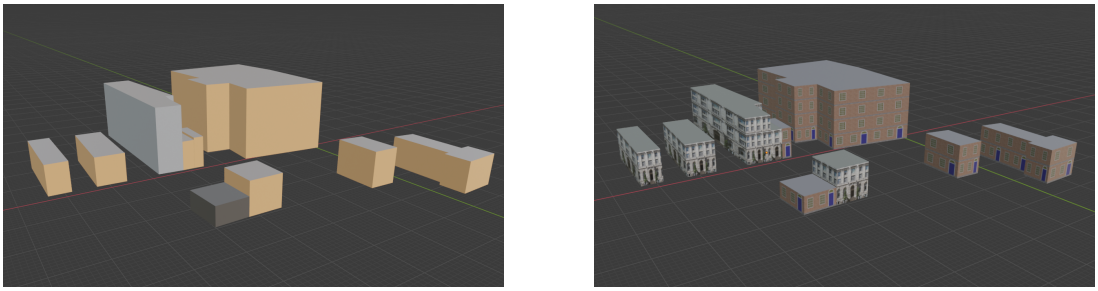
```
1: divide mesh by "loose parts"
2: for each green area obtained: do
3:     calculate area
4:     number of green items = (green area surface/average green item surface)
5:     add hair particle system to the green area
6:     select green items collection as rendered object
7:     set number of green items as number of particles
8: end for
```

4.6 Buildings management

The buildings initially imported by Blender OSM are simple LoD1 geometries (Figure 4.15a), therefore, in order to make them more realistic textures need to be applied.

For each building a series of steps are iterated via code. Firstly, the value of the building height is saved, then the building is flattened on the ground. The "delete by distance" tool is applied to the mesh, obtaining a flat plane. The plane is then extruded according to the original height of the building: assuming that each floor is 3m tall, the original height is divided by this value, obtaining the number of floors needed to depict the building, for each floor the plane is extruded.
For every extrusion, the new piece of geometry is selected and a material is applied to it achieving the result shown in figure 4.15b.
Several textures are created, each set of textures contains a ground floor and one for upper floors. For each building a specific set of textures is chosen randomly and applied accordingly to the floor level.

The algorithm's pseudo-code is described in Algorithm 8.



(a) Initial appearance of buildings

(b) Textured buildings

Figure 4.15: Output of the algorithm, randomly picking from two different sets of materials

Algorithm 8 Algorithm to texture LoD1 buildings

```
set floor height = 3m
for each building: do
    save height information
    flatten building on the ground
    delete extra vertices using the merge "by distance" tool
    calculate number of floors to create = building original height/floor height
    pick random number to select materials set
    set counter = 0
    while counter < number of floors: do
        extrude previous floor for 3m
        select extrusion and apply material according number of floor and material set
        counter = counter + 1
    end while
end for
```

4.7 Export to Unity

To export the VE to Unity, fbx files of the various assets are created.
The ground, made out of the sidewalk item and the road item, is easily exported and uploaded to Unity manually.

To export the CCs, the various OMs are individually exported via code as fbx items and uploaded into the Unity project.
The export process of CCs is based on OMs: a JSON file is created, containing the OMs name and all of its replicas' position and rotation (Figure 4.16). Iterating this new JSON file in Unity, it creates a copy of each OMs and places and rotates it according to the information provided.

The algorithm developed for Unity was written using C#and its pseudo-code is described in Algorithm 9.

To export green areas, each one of them is algorithmically exported as a fbx item and then uploaded into the Unity project.

Algorithm 9 Algorithm to place meshes in Unity

```
for each item in JSON file: do
    check name value
    duplicate fbx according to name
    place and rotate duplicate according to values shown in the JSON file
end for
```

```
{
  "name": "object--street-light",
  "position": {
    "x": -438.5954284667969,
    "y": 35.41669845581055,
    "z": 0.0
  },
  "rotation": {
    "x": 0.0,
    "y": -0.0,
    "z": -1.7221474621843247
  }
},
```


Figure 4.16: Example of object's information in the JSON file

4.8 Colchester

To test the developed algorithms, the British city of Colchester was built.
This city was chosen as the PEDAL project is developed at the University of Essex, based in Colchester and, therefore, tests will take place in this town.
A surface that measures approximately $420m^2$ of the city was replicated (Figure 4.17).

