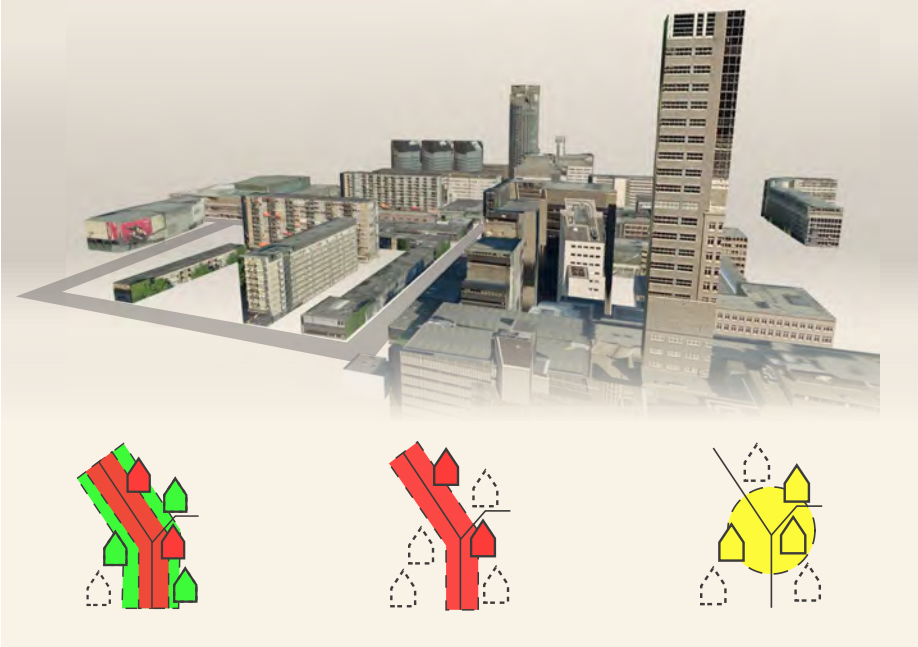
Master of Science Thesis



Knowledge-based optimisation of three-dimensional city models for car navigation devices

Simeon Nedkov

TU Delft

August 2012

Knowledge-based optimisation of three-dimensional city models for car navigation devices

Master of Science Thesis

Simeon Nedkov

August 27, 2012

Graduation Committee
Prof. dr. Peter van Oosterom (Graduation professor)
GIS-Technology, OTB
Dr. Hugo Ledoux (Supervisor)
GIS-Technology, OTB
Dr. Gerwin de Haan (Co-reader)
Computer Graphics and Visualization Group,
Faculty of Electrical Engineering, Mathematics and Computer Science
Prof. Massimo Menenti (External examiner)
Geoscience and Remote Sensing, Faculty of Civil Engineering and Geosciences

TomTom contact
Boris Menkov (Reader)
Principal Architect, TomTom

Department of GIS Technology
OTB Research Institute for the Built Environment
Jaffalaan 9, gebouw 30 2628 BX Delft

TU Delft

Abstract

Three-dimensional maps are deemed better for navigation purposes as they offer a larger number and more realistic navigation cues than two-dimensional maps. Improvements in two key technologies have opened the doors towards utilization of 3D maps for car navigation devices. Advances in data acquisition technologies and data processing methods have made creating photorealistic three-dimensional city models cheaper and to a large extent automatic, while advances in mobile technologies have made e.g. modern smartphones powerful enough to visualize photorealistic 3D graphics. Despite the latter improvements, making three-dimensional mobile maps remains a challenge due to the large amounts of data and the device's limited amount of memory and processing power. These limitations can be overcome by intelligently reducing the amount of information that is handled and displayed by the device.

This thesis presents an information reduction and prototyping framework that reduces the amount of information contained in city models so a to enable their loading and display on car navigation devices. The information reduction method consists of two steps. The first step selects buildings that are close to the driver's route with the idea that these aid the driver in navigating. Buildings that are far from the route are discarded. In the second step, the selected buildings' external representation is adapted to match their navigational value that is based on their thematic, semantic and cognitive properties. For instance, a building of type 'restaurant' and 'brand' McDonald's offers more navigational cues than a block of gray, anonymous residential buildings. The latter are styled in generic textures whereas the former is styled in photorealistic textures. The relations between a building's semantic and thematic properties and its external representation are captured in visualisation rules.

A prototype is built that implements the designed information reduction methods and tests their effectivity. The selection step is performed using a spatial database while the visualisation rules are processed by an expert system. The reduced 3D scenes are displayed in a game engine that also performs performance measurements. The obtained results are conclusive: the performance of a visualisaion in terms of frame rate and used graphics memory is governed by the the amount of textures, much more so than the number of geometries. Effort should therefore be directed towards the reduction and/or simplification of textures rather than geometries.

Contents

1 Introduction **1**
- 1.1 Related work . . . 2
- 1.2 Context and methodology . . . 4
- 1.3 Research objectives . . . 7
- 1.4 Project scope, design decisions and data sources . . . 8
- 1.5 Thesis outline . . . 9

2 Data sources and information organisation **11**
- 2.1 Geographical data . . . 11
  - 2.1.1 Advanced City Model . . . 14
- 2.2 Semantic and thematic information . . . 16
  - 2.2.1 Principles of Linked Data . . . 16
  - 2.2.2 SPARQL . . . 18
  - 2.2.3 DBpedia and LinkedGeoData . . . 19
- 2.3 Data storage and information model . . . 22

3 Information Reduction Pipeline **25**
- 3.1 Information reduction . . . 25
- 3.2 Information Reduction Pipepline . . . 27
  - 3.2.1 Selection . . . 29
    - 3.2.1.1 Operation . . . 31
  - 3.2.2 Extraction and styling . . . 32
    - 3.2.2.1 Operation . . . 33
  - 3.2.3 Visualisation preparation . . . 33
- 3.3 Designing and extending the IRP . . . 33
  - 3.3.1 Chaining . . . 34
  - 3.3.2 Operator extension . . . 34

4 Prototyping, Benchmarking and Testing framework **35**
- 4.1 Context . . . 35
- 4.2 Methodology . . . 36
- 4.3 Execution phase . . . 38
  - 4.3.1 Spatial controller . . . 38
  - 4.3.2 Semantic controller . . . 38

4.3.2.1 Expert system 39
4.4 Visualisation, testing and benchmarking 40
4.4.1 Visualisation and testing 41
4.4.2 Benchmarking 41

5 Prototype **43**
5.1 Preparation phase 45
5.1.1 Footprints 45
5.1.2 Semantic data 49
5.2 Execution phase 50
5.2.1 Spatial controller 50
5.2.2 Semantic controller 52
5.2.3 IRP 54
5.2.3.1 Route preparation 55
5.2.3.2 Selector 56
5.2.3.3 Extractor and themer 60
5.2.3.4 Preparation for visualisation 60
5.2.4 Example IRP results 61
5.3 Visualisation, simulation and benchmarking 63

6 Use case and PBT results **67**
6.1 IRP operation 67
6.2 Performance results 76

7 Conclusion **81**

8 Future work **85**

A Geometry extraction **89**
A.1 Buildings 89
A.2 Tiles 91

B Performance results **93**

Chapter 1

Introduction

Three-dimensional city models are becoming easier to create. Advancements in laser scanning, photogrammetry technologies and point cloud processing techniques have resulted in a significant reduction in production cost. The whole three-dimensional acquisition and modeling pipeline is becoming increasingly more standardized and automatic. Efforts in city model standards have produced the semantically rich CityGML information model. As a result, municipalities and companies are gaining confidence in this field and are investing in three-dimensional models of their cities. Companies like Apple, Google and Nokia are slowly starting to use 3D models for their map making. An increasing number of cities is thus becoming available in digital standardized form (Stoter et al., 2011).

At the same time, navigating urban spaces using 3D information is thought to be more intuitive than two-dimensional maps as it provides more natural and realistic navigation cues (Jobst and Germanchis, 2007; Oulasvirta et al., 2009; Schilling et al., 2005) thereby lowering the cognitive load which is inherent in the constant mapping and transformation of 2D information, i.e. traditional maps, to a 3D world (Crampton, 1992; Bessa et al., 2004). Companies such as BMW are starting to experiment with 3D urban models for car navigation applications.

The research presented in this thesis is conducted for the Dutch car navigation company TomTom that seeks to combine 3D models with their existing car navigation offerings. TomTom has 3D models of numerous major cities that they want to incorporate in their 2D maps. TomTom is facing two challenges: poor visualization performance of the 3D models on their mobile devices and a lack of prototyping tools.

Visualization performance The first problem is caused by the sheer amount of data: a city model may become as large as 8GB (Schilling et al., 2009). Loading and displaying 3D models on mobile devices is therefore challenging (Nurminen, 2008). The amount of information that can be visualized on a mobile device is constrained due to limited memory and 3D rendering capabilities

(Capin et al., 2008). A model may not fit in memory or if it does, it will render slowly. Displaying too much information at once may also overload drivers and hamper their navigation activities (Döllner and Kyprianidis, 2010). Simply loading raw city model data to mobile devices is therefore not an option.

Hence, the two major challenges that need to be addressed are 1) an increase in 3D model visualization performance on mobile devices while 2) limiting the amount of shown information such that drivers are not overloaded. Tackling the first challenge is the field of computer graphics where sophisticated algorithms that run on dedicated hardware speed up the visualization of 3D models. The second challenge, information overload, is tackled by the field of cartography where information is reduced by techniques such as conceptualization and generalization.

