POLITECNICO DI MILANO

Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

Generazione Automatica dei Contenuti per Mondi Virtuali 3D

Relatore: Prof. PIER LUCA LANZI

Autore:
CRISTIANO PEDERSINI
Matricola: 783210

Anno Accademico 2013 - 2014

This dissertation is dedicated to my parents, who always supported me and encouraged any of my choices.

Dedico questa tesi ai miei genitori, che mi hanno sempre sostenuto ed hanno sempre incoraggiato ogni mia decisione.

Acknowledgements

My heartfelt thanks to my friend and supervisor Pier Luca Lanzi, for his support and guidance during these amazing years.
I must acknowledge my family, Guendalina and my friends Daniele, Simone and Ivano, who always blindly believed in me, too often suffering my bad temper and lack of sleep.
A very special thanks also goes out to my colleagues Eurico Doirado, Edern Gray and Satoko Ohtsuki, who helped me to get through this project and supported me during this last year of deep change.

Un ringraziamento di cuore al mio supervisore ed amico Pier Luca Lanzi, per il suo supporto e la sua guida durante questi anni eccezionali.
Devo riconoscere il merito della mia famiglia, di Guendalina e dei miei amici Daniele, Simone ed Ivano, che hanno sempre creduto ciecamente in me, pur dovendo, troppo spesso, sopportare il mio pessimo carattere e la mia mancanza di sonno.
Un ringraziamento speciale va anche ai miei colleghi Eurico Doirado, Edern Gray e Satoko Ohtsuki, che mi hanno aiutato a finire questo progetto e mi hanno supportato durante quest'ultimo anno di profondo cambiamento.

Preface

This dissertation is original, unpublished, independent work developed at National Institute of Informatics (国立情報学研究所), 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, by the author Cristiano Pedersini.

Questa tesi presenta i risultati del nostro studio sulla generazione automatica dei contenuti per ambienti tridimensionali online basati su Unity, nell'ambito delle applicazioni per Smart Cities.

Il capitolo 2 descrive l'evoluzione degli sforzi di ricerca nell'ambito delle Smart Cities per sviluppare simulatori di traffico e di guida distribuiti ed altamente accessibili. L'ananlisi dello stato dell'arte dimostra la validita di questo tipo di applicazioni come strumenti di supporto per testare l'introduzione di nuove strategie per STI (Sistemi di Trasporto Intelligente).

Il capitolo 3 descrive ICO2, una piattaforma software che integra simulazioni di traffico e di guida, implementate con Unity sulla base di tecnologie pervasive. ICO2 permette di studiare la risposta dei cittadini analizzando il loro comportamento durante simulazioni virtuali in un ambiente tridimensionale estremamente realistico. Vengono forniti dettagli architetturali ed implementativi del simulatore e delle sue principali componenti, quali il modulo per la gestione delle comunicazioni di rete ed il software di generazione del contenuto virtuale per le simulazioni.

Il capitolo 4 mette in evidenza le principali limitazioni dei risultati descritti nel capitolo 2, individuando il processo di generazione dei contenuti 3D, per simulatori ad alto realismo, come uno dei fattori piu condizionanti per la futura evoluzione di queste applicazioni. Nel capitolo vengono descritti gli obiettivi del nostro progetto spiegando come i risultati ottenuti possano aiutare a superare i limiti della tecnologia attuale, in riferimento allo stato dell'arte, all'architettura del simulatore ICO2 ed alla generazione del contenuto virtuale, introdotti nei capitoli precedenti.

Il capitolo 5 descrive i principali dettagli implementativi e le scelte ingegneristiche adottate in questo progetto per estendere ed ottimizzare l'attuale generazione dei contenuti per le applicazioni del simulatore ICO2 su piattaforme mobili. Vengono introdotte le caratteristiche innovative del simulatore e del software per la generazione del contenuto, che viene adattato per offrire un servizio online in tempo reale.

Il capitolo 6 infine descrive risultati, limiti e possibili sviluppi futuri derivanti dal lavoro svolto in questo progetto. I risultati vengono valutati in relazione agli obiettivi preposti nel capitolo 4, descrivendo vantaggi e svantaggi introdotti attraverso le nuove caratteristiche del simulatore ICO2 ed alla rivisitazione del processo di generazione dei contenuti.

Contents

List of Figures

List of Algorithms

Listings

Chapter 1

Introduction

In this thesis we will present our study on automatic content generation for 3D online environments based on Unity, in the context of Smart Cities research applications.

Chapter 2 describes the main research efforts in the Smart Cities field, to employ traffic and driving simulators as tools for behavioral studies and ITS (Intelligent Transportation Systems) strategies benchmarking.

Chapter 3 introduces ICO2, an integrated traffic and driving simulator, based on the Unity game engine and pervasive technologies, which allows to study drivers' behavior in highly realistic virtual environments.

Chapter 4 highlights the main research purposes of this project and explains how our results can help to overcome limitations of current approaches, with reference to the revised literature and the described ICO2 architecture.

Chapter 5 describes and motivates our design and implementation choices to improve the current content generation solutions for ICO2 mobile clients.

Chapter 6 presents results, limitations and new challenges derived from our project, with reference to the introduced research goals.

Chapter 2

Related Work

In this chapter we will present a short study of the state of the art regarding 3D traffic and driving simulators. We will focus on the role that this type of applications can have in the Smart Cities evolution, for which they represent a cheap and practical option for developing and testing ITS (Intelligent Transportation Systems) solutions.

With the increase of urbanization and the continuous migration of people to urban areas, most of the governments and research organizations are challenged to find "smarter" solutions to better understand and address people's requirements in modern urban environments. Celino and Kotoulas [2013] try to give the simplest definition of *Smart City* in the context of internet computing: a city which can effectively process networked information to improve the outcome of city operations such as traffic management. They illustrate a set of research efforts representing the state of the art in the *Smart Cities* field and underline the central role of ICT technologies to deal with the highly interdisciplinarity of this context. Internet is fundamental for communication, information sharing, processing, data analysis and distributed computing.

Celino and Kotoulas [2013] also explain how increasingly *ubiquitous* and *pervasive* city systems are shaped by their inhabitants. For this reason investigating *social aspects* of urban dynamics and information systems is necessary to address citizens' needs, but also their role. In a pervasive smart environment citizens in fact are not only passive actors, they can play an active and fundamental role in improving the urban ecosystem and addressing new limits and challenges. Citizens can collect, filter, and assess information with a crowdsourcing approach, where ICT (information and communication technologies) do not allow for an automated process.

ITS (Intelligent Transport System) strategies can play a significant role in improving quality of life in smart-cities. Their effectiveness however is deeply

influenced by the users' compliance. At present there are no low-cost methods to investigate human reaction to this traffic management strategies. Conventional methods like single-user, highly realistic, expensive driving simulators are not easily accessible. On the opposite side, web based surveys are cheap and highly accessible, but they cannot capture real time behavioral information.

Prendinger et al. [2011] investigate the validity of multi-user 3D virtual environments as a cost effective method to study human behavior in simulated traffic situations. In their work they show the advantages of using "OpenEnergySim", as a benchmark for ITS strategies aimed at reducing CO_2 emissions. OpenEnergySim is a behavioral data collection platform, based on OpenSim virtual simulator. It supports multi-user immersive driving and integrated visualization of simulation results. Prendinger et al. [2011] try to capture the real-time social and human behavior in traffic simulations, which distinguish their single-step approach from those which log the driving data, build behavioral models and only then run traffic simulations using those models.

To prove the effectiveness of OpenEnergySim, Prendinger et al. [2011] run an experiment on simple driving tasks like car-following, for which real-world data and mathematical models are well known. The 3D simulation space for this experiment, integrated in the OpenEnergySim tool, is made by an ad hoc built model of a highway section in Kashiwa, a city in the Tokyo prefecture. The structure of the virtual space is very simple, and the size limited, in fact the two open ends of the highway section are joined with two close curves, to let the users drive for enough time on a street loop. Also the number of the user involved in the experiment is limited to three drivers, which is the suggested smallest number to study car following interactions. The three drivers follow the car ahead in the simulator, driving for 3 hours and 15 minutes. After data is collected it is preprocessed to remove information relative to the curve sections and avoid corruption by synthetic conditions. The car following behavior from the examined data is then compared to real-world data and to the Gipps' Car-Following Model, a collision avoidance model. The results show a fast learning curve for the users, who can adapt easily to the simulated environment, and confirm the behavioral validity of virtual worlds for driving experiments.