Figure 4.17: Selected area to import as seen in Google Maps

4.8.1 City components

Because the algorithm was tested building a British town, the TSs were rotated according to the right-hand traffic system.
Using the developed algorithm to place CCs, more than 350 items were placed.

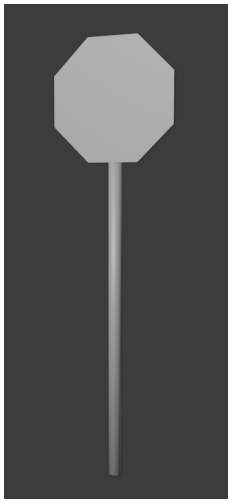
Overall 26 different items categorized as points and 72 distinct traffic signs were modeled.

- Points:
  - construction–barrier–temporary
  - marking–discrete–arrow–left
  - marking–discrete–arrow–right
  - marking–discrete–arrow–split-left-or-straight
  - marking–discrete–arrow–split-right-or-straight
  - marking–discrete–arrow–straight
  - marking–discrete–crosswalk-zebra
  - marking–discrete–give-way-row
  - marking–discrete–give-way-single
  - marking–discrete–stop-line
  - marking–discrete–symbol–bicycle
  - object–bench
  - object–bike-rack
  - object–catch-basin
  - object–junction-box
  - object–mailbox
  - object–manhole
  - object–street-light
  - object–support–utility-pole
  - object–traffic-cone
  - object–traffic-light–general-horizontal
  - object–traffic-light–general-single
  - object–traffic-light–general-upright
  - object–traffic-light–other
  - object–traffic-light–pedestrians
  - object–trash-can
- Traffic signs:
  - complementary–bicycles-and-pedestrians-detour–g1
  - complementary–dead-end–g1
  - complementary–maximum-speed-limit-80–g1
  - information–bicycles-both-ways–g1
  - information–dead-end–g1
  - information–disabled-persons–g1
  - information–minimum-speed-60–g1
  - information–parking–g1
  - information–parking–g5
  - regulatory–bicycles-and-buses-only–g1

- regulatory–buses-only–g1
- regulatory–dual-path-bicycles-and-pedestrians–g1
- regulatory–dual-path-pedestrians-and-bicycles–g1
- regulatory–give-way-to-oncoming-traffic–g1
- regulatory–keep-left–g1
- regulatory–keep-left–g2
- regulatory–keep-right–g1
- regulatory–maximum-speed-limit-10–g1
- regulatory–maximum-speed-limit-20–g1
- regulatory–maximum-speed-limit-30–g1
- regulatory–maximum-speed-limit-40–g1
- regulatory–maximum-speed-limit-5–g1
- regulatory–no-bicycles–g1
- regulatory–no-entry–g1
- regulatory–no-heavy-goods-vehicles–g1
- regulatory–no-left-turn–g1
- regulatory–no-motor-vehicles–g1
- regulatory–no-motor-vehicles-except-motorcycles–g1
- regulatory–no-parking–g1
- regulatory–no-pedestrians–g2
- regulatory–no-right-turn–g1
- regulatory–no-right-turn–g2
- regulatory–no-stopping–g1
- regulatory–no-u-turn–g1
- regulatory–no-u-turn–g3
- regulatory–one-way-left–g1
- regulatory–one-way-left–g2
- regulatory–one-way-left–g3
- regulatory–one-way-right–g1
- regulatory–one-way-straight–g1
- regulatory–priority-over-oncoming-vehicles–g1
- regulatory–priority-road–g1
- regulatory–road-closed-to-vehicles–g1
- regulatory–road-closed-to-vehicles–g3
- regulatory–roundabout–g1
- regulatory–shared-path-bicycles-and-pedestrians–g1
- regulatory–shared-path-pedestrians-and-bicycles–g1
- regulatory–stop–g1
- regulatory–stop–g10
- regulatory–trams-and-buses-only–g1
- regulatory–turn-left–g1
- regulatory–turn-left-ahead–g1
- regulatory–turn-right–g1
- regulatory–yield–g1
- warning–bicycles-crossing–g1
- warning–bicycles-crossing–g4
- warning–children–g1
- warning–crossroads–g1
- warning–curve-left–g1
- warning–curve-right–g1
- warning–junction-with-a-side-road-perpendicular-left–g1