Prototyping tools Two-dimensional cartography is a mature field that over the years has developed a rich palette of visualization paradigms and techniques that ensure (spatial) information is presented in a clear way. Such paradigms and techniques are lacking in the field of three-dimensional maps. This is in part due to the novelty of 3D information as well as the increase in complexity i.e. the extra dimension. Finding suitable and effective visualization paradigms is for now a highly iterative process: visualization experts ask the developers to implement a visualization paradigm that is validated by presenting it to the visualization's users who's comments are implemented, are then again presented to them and thus the whole process repeats. This process repeats until both the producers and the users of a map are satisfied. Implementing and reimplementing changes, however, is a labor and resource intensive process as no standard and streamlined prototyping tools exist.

This research aims to address both issues. TomTom's first challenge, i.e. slow visualization and information overload, is addressed by noting that both issues are connected i.e. displaying a lot of information burdens the mobile device as well as the driver. Reducing the amount of information that travels around the device-driver system results in better performance on both ends. This research thus sets out to explore data reduction mechanisms that use geographical, semantic and thematic information to reduce the amount of data that is presented to the driver.

TomTom's second challenge is addressed by building a framework that 1) allows for the assessment of the implemented data reduction strategies and 2) acts as a rapid prototyping environment. Section 1.2 outlines the information reduction method and prototyping tools in more detail.

1.1 Related work

Displaying three-dimensional maps i.e. city models on handheld devices, requires knowledge and praxis from the three distinct fields of cartography, computer graphics and geo-information modeling. The challenges posed by three-

dimensional mobile maps have thus far been tackled from these two angles: computer graphics literature is exhaustive and therefore the current review has been limited to mobile graphics. The same goes for cartography, and therefore the current review is limited to three-dimensional cartography techniques. The review as a whole is limited to city models and terrain models.

Mobile graphics pose a distinct set of challenges due to a lack of computing power, memory, limited battery capacity and a lack of dedicated graphics processing units (Capin et al., 2008). City models are difficult to render on mobile devices as the amount of data is often much larger than the available device memory (Nurminen, 2008). Speed up and data reduction techniques are needed in order to successfully display and interact with 3D city models.

Several different methods of speeding up graphics loading and rendering exist. The most obvious and easy to implement is to wait for better hardware, but as Nurminen (2006) and Bessa et al. (2004) point out, the increase in graphics capabilities goes hand in hand with an increase in user's expectations. Users will always demand more than is possible on mobile devices as they expect to see PC and gaming consoles quality graphics. Another speed up technique is the development of highly efficient low-level rendering algorithms. Capin et al. (2008) give an overview of some common algorithms and techniques. Compression techniques reduce the amount of data that travels between the device's memory and processing elements. Culling techniques such as z-buffering and occlusion queries prevent the loading and processing of geometries which will not be visible to the user. Nurminen (2006) describe a mobile 3D city model rendering engine called m-LOMA which makes heavy use of culling. Two types of culling are applied: preprocessed and runtime. Preprocessed culling is performed by subdividing the space in a 3D grid and running visibility analyses from the corners and center of each cell. Runtime culling is performed by intersecting geometry with the viewing angle or frustum which is commonly known as view frustum culling. Marvie and Bouatouch (2004); Burigat and Chittaro (2005) store the results of preprocessed culling directly in the data structure. This technique is suited for static scenes where the results of visibility algorithms do not change over time. Preprocessed culling's main advantage is the possibility of knowing beforehand which geometries will be invisible to the user and preventing these from being loaded thereby removing the need for runtime culling. Also, no unneeded geometries are retrieved and processed. Another speed up strategy is the utilization of specialized hardware (Capin et al., 2008). Some phones have a separate graphics rendering processor. Nurminen (2007) discuss how the m-LOMA system has been deployed on hardware accelerated devices. Recently, the PC world has experienced the rise of the so-called Graphical Processing Unit (GPU). The GPU is a specialized processor that is tailored specifically for performing computer graphics calculations (Owens et al., 2008). However, mobile devices often do not have GPUs or if a GPU is present it is not yet advanced enough to utilize the full potential of modern speed up algorithms (Noguera et al., 2011).

Another way of relieving the mobile device from performing complex rendering calculations is to outsource these to a more powerful server or a cluster

of servers. In this set-up, the mobile devices sends its position and viewing direction to the server, the server performs the rendering and sends back an image of the rendering. Examples of implementations can be found in Lamberti and Sanna (2005); Jeong and Kaufman (2007). Hildebrandt et al. (2011) have implemented the client-server architecture as a web service using existing OGC standards.

They use OGC's Web View Service to serve panorama image renderings of a three-dimensional scene. However, these techniques require a device with a data connection. Also, the latency in obtaining images prevents the application from being responsive. Noguera et al. (2011) combine the best of the client-server rendering set-up by building a hybrid server-client system. In their implementation, the foreground of the scene is rendered locally, while the background of the scene is rendered remotely. A data reduction rate in the order of magnitude of 100 is reached.

In general, reducing the complexity of the geometry results in a decrease in handled data. Methods borrowed from cartography are implemented by Glander and Döllner (2009, 2008). Here, the urban geometry is simplified by grouping similar buildings together and representing them as a single block. The city scene is kept recognizable by using local and global landmarks and adjusting the size of the landmarks. Nurminen (2006) suggest to automatically cut away geometry that will not be visible.

Oulasvirta et al. (2009) have determined through user trials using their m-LOMA system that low-complexity geometries with high quality textures are better suited for navigation purposes than complex geometries with low-quality textures. Utilising textures is thus warranted. However, textures are in essence raster images, they take a lot of memory to store and are bulky in transport. Nurminen (2008) replace building textures by the dominant color in the building's texture. Another approach is deploying automatically generated generalized textures (Coors and Zipf).

Amri Musliman et al. (2010) use GIS data and spatial analysis to achieve data reduction and speed ups. A 3D buffer of the street network is calculated and only textures of buildings that intersect the buffer are loaded and displayed. Buildings which do not intersect the buffer are displayed without textures. Fisher et al. (2005) automatically find a landmark for each crossing along the route. The saliency of buildings is based on GIS data and visibility analyses performed on a Digital Elevation Model. However, only one landmark is loaded for every crossing.

1.2 Context and methodology

The information reduction technique presented in this thesis revolves around the idea that in the specific case of driving instructions drivers do not need to see a highly detailed representation of the surroundings. Instead, only a detailed representation of the actionable points such as junctions, lane switches, etc. on the route is desired.

The main information reduction strategy I present and apply here is closely related to Choreme map theory (Klippel et al., 2006). Choreme map theory is a formalization of the mechanism by which humans exchange routing information. When someone asks for the route to a certain destination, we do not give a detailed account of the whole route. Rather, we mention only the points on the route where the navigator has to take action i.e. change his driving direction. Directions are often expressed as *take a right turn, then follow the road to the red gas station and turn right again.* In this description it does not matter how far the gas station is or what other buildings stand between the first and the second right turn. The only thing that matters is the point (the red gas station) where action (make a right turn) is needed. This mode of communication in effect compresses the driving instructions i.e. it reduces the amount of conveyed information.

I deploy the same strategy to reduce the amount of information that is displayed by car navigation devices. A driver who is following driving instructions only needs to see 3D information that is 1) close to his route and 2) is relevant for his navigation task; the remaining information is not useful and is thus not visualized.

The information reduction process is split in three steps: *selection*, *extraction and styling* and *visualization preparation.* The selection step determines which buildings are relevant to the driver, the extraction step retrieves their geometries from storage and styles them according to predefined rules, while the last steps prepares them for visualization. A building's relevancy is determined based on its geographical, semantic and thematic properties. Far away buildings are deemed less relevant and are thrown away. This information reduction process is termed the Information Reduction Pipeline (IRP).

Two extraction and styling decision modes are defined, namely spatial and semantic. In the the spatial mode, a building's appearance depends solely on its distance from the driver's route and its action points. The semantic extraction mode takes a building's thematic and semantic properties into account when deciding on the complexity of the building's external representation. Salient and popular buildings such as restaurants, malls, theaters, etc., are thus represented in more detail than non-salient buildings such. Although highly realistic representations help drivers to navigate, they are difficult to render and take up a lot of storage space and processing power. Fortunately, complex representations are not always needed. Buildings next to a highway, for instance, do not need a complex representation as the driver passes them quickly and does not use them to navigate. Replacing these buildings in with less details is beneficial for the visualization performance and driver's experience. I incorporate semantic information when deciding how to visualize a structure by defining rules such as *if building is a restaurant, show in full detail* that are evaluated by an expert system.

The IRP's input is a photorealistic three-dimensional urban scene containing building geometries and a route that pases through the scene. The output is a simplified scene that contains only buildings that are located close to the road, some of which have simpler external representations. The IRP aims is to solve

TomTom's first challenge i.e. improve the visualization performance.

Conceptually, the IRP is designed to run on the car navigation device. However, in this thesis it is implemented as a proof-of-concept that runs on a desktop computer as part of the prototype presented below.

I address TomTom's second challenge i.e. effective prototyping, by proposing a prototyping framework that allows one to easily design and test different visualization configurations. It is here termed the Prototyping, Benchmarking and Testing framework (PBT). As discussed above, the experimental nature of 3D cartography combined with the quirks of developing for mobile devices require an iterative design process. The best visualization configuration is found by creating many different visualizations, comparing them to each other using performance metrics, showing them to drivers and reimplementing their feedback. The PBT achieves this by allowing one to run many different IRP configurations, visualize their results and collect performance measures. Figure 1.1 shows a schematic overview of the IRP and PBT.