Madruga Filho et al. [2012] focus on the challenge of teaching eco-safe driving style in order to reduce transport systems' environmental impact. Eco-driving involves few simple guidelines, such as accelerating smoothly and keeping speed constant, but it can impact fuel consumption and emissions up to 25%. The main drawback in this context is the need for *lasting change of driving habits*, which usually takes long time and effort to happen. Madruga Filho et al. [2012] point out the limits of the existing approaches: Written online guidelines are informative, but not really effective. Real-time feedback mechanisms, implemented

either on in-car devices or as smartphone applications, are helpful but they require previous training to let the user know what to do. Eco-driving courses and events can use simulators or real cars, but they only reach a limited audience and they are too short to change long lasting habits.

Madruga Filho et al. [2012] show their teaching approach through the BeGreen simulator. The innovative aspects of their proposal is the concept of driving as a social activity, involving many people at the same time, who share the same virtual environment. The BeGreen application is highly accessible, needing only an internet connection and standard gaming input devices. It uses real world emission data and can potentially support MMO (Massively Multiplayer Online) features. Using this simulator Madruga Filho et al. [2012] empirically investigate the impact of different graphical feedback on users' eco-safe awareness and driving behavior. However, they can collect meaningful data for only thirty users, which is an improvement compared to single user simulators, but still far from being a social interaction study.

Prendinger et al. [2013] present the "Tokyo Virtual Living Lab" (TVLL), a 3D online experimental space to conduct controlled driving and travel studies. With this tool they aim to contribute to the achievement of eco sustainable and optimized transportation for future smart cities, giving the possibility to analyze interactive driving behavior. Their innovative approach consists in using the same shared environment to have contemporary traffic simulation and multi-user driving experience. In their work they show how users' driving behavior and eco-friendliness can be affected by the surrounding traffic in the shared 3D environment.

The TVLL is embedded in a massively multi-user 3D virtual environment. Although the potential of this kind of environments for testing smart transport solutions is clearly understood [see Prendinger et al., 2011] the effort to create the city-like road network and environment needed to integrate both traffic simulation and immersive user driving can be prohibitive. Prendinger et al. [2013] generate the 3D environment, representing one square kilometer of Tokyo, "feeding" CityEngine with real-world GIS data from OpenStreetMap. Similarly, they extract a navigation network in which the generated trajectories and meta-information is stored. The AI driven cars then use this information to behave in the virtual environment following a perception-cognition-action model.

The pipeline developed for the TVLL takes one day to extract the navigation network and one day to generate the 3D city, which represents a one square kilometer area in the center of Tokyo. The 3D space also needed to be optimized by a graphic designer inside Unity, in order to get the best performance gains through the occlusion culling rendering technique, allowing real-time smooth sim-

ulations. The 30 : 1 ratio between optimization and generation time points out how the optimization and integration of automatically generated 3D content in game engines like Unity for this kind of applications is still a big challenge that researchers and companies need to address.

In Prendinger et al. [2013] the TVLL is used to research the impact of ubiquitous eco-driving on users' behavior while performing simple driving tasks. Thirty users participated in the experiment, where groups of three of them had to drive in the virtual environment following each other at a close but safe distance. The users had to drive on a street loop inside the virtual environment mentioned before. This involves lane changes, observing the traffic light signals and interacting with other vehicles, but no route changes nor turns. The statistics showed an average of three users and twenty-one AI driven cars present on a single road section at the same time. The results of the experiment show that users adapt their driving behavior to the eco-friendliness of AI driven cars, although no significant changes are registered in the simulated CO_2 emissions.

Madruga and Prendinger [2013] show the potential of 3D virtual environments applied to serious games. They introduce ICO2, a serious game for training eco-friendly driving in a multiuser 3D virtual world, built on the same network middleware and the same content pipeline presented in Prendinger et al. [2013]. From the serious games perspective, letting users practice in an engaging and motivating way is fundamental to affect long term driving habits. Their research purpose is to find an adequate challenge level for all the users, in spite of the complexity of traffic situations in a shared environment, with both human drivers and computer controlled entities.

Madruga and Prendinger [2013] use agent-based opponents (both traffic lights and NPC cars) to challenge players, making eco-friendly driving more or less difficult to match their skill level. The method adopted to train the opponents is called "challenge sensitive action selection". In the presented implementation, the opponents are trained offline by standard tabular Q-learning to learn a winning strategy against the user. When online, the system keeps track of each user's skill level n in real-time and, by periodically checking the challenge function, updates it if the two are not matching. When an opponent plays against a driver, the agent chooses the n-th best action for that state from the Q-table, where n is the skill level of the user. This method is tested and effective in a one-to-one game situation. In this particular scenario, which involves multiple agents, either controllable or not (users), every action chosen can have "side effects" on many entities at the same time. This means that instead of finding an optimal action for any agent, the system needs to find a globally optimal solution, considering the global challenge level over the pool of users. To solve this problem they model the scenario as a DCOP (Distributed Constraint Optimization Problem)

and apply the DTREE algorithm to find a globally optimal solution.

To test the feasibility of the designed solution in such a context, which requires real-time performance and scalability to hundreds of users, Madruga and Prendinger [2013] conduct an empirical test with human drivers, on a close circuit street inside a one square kilometer virtual replica of Tokyo. The hypothesis for the study is that the linear time complexity of the DTREE algorithm should not generate a performance bottleneck. The size of the graph representing the DCOP problem and the frequency of the computation to solve it instead, grow together with the number of entities, hence the DCOP approach may become a limit to the scalability of the application.

As expected the study shows that the DTREE algorithm does not represent a limit to the real-time performance needed. However, the size (number of variables) of the DCOP problem and the time needed to solve it remains almost constant on average, instead of growing as expected. This unexpected result probably depends on the fact that the study invelved 10s instead of 100s of entities. The researchers also investigated how the DCOP influenced the value of the challenge function over time. Even if the challange balancing was not the focus of the empirical test, they found that the challenge level perceived by the users was lower than expected on average. This secondary result depends on the sorrounding environment conditions. The variability of the "ambient traffic" (the NPC cars do not change lane), of the environment structure itself, which has limited size and does not allow turns, the limited number of human drivers in the test are all conditions which make the driving experience more predictable, preventing the correct adaptation of the game difficulty.

Prendinger et al. [2014] try to identify the performance limits of DiVE based environments, testing different data communication strategies and running a marketing campaign to benchmark test the ICO2 simulator with real-world users on mobile platforms. The most innovative aspect of ICO2 lays in the capability of collecting human driving behavioral data on a large scale, outside of a controlled research environment, through mobile or web platforms. In order to do this the game needs to be fun and make the driving tasks socially engaging. From a game design point of view, ICO2 has two different playing modes, "free driving" and "campaign". In the first mode incentives and penalties (in form of fuel) are given to the driver, based on the its eco-friendliness and driving rules compliance. In campaign mode, aside from incentives and penalties, the game "missions" present driving and route planning tasks of increasing difficulty level. This mode also integrates an achievement system, which rewards the player for its performance on side tasks (e.g. saving more fuel), and a RCB (Real-time challenge balancing) feature, which adapts game difficulty acting on the behavior of traffic lights and NPC cars.

The DiVE networking middleware presents a simple and extensible architecture but it also needs to meet certain performance constraints in order to guarantee seamless interaction between multi-platform ICO2 clients, providing a smooth immersive user experience. For this reason Prendinger et al. [2014] propose and test a region-based approach to the continuous data sharing mechanism, opposed to the basic one which only uses the entities' "area of interest". The DiVE framework models entities and events. With the basic approach any entity has an area of interest (a square tile in the virtual world) and it receives events from any other entity inside its area of interest. With a region-based approach instead, the virtual world is partitioned into non overlapping regions. In this way a generic entity gets notifications regarding any other entity present in the regions which intersect its own area of interest. With the most naive approach, with n entities, when all entities gets updated about all other entities in the world, the number of messages sent is in the order of $\mathrm{O}\big(n(n-1)\big)$. Prendinger et al. [2014] make an empirical comparison of the region-based approach and the basic one (only area of interest) showing a linear complexity of the first over a quadratic complexity of the former. Applying the region based technique to the DiVE environment and tuning the size of regions and areas of interest, they identify a performance upper bound of thirty frames per second.