- warning–junction-with-a-side-road-perpendicular-right–g1
- warning–other-danger–g1
- warning–pedestrians-crossing–g1
- warning–pedestrians-crossing–g5
- warning–pedestrians-crossing–g6
- warning–road-narrows–g1
- warning–road-narrows-left–g1
- warning–road-narrows-right–g1
- warning–roundabout–g1
- warning–traffic-merges-left–g2
- warning–two-way-traffic–g1

In order to model the OMs, a thorough analysis of actual CCs' appearance was carried out, so as to obtain models as true as possible to the real ones.
To achieve this, a series of references from governamental and reliable sites were taken into consideration (e.g gov.uk). Thanks to these references, it was possible to create OMs whose dimensions are true to real CCs and to use appropriate texture, so as to create items that properly mimic the British ones. In the figures in 4.20, it is possible to see various modeled CCs and the references taken into consideration.



(a) Not textured traffic sign

(b) Textured traffic sign

Figure 4.18: Example of textured and not textured asset

In order to model the OMs, it was necessary to go through modeling and texturing. Starting from a primitive shape (mainly cubes and cylinders) and using the tools provided by Blender, the object's geometry was modified operating on its vertices, edges and faces.
Once the mesh properly represented the wanted item, a material was added to the object, which carries out information about the behavior of the surface in relation

to light input (e.g. whether it reflects the light or not).
Then it was possible to UV map the model, meaning that the 3D model's geometry was projected to a 2D space. Thanks to this step, it was possible to texture the object, where the created UV map reflected the placing of the texture on the 3D model (Figure 4.18). The texture information, combined with the material data, depicts a pattern or an image that also responds to the light behavior.

4.8.2 Sidewalks and roads

Differently from the street setting seen in section 4.4, in this case the streets' width was fixed for all roads, so as to let the user navigate the city more easily avoiding narrow streets.
To obtain the same width for all the streets, the curves initially imported using the add-on were shaped as the same object that mimics a wide street.
The algorithm described in section 4.4 was applied and appropriate textures were used to depict sidewalks (Figure 4.19b) and roads (Figure 4.19a).
In addition to the asphalt material, another one was used on the ground object to depict pedestrian roads (Figure 4.19c).

(a) Asphalt material

(b) Sidewalks material

(c) Pedestrian road material

Figure 4.19: Materials used on sidewalks and roads

(a) Trash can reference

(b) Modeled trash can

(c) Traffic light reference

(d) Modeled traffic light

(e) Construction barrier reference

(f) Modeled construction barrier

(g) Modeled cone

Figure 4.20: Modeled assets and their references (as seen in Google Street View and Mapillary)

4.8.3 Manual changes

In order to create a more immersive environment, other assets were manually added to the scene.
In particular, extra CCs were added, especially: street lights, missing traffic lights at intersections, bike racks, trash cans and benches.

Moreover, buildings provided by Blender OSM were replaced with LoD3 meshes. A set of more than 13 different buildings was placed in the urban area (Figure 4.21). Because these buildings are copies of the same original meshes, the algorithm to export them into Unity was the same used to export CCs.

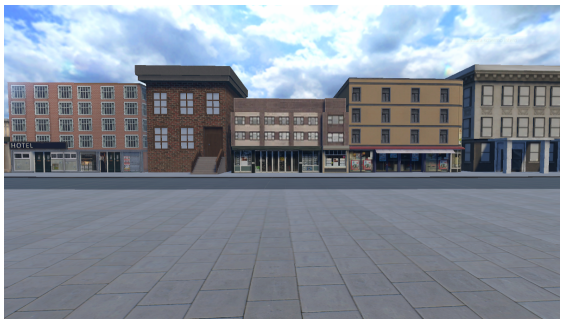


Figure 4.21: Some of the buildings used to replace the LoD1 ones produced by Blender OSM

During the PEDAL experience the patient has to reach certain points in the VE, therefore it was necessary to model precise landmarks. These models also help the patient to orient themselves in the city.
Thanks to the PEDAL team, a list of landmarks was defined and these important places were modeled.
The modeled landmarks were: Colchester castle, Culver square, Trinity church, Colchester library, Mark & Spencer store in High street, Odeon movie theater.
Store landmarks were depicted adding the store sign to the buildings (Figure 4.22), while other distinctive buildings were modeled from scratch (Figure 4.23).