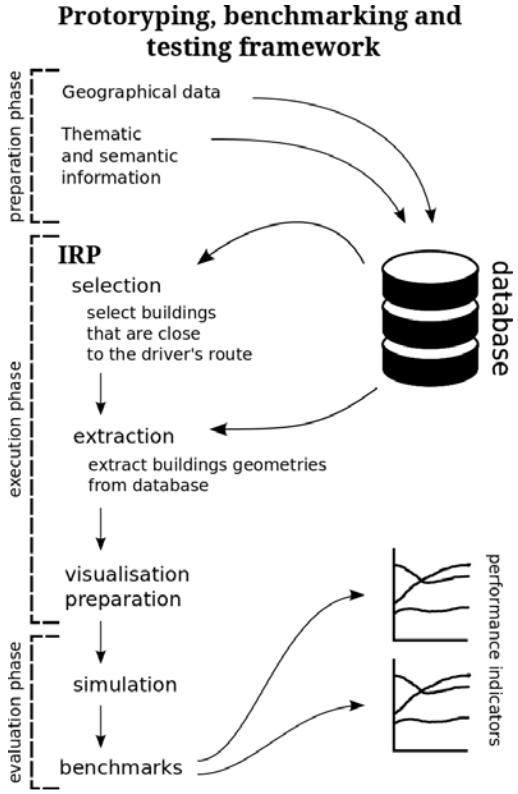


Figure 1.1: A schematic overview of the PBT and the therein contained PBT.

I have built a prototype application that implements both the PBT frame-

work and the Information Reduction Pipeline in order to test the developed information reduction theories and provide the envisioned prototyping functionality. The prototype is built using Python (Lutz, 2006) in an object-oriented design and runs on a desktop computer. The application consists of three phases: preparation, execution and assessment.

In the preparation phase, TomTom's geographical data is extracted from COLLADA and KML files and stored in a spatial database. It is then enriched with thematic and semantic data from Linked Data sources to allow the execution of the information reduction algorithms contained in the IRP. I have written custom classes for parsing COLLADA and KML files, interacting with the database and extracting semantic information from Linked Data.

In the execution phase the prototype runs the IRP and the information reduction algorithms stored therein: information is selected, extracted and styled according to spatial and semantic selection and visualisation rules. Spatial operations, such as buffers and intersections, are performed within the database and by Python spatial libraries. The result of this phase is a three-dimensional scene that is simpler i.e. contains less information, than the input scene.

In the the last phase of the prototype, the IRP results i.e. the reduced scenes, are run in a simulation in order to asses their performance and cognitive effectivity. In the simulation, a vehicle moves through the urban scene following the shortest path. The viewport simulates a traveling vehicle and mimmicks the display of a real car navigation device. As the vehicle moves around, geometries are dynamically loaded and unloaded as a means of keeping the used virtual memory to a minimum. While the simulation runs, performance measures such as frames-per-second, used virtual memory, etc. are collected with the purpose of comparing the different visualization configurations to each other. The simulation is built using the Python game engine Panda3D.

The prototype plays a double role of an information reduction tool by implementing the IRP, and a prototyping and experimentation framework by implementing the PBT.

1.3 Research objectives

The main research objective is to investigate and implement data reduction methods and mechanisms that

> use *geographical, semantic and thematic information*, knowledge about the *routing solution*, *spatial analysis* and *pre-processing strategies* to reduce the information handled by the car navigation device, thereby speeding up the loading and visualization of 3D navigational information.

Geographical and semantical information refers to knowledge about road and building types: whether a city surface is a wall or a roof, speed limits, junction complexity, etc. The routing solution is the result of the shortest path

analysis (be it in terms of distance or time) performed by the car navigation system after receiving a departure and arrival location from the user.

The objective is reached by answering the following research questions:

- What knowledge about the road network and specifically about the navigation solution can be used to improve city model loading and rendering speeds?
- Which semantic and thematic city model attributes and features (i.e. building and road type, knowledge of walls, roofs, etc.) are usable for the purpose at hand?
- What pre-processing techniques can be designed and implemented to improve city model loading and rendering speeds?
- What is an efficient and intelligent way of handling textures using road semantic and thematic information? What (automatic) techniques and strategies exist to reduce the amount of needed textures?
- What is the best of way of assessing the obtained results and comparing these to one another?

1.4 Project scope, design decisions and data sources

The focus of this research lies in incorporating spatial analysis techniques to speed up the visualisation of 3D urban models on mobile devices. These techniques are set forth as an alternative to the more traditional speed up techniques as are found in the field of computer graphics and cartography. This research therefore does not aim to make advancements in these fields. It is assumed that the majority of computer graphics (speed up techniques) and data reduction methods are implemented in the chosen visualisation engine. The work carried out is also not aimed at providing new ways of navigation or enhancing existing navigation paradigms other than making 3D information fit for navigational use. Finally, this research does not aim to provide new insights into user perception of three-dimensional information. The proposed (spatial) techniques are not meant to replace existing cartographic and data reduction techniques, but to exist next to and on top of them.

Design constraints The researched and developed information reduction methods are targeted to run on a car navigation device. Traditional mobile devices are generally limited in processing power, battery capacity and available storage space. Car navigation devices, however, do not suffer from all of these restrictions. Battery capacity is not a problem since the device draws power from the vehicle as it drives through the urban landscape. Storage space is not a limitation either as TomTom plans to offer 3D navigation only to high-end systems that are built in luxurious vehicles where there is enough space for a physical hard disk. An internet connection will not be available as that

is considered unreliable. The information reduction computations are therefore performed locally on the device after the shortest-path calculations.

Semantic information Semantic information gives meaning to the data i.e. it specifies the type, purpose of use, age, architectural style, etc. of the building under consideration. Using semantics, the information reduction algorithms are able to make choices that resemble human decision making. For instance, semantic information makes it possible to adapt the external representation of a building based on its type.

This information is already available in car navigation devices in the form the so-called Points of Interest (POIs): a large database of categorised buildings. This thesis sets to explore ways in which this information can be incorporated in the decision making process.

TomTom's dataset does, however, not contain enough semantic information to enable the desired semantic and thematic data reduction operations. I therefore enriched it with information that comes from Linked Data (LD) data sources (see Section 2.2.1). The so enriched data set aims to demonstrate the possibilities of semantic information rather than deliver a working product.

1.5 Thesis outline

The thesis begins by outlining the used data, its extraction and storage in a spatial database (Chapter 2). The Information Reduction Pipeline and the main information reduction functionality is discussed in Chapter 3. The prototyping framework and its linkage to the IRP are discussed in Chapter 4. The prototype that implements the ideas of the previous two chapters is introduced in Chapter 5. Chapter 6 provides a step-by-step discussion of the development methods and the obtained results. The conclusions are presented in 7 while recommendations and future work is presented in Chapter 8.

Chapter 2

Data sources and information organisation

The previous chapter briefly introduced the main information reduction method and termed it the Information Reduction Pipeline (IRP). The IRP, further discussed in Chapter 3 makes decisions about which buildings to display and how to texture them based on two types of information: geographical and semantic.

Geographical knowledge is here taken to mean the routing solution as calculated by the navigation device, but also spatial methods that operate on it and the 3D dataset itself. Geographical information is what is shown on the device's screen.

Semantic knowledge on the other hand is defined as information that describes the meaning of the model and the buildings stored therein. Semantic information is for example knowledge about the type of buildings (i.e. commercial, industrial, residential, etc) and roads, but also information regarding the state of the driver i.e. is he in a hurry, looking for a restaurant, etc. In this thesis semantic information is used to make decisions about how to show the geographical information.

This chapter describes the two types of information, how they are obtained and stored in a spatial database. Section 2.1 describes the geographical data's extent, number of objects it stores, their properties and the used tiling scheme. Section 2.2 discusses the used semantic data and its sources, and gives a short introduction to Linked Data. Section 2.3 outlines how both types of data are stored in a spatial database.

2.1 Geographical data

The primary data source used in this thesis is TomTom's three-dimensional COLLADA (Banes and Finch, 2008) model of the Dutch city Rotterdam. The dataset covers an area of roughly 55 km^2 that is divided in 333 tiles (Figure 2.1).

Figure 2.1: The extent of the dataset.

The model contains 3D textured boundary representations of buildings situated in the city. The dataset harbors 78566 buildings, 916924 faces and takes 7.3 GB of hard disk space, most of which is taken by the textures. The dataset is a fourth class level-of-detail as defined below

LOD1 Geometry: simplified building ground structures with generalized heights. Textures: none.

LOD2 Geometry: ground structures (footprints and facades), super structures, pediments and roofs. Textures: buildings are colored using facade and roof dominant colors.

LOD3 Geometry: same as LOD2. Textures: generalized image library for facade and pediment textures.

LOD4 Geometry: same as LOD2. Textures: photorealistic image library for facade, pediment and roof textures.

Note that these differ from the LODs defined by CityGML; TomTom's LOD definitions do not include building interiors. Hence LOD4 mathches to CityGML's LOD2.

COLLADA is an XML data model for exchanging digital assets. An asset is anything from 3D geometries, lights, cameras, animations, materials but also rigid bodies meant for inclusion in physics engines and simulations. Assets are

stored in libraries within the XML file and are linked to other assets. Each COLLADA file contains one or more *visual scenes*. Each scene is a tree-like conglomeration of nodes where each node represents a group. A scene may thus contain a 3D model of a building as well as a definition for a camera which determines how the scene should be viewed. COLLADA's aim is to be a "one-stop shop" for seamless exchange of complete scenes. Emphasis is put on information transfer rather than processing. As such, COLLADA files are generally transformed in more efficient (binary) render engine specific file formats.

XML files have a tree-like structure that is queried with the XML Path Language (XPath). XPath enables one to search an XML document and select nodes and elements based on numerous criteria. An XPath expression specifies a pattern that selects a set of XML nodes much like one addresses the hierarchical directory structure of an operating system. For instance, a forward slash (/) is used as a path separator, a double period (..) addresses the parent of the current node, etc. XPath furthermore defines functions that operate on nodes and their values e.g. max(), substring(), etc. and supports mathematical and logical operators such as addition, subtraction, AND, OR, etc. Consider the hypothetical XML document shown in Listing 2.2.

```
<buildings>
  <building id=1>
    <position>51.900409 4.476714</position>
    <geometry_collection>
          ...
    </geometry_collection>
    </building>
    <building id=2>
      <position>51.917591 4.43706</position>
      <geometry_collection>
          ...
      </geometry_collection>
    </building>
    <building id=n>
      ....
    </building>
</buildings>
<trees>
   ...
<trees>
```

Figure 2.2: Fictional XML document that contains a collection of buildings and their geometries, and a collection of trees.