Using the aforementioned network settings, Prendinger et al. [2014] run a small marketing campaign to test the game with real users and show the validity of the collected data. For every driver the information logged includes timestamp, user ID, speed, CO_2 emissions, fuel and 3D position. Analyzing this data they were able to partition the users into four clusters using k-means clustering. The number of clusters was chosen by minimizing the SSE (sum square error) over k. From the results it is clear that clusters with less average CO_2 emissions are those whose drivers present a smoother acceleration/deceleration curve. One of the clusters however represents users who use ICO2 as if it was a racing game. This last outcome point out new challenges for real-time driving behavior classification and feedback system. The DiVE middleware did not show any performance issue up one hundred simultaneous players.

Chapter 3

Background

In this chapter we will present ICO2, a 3D, online, integrated traffic and driving simulator, which allows researchers to conduct behavioral studies in a highly realistic virtual environment. We will describe the architecture and the components of the simulator focusing on its virtual environment and on the details of its generation process.

In chapter 2 we described how 3D virtual worlds are evolving towards benchmarking and training tools to support Smart Cities' growth and innovation. Serious games appear to be the natural product of this evolution. Compared to other approaches, serious games have the potential to motivate the direct participation of citizens in the innovation process, supporting their social interaction in highly accessible and realistic driving simulations.

The ICO2 simulator [see Madruga and Prendinger, 2013, Prendinger et al., 2014] has been developed at Prendinger Lab., National Institute of Informatics, Tokyo. This application encloses typical features of a serious game into a research framework through which simulations and user behavior studies can be run, collecting meaningful research data (see chapter 2). In this chapter we describe the structure of the simulator, focusing especially on how the 3D environment, in which the simulations take place, is built.

3.1 ICO2 Simulator Architecture

The architecture and features set of the ICO2 framework derive from those of previous works like "OpenEnergySim" [see Prendinger et al., 2011], "BeGreen" [see Madruga Filho et al., 2012] and "Tokyo Virtual Living Lab" [see Prendinger et al., 2013]. These features has been evolving together with the simulator [see Madruga and Prendinger, 2013, Prendinger et al., 2014] to match the greater

performance demand in the 3D internet applications. The researchers tried to use the scalability of the underlying architecture to reach a more direct participation of citizens, based on pervasive technologies. With the latest version of the application, it is possible to run simulations on Android smartphones. It is possible to use the previous 2013 version instead on Apple's iPads, on Facebook, as a web browser application and at "The Cube" (www.thecube.qut.edu.au), in the heart of QUT's (www.qut.edu.au) Gardens Point campus (Brisbane, QLD, AU).

3.1.1 The DiVE Middleware

DiVE (Distributed Virtual Environment) is the real-time communication middleware on which ICO2 is based. The communication is based on Photon (see www.exitgames.com) and Protobuf which makes it highly scalable, fast and cross platform. Photon provides serialization for the object-based continuous data-sharing mechanism whereas Protobuf is used to serialize objects inside events. The architecture of DiVE is client-server, with the server entity in charge of broadcasting messages from the various clients. The DiVE framework allows to add a different client for any functionality which needs to be implemented. The basic functionalities needed by the ICO2 simulator are implemented in five different types of client entities: The host client, responsible for the generation of the spawn points in the virtual environment. The traffic simulator client, which manages the NPC cars and the traffic lights. The ranking client, which maintains a leaderboard and the progress data for the different driving modes, depending on the simulation being run. The logging client, which constantly receives information about each player and logs it into a persistent database for future analysis.

3.1.2 Content and Metadata

The ICO2 simulator needs two types of content to run realistic simulations and collect meaningful data: The 3D models for the city buildings, the roads and street furniture, which compose the virtual environment; The navigation network containing all the meta-data necessary to run the traffic simulator and to estimate the performance of each user (average consumption, emissions, eco-friendliness etc.). The two type of content are very different, but the workflows through which they are generated and exported are strictly coupled.

The Navigation Network

The navigation network structure is organized on two different levels of abstraction which are reflected in the XML representation used to store the meta-data. This hierarchical structure is used by the traffic simulator, which implements a near-optimal hierarchical pathfinding algorithm (HPA*) to drive the AI cars through the city.

On the low level there is a waypoint network, composed by nodes and links, which form an undirected graph. Listing 3.1 shows a simple example of waypoint network. Any `node` tag, in the waypoint network xml representation, has an unique `id` attribute and three other attributes, `x`, `y` and `z`, representing its three-dimensional coordinates in the scene. Every `link` node has two attributes, `from` and `to`, which identify the ids of the starting and ending nodes of the link. Every `link` node has also a `costs` child, which defines the type and the cost (length) of the link, used by the pathfinding algorithm, through its `flag` and `cost` attributes.

```
<WaypointNetwork>
    <node id="2615" x="-60.07444" y="0" z="93.3046" />
    <link from="2615" to="2618" >
        <costs flag="LaneConnection" cost="74.46095" />
    </link>
    <link from="2615" to="2614" >
        <costs flag="LaneConnection" cost="69.96095" />
    </link>
    <link from="2615" to="2616" >
        <costs flag="LaneConnection" cost="65.46095" />
    </link>
    <node id="1569" x="-496.7339" y="0" z="49.8018" />
    <link from="1569" to="1570" >
        <costs flag="Traversable" cost="56.49391" />
    </link>
    <node id="2592" x="238.6548" y="0" z="-33.80183" />
    <link from="2592" to="2597" >
        <costs flag="LaneConnection" cost="16.89153" />
    </link>
    <node id="1546" x="221.7742" y="0" z="-38.25858" />
</WaypointNetwork>
```

Listing 3.1: WaypointNetwork example

The higher level structural information is stored instead inside the `RoadNetwork` tag. The structure of the `RoadNetwork` subtree is directly extracted from the GIS data tags, which describe in detail all the different types of roads and crossings. Listing 3.2 shows a simple example of road network compliant to the xml schema used.

```
<RoadNetwork>
    <section >
        <tangent >
            <node id="1480" />
            <node id="1479" />
        </tangent>
        <lane from="3347" to="3348" offset="4.5" />
        <lane from="3349" to="3350" offset="-4.5" />
        <lane from="5476" to="5477" offset="0" />
    </section>
    <section >
        <tangent >
            <node id="5975" />
            <node id="5976" />
        </tangent>
        <lane from="5980" to="5981" offset="-3.46" />
        <lane from="5984" to="5985" offset="3.06" />
    </section>
    <section >
        <tangent >
            <node id="5979" />
            <node id="5977" />
        </tangent>
        <lane from="5982" to="5983" offset="0" />
        <lane from="5986" to="5987" offset="4.82" />
        <lane from="5988" to="5989" offset="-4.49" />
    </section>
</RoadNetwork>
```

Listing 3.2: RoadNetwork example

The `RoadNetwork` contains many `section` nodes, each one representing a carriageway segment, identified by a single driving direction and no crossings. A road section is basically the simple directed link in the top level digraph representation. Each `section` has one `tangent` child, which represents the mid-line of a road section, and one or more `lane` children. Every `lane` tag has `from` and `to` attributes (similarly to the `link` of the waypoint network) and one `offset` attribute, which can be positive or negative and indicates the shift distance of a lane from the tangent of the road section. By convention, positive offset means that the lane is shifted on the left of the tangent. The `tangent` node contains two or more `node` children, each one referring to a node in the underlying waypoint network through an `id` attribute. The tangent implicitly contains the information about the section's driving direction (given by the order of the nodes) and the allowed trajectory for each of the lanes (they share the same trajectory of the tangent but shifted by the lane's offset). A drawback of this schema definition is the lack of robustness, because changing the order in which the xml nodes are listed, inside a `tangent` subtree, will also change the road section driving direction. Adding an `order` attribute to the `node` tags can easily fix this issue.