(a) Mark & Spencer store reference

(b) Modeled Mark & Spencer store

(c) Odeon movie theater reference

(d) Modeled Odeon movie theater

(e) Culver square reference (licensed image taken from the visit Colchester website)

(f) Modeled Culver square

Figure 4.22: Store landmarks and their references (as seen in Google Street View and in the visit Colchester website)

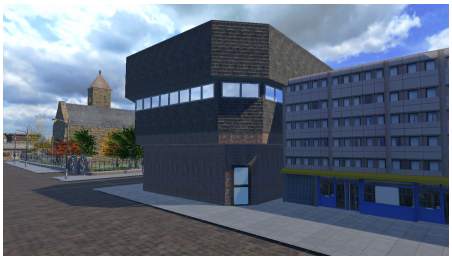
(a) Colchester castle reference

(b) Modeled Colchester castle

(c) Trinity church reference

(d) Modeled Trinity church

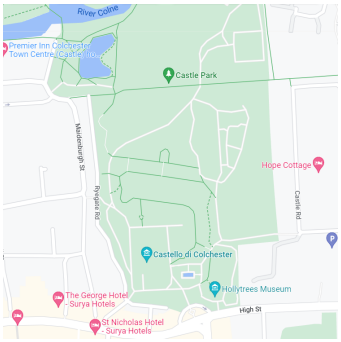
(e) Colchester library reference



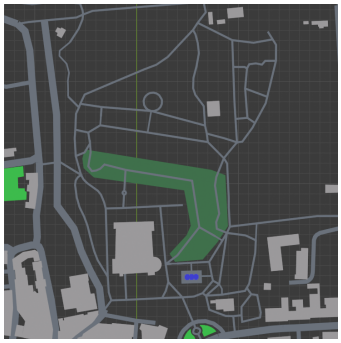
(f) Modeled Colchester library

Figure 4.23: Modeled landmarks and their references (as seen in Google Street View)

As it is possible to see in figure 4.24, the green area detected by Blender OSM did not properly depict the real size of the Colchester castle park. Because of this, it was replaced with another plane to which the algorithm described in section 4.5 was applied.
In addition to this, the Trinity church park was added, as that green area was not detected by the add-on.



(a) Castle park's green area as seen in Google Maps



(b) Castle park's green area as imported by Blender OSM

(c) Castle park's green area manually replicated

Figure 4.24: Castle park's green area

Fences and gates were added to outline these parks. An initial fence object was modeled and the perimeters of the parks were converted into curves. To the fence object, array and curve modifiers were applied, using the perimeter curves as values for the curve modifier. In this way, a continuous fence was added around each park. These objects were then exported to Unity and placed in the scene.

Furthermore, other smaller sets of trees were added to the scene directly in Unity.
Thanks to the terrain GameObject provided by Unity, it was possible to quickly add smaller green areas and fill them with trees. In this way a more diverse environment was obtained.

To apply materials to all the assets, CC0 textures by ambientCG were used.
Once in Unity, the mesh collider component was added to all the assets.

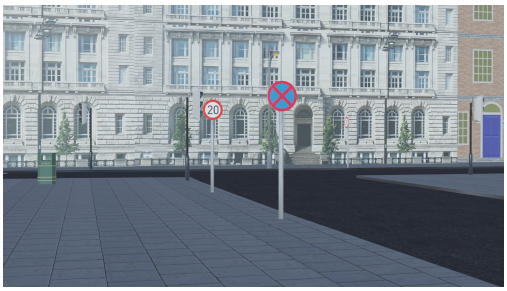
Chapter 5

Comparison between output and real environment

To evaluate the output of this work, a comparison between real photos and the VE was made.
In order to evaluate the similarity between virtual and real environments, the comparison focuses on street level point of view.
The images from real places are taken from Google Maps Street View. The VE is considered both before and after manual changes.
The initial outputs are captured in Blender using an HDRI image as source of light.
The final outputs are captured in Unity.