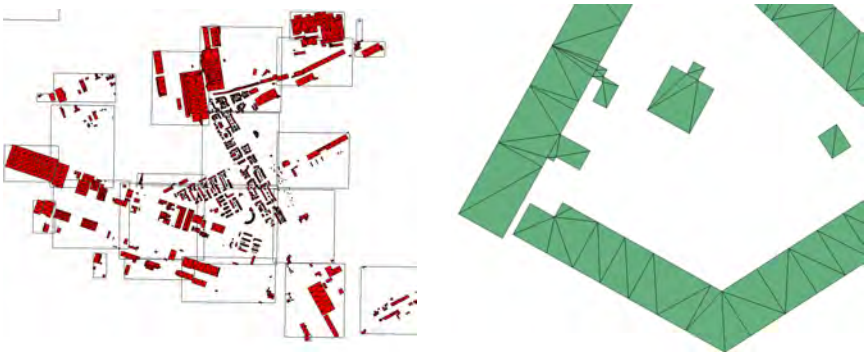
Following are examples of XPath queries that query Listing 2.2.

- */buildings* - selects all building nodes.

- */buildings/building[@id=1]* - select all buildings nodes that have an attribute with a value of 1.
- */buildings[n]* - selects the n-th building node.
- */buildings/building[@id=1]/position/text()* - selects the text value of the first building's position node.
- */buildings/building[max(@id)]* - return the max id value of a building node

2.1.1 Advanced City Model

The TomTom data set defines a city by subdividing an area in tiles that are filled with buildings. The tiles are defined and governed by a top-level Keyhole Markup Language (KML) file containing the location of each tile as a WGS84 latitude/longitude pair, its extent and a link to the COLLADA model that contains building geometries. The geometries themselves are stored in a local coordinate reference system that is centered at the tile's center. Figure 2.3a shows a top-down 2D view of a selection of tiles. Figure 2.3b shows a close-up of footprints triangles.



(a) COLLADA tiles. (b) An example of triangulated footprints

Figure 2.3: Top down views of the dataset.

Buildings are defined as aggregations of so-called building blocks. Each block represents a building structure and is itself defined as collection of footprints, facades and a roof (Figure 2.4).

Textures are stored as external PNG files that act as texture libraries (Figure 2.5).

Section (A.1) discusses the technical details involved in extracting information from the XML files.

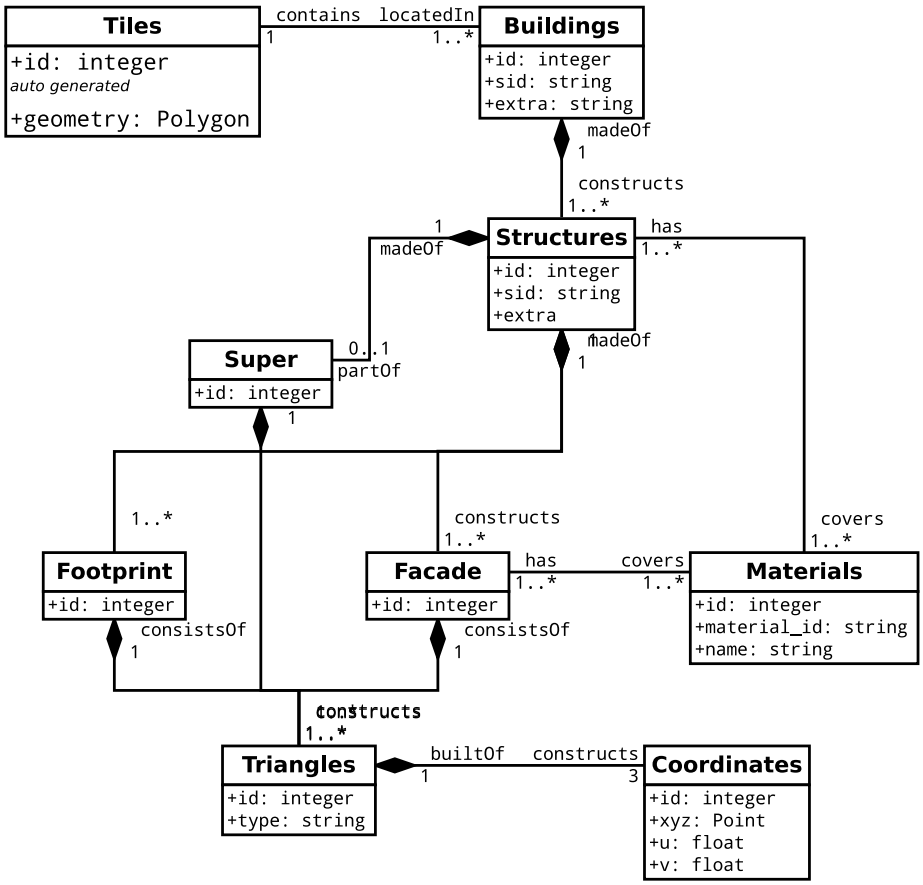


Figure 2.4: UML diagram of the dataset.

Figure 2.5: An example structure material.

2.2 Semantic and thematic information

TomTom's 3D model contains basic implicit and explicit semantic information. Implicit information here means knowledge that is indirectly deduced from the model. For instance, building height is not explicitly stored but can be extracted by finding a structure's largest z-coordinate.

Explicit information is here defined as knowledge that is plainly accessible. All stored buildings have an *extra* field that stores the building's wall color, roof color, function type and others (see Table 2.1).

Semantic and thematic information is used for visualisation purposes in Chapter 3 and for making information reduction decisions in Chapter 4.

A part of these semantic properties are used in the styling part of the information reduction algorithms presented in this thesis (Section 3.2.2).

However, the semantic information reduction algorithms presented in later chapters require more and richer semantic information to function (see Section 4.3.2) such as the currently missing *Architectural type* and *Functional type*. To obtain these properties I have extended the TomTom COLLADA dataset with semantic data from the Linked Data store linkedgeodata.org.

2.2.1 Principles of Linked Data

Linked Data (Bizer et al., 2009a) captures meaning by storing the relation between pieces of information in triplets as *cat is an animal*. The base of Linked Data is formed by the Resource Description Framework (RDF). The following discussion is largely based on Manola and Miller (2004). RDF is a language for

Item	Description	Type
levels*	Building storys	int
wall color	Wall color of real world structure	hex
roof color	Roof color of real world structure	hex
door position	Position of main building entrance	percentage from left edge
type door	Door type (4 types are defined)	hex
type window	Window type (7 types are defined)	hex
mapped color		hex
arch type*	Architectural type	int
mate type*	Material type	int
func type*	Functional type	int
fene type*	Fenestration type	int
geo lod	The building's LOD as defined in Section 2.1	int

Table 2.1: Basic building semantic information. Options marked with an asteriks are reserved for future use and currently have a null value.

representing information about resources (on the Web) by making statements about their properties viz.

```
http://www.example.org/index.html has a creator whose value is
    John Smith
http://www.example.org/index.html has a creation-date whose
   value is August 16, 1999
http://www.example.org/index.html has a language whose values
   is English
```

These statements consists of three parts:

- the *subject*: the thing described by the statement. In this case this is the webpage.
- the *predicate*: the thing's property. In this case these are the creator, creation-date and language.
- the *object*: the value's property. In this case John Smith, August 16 and English, respectively.

These statements are made machine-readable by replacing each part with its Uniform Resource Identifiers (URI). URIs can refer to network-accessible things (e.g. electronic documents, images, groups of other resources, etc.), things that are not network-accessible (e.g. human beings, corporations, books, etc.) and abstract concepts such as "creator". URIs are machine-readable address pointers much like a webpage's URL. In fact, URLs are As such, the subject, prediacte and object can be located anywhere on the web and in turn be RDF statements themselves.

Rewriting the above statements using URIs results in:

```
<http://www.example.org/index.html>
<http://purl.org/dc/elements/1.1/creator>
<http://www.example.org/staffid/85740>
```

Here the URIs point to a machine readable definition of the concept "creator" and a data record of John Smith. The abstract concept "creator" is itself defined by a series of RDF triplets. The mechanism facilitating this is called RDF Schema. RDF Schema defines several base classes and properties which are used to define everything else. RDF Schema provides predefined RDF resources with special meaning which can be used to construct new classes. One of the built-in RDF Schema resource is a class which is used to denote "kinds of things" i.e. types and categories. An important built-in property/predicate is *type* which is defined to mean that an object is an instance of a class. It allows one to define his own classes and properties i.e. it defines a type system.

2.2.2 SPARQL

Linked data is querryed with SPARQL Protocol and RDF Query Language (Quilitz and Leser, 2008). SPARQL allows for SQL-like select statements to be fired at a RDF triplet database. SPARQL statements consists of patterns that are matched to entries in the database. Consider the following example, where an answer is sought to the question "What are all the country capital in Africa?"

```
PREFIX abc: <http://example.com/exampleOntology\#>
SELECT ?capital ?country
WHERE {
     ?x abc:cityname ?capital .
     ?x abc:isCapitalOf ?y .
     ?y abc:countryname ?country   .
     ?y abc:isInContinent abc:Africa .
}
```

Every SPARQL query starts by defining the needed prefixes. A SELECT statement defines the sought values i.e. result. A WHERE statement defines the conditions to which a matched triplet must conform. Evaluation of the WHERE conditions is performed by pattern matching each clause to the entries in the database. If a match is found, the result is bound to one of the variables designated with a *?* and used for the remaining matches. The WHERE clause is successful only when all statements evaluate to true. A step-by-step execution of the above WHERE clause is thus narrated as

- the first line finds all triplets the subject, ?x, of which is linked to the object ?capital by a cityname predicate. In other words: find all objects

?x that are known to be names of capitals, and store the found subject in ?x and the capital's name in ?capital.