3.2 Content Generation Pipeline

The content generation pipeline is split into three main components. The next sections gives a brief introduction on every component focusing particularly on those features used during the content generation process.

3.2.1 Open Street Map

OpenStreetMap is a free, editable map of the whole world. Its knowledge management system is based on the crowdsourcing paradigm. Unlike proprietary datasets like Google Map Maker, the OpenStreetMap license allows free access to the full map dataset. This massive amount of data can be downloaded in full, but it is available in immediately-useful forms like maps and commercial services. OpenStreetMap represents physical features on the ground (e.g. roads or buildings) using tags attached to its basic data structures (nodes, ways, and relations). Each tag describes a geographic attribute of the feature being shown by that specific node, way or relation.

A node is one of the core elements in the OpenStreetMap data model. It consists of a single point in space defined by its latitude, longitude and node id. A third, optional dimension (altitude) can also be included. A node can also be defined as part of a particular layer or level, where distinct features pass over or under one another (e.g. at a bridge). Nodes can be used to define standalone point features, but are more often used to define the shape or "path" of a way.

A way is an ordered list of nodes which normally also has at least one tag or is included within a Relation. A way can have between two and two-thousand nodes, and can be open or closed. A closed way is one whose first and last nodes overlap. A closed way may be interpreted either as a closed polyline, or an area, or both (e.g. used to define buildings' perimeter).

A relation is an element that consists of one or more tags and an ordered list of one or more nodes and/or ways, which is used to define logical or geographic relationships between other elements. A member of a relation can optionally have a role which describe the function of that particular element within the relation.

OpenStreetMap's free tagging system allows the map to include an unlimited number of attributes describing each feature. The community agrees on certain key and value combinations for the most commonly used tags, which act as informal standards. OpenStreetMap's quality in the Tokyo area compares well to

the available proprietary sources, but it's general data quality varies considerably worldwide.

3.2.2 City Engine

Esri CityEngine is a stand-alone software product that provides a unique conceptual design and modeling solution for the creation of 3D cities and buildings. More specifically, it allows users to create 3D content based on its existing 2D/3D GIS representation. CityEngine uses a procedural modeling approach which means it automatically generates models through a predefined rule set. The rules are defined through a CGA shape grammar system which allows the creation of complex parametric models. A user can change or add shape grammar rules as much as needed, providing room for new design possibilities.

The main CityEngine's features used by the pipeline are the import function from OSM (OpenStreetMap), the Python scripting interface, and the script based export. Street networks and footprints of real world cities can be easily imported from OpenStreetMap into CityEngine. This allows a quick generation of urban surroundings based on existing streets or building footprints. CityEngine also provides the user a Python scripting interface which can execute streamline repetitive or pipeline-specific tasks. The python scripts can automatically create different levels of detail in batch mode, import Autodesk FBX cameras, and so on. The pipeline uses CityEngine's scripting interface to preprocess data imported from OSM, classifying the ways through their semantic tags and assigning the starting CGA generation rule for each different class (e.g. ways which compose a road, rather than a building or a sidewalk). CityEngine's script-based export feature allows scripting of complex export processes via Python, such as writing out additional meta-data or instancing information for each building. The existing pipeline uses this feature to customise the export of 3D models and navigation network.

3.2.3 Unity

Unity is a cross-platform game creation system developed by Unity Technologies. It includes a game engine and an integrated development environment (IDE). It is used to develop video games for web plugins, desktop platforms, consoles and mobile devices. It claims to have 630 thousand monthly active developers and nearly 2.9 million registered developers.

Unity's graphics engine targets the following APIs: Direct3D, OpenGL, OpenGL ES and proprietary APIs on video game consoles. Unity allows specification of

texture compression and resolution settings for each platform the game supports. The game engine's scripting is built on Mono, the open-source implementation of the .NET Framework. Programmers can use UnityScript (a custom language with ECMAScript-inspired syntax, referred to as JavaScript by the software), C#, or Boo (which has a Python-inspired syntax).

Unity can import models and animations from almost any 3D application through its asset pipeline: Maya, 3ds Max, Modo, Cinema 4D and Blender. Sprites can be automatically imported by Unity by simply putting them into the relevant folder inside the project hierarchy. The assets can then be modified at any time and the changes are automatically applied to the project. A model file may contain a 3D model, such as a character, a building, or a piece of furniture. Unity imports models as multiple assets. In the Project view the main imported object is a Model Prefab, that is a "container" which allows to store an object complete with components and properties. Usually there are also up to several Mesh objects that are referenced by the Model Prefab. A model file may also contain animation data which can be used to animate this model or other models. The animation data is imported as one or more Animation Clips.

3.3 The Pipeline's Workflow Explained

The content generation workflow involves the three components we have introduced. We can define this workflow as semi-automatic, because it uses the CityEngine scripting interface to automatically batch process the GIS data and export the 3D content. However, the data retrieval and import processes need the human intervention. All of the Python scripts, which generate and export the city model, need to be run in a specific order by a user through the CityEngine GUI. The following subsections explain step by step the pipeline's workflow.

3.3.1 Downloading GIS Data From OSM

A user can gather data from the OpenStreetMap's online API through a normal http request or using a client application for OpenStreetMap. The latter option, which is the one chose by Prendinger et al. [2013], has the advantage of letting the user see and easily select which part of the world map to download. The user needs first to download JOSM which is a multiplatform client for OSM, implemented in Java. We suppose that the user already has the Java environment installed and set up on his machine. After opening the JOSM GUI he should select a certain area of the world to download the relative GIS data. To do this the user only needs to select the boundary of the area, with his mouse pointer, on

the map shown inside JOSM. The GIS data is organised in semantic layers which can be selected or not to filter out not relevant information (e.g. underground subway network). In the case of ICO2 the user needs to keep only the information relative to the buildings and the road network of the portion of city considered, in this case a part of Tokyo. The last step to get the GIS data locally is to click on the download button.

3.3.2 Generating Content with CityEngine

In the actual implementation of the pipeline, the generation of the 3D content and that of the navigation network are separated. This means that even if the data fed to CityEngine is exactly the same, the two generation workflows are split into two different CityEngine projects.

Generating the 3D City

A CityEngine project is composed of assets, data, images, maps, models, rules and Python scripts, organised into the project's folders structure. The Assets folder contains all those basic resources used in the generation process. The textures for the buildings and roads are organized in dirtmaps, facades, flatroofs, ground, roadmarks, signs and misc textures. The assets also contain some basic 3D models made by a 3D graphic designer, which represent road signs, traffic lights, railways and some parts of the sidewalks. The rules folder contains the files with the CGA rules needed to build all the different parts of the city, in this case: assets, bridge, buildings, flatroof, intersection, road, sidewalk, signs, street and shared.

The first step of the workflow is to import the .osm file downloaded from OSM through CityEngine's GUI. The import dialog displays the GIS data topology accordingly to the OSM meta-data. From this window we can filter out data which can create some problems during the generation. In this case the user needs to deselect all the tunnels and the bridges because the pipeline and so the ICO2 simulator only model flat road networks. Importing the .osm file into CityEngine converts it into a suitable format to be manipulated by the software. Depending on the semantic tags attached, OSM closed ways are converted into "shapes", basically polygons to which CGA rules can be attached, and "lots", sets of shapes which usually represent a building's footprint and which are grouped into a "buildings" layer. Nodes and open ways labelled as streets in OSM are converted into a "street network" layer instead. The OSM relations are used to semantically connect all the elements inside these two layers (e.g. adjacent buildings, linked or intersected roads).