As it is possible to see in comparison 1 (Figure 5.1), the geometry of the ground was not imported properly by Blender OSM, but, replacing LoD1 buildings with LoD3 ones, the scene looks more realistic.
Moreover, this comparison highlights how not every CCs is detected by Mapillary and it is necessary to add more assets manually. In fact, in this comparison extra traffic lights were added as not every one of them was captured by Mapillary.
In figure 5.2 it is possible to see a photo uploaded to Mapillary from which the traffic light in the foreground was detected. In this same photo, another traffic light is seen (on the left), but Mapillary was not able to detect it. Moreover, from this same figure it is possible to see two TSs (speed limit and restricted parking zone) that were detected by Mapillary. Their location in latitude-longitude was not precise and, therefore, the placed assets are not precisely where they are supposed to be.

(a) Real place (image taken from Google Street View)



(b) Initial output

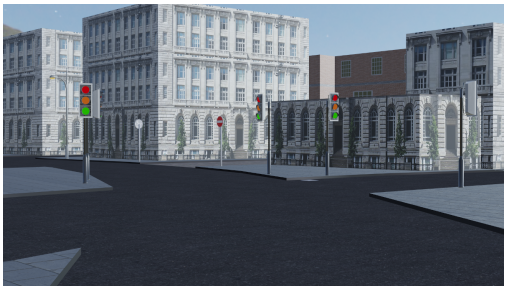
(c) Output with manual changes

Figure 5.1: Comparison 1

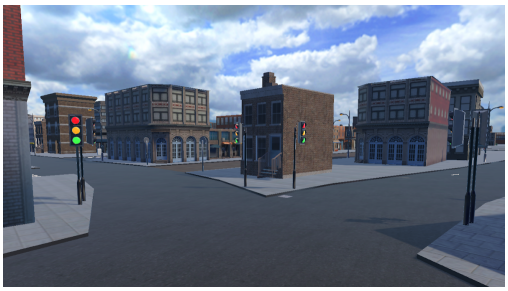
Figure 5.2: Example of Mapillary photo

On the other hand, in comparison 2 (Figure 5.3), all the traffic lights are detected by Mapillary and therefore there was no need to place extra assets.
In this case as well, replacing the LoD1 buildings with LoD3 ones makes the scene more similar to the real one.
In addition to this, in this case the street's geometry was properly imported by Blender OSM.

(a) Real place (image taken from Google Street View)



(b) Initial output



(c) Output with manual changes

Figure 5.3: Comparison 2

Comparison 3 (Figure 5.4) takes place in front of the castle park. In this area many CCs were detected by Mapillary. Despite this, the location in real life presents other assets that are not detected by Blender OSM and neither by Mapillary, like the statue, distinctive lamps and fences.
In this scene there are two green areas detected by Blender OSM: one is the castle park itself and one is the flowerbed outside the park. In this case, the algorithm that adds trees to green areas (described in section 4.5), was applied to the castle park green area but not to the flowerbed as it only needed grass. In order to depict the grass, a proper material was used on this area (Figure 5.5).
In the final result, it is possible to see how adding the fence that delimits the park makes the scene more realistic.
In this case, to create a complete replica, the statue needed to be modeled manually.

(a) Real place (image taken from Google Street View)

(b) Initial output

(c) Output with manual changes

Figure 5.4: Comparison 3

Figure 5.5: Grass material

Comparison 4 takes place near the library and the Trinity church. In this case all the present CCs were captured by Mapillary and placed accordingly.
In this case, it is possible to see how the ground was textured differently from other areas as this is a pedestrian area and, therefore, the pedestrian road material was used (Figure 4.19c).
Because the street width was fixed to allow easier navigation, the final result appears wider than the original one.

Moreover, this scene highlights how, even from the street point of view, it is important to represent roofs, as they are really characteristic and may recall the real location more easily.

(a) Real place (image taken from Google Street View)

(b) Initial output

(c) Output with manual changes

Figure 5.6: Comparison 4

In addition to these comparisons, the overall urban area resembles the real one, also thanks to the modeled landmarks. These were modeled inspired by the real ones and, therefore, they are effectively represented. The comparison figures between references and modeled assets can be found in figure 4.23.

Overall, it is possible to notice some trends in the output provided by this method.
Generally, manual fixes are required for various reasons. Moreover, data taken from Mapillary does not always comprehend all the present CCs and, therefore, extra ones need to be added.
In fact, in the beginning of building Colchester, there were some areas where no street lights were captured, while others presented many street lights. Adding extra assets solved this inconsistency and made the overall result more uniform. Moreover, replacing LoD1 buildings with LoD3 ones makes the scene more realistic and immersive.