- the second line checks whether any of the matched objects stored in ?x have a relation isCapitalOf. If so, the name of the capital is saved in the ?y variable.
- The third line finds the country that belongs to the capital stored in ?y by searching for a *countryname* relation. The result is stored in ?country.
- the last line checks whether the city stored in ?y has a relation *isInContinent* that points to the object Africa.

The result of this query is a table with the columns *capital* and *country* where the rows list all cities cities that are the capital of African countries. SPARQL queries are directed towards web-based endpoints that return the results of a query in various formats.

2.2.3 DBpedia and LinkedGeoData

In this thesis two linked data sources are used: DBpedia and LinkedGeoData.

The DBpedia (Bizer et al., 2009b) project extracts and links the information contained in Wikipedia's infoboxes and categorisation information boxes and stores this as RDF. Infoboxes contain the key facts of a Wikipedia article whereas categorisation boxes describe to which categories the "thing" described in the article belongs. Figure 2.6 shows a part of Rotterdam's infobox.

Area[1]	
• **Municipality / City**	319 km^2 (123 sq mi)
• **Land**	206 km^2 (80 sq mi)
• **Water**	113 km^2 (44 sq mi)
Population (1 February 2012)[1][2]	
• **Municipality / City**	617,347
• **Density**	2,850/km^2 (7,400/sq mi)
• **Metro**	1,211,523
• **Demonym**	Rotterdammer
Time zone	CET (UTC+1)
• **Summer (DST)**	CEST (UTC+2)
Area code(s)	010
Website	www.rotterdam.nl

Figure 2.6: A Wikipedia infobox. Not shown here are facts about the city's location and geographical extent.

Rotterdam belongs to the categories *Cities in the Netherlands*, *Populated places of South Holland*, *Port cities and towns in the Netherlands*, etc.

Information about other objects, such as buildings, is stored in the same manner. DBpedia thus "knows" whether a building is a restaurant, theater, cinema, etc. i.e. it contains the semantic information that TomTom's dataset lacks. The result of dbpedia.org's efforts is a linked data version of Wikipedia that is queryable with SPARQL and is linkable to other datasets.

LinkedGeoData (Auer et al., 2009; Stadler et al., 2012) transforms the OpenStreetMap dataset into linked data and links it to DBpedia. The linkage process begins by first transforming OpenStreetMap to RDF. Auer et al. (2009) first subdivide OpenStreetMap's attributes in three classes:

- *classification attributes*, identify the class a certain element belongs to. For example, the elements *motorway*, *secondary*, *path*, etc. belong to the *highway* class.
- *description attributes*, describe an element by attaching values to it that have some predefined value. For example, an attribute *lit* with a value of yes/no indicates whether a streetlight is on or off.
- *data attributes*, annotate the attribute with free text or values e.g. the opening hours of shops.

The RDF class hierarchy is derived from the *classification attributes*: each *classification attribute* becomes an RDF class, while its values become its subclasses. Thus *secondary*, *motorway* and *path* become subclasses of the class *highway*. *Description attributes* are converted to object properties and their values to sources, whereas *data attributes* are converted to datatype properties and their values to RDF literals. Listing 2.7 shows the result of the transformation.

This schema is matched to DBpedia using machine learning techniques. The training set consists of OSM entities that already have user-created links to Wikipedia. These OSM entities are linked through *owl:SameAs* relations to DBpedia during the transformation described above. The matching heuristic is defined as a combination of three criteria: type information, spatial distance and name similarity. A large heuristic value is obtained if both entities are of the same time, are located close to one another and have a similar name. Matching is then done by computing a heuristic value for all LinkedGeoData points within the vicinity of a DBpedia point, and selecting the one with the highest score that exceeds a certain threshold. The two entities are linked through a *owl:SameAs* relation if the computed heuristic is high enough.

```
lgd-node:26890002 rdfs:comment "Generated by Triplify V0.5" .
lgd-node:26890002 cc:license cc:by-sa/2.0 .
lgd-node:26890002 lgd-vocabulary:attribution "This data is
    derived" .
lgd-node:26890002#id rdf:type lgd-vocabulary:node .
lgd-node:26890002#id geo-wgs84:long "13.7416"^^xsd:decimal
lgd-node:26890002#id geo-wgs84:lat "51.0519"^^xsd:decimal
lgd-node:26890002#id lgd-vocabulary:religion lgd:christian .
lgd-node:26890002#id lgd-vocabulary:name "Frauenkirche" .
lgd-node:26890002#id lgd-vocabulary:tourism lgd:viewpoint .
lgd-node:26890002#id lgd-vocabulary:amenity lgd:
    place_of_worship .
lgd-node:26890002#id lgd-vocabulary:wikipedia%2525en .
"http://en.wikipedia.org/wiki/Frauenkirche_Dresden" .
lgd-node:26890002#id lgd-vocabulary:denomination lgd:lutheran
    .
lgd-node:26890002#id lgd-vocabulary:url "http://www.
    frauenkirche-dresden.de/" .
lgd-node:26890002#id lgd-vocabulary:locatedNear lgd-way
    :23040893> .
lgd-node:26890002#id lgd-vocabulary:locatedNear lgd-way
    :23040894> .
```

Figure 2.7: An excerpt from the result of transforming OpenStreetMap to RDF (Auer et al., 2009).

The TomTom dataset is enriched by retrieving all buildings from DBpedia and LinkedGeoData that are close to Rotterdam, and intersecting them with the COLLADA footprints. A semantic building's properties are matched to a TomTom building through the intersection between the two. Each semantic building has a location, label and a type. The latter is later used in the information reduction algorithms (Chapter 4).

Figure 2.8 shows an overlay of data from LinkedGeoData (red circles) and TomTom's footprints (yellow triangles). Note that many TomTom buildings do not form pairs with a building from one of the linked datasets. This enrichment method is meant as a showcase of the possiblities rather than a mature solution.

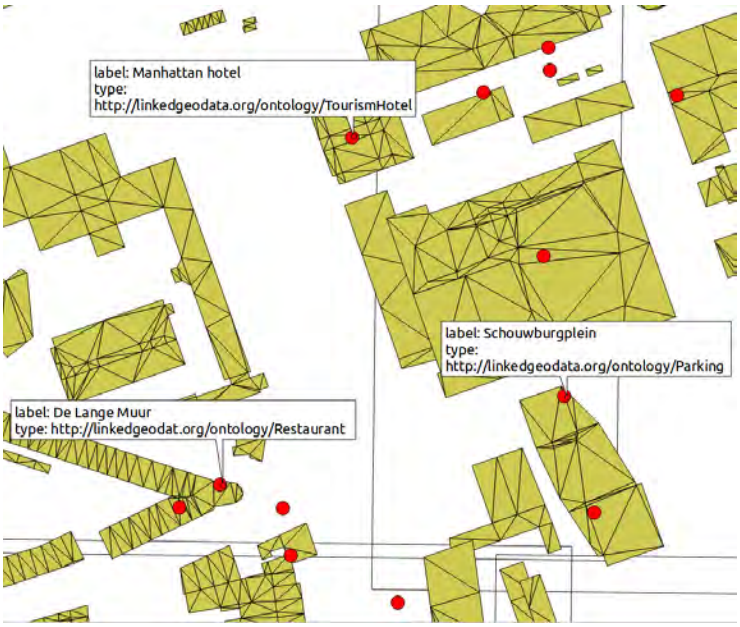


Figure 2.8: An overlay LinkedGeoData buildings (red circles) with TomTom's dataset (yellow triangles). The image shows a hotel, a parking place and a restaurant.

2.3 Data storage and information model

The geographical and semantic data are stored in a spatial database. Spatial databases have many advantages, the most interesting for this project being spatial indexing and the ability to deal with the dataset as a whole rather than as single files.

The database stores the dataset's tiles and buildings as the schema shows in Figure 2.9. The schema is modelled after the UML representation of the dataset as displayed in Figure 2.4.

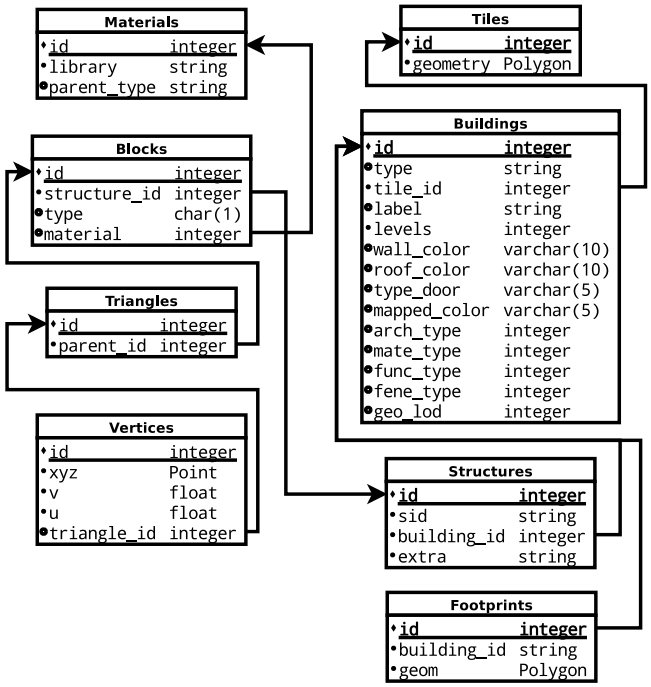


Figure 2.9: Database schema.