The user can tune some parameters in the CGA rules to slightly change the look and feel of the city before running four different scripts, from the Python editor or the Python console inside CityEngine. The scripts need to be run in a specific order to correctly process the data and generate the wanted result. First `removeTunnels.py` checks if there are still tunnels or unwanted data before generating the city. `cityGenerator.py` then assigns the correct starting rule to lots and shapes based on their tag (sidewalk, road, intersection or building) and triggers the execution of those rules to extrude buildings and create roads. The starting rule for lots is `building.cga`, for sidewalks is `sidewalk.cga`, for streets and crossings is `road.cga`. The `correctPos.py` script resets the offset of the generated models, based on the current scene origin (this prevents the objects to have a wrong offset after being imported in the Unity scene). The last script which need to be run is `exportCity.py` which separately exports blocks (buildings), streets, crossings and sidewalks, dividing them into clusters of at most `P` elements, with a maximum distance `D` from the pivot element chosen. `P` and `D` parameters can be set in the export script to customise the export process. This last step lets the user import the clusters into Unity avoiding that the Unity's editor runs out of memory while importing big files.

Generating the Navigation Network

The workflow for the navigation network generation is similar to the one described above. The corresponding CityEngine project does not contain any specific asset other than the Python scripts which implement the generation and export workflows of the navigation meta-data. The project contains specific scripts for any different element of the street network layer. Some of the scripts implement the class which models a specific street element (e.g. a road section) and others implement the class serialization in xml format. The CityEngine street network elements modelled are: Corridor, Intersection, RoadSection, LaneConnection, Link, Node, RoadSegment, TraversableLink and Network.

After importing a .osm file as described above, the user can start the generation and export workflow running the `main.py` script from CityEngine's Python interface. The script runs the generation and the export of the elements in the scene. During the extraction of the network some extra meta-data describing trajectories and the road network structure is generated, starting from the raw shapes and nodes. First the "drivable" trajectories for simulated vehicles are determined [see Prendinger et al., 2013]. The software has to identify the paths and the lanes where the vehicles are allowed to drive from the shapes' position and perimeter. This issue does not arise in traditional 2D traffic simulators, which represent cars as "dots" moving along links with no lane information.

In this case trajectories are represented as drivable links (`d-links`) and associated to `d-nodes` [see Prendinger et al., 2013]. Every `d-links` will correspond to a single `lane` node in the xml representation of the network. The routing information instead is represented by traversable links (`t-links`) and associated with `t-nodes`. Every `t-link` will correspond to a `tangent` node of a specific road section, and the associated `t-nodes` to the first and the last `node` tags inside that midline (see section 3.1.2). In CityEngine shape objects are identified as "street" shapes or "intersection" shapes, accordingly to the OSM meta tags imported with the data. Each shape contains a vertex pair, and adjacent shapes will share some of the vertexes. In addition, a street shape has two travel directions. The algorithm first builds the `t-nodes`, `t-links`, `d-nodes` and `d-links` for both directions of a street shape. These elements are then split into two different road sections, each one representing one direction. In this way, the algorithm can model two-way streets with arbitrary width and number of lanes.

The generation of the navigation network is a two-step process. First road sections are generated for the entire network. In a second step, shape objects which are identified as "intersection" are processed by checking all incoming and outgoing road sections. This step uses the references that every shape object contains to all of the adjacent shapes (those sharing vertices with it). At last the algorithm builds all links and nodes for each side of an intersection shape, generating the corridors that connect every incoming lane to every allowed outgoing lane.

3.3.3 Importing Content in Unity

Once the navigation network and the 3D models have been exported respectively in XML and FBX format, they can be easily integrated into a Unity project. FBX is a self-contained 3D format, this means that unlike other formats like OBJ, it can incorporate 3D meshes, textures and uv mapping in a single file. The user needs to put all the generated files into the assets folder of the ICO2 client project to allow Unity to automatically import them. FBX files are imported and converted by Unity into Prefabs, which are then shown in the project's hierarchy. Prefab asset type allows to store a generic GameObject (the class of objects which can compose a Unity scene) with components and properties' values. The prefab acts as a template from which a new GameObject can be instantiated in the 3D scene. Any change made to a prefab asset is reflected in all the GameObjects instantiated from that prefab. Once the prefabs for all the FBX files are created, the user can drag them into the Unity scene, using the drag and drop functionality provided by the Editor. All the models are instantiated as GameObjects, translated to the position and rotated towards the orientation written in the FBX file, and finally rendered by the graphic engine.

The 3D representation of a one square kilometer section of Tokyo [see Prendinger et al., 2013] contains a number of triangles and vertices which can introduce a performance bottleneck even when rendered inside the Unity editor and make the application run out of memory. The export process seen in 3.3.2, splits the 3D scene using its elements' classification into streets, sidewalks and buildings. The elements are clustered using a distance metric and only then each cluster is exported into a single FBX file. With the actual implementation, the export is not optimal and can compromise the rendering performance of the city. For this reason after importing the models into the ICO2 client as described above, the whole 3D scene needs to be manually optimised by a 3D designer.

The navigation network is used both by the Unity based ICO2 client and by the traffic simulator client, which is built on the .NET framework. The ICO2 client's scene contains a "Navigation Network" GameObject which has a C# script attached: "NetworkLoader.cs". The user needs to reference the XML file location, inside the project's hierarchy, as a variable in this script. When the ICO2 application starts, the script loads the file and parses it with a basic XML parser to create a class instance of the network to be used at runtime. In the Unity editor view is also possible to render the navigation network and directly edit it, adjusting nodes position and links structure.

Chapter 4

Problem Statement

In this chapter we will highlight the main limitations of the research works presented in chapter 2. We will explain how this project aims at designing a better solution for dynamic content generation which is compatible with the ICO2 simulator. In section 4.2 we motivate our research purpose explaining how this project can contribute to overcome the aforementioned limitations.

After looking at the implementation details of the ICO2 simulator in chapter 3, we can notice that its behavior is heavily dependent on the 3D world in which the simulations take place. The coupling of this two different components represents a limit to the future applications of the simulator because it makes the simulations depend on the specific portion of the real world replicated.

While the proposed approach for immersive driving and traffic simulation has been evolving in the last four years, from "OpenEnergySim" [see Prendinger et al., 2011] to "BeGreen" [see Madruga Filho et al., 2012], "Tokyo Virtual Living Lab" [see Prendinger et al., 2013] and finally ICO2, becoming more and more accessible and proving the validity of virtual worlds, the virtual world itself did not show any particular evolution. The content creation pipeline described in section 3.2, used to create the virtual world for [see Prendinger et al., 2013] and Prendinger et al. [2014] uses a more complex and cheaper procedure compared to a completely manual solution, which on the other hand needs a big effort from a 3D designer. Even so, the pipeline workflow is still not completely automatic and the implicit fixed costs (e.g. licenses for CityEngine and Unity Pro) are not worth if every generated replica still needs to be optimized and manually embedded into Unity to work properly.

The import process of the content, both 3D and meta-data, into Unity uses the game engine's editor GUI, which provides drag and drop functionalities. This aspect makes it very simple and fast to embed the virtual world in the application, but it brings many drawbacks. Although deploying the navigation network meta-

data together with the ICO2 simulator does not affect its performance, deploying the 3D models in the same way, as it is done with the current implementation, heavily affects the size of the application package. This means that on most of the mobile platforms, it is not possible to install the simulator without a Wi-Fi connection, and even having that, the app will need a large amount of storage, compared to the limited resources available in this context.

Having big 3D data deployed inside the application package also brings computational performance limits, indeed the one square kilometer size [see Prendinger et al., 2013], corresponding to the city area replicated, seems to be the upper bound (in terms of triangles and textures) that can be handled by both the Unity editor and mobile platforms, making the optimization effort even more expensive and critical. The latter issue primarily depends on the ICO2 client implementation, but changing it, to provide the 3D content dynamically, will also affect how the content needs to be generated and preprocessed.

Looking at the state of the art, at the up-to-date implementation of the ICO2 simulator and the relative content generation, we can easily notice some limitations that represent a bottleneck for the future development of similar applications. It is clear that we need a different approach rather than generating a one time environment ad hoc for a specific application. With the actual pipeline we can potentially generate any city, given a sufficient quality and completeness of the GIS data in that area, which can be improved with a crowdsourcing approach. However, the size of the city that can be generated is bound by the limited computational and memory performance of the mobile ICO2 clients, and indirectly, by the level of detail of the GIS data: more detail means more data per area unit so higher memory consumption at runtime.