Chapter 6

Conclusion and future developments

This study proposed a method to algorithmically replicate real urban areas as virtual environments thanks to the Blender OSM add-on and the Mapillary platform. This result was developed for the PEDAL project, which requires the user's real city to be used as virtual environment.
This method mainly focused on the placement of city components in the city, such as traffic signs, benches, traffic lights, street lights and more. In addition to this, other algorithms were developed to place trees on green areas and to texture the buildings.
It was possible to create a plausible replica of the city of Colchester using these algorithms and applying manual changes to depict specific landmarks and to make the environment more visually appealing.

Blender OSM provided a good initial input to properly portray the geometry and dimensions of buildings, streets and other assets as waterways and large green areas. In addition to this, a good documentation of the add-on is available online and, therefore, it was possible to use the method of the add-on to convert from latitude-longitude coordinates to Blender coordinates, based on the transverse Mercator projection.
Yet, the add-on was not able to portray small green areas and, building the city of Colchester, some parts of the street structure were not imported properly. Despite these flaws, the general appearance of the city was not affected and the result effectively depicted the real city.

Thanks to Mapillary, data about CCs and their location was obtained. Combining this data with the code provided by Blender OSM, it was possible to place them in the scene. In particular, in the replica of Colchester, more than 350 assets

were placed, according to their location in real life.
Although, as seen, Mapillary is not capable of detecting all the CCs captured in the photos provided by its users and, therefore, extra objects might need to be added. Yet, since Mapillary is based on crowdsourcing it is possible to obtain a more precise mapping of items contributing directly to the project.

Creating Colchester, the output provided by this method needed some manual changes to create a more pleasant and realistic environment. In fact, LoD1 buildings were replaced with LoD3 ones to create a more immersive and realistic scene, small green areas were manually added as well as some specific landmarks, which were useful for the purpose of the PEDAL project and also help the user to orient themselves more easily. In addition to this, extra CCs were placed as well as fences around parks.

As these changes were needed to create a better environment, in the future it might be necessary to develop methods to apply these changes algorithmically.
As seen, LoD3 buildings create a more immersive and realistic environment. It would be ideal to develop a method to replace the LoD1 buildings with appropriate LoD3 ones.
Knowing the dimension and location of the LoD1 buildings, they could be replaced using a set of LoD3 buildings chosen according to the LoD1 dimension. In this way, the position and dimension of original buildings would be respected, but the overall scene would look more appealing.

Adding extra CCs algorithmically would be more challenging, as they would be placed procedurally and, therefore, would not be true to the real urban area. Despite this, there are some assumptions that can be made.
For instance, as seen, Mapillary might not capture all the traffic lights at intersections, but if at least one is detected, it is possible to suppose that more traffic lights are placed at the same intersection, which can be added procedurally.
Moreover, at intersections it is possible to suppose that pedestrian crossings need to be placed, which could be done procedurally once intersections are identified.

Moreover, markings on the street could be placed, so as to define roads more clearly. To do so, it would be necessary to follow the original street structure and replace it with textured planes. This might be hard to do as the original streets imported by Blender OSM are made out of various curves and, therefore, the planes that would make up the streets would not be continuous.

In the future, it would be useful to automatically add fences along parks and green areas so as to delimit these areas. This can easily be achieved with the green areas imported by Blender OSM, but it would be necessary to also be done around small green areas, which now are manually created directly in Unity using terrain GameObjects.

In addition to automating these processes, in order to create a more realistic environment, cars and pedestrians will be added to the scene. The logic behind their behavior was developed by the members of the PEDAL team and was also tested on test scenes and will be now applied to the Colchester environment.

Moreover, the PEDAL team developed the AI system to calculate the path that the patient has to follow to reach specific points in the scene, which will now be applied to the Colchester scene. The AI system will determine the route to reach the points of interest that were modeled. The goal of each route will be highlighted in green to further help the user to identify them in the scene.

It will be possible to navigate the environment using a custom made stationary bike, which will allow the patient to move around the city. This will be equipped with a smart trainer and, together with the VR headset, will create a complete immersive experience.