The *Tiles* tables stores the COLLADA tiles (Figure 2.1). The concept "building" is stored in the *Buildings* table. The semantic and thematic properties from Table 2.1 are also stored in the *Buildings* table, alongside the type extracted from the Linked Data sources described in the previous section. A building is made from several *Structures* each of which contains one or more *Blocks*. Structures are linked to buildings through the foreign key relation *building_id* in the *Structures* table. Each block consists of footprints, facades and super structures. Blocks are linked to a structure through a the *structure_id* foreign key relation. The *type* column stores the block's designation i.e. whether it is a collection of footprints, facades or super structures. Facades and super structures have a material assigned to them which is a reference image containing the structure's texture (Figure 2.5). The table *Materials* stores a reference to the texture which reside outside the database.

The *Triangles* table stores the triangles that make up the block's geometry. Each triangle is a collection of vertices and texture coordinates. The tables and classes presented up to this point are conceptual classes i.e. they do not define any geometry, this is defined in the *Vertices* table. Vertices are related to triangles through the foreign key relation *triangle_id* in the *Vertices* table.

The *Footrpints* table stores the building's footprint triangles as polygons. This table is used extensively in the spatial lookups involved in the information reduciton process and is therefore indexed with an R*Tree spatial index.

Note that when transforming the COLLADA data structure into a database schema effort is put into optimizng the database for needed applications and not in the creation of a complete COLLADA-to-database converter. Therefore only information that is relevant to the task at hand is extracted and stored in the database.

Chapter 3

Information Reduction Pipeline

Displaying 3D graphics on mobile devices is challenging. Mobile devices often have limited processing power and storage space. Visualisations for mobile devices need to take these limitations into account in order to run smoothly. A selection of optimization strategies is presented in section 1.1. The presented research aims to investigate alternative methods optimization methods that are based on geographical and thematic information. This chapter presents the conceptual basis of the Information Reduction Pipeline presented in the previous chapter. This chapter aims to give a bird's-eye-view of the main information reduction method. It abstains from discussing implementation details, these are outlined in Chapter 5.

This chapter starts by discussing the merits of 3D information for urban navigation purposes. Section 3.1 presents the main information reduction idea on a conceptual level. Section 3.2 presents the information reduction methods, here collectively termed as the Information Reduction Pipelien, by which the envisioned information reduction is achieved. The chapter ends with a discussion on the extensibility of the proposed pipeline.

3.1 Information reduction

The act of information *reduction* can be modified so that it becomes an act of smart and intelligent information *selection*. Instead of presenting information to the driver as-is in full detail, one can choose to show only information that is relevant to the driver's current context and that aids him in performing his current task. When a person is navigating "less is more".

That is why we use maps to navigate instead of raw satellite images. The latter contain too much information which is presented in a flat manner i.e. as if all of it is of equal value. The value of information, however, depends on the task for which it is used. For instance, knowing where cycleways and parklanes are

when driving a car is not relevant as these are not traversable by car. Traditional two-dimensional maps, therefore, display a selection of the available information that supports the map's purpose, be it navigation, reference, wayfinding or other.

I apply the same strategy to 3D city models that are used for navigation purposes. Instead of showing the complete 3D model to the driver and running the risk of overloading him as well as the device, a smart information selection is performed based on the shortest route as calculated by the device between the driver's current position and desired destination. Objects are loaded and displayed only if they are deemed relevant to the navigation process. A building's relevancy depends on its location and semantic and thematic properties (Table 2.1). For instance, buildings located close to the driver's route are probably more useful to his navigation than buildings that are far off. Its visual appearance is varied according to the context of the situation.

This idea shows parallels with Choreme map theory (Klippel et al., 2006). Choreme maps convey information much like humans exchange routing information. Consider the Choreme map shown in Figure 3.1.

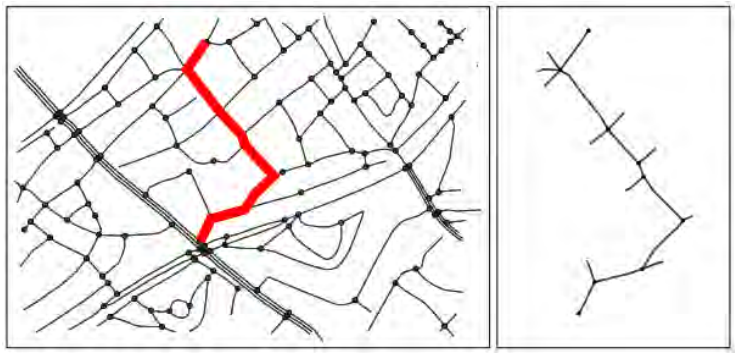


Figure 3.1: Left: bold line represents a "conventional" shortest-path visualisation. Right: The Choreme map representation of the same route. (Klippel et al., 2006)

The desired route (in bold) is shown to the left of the image alongside the city's complete road network. Starting from the top-left corner, this route can be conveyed to a driver as *drive to the first crossing and make a left turn, make a right turn at the fourth crossing, take a right turn at the crossing after that and a left at the crossing after it.* The image to the right of Figure 3.1 shows the visualisation of this description. The actionable points in this description are the crossings at which the driver needs to turn. Information reduction is achieved by 1) showing/mentioning only the streets the driver passes and 2) showing/mentioning only the action points on these streets and diminishing non-actionable portions of the route. Choreme map theory thus acts as a compressor of navigation information: actionable and important parts of the route are emphasized and enlarged while less important and non-actionable parts are

minimized or not mentioned at all.

Choreme map theory is used as a starting point for the here devised information reduction/selection methodology as it closely resembles the driver's mode of operation when using a navigation device, where driving instructions are presented as a list of actions that the driver needs to perform in order to reach his destination.

3.2 Information Reduction Pipepline

The information reduction strategies discussed in the previous section are bundled and formalised in the Information Reduction Pipeline (IRP). The IRP is a set of operations that select relevant buildings for navigation and optimize their external representation with the aim of reducing the amount of displayed information in terms of number of buildings as well as their external complexity. The IRP functionality is based on two information compaction strategies, namely

1. Select buildings that are close to the driver's route or actionable points on the route

2. Adapt building external representation to fit their navigational function

The rationale behind the first strategy, as discussed in the previous section, is that a driver who is following driving instructions does not need to see information that is not related to the driving instructions, and thus does not aid him in his navigation activities. Such is the case with information about objects that are, for example, on the opposite side of the region that is being navigated. Discarding this information removes the burden on the device and driver. The device does not have to load the information while the driver does not have to interpret and process it.

The second strategy adapts the appearance of buildings to their role in the navigation process. A building's role is determined based on its location as well as its semantic and thematic properties. Buildings that are close to Choreme actions points and thus aid the navigation process are shown in full detail. Buildings that are situated in the *compacted* portion of a route are shown in less detail. For instance, the verbal instruction *drive straight for 10 km* compacts a route of 10 km into a single statement. The visual equivalent of this is to simplify the appearance of the buildings in that stretch.

A building's semantic and thematic properties such as type, dominant facade color, etc., determine its suitability for navigation purposes. Well known buildings, such as restaurants and malls, act as navigational aids. These buildings are rendered in greater detail than less significant buildings such as residential blocks.

The IRP is designed to run on the car navigation device. It starts functioning right after the shortest route is calculated by the device:

1. driver requests a route from the device

2. the device calculates a shortest path from the driver's current position to the desired destination
3. the devices passes the route to the Information Reduction Pipeline
4. **for each route segment**
    1. *select* buildings that are close to the route
    2. *extract* building geometry from storage and apply *visualisation styles*
    3. *create visualisation asset* and pass to navigation device
5. navigtion device displays visualisation asset

The Information Reduction Pipeline forms the center of the information reduction functionality. Its input is a 3D urban model that contains building geometries, and the result of the shortest path calculations as performed by the device. The IRP consists of three steps each defining operators that together implement the two information reduction strategies introduced above (Figure 3.2). In the *selection* phase, buildings that are close to the driver's route are selected for display (Section 3.2.1). In the *extraction* phase, the selected buildings are retrieved from geometry and themed according to visualisation rules (Section 3.2.2). The last IRP phase creates a visualisation asset that is passed back to the navigation device for display (Section 3.2.3. A visualisation asset is the result of the IRP operations encoded in a format that can be displayed by the navigation device's rendering engine.

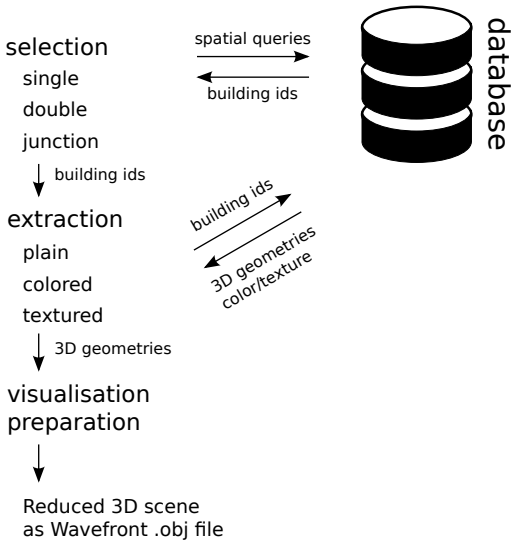


Figure 3.2: A schematic overview of the Information Reduction Pipeline.

The functionality of the *selection* and *extraction* phase is implemented as a set of operators. Each operator performs a single action; it is the smallest unit

of functionality that can perform a selection or extraction. These operators are introduced in order to make the IRP phases more versatile and open ended. For instance, the *selection* phase defines three operators (see Section 3.2.1) each of which selects geometry in a different and distinct way. Running the IRP three times and invoking a different *selection* operator produces three distinct IRP results. This functionality allows the rapid prototyping introduced in the Introduction to work and is a part of the prototyping framework discussed in Chapter 4.

The product of the Information Reduction Pipeline is a 3D scene which contains less information, both in terms of number of buildings as well as their complexity, than the original and unprocessed scene.

The remainder of this section discusses each of the three IRP phases and the operators they define. The discussions are on a conceptually high level. Implementation details are discussed in Chapter 5.