Some of the core features of the pipeline instead are strictly related to the structure of the ICO2 simulator and to the evolution of online 3D simulators, as seen in chapter 2. Using OpenStreetMap to retrieve GIS data provides the advantage of crowdsourcing information. Using Unity as development tool allows researchers to use widely known object oriented programming languages. Unity also provides an easy and usable GUI, allows to export the client as a native application for the all the major architectures on the market and reach a wide pool of users. These features concur to distinguish ICO2 from other research efforts in this field and represent a constraint we want to keep for our study.

4.1 Objectives

Given the limitations presented above, our research purpose is to design a more scalable solution to build bigger real-world, semantically enriched, 3D content for traffic simulations and driving behavior studies, with hundreds of users, on mobile platforms.

More in detail, to address this challenge, taking into consideration the state of the art and the present implementation, we need to: Provide cost-effective 3D content, compatible and optimized for the Unity game engine. Generate content using a completely automatic workflow. Embed the semantic information representing the navigation network into the generated content. Provide a mechanism to download 3D content and metadata dynamically at runtime. Provide content optimized for mobile platforms. Generate content from real-world, open-source and detailed GIS data, like that offered by OpenStreetMap.

In this work we want to study if, given the underlined implementation constraints, which define the compatibility with the ICO2 simulator, it is possible to design a more scalable content creation solution, providing good enough performance for the future development of this application on pervasive technology platforms.

4.2 Motivation

The research challenge we address can bring considerable advantages to the actual implementation of traffic simulators like ICO2. These innovations represent the natural evolution of 3D virtual worlds applications in the Smart Cities field.

As pointed out by Celino and Kotoulas [2013] Smart Cities applications are becoming increasingly pervasive and they are shaped by their inhabitants. The ICO2 simulator lets researchers analyze citizens' driving behavior and help adapting users' driving habits at the same time, using widespread mobile devices. Being able to generate 3D content, starting from OpenStreetMap's GIS data, gives a more central role to the citizens which can contribute to collect this information, improve data quality and directly feel the results using the simulator. Generating this 3D data dynamically and in a more scalable way will be, on the other side, fundamental to let citizens use ICO2 everywhere and at anytime on mobile platforms, making this application really pervasive and overcoming the limits shown at the beginning of this chapter.

Prendinger et al. [2011] uses an ad-hoc built replica of an urban area, specifi-

cally crafted for a single experiment. This can have a big impact on the effectiveness of 3D virtual worlds for the purpose shown. If the cost of creating virtual worlds is too high, only small and very simple urban areas can be affordably replicated. This kind of environments can limit the users' set of actions and the possible social interactions in the simulation, impacting also the results of the experiments. In Prendinger et al. [2011] three users drove on small street loop for more than three hours. It is evident that in this conditions only trivial driving behaviors can be researched, and the motivation of the users is deeply affected. A scalable approach to dynamically replicate different, larger and more interconnected urban areas will easily overcome this limit.

Madruga Filho et al. [2012] consider driving as a social activity, which is proposed through a highly accessible online virtual environment to teach eco-safe driving. Also in this work, as stated above, having a not big and complex enough environment limits the social interactions and affects the overall experience. Compared to nowadays standards for pervasive technologies, the approach proposed is not highly accessible anymore, so also in this case the 3D virtual world should be compatible with more convenient mobile platforms.

Prendinger et al. [2013] introduces a simulated traffic environment using AI cars. Despite registering twenty-one NPC cars at the same time on a single road section on average, considering the human drivers, this statistic has an average value of three. Even if the drivers seem to adapt their driving style to the traffic conditions, Prendinger et al. [2013] do not register any difference in the simulated CO2 emissions. This probably depends on the very limited number of users and the poor interactivity of the environment. The experiment shown in fact takes place on a small street loop, regardless of the effort shown to create a content generation pipeline, which can potentially generate a urban environment of arbitrary size and complexity.

Madruga and Prendinger [2013] try to improve the social experience of the driving simulation by adjusting the challenge level modeling a DCOP. The results of their experiment show that the size of the DCOP did not increase as expected. This result was probably affected by the limited number of users, which is constrained by the limited size of the 3D environment. Madruga and Prendinger [2013] also report that the challenge level perceived by the users was low on average, despite of the adaptive challenge technique introduced. Given that the DCOP was correctly modeled, its ineffectiveness may depend on the limited set of actions that users and opponents can take in the virtual worlds. This limitation directly derives from the simple structure of the urban environment: Driving on a street loop did not allow users and NPC cars to take turns or, for the latter even to change lane.

Prendinger et al. [2014] tests the potential and limitations of the ICO2 simulator on mobile devices, collecting data outside of a controlled laboratory environment. Their results show that the underlying networking middleware does not represent a performance bottleneck even during a stress test with many mobile clients. To fully exploit the potential of this architecture through pervasive technologies, the next challenge is to handle a greater number of simultaneous users. To do so, it is necessary to provide a bigger environment, which cannot be deployed on mobile devices, with the actual state of the implementation. Even optimizing the mobile clients to handle the number of vertices and draw calls needed to render a bigger environment, deploying the 3D content and metadata inside the client application will still increase the size of the application package. The increased size will make it impossible to download the application through mobile networks, limiting the application availability. Re-engineering the way the content is fed into the client application is a primary challenge to overcome the limitations underlined in this study.

Propietary MMO videogames already implement effective solutions to dynamically provide 3D content through a client server architecture. Although similar solutions already exist also for Unity based environments, the information provided does not faithfully depict the real world and does not contain meaningful meta-data for behavioral studies, which instead allows ICO2 to integrate realistic driving experience with traffic simulation. In any case, usually MMO games with big worlds do not run on mobile platforms, because of the performance limits underlined above.

Chapter 5

Implementation

In this chapter we describe how we adapted the current implementation of the content generation pipeline described in section 3.2, to dynamically provide 3D content and navigation meta-data in one single package. The target of the generation pipeline are the latest ICO2 clients, which can run on different, geographically-spread platforms, mostly based on mobile pervasive technologies. These properties introduce highly demanding performance and memory constraints.

5.1 Dinamically Generate Content Upon Request

The generation of 3D content starting from GIS data, as described in chapter 3 is a computationally intensive task. From Prendinger et al. [2013] we know that generating and manually refining the model of the city, used in the previous versions of ICO2, needed one month of work.

We adapted the generation pipeline to automatically download GIS data from OpenStreetMap's web API upon request, generate the 3D models and navigation meta-data and export them into self contained bundles. To provide the CityEngine based features as a service we implemented a socket server through CityEngine's Python interface and we completely automatized the workflow already available.

The new version of the pipeline can elaborate data for any generic geographical region. The time needed to generate the content for a specific area depends on its extension and on the level of detail of the GIS data. To make a comparison, to generate data for a chunk of the city with an extension of approximately one square kilometer and the GIS data quality of the Tokyo area, the improved pipeline takes on average six hours. However, the improved generation performance is not good enough to provide a dynamic generation service for highly

realistic online applications.

Let's consider a generic square area and compare the average generation time with the performance demand of the clients in the best case scenario. The users can drive in virtually any direction allowed by the street network structure. Supposing a maximum driving speed of $30km/h$ which corresponds to $8.33m/s$, in the best case, the longest distance that can be driven corresponds to the diagonal of the one square kilometer area (approximately $1414m$). Given this data, in the best case, it will take around $170s$ for the user to cross from side to side along that path, which is, in any case, a too short interval to generate the next chunk of the city. To overcome this problem it is necessary to separate generation and retrieval of the bundles. We used a caching sever to quickly retrieve pre-generated 3D models and meta-data upon request. For every area which needs to be accessible from the simulator we need to run a script on the pipeline server, which builds, exports and caches the content, ready to be retrieved.