In addition to this, during their navigation, the user will be guided by a virtual assistant developed by the members of the PEDAL team. This virtual assistant, which is represented by a robot, will communicate with the patient, so as to tell them if they're following the right direction or not, according to the path identified by the AI system.
Because it might be hard for elderly patients to process so many digital inputs, the virtual assistant will communicate using a green circle that will appear if the patient is following the right path, or, in case they're following a wrong path, a red no entry sign will appear and the virtual assistant will point in the right direction. In addition to this, in case the user is not moving in any direction, the assistant will ask them if they need help.

In the future, the virtual assistant will be controlled by a supervisor, so as to watch over the patient and obtain the best results. In this way it will be possible to monitor more accurately the behaviour of the patient and their response to this experience. To achieve this result a continuous connection with a remote server will be used to create a multiplayer system.

Thanks to these improvements it will be possible to fully experience PEDAL.

Bibliography

[1] Tarja Susi, Mikael Johannesson, and Per Backlund. «Serious Games : An Overview». In: 2007 (cit. on pp. 1, 5).

[2] Perla Werner, Sarit Rabinowitz, Evelyne Klinger, Amos Korczyn, and Naomi Josman. «Use of the Virtual Action Planning Supermarket for the Diagnosis of Mild Cognitive Impairment». In: *Dementia and geriatric cognitive disorders* 27 (Apr. 2009), pp. 301–9. DOI: 10.1159/000204915 (cit. on p. 2).

[3] David Howett et al. «Differentiation of mild cognitive impairment using an entorhinal cortex-based test of virtual reality navigation». In: *Brain* 142 (2019), pp. 1751–1766 (cit. on p. 2).

[4] Marshal F. Folstein, Susan E B Folstein, and Paul R. McHugh. «"Mini-mental state". A practical method for grading the cognitive state of patients for the clinician.» In: *Journal of psychiatric research* 12 3 (1975), pp. 189–98 (cit. on p. 2).

[5] John C. Morris. «The Clinical Dementia Rating (CDR)». In: *Neurology* 43.11 (1993), 2412–2412-a. ISSN: 0028-3878. DOI: 10.1212/WNL.43.11.2412-a. eprint: https://n.neurology.org/content/43/11/2412.2.full.pdf. URL: https://n.neurology.org/content/43/11/2412.2 (cit. on p. 2).

[6] Paul White and Zahra Moussavi. «Neurocognitive Treatment for a Patient with Alzheimer's Disease Using a Virtual Reality Navigational Environment». In: *Journal of Experimental Neuroscience* 10 (Nov. 2016), p. 129. DOI: 10.4137/JEN.S40827 (cit. on p. 2).

[7] Larrue Florian, Hélène Sauzéon, Grégory Wallet, Déborah Foloppe, Jean-René Cazalets, Christian Gross, and N'Kaoua Bernard. «Influence of body-centered information on the transfer of spatial learning from a virtual to a real environment». In: *European Journal of Cognitive Psychology* 26 (Oct. 2014). DOI: 10.1080/20445911.2014.965714 (cit. on pp. 2, 3).

[8] Grégory Wallet, Hélène Sauzéon, Prashan Pala, Larrue Florian, Xia Zheng, and N'Kaoua Bernard. «Virtual/Real Transfer of Spatial Knowledge: Benefit from Visual Fidelity Provided in a Virtual Environment and Impact of Active Navigation». In: *Cyberpsychology, Behavior, and Social Networking* 14 (Feb. 2011), pp. 417–423. DOI: 10.1089/cyber.2009.0187 (cit. on p. 3).

[9] Jung Woon Park and Seok Hee Oh. «A Study on Creation and Usability of Real Time City Generator via Procedural Content Generation: – Focus on virtual reality contents for senior». In: *2019 International Symposium on Multimedia and Communication Technology (ISMAC)*. 2019, pp. 1–4. DOI: 10.1109/ISMAC.2019.8836162 (cit. on p. 5).

[10] Kurt Bauer. *City Planning for Civil Engineers, Environmental Engineers, and Surveyors*. Sept. 2009, pp. 1–495. ISBN: 9781439808931. DOI: 10.1201/b15801 (cit. on p. 6).