3.2.1 Selection

The basic idea of the selection phase is to use the routing solution to smartly select information that is relevant to the driver and disregard information that is of no use to him. Relevancy is defined as stating that relevant buildings are

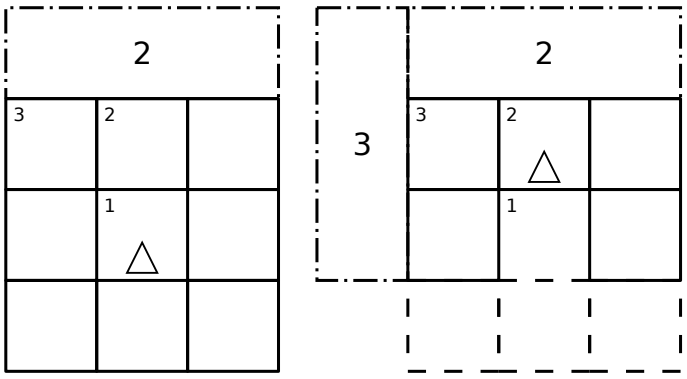
> *those buildings that are close to the road on which the driver is driving or going to drive.*

This IRP phase is the practical manifestation of the first information reduction strategy introduced in Section 3.2.

Vast 3D geographical information is typically visualised by tiling it and showing a small selection of the tiles to the user at any given time. New tiles are loaded and displayed as the user explores the data. Since it is impossible to predict in which direction the user will drag the map, new tiles are only loaded once the user's viewport hovers over them. The loading process becomes visible as every new tile "pops" into existence.

One way to prevent new geometries from popping into existence when not knowing beforehand where the driver will travel is to load the current tile's neighbouring tiles before they are needed and smoothly bring their geometries into view as the driver travels through the 3D model. Figure 3.3a shows this arrangement; the driver's vehicle is represented by the triangle located in tile 1. The tiles adjacent to tile 1 are all loaded and ready for display as it is yet unknown in which direction the driver will travel. In this case, the driver decides to travel North to tile 2. As he enters tile 2, its not yet loaded neighbors, depicted by the dashed region to the North are loaded and prepared for display. The dashed tiles on the bottom are unloaded to free resources. Instead of continuing North, the driver decides to travel West to tile three instead (Figure 3.3b). Keeping with the tile loading scheme, the engine loads and prepares the tiles in region 3 and discards the right most tile of region 2. That tile was loaded in vain and so are the remaining tiles in region 2 if the driver continues traveling

West.



(a) Initial configuration.

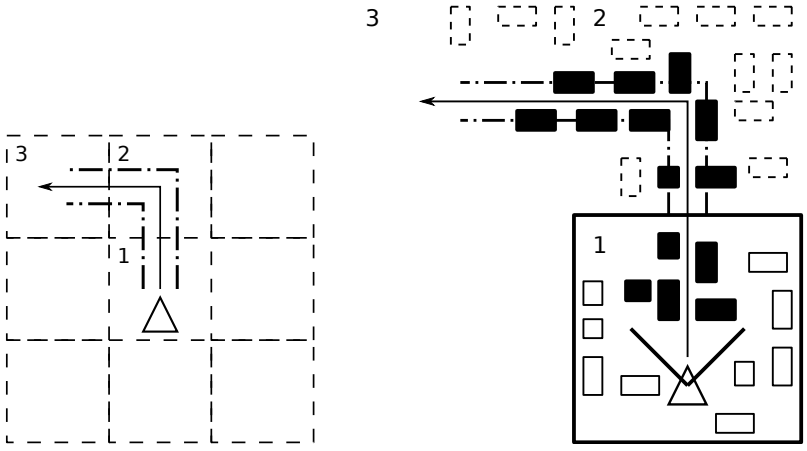
(b) Position of vehicle prior to turning left.

Figure 3.3: A vehicle approaches a junction following a shortest path as calculated by a navigation device.

In the specific case of car navigation devices, however, knowledge about the driver's traveling direction is available in the form of the shortest route as calculated by the device. Loading only buildings that are close to the driver's route greatly decreases the amount of loaded and displayed information while at the same time minimizes unneeded loading and unloading of geometry. Figure 3.4a shows a schematic of the proposed method: the navigation route is used to load a fraction of the tile-based approach which loads nine tiles at the beginning of every journey and three for every tile transition.

The selection method also has merit on the street level. Figure 3.4b shows the gains of the proposed method when compared to loading geometry in tiles. The triangle again represents the driver's vehicle while the two diagonal lines represent the driver's field of view. The bottom of the image shows the situation in the tile-based case (location 1). Whenever a driver enters a tile the render engine loads all of its geometries into memory. It then checks which buildings intersect the driver's field of view and displays them on the screen (shown in black). The major drawback of this approach is that it loads and processes buildings that in the end may not be visible to the user (shown in white). And even if they are visible, they may be unnecessary for his navigation task.

The top of Figure 3.4b shows the effect of the proposed selection method (location 2): only buildings that are close to the road are selected (shown in black, all remaining buildings (dashed) are left untouched i.e. they are not loaded into memory and processed by the rendering engine. The result is a smaller memory footprint and less visibility calculations. Note that using the route for building selection makes tiles obsolete.



(a) The vehicle travels from point 1 to point 3. The dashed rectangles represent the tiles that are no longer needed. The dash-dotted line depicts the new building search region.

(b) Tiled case (bottom) versus non-tiled case (top). Black buildings in location 1 are visible, white building are loaded but not visible. Black buildings in location 2 are loaded as they are close to the route, dashed buildings are not loaded.

Figure 3.4: Schematics of the proposed selection method.

Although simple, this idea is powerful as it is straightforward to implement and results in a significant reduction of information (Chapter 6). This approach has the following benefits:

- we load a small subset of all the available information
- we can deploy smart geometry pre-fetching and caching techniques
- deploy just-in-time geometry preparation and loading
- we can deploy visualisation rules based on building positions

At the same time it is easily extended with more sophisticated building selection strategies.

An evident drawback of this method is the case in which drivers do not follow the navigation route due to a mistake, closed road, faulty map data, etc. This situation can be remedied by segmenting the route in short segments and loading small chunks of data at a time. Whenever a driver misses a turn, a new route is calculated and the process starts over. Responsiveness is guaranteed by making the segments sufficiently short.

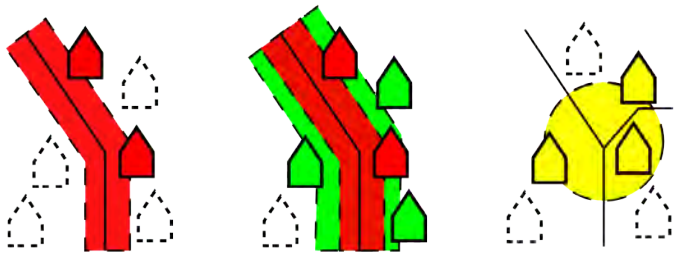
3.2.1.1 Operation

The *selection* phase is implemented by calculating a 2D buffer around the route and intersecting it with nearby building footprints. Buildings that intersect the

buffer are, by the definition given above, selected as important and passed to the next IRP phase.

The IRP *selection* phase defines three operators. The *single* operator is a simple buffer of width w as shown in Figure 3.5a. The *double* operator is a more complex selector that consists of two buffers with different widths as shown in 3.5b. Buildings in red are located closer to the road the green buildings. Figure 3.5c shows the *junction* operator. This operator selects buildings that are close to junctions.

More complex buffers can easily be designed and implemented by visualisation experts (Chapter 6) as the need arises.



(a) The *single* selection buffer

(b) The *double* selection buffer.

(c) The *junction* selection buffer.

Figure 3.5: The three Selection operators.

The *selection* phase's input is the shortest path as calculated by the device, a 3D (tiled) urban scene and the location of junctions. It outputs a list of building IDs that are passed to the next IRP phase.

3.2.2 Extraction and styling

The IRP's *extraction and styling* fetches building geometries from storage and applies visualisation rules designed by visualisation experts (Chapter 6). The visualisation rules define how detailed the exterior of a building should be based on its thematic and semantic properties. The extraction and styling IRP phase is the implementation of the second information reduction strategy presented in Section 3.2. It forms the second step in the information reduction process: after selecting a limited amount of buildings to display in the *selection* phase, this step aims to simplify their appearance in order to reduce the burden on device and driver. The designed simplifications act on a building's texturing style. The extractor defines, just like the selector, a set of operators that harbor the functionality:

Plain extract and paint all buildings in the same color theme (lowest level of detail)

Colored extract and paint each building in its dominant color

Textured extract and paint each building using its textures (highest level of detail)

The plain operator paints the selected buildings in a pre-defined color scheme with different colors for the roof and facades. As a result, all buildings look the same. This is the lightest operator in terms of amount of extracted and visualised information as all buildings have the same appearance.

The colored operator fetches buildings from storage and paints them in their dominant color. The dominant color of a building is defined as the color that best characterizes the building's texture color. Dominant colors are defined for building facades and roofs and were available from TomTom's dataset (Table 2.1). Since each building has its own color, this style generates more traffic between storage and renderer and also requires more storage space.

The textured operator is the heaviest of all defined extractors. It paints the building using their textures as supplied in the COLLADA files.

These extraction and theming operators together with the visualisation rules allow the building relevancy measure presented in Section 3.2.1 to be extended. A relevant building is now defined as a building that

> *is close to the road on which the driver is driving or going to drive*
> *AND has certain semantic and thematic properties*

In other words, the extraction and styling step allows for a variation of a building's external representation based on its type, age, use, etc.

3.2.2.1 Operation

The extractor mainly acts as an interface to the the database. It receives building ids from the selector, extracts their 3D geometries from and styling information from storage and sends it to the next IRP phase.