Being the content generated and retrieved in two different moments introduces another problem: Binding 3D content and meta-data to an infinite number of arbitrary grid squares, to which they can belong to, decoupling their retrieval and generation processes. For example, a certain content bundle can be generated on the pipeline server as part of an area we want to cover, and then it needs to be retrieved when some clients ask for an arbitrary different area, which partially overlaps the generated one. Identifying geographical areas by their bounding box allows great flexibility, because the longitude and latitude bounds of an area are represented as decimal numbers. This means that, if we represent geographical areas using a grid pattern, we can have a variable "resolution", specifying an arbitrary number of decimal digits. This flexibility lets easily address geographical areas, but not content bundles, which can belong to infinite areas of arbitrary size.

We decided to solve this problem introducing a maximum resolution for the grid representation of the world map. We fixed the size of the smallest unit which can be addressed through a bounding box. This corresponds to fixing the number of digits in the decimal representation of longitude and latitude bounds. From now on we will call this basic unit "base box". At generation time, the script on the pipeline server receives a generic bounding box and computes the area for which the content should be generated. This area is the smallest grid section which contains the required geographical area and whose size is a multiple of the base box. The area is then divided into its composing base boxes, for which the pipeline downloads the corresponding GIS data and runs the generation and export tasks. In this way the content bundles can be indexed at generation time with the relative base box ID.

5.2 Uniquely Identify Geographical Data

Once content bundles have been generated and cached, they can be retrieved using the corresponding base box unique ID. Upon receiving an HTTP request, with the min and max latitude and longitude bounds as parameters, the caching server controller generates the unique ID through a bijective function which binds a base box to an ID.

Being any base box a bounding box, we can always be sure that, just concatenating its latitude and longitude bounds in the fixed order defined by the OpenStreetMap's documentation, we will generate a unique ID. Reversely, having a generic ID, which we can decompose into its four bounds, we cannot be sure that it will correspond to a base box, but only to a generic bounding box. To ensure that from any tuple of parameters we can retrieve one and only one base box (so the set of content bundles generated for that box), we need to round the decimal parameters to the specified precision. In this way we approximate any generic bounding box to its closest base box.

To manage more easily the request and the merging of the content in the ICO2 clients, we decided to keep both sides' length of a base box the closest possible, giving the base boxes a square shape. As long as the decimal representation of longitude and latitude corresponds to an angle measure in degrees (-90/90 for latitude and 0/360 for longitude), the same decimal interval, along the two angular dimensions, does not correspond to the same distance measure on the earth surface. For this reason we decide to keep three fractional digits for latitude and two for longitude. In this way the side of a base box will measure approximately $1.112km$ from north to south and approximately $0.9034km$ from east to west, which corresponds to a surface extension of almost $1km^2$.

5.3 Request and Re-Assemble Data in the ICO2 Clients

Our objective is testing the feasibility of our content retrieval approach, in terms of seamless loading of the 3D virtual world. To achieve this goal we developed a test version of the ICO2 client, stripping off the training and traffic simulation features and leaving only the 3D driving environment. We also implemented the new functionalities to retrieve and manage the content during the driving simulations.

At loading time the client knows the real world GPS position corresponding to the user's virtual location. This information is used to compute the GPS bounds of the starting base box and it is updated at runtime to retrieve new content

bundles, when necessary, and populate the 3D environment.

Unity based applications use the `Start()` function to execute all the initialization tasks and the `Update()` function, which is called every frame, to keep track of game logic and interactions.
The pseudocode implementation of the `Start()` function, in algorithm 2, shows the initialization of the ICO2 client. The pseudocode in algorithm 1 instead describes how the client application keeps track of the GPS boundaries of the current base box, retrieves the content needed and deletes that which is not used anymore, to optimize the memory usage of the platform.

```
void Update()
    foreach (bound in GetBounds(currentBaseBoxID))do
        if (Crossed(bound, threshold))then
            id ← NextBaseBoxID()
            LoadAssetBundles(id)
            Render3DData(id)
            LoadNavigationNetwork(id)
        end
        if (Crossed(bound))then
            id ← UpdateCurrentBaseBox()
            foreach (adjacentID in AdjacentBaseBoxes(id))do
                if (!IsStored(adjacentID))then
                    RequestData(adjacentID)
                end
            end
            foreach (storedID in GetStoredBaseBoxes())do
                if (! storedID in AdjacentBaseBoxes(id))then
                    DeleteStoredBundles(storedID)
                end
            end
        end
    end
end
```

Algorithm 1: Update() Function

```
void Start()
    (long, lat) ← StartingGPSCoordinates()
    minLong ← ToInt(Truncate(long))
    minLat ← ToInt(Truncate(lat))
    maxLong ← min long +1
    maxLat ← min lat +1
    id ← BaseBoxID(min long, min lat, max long, max lat)
    RequestData(id)
    foreach (adjacentID in AdjacentBaseBoxes(id))do
        RequestData(baseBoxID)
    end
    LoadAssetBundles(id)
    Render3DData(id)
    LoadNavigationNetwork(id)
end
```

Algorithm 2: Start() Function

After downloading the asset bundles for the requested base box, the Unity based client can load the 3D models in background. Once the 3D models are loaded, the software computes the translation vector to be applied to the models, in order to make their position consistent with that of the content already present in the virtual world. This vector has a null z component, because the virtual world replicated is flat. The x and y components of the vector instead are computed as the difference between the GPS coordinates of the loaded base box and those of the starting base box. The latter is actually aligned with the Unity scene's origin, so with the origin of the virtual world. Being the GPS bounds of the base boxes stored as an integer, the vector components represent the number of base boxes present in between the loaded base box and the starting one. Before applying the translation vector, we multiply its components respectively for the length and width extension of a base box.

When loading the navigation network meta-data, the client has to deal with some redundant information. Being the generation and export algorithm conservative, merging the navigation network of two adjacent base boxes, may result in some duplicated elements. To solve this problem the client checks all the loaded nodes and links to find duplicates and removes them. After the duplicates are removed it can re-load the roadnetwork on top of the updated waypoint network, the same way it does in the `Start()` function.

5.4 Generate Unity-compatible content bundles

In the new version of the pipeline we automatized the content export workflow from CityEngine and we added a post-processing phase to make the content, both 3D models and navigation network, compatible with Unity based simulators. Unity based applications can load assets at runtime only after they have been processed by Unity's Assets Pipeline into Asset Bundles. The CityEngine pipeline exports 3D models, as self-contained FBX files, and the XML representation of the navigation network, as described in 3.2. After the export for a base box is completed, the new pipeline runs a Unity application in batch mode (without showing any GUI), through a bash command. This Unity application gets the location of the exported files as a command line parameter, loads them from that location and uses the Unity's Assets Pipeline to pre-process them into Assets Bundles not bigger than a maximum size. In this case we chose a maximum size of 1MB, to avoid ICO2 mobile clients downloading big files over a mobile network.

Once the Asset Bundles have been downloaded by the ICO2 client, they can be loaded in the Unity scene as GameObjects. The ICO2 client can then access objects position in the scene and toggle their rendering based on the user's field of view. With the previous implementation of the pipeline, the 3D models needed to be manually optimized by a designer in order to fully exploit the performance gains of occlusion culling rendering. Occlusion culling is a conservative rendering technique, so it renders all the triangles which can be seen from the camera perspective in the scene. The 3D designer had to modify the structure of the models, splitting the triangles of which they are composed, to find the right trade-off between performance gains and number of triangles to be handled, which is especially critical for mobile devices. The new approach implemented is simpler and it aims to avoid ad hoc optimization still getting good visual realism. In the new client implementation only objects which are closer than a certain threshold to the user and inside his field of view are rendered. In this way it is possible to tune the user's field of view to minimize the number of triangles to be rendered, without the need of using occlusion culling rendering.

Chapter 6

Conclusions

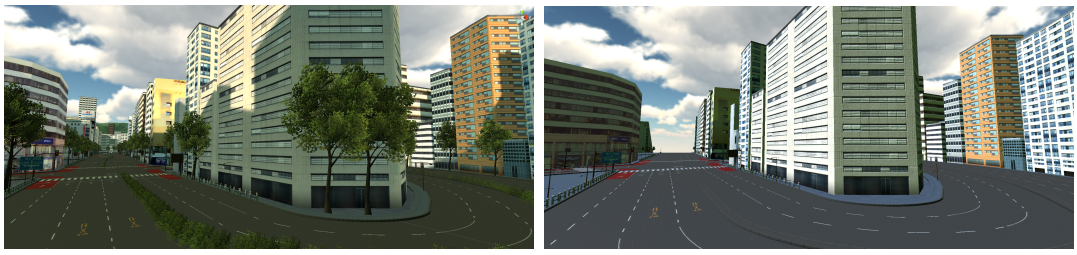
In this chapter we draw some conclusions regarding the described work, presenting achieved results, with reference to the research challenges introduced in chapter 4. We also address limitations, future research possibilities and present efforts to continue and extend this project.