[11] Jerry Talton, Yu Lou, Jared Duke, Steve Lesser, Radomir Mech, and Vladlen Koltun. «Metropolis Procedural Modeling». In: *ACM Transactions on Graphics* 30 (Apr. 2011). DOI: 10.1145/1944846.1944851 (cit. on p. 6).

[12] Filip Biljecki, Hugo Ledoux, and Jantien Stoter. «An improved LOD specification for 3D building models». In: *Computers Environment and Urban Systems* 59 (Sept. 2016), pp. 25–37. DOI: 10.1016/j.compenvurbsys.2016.04.005 (cit. on p. 6).

[13] Markus Lipp, Daniel Scherzer, Peter Wonka, and Michael Wimmer. «Interactive Modeling of City Layouts using Layers of Procedural Content». In: *Comput. Graph. Forum* 30 (Apr. 2011), pp. 345–354. DOI: 10.1111/j.1467-8659.2011.01865.x (cit. on p. 6).

[14] National Oceanic and Atmospheric Administration (NOAA) Coastal Services Center. «Lidar 101: An Introduction to Lidar Technology, Data, and Applications.» In: (2012). Revised. Charleston, SC: NOAA Coastal Services Center. (cit. on p. 7).

[15] Bingyu Shen, Boyang Li, and Walter J. Scheirer. «Automatic Virtual 3D City Generation for Synthetic Data Collection». In: *2021 IEEE Winter Conference on Applications of Computer Vision Workshops (WACVW)*. 2021, pp. 161–170. DOI: 10.1109/WACVW52041.2021.00022 (cit. on p. 7).

[16] Radim Tyleček and Radim Šára. «Spatial Pattern Templates for Recognition of Objects with Regular Structure». In: vol. 8142. Sept. 2013. ISBN: 978-3-642-40601-0. DOI: 10.1007/978-3-642-40602-7_39 (cit. on pp. 7, 16).

[17] Vladimir Vezhnevets, Anton Konushin, and A. Ignatenko. «Interactive image-based urban modelling». In: (Jan. 2007) (cit. on pp. 7, 8, 16).

[18] Fuan Tsai, Tea-Ann Teo, L. Chen, and Szu-Jen Chen. «Construction and visualization of photo-realistic three-dimensional digital city». In: June 2009, pp. 1–7. DOI: 10.1109/URS.2009.5137674 (cit. on p. 8).

[19] L. De Beer. «Automatic Generation Of LoD1 City Models And Building Segmentation From Single Aerial Orthographic Images Using Conditional Generative Adversarial Networks». In: *GI_Forum* (2019) (cit. on p. 9).

[20] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. «Improved Techniques for Training GANs». In: (June 2016) (cit. on p. 9).

[21] Daniel Aliaga, Ilke Demir, Bedrich Benes, and Michael Wand. «Inverse procedural modeling of 3D models for virtual worlds». In: July 2016, pp. 1–316. DOI: 10.1145/2897826.2927323 (cit. on p. 9).

[22] Ilke Demir. «Inverse Procedural Modeling for 3D Urban Models». In: Oct. 2017 (cit. on p. 9).

[23] Suzi Kim, Dodam Kim, and Sunghee Choi. «CityCraft: 3D virtual city creation from a single image». In: *The Visual Computer* 36 (May 2020). DOI: 10.1007/s00371-019-01701-x (cit. on p. 9).

[24] Maria Valueva, Nikolay Nagornov, Pavel Lyakhov, G.V. Valuev, and N.I. Chervyakov. «Application of the residue number system to reduce hardware costs of the convolutional neural network implementation». In: *Mathematics and Computers in Simulation* 177 (May 2020). DOI: 10.1016/j.matcom.2020.04.031 (cit. on p. 10).

[25] Francesco Bellotti, Riccardo Berta, Rosario Cardona, and Alessandro De Gloria. «An architectural approach to efficient 3D urban modeling». In: *Computers Graphics* 35 (Oct. 2011), pp. 1001–1012. DOI: 10.1016/j.cag.2011.07.004 (cit. on p. 10).

[26] National Geospatial-Intelligence Agency (NGA). «Web Mercator Map Projection». In: (Feb. 2014) (cit. on p. 13).

[27] Peter Osborne. *The Mercator Projections*. Jan. 2013. DOI: 10.5281/zenodo.35392 (cit. on p. 13).