6.1 Results

The first result achieved, thanks to the re-designed content generation pipeline, is the possibility of using the ICO2 client to navigate any part of the world which has been pre-generated on the pipeline server, without being constrained on a single, pre-determined and limited area only. For example figure 6.1 shows a top view of the 3D content generated from a base box in Jakarta, which has a different urban topology compared to the area of Tokyo previously replicated in the ICO2 simulator. Figure 6.1 also shows how the client triggers the rendering of the 3D content when the user's car travels inside the virtual world. The position of the user and its field of view in figure 6.1 have been highlighted. The three consecutive shots were captured from a top view camera we positioned in the Unity scene to better visualize the rendering strategy in the ICO2 client.

We generated the content relative to some different base boxes in the area of Tokyo to test the performance of the generation server. Being the extension of a base box close to one square kilometer and the quality of OpenStreetMap's data uniform across Tokyo metropolitan area, we could compare the average generation time for these areas to the pre-existing performance, reported in Prendinger et al. [2013]. To generate the content for each of the aforementioned base boxes, the pipeline server took six hours on average. We can assess that the generation performance has improved compared to the previous implementation, thanks to the fully automatized workflow which proocess 3D content and navigation metadata in parallel, without any human intervention.

Figure 6.1: 3D data generated from Jakarta



(a) Previous Content Pipeline (b) Improved Content Pipeline

Figure 6.2: Comparison of the look and feel of a Tokyo replica

The second result achieved is being able to use accurate realistic content for simulations on mobile devices, without the need of manual optimization by a 3D designer. Depending on applications performance demand and on the technical specifications of the target mobile device, the number of draw calls and vertices to be rendered in the scene are critical to obtain seamless and realistic user experience. Figure 6.2 shows a visual comparison of the 3D environment generated with the two versions of the pipeline, from the same perspective. Subpicture 6.2(a) shows the manually optimised 3D content; Subpicture 6.2(b) instead shows the 3D content generated by the fully automatic pipeline we described in chapter 5. The two pictures are screenshots taken from the ICO2 client application running on an iPad2 device. We can notice that the level of visual detail is slightly lower in picture 6.2(b) compared to that in picture 6.2(a), mostly because of missing decoration details (e.g. trees) and lightmap. Besides this difference, the scene in picture 6.2(b) still looks realistic. The scene generates a number of draw calls lower than 1000 and a number of vertices lower than $200k$, which represent, on average, a performance demand equal to or lower than the one from the previous versions of ICO2.

The last result we achieved was the decoupling of the 3D content and meta-data from the application logic. With the implementation described in chapter 5 the ICO2 client can download content at runtime, avoiding the necessity of deploying it inside the application package. The size of the application package can limit the performance of the its download and installation on mobile devices.

The maximum allowed size to enable download over-the-air for iOS apps is $100MB$. For Android applications there is no such a limit, but too large application sizes can discourage its installation. The size of the ICO2 application (inclusive of the 3D content and navigation meta-data) is $100MB$ for iOS devices (only tablets) and $75MB$ for Android devices. We do not know the size of the ICO2 mobile client application after introducing the changes described in chapter 5, because they have not been integrated yet with the simulation and gamifica-

tion features of the framework. Being the size of the 3D environment deployed into the current version of the application equal to $700MB$ (lightmap, 3D models and materials); Being the overall size of the ICO2 client Unity project, before compression, equal to $2.5GB$, we expect a size reduction in the order of 30% in the application package size. Given this data we can ensure that the iOS application can keep on being downloaded over-the-air and we expect a reduction in the download time for both iOS and Android clients in the same order of magnitude of the achieved size reduction.

6.2 Future Developments

The features described in chapter 5 have been implemented in a simple version of the ICO2 client which we used to test the performance and feasibility of our content generation approach. In the actual state of implementation, acknowledged that the results presented in section 6.1 fulfill the research objectives described in chapter 4, the next development step will be integrating our work with the simulation environment described in Prendinger et al. [2014] to make it available to the general public.

In order to make the aforementioned results accessible for behavioral studies and simulations, we need to use the pipeline flexibility to reach a better coverage of urban areas of high research interest, generating and caching the relative content. However, the covering of arbitrary geographical areas of the world brings out one of the main limitations of our work. The ability of the pipeline server to generate meaningful and realistic content is constrained by OpenStreetMap's coverage and data quality in the area. Figure 6.1 for example shows the content generated from an area in Jakarta, whose GIS data quality and level of detail are clearly lower than those form Tokyo urban area. We can notice for example that some parts of the street network are missing, which underlines gaps in the GIS data.

There are some important considerations regarding the decoupling of 3D content and application logic, described in this chapter. We pointed out that the assets size reduction, achieved for the ICO2 client, was approximately $700MB$ before compression. The 3D content we removed from the application, corresponds to a one square kilometer sized area of Tokyo (see Prendinger et al. [2013]). However, the size of the 3D models and materials only reaches approximately $200MB$, much less than the size of the lightmap, which we excluded from the generation. Considering that the size of the asset bundles generated may vary depending on the geographical area and on the compression rate achieved by the Unity Asset Pipeline, the download time of asset bundles, relative to one base

box, may become a bottleneck for mobile clients. For this reason, we will need to test the new client implementation with different available network performance, and tune both the size of the base box and the amount of data which needs to be stored as a buffer in the client, to achieve the best user experience.

Researchers at National Institute of Informatics are also developing a GPS mapping system to extend the content generation pipeline we presented in this project. Their target is to study the evolving role of online driving simulators as tools to support the growth of Smart Cities. This project, which follows our work, aims to bind GPS coordinates to the nodes of the navigation network described in section 3.1.2. Through this enriched navigation meta-data, the ICO2 simulator would be able to provide real-time, personalised feedback, based on simulations' results, while users are driving through sensible areas in the real world. One way to implement this feedback mechanism will be using the notification features available on most of the mobile operating systems.

Bibliography

Irene Celino and Spyros Kotoulas. Smart cities [guest editors' introduction]. *IEEE Internet Computing*, 17(6):8–11, 2013.

Marconi Madruga and Helmut Prendinger. ico2: multi-user eco-driving training environment based on distributed constraint optimization. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, pages 925–932. International Foundation for Autonomous Agents and Multiagent Systems, 2013.

Marconi Madruga Filho, Helmut Prendinger, Todd Tilma, Martin Lindner, Edgar Santos, and Arturo Nakasone. Practicing eco-safe driving at scale. In *CHI'12 Extended Abstracts on Human Factors in Computing Systems*, pages 2147–2152. ACM, 2012.

H. Prendinger, A Nakasone, M. Miska, and M. Kuwarhara. Openenergysim: Conducting behavioral studies in virtual worlds for sustainable transportation. In *Integrated and Sustainable Transportation System (FISTS), 2011 IEEE Forum on*, pages 382–387, June 2011. doi: 10.1109/FISTS.2011.5973599.

H. Prendinger, K. Gajananan, A Bayoumy Zaki, A Fares, R. Molenaar, D. Urbano, H. van Lint, and W. Gomaa. Tokyo virtual living lab: Designing smart cities based on the 3d internet. *Internet Computing, IEEE*, 17(6):30–38, Nov 2013. ISSN 1089-7801. doi: 10.1109/MIC.2013.87.

Helmut Prendinger, Joao Oliveira, Joao Catarino, Marconi Madruga, and Rui Prada. ico2: A networked game for collecting large-scale eco-driving behavior data. *Internet Computing, IEEE*, 18(3):28–35, May 2014. ISSN 1089-7801. doi: 10.1109/MIC.2014.21.