3.2.3 Visualisation preparation

The final IRP phase receives the building geometries from the extractor and prepares these for visualisation in a rendering engine. It transforms the geometries into a format that the device's rendering engine understands and creates the visualisation asset. The visualisation step thus acts as a link between the processing and visualisation of the data making the IRP rendering engine agnostic. This is also beneficial for extension purposes where the visualisation can be extended to support several output modes.

3.3 Designing and extending the IRP

The Information Reduction Pipeline is a collection of functionalities that aim to reduce the amount of information that is displayed to the user. As discussed in the Introduction and further detailed in Section 1.4, this thesis aims to create a framework for performing 3D experiments quickly and efficiently. It is therefore

important that the IRP can be extended so it becomes a general methodology instead of a one-off specific tool. The IRP is designed so it can easily be extended with extra functionality.

The operators presented above are presented as examples and points of departure that are to be extended by visualisation experts. Extending the IRP's functionality is achieved through *chaining* and *operator extension*.

3.3.1 Chaining

The first mechanism of extending the IRP is by invoking a number of operators of the same type one after the other using different parameters. Linking two selector operators with each other results in a more complex selection pattern. For instance, linking two *plain* selectors (introduced in section 3.2.1.1) of 5 m and 10 m each, results in a two-tier selection pattern in the spirit of the *sized* selector.

Chaining keeps the selectors simple, while enabling complex selection patterns to be achieved. Instead of predefining and hard coding each selector, the IRP is designed such that selectors can be chained to conjure more sophisticated selections. Chaining is beneficial as it reduces the number and complexity of selectors, and prevents duplication of functionality.

3.3.2 Operator extension

The second extension mechanism is one of designing completely new IRP selector and extractor operators. The simple selectors and extractors presented above may not suffice. More powerful selectors allow the construction of more sophisticated and refined selection and extraction scenarios.

As long as each operator respects the information exchange format, operators can be extended and linked in any way wished.

Extending the extraction operators is done much in the same way as the extension of selectors as described above. The extractors described above retrieve the whole building from storage. In some cases, however, retrieving the roof of a building is not needed as it will not be visible to the driver. An additional extractor is thus one that selects only the facades of buildings. An advanced version is one that selects only the facades that are facing the street.

Chapter 4

Prototyping, Benchmarking and Testing framework

The Information Reduction Pipeline presented in the previous chapter reduces the amount of information that a car navigation devices displays to the driver by selecting only information that is close to the driver's route. A further decrease is realized by varying the complexity of the 3D building models based on their usefulness in the navigation process. Both reduction strategies are governed by selection and theming rules. These rules are written by visualization experts that seek to create a 3D map that renders smoothly on a mobile devices, while at the same time is useful to the driver. Writing these rules and discovering the correct IRP configurations is an iterative process that is facilitated by the Prototyping, Benchmarking and Testing framework (PBT).

This chapter begins by discussing the need for the PBT framework. Section 4.2 gives a high-level overview of the framework and presents the three phases that contain its functionality. Section 4.3 discusses the *execution* phase that is responsible for running the numerous IRP configurations. Section 4.4 concludes this chapter by presenting the *evaluation phase* which runs the IRP configurations in a game engine and collects performance indicators.

4.1 Context

A visualization designer's task is to find the *best* visualization configuration in terms of technical performance and cognitive aspects. This is a highly iterative process that on one hand calls for hard performance measurements, while on the other hand requires soft knowledge in the form of user questionnaires and interaction design assessments that gauge the cognitive aspect of a visualization produced by the IRP.

What both cases have in common is that they need to be compared to other configurations in order to determine what the influence of, for example, a fully textured representation of the data is on the visualization performance, but also

on the driver i.e. is the textured model useful and worth the extra burden on the device. While the IRP acts as a tool to decrease information, it is not designed and equipped for answering such questions. I designed the The Prototyping, Benchmarking and Testing (PBT) framework to address these questions.

The PBT framework helps 3D map designers to find an IRP configuration that suits their needs by providing facilities with which to control the behavior of the IRP, automatically run many different configurations, visualize and test their performance.

To this end, the framework allows visualization designers to create custom controllers that govern the IRP's behavior. It facilitates searching for an optimal IRP configuration by automatically running multiple IRP configurations while at the same time recording per-configuration performance measures. The PBT thus wraps the Information Reduction Pipeline and extends it with automatisation, benchmarking and comparison facilities. Visualization designers can then use these measurements, here termed performance indicators, to quickly understand the impact of a design decision, compare it to other configurations and choose a "best" configuration.

4.2 Methodology

The PBT consists of three phases, namely *preparation*, *execution* and *evaluation* (Figure 4.1).

In the preparation phase or pre-processing phase all the required information is collected, organized and stored in the spatial database. The building geometries are extracted from the COLLADA files while thematic information is retrieved from the LinkedGeoData Linked Data source (Chapter 2). This phase is executed only once when the system is set up, consecutive runs of the PBT start at the execution phase.

In the execution phase the IRP is run for each distinct configuration. The IRP is controlled by the rules set forth in the controllers by the visualisation designers.

The evaluation phase runs the different configurations in a simulator. The simulation's performance in terms of e.g. FPS and used virtual memory depend on the the number of loaded geometries and textures and their complexity as complex models require more processing power and virtual memory to run. The simulation's performance is thus a measure for the performance of an IRP configuration. These measures of a configuration's technical performance are termed *performance indicators* (4.4.2).

The PBT framework runs on the development machine. Its aim is to allow to quickly run and test different scenarios in order to determine the correct IRP parameters. Once these are known the IRP and the found rule sets are uploaded to the device, integrated with the navigation application and executed each time the user requests a routing solution.

This thesis does not aim to define what a "best" configuration means as that requires one to also measure the cognitive effectivity of a visualisation

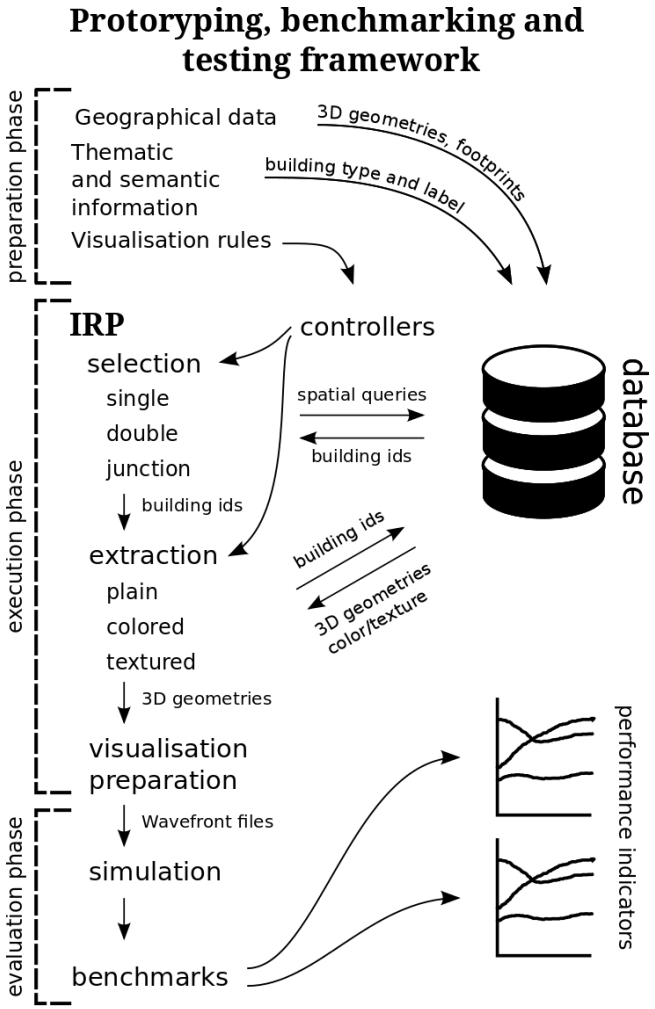


Figure 4.1: A schematic view of the PBT framework.

i.e. a technically well performing configuration may be ineffective in conveying information to the use (Section 1.4). Rather, I aim at creating a framework that allows technical and non-technical visualization experts to apply their knowledge to the case of 3D maps by quickly building numerous prototypes which can be presented to potential drivers. The idea is to facilitate and enable the iterative workflow described in the Introduction.

4.3 Execution phase

The PBT has two modes of operation that use the same IRP operators. The *spatial mode* uses geographical information and operations to select and visualize buildings. In this mode, an object's distance from the driver's route is the only discriminating factor and relevancy measure. The IRP is controlled by an execution plan that is written by the framework users i.e. designers and developers.

The *semantic mode* uses knowledge about the semantic and thematic properties of buildings and roads to decide what to select and how to visualize it. In this mode, an object's relevancy depends not only on its distance from the route, but also on its thematic and semantic properties. This is mode is richer as it is able to incorporate context in the selection and visualization procedure. The IRP is controlled by rules such as *if road type is local and building type is residential, render building in full detail.*

The link between the execution plan and rules, and the IRP are the so-called controllers. The spatial controller is a simple execution plan parser, whereas the semantic controller is an expert system that links the rules to IRP operators.

4.3.1 Spatial controller

The spatial controller is implemented as an execution plan that explicitly specifies which IRP operators are invoked with what parameters. An execution plan is a branch-like structure that lists which operators are to be invoked in each phase of the IRP. The execution plan is written by a visualization expert who wants to test different configurations. An example plan is shown in Figure 4.2.

Each branch signifies a different visualization configuration. Running several different configuration is done by extending the plan with more branches.

As discussed in section 3.3, the IRP can be extended to wield more functionality by chaining the atomic operators. Chaining is achieved by calling a single selector/extractor multiple times.

4.3.2 Semantic controller

The semantic controller governs the Information Reduction Pipeline by interpreting rules which encode the desired type of visualization. These rules translate semantic building properties into IRP selection and extraction operations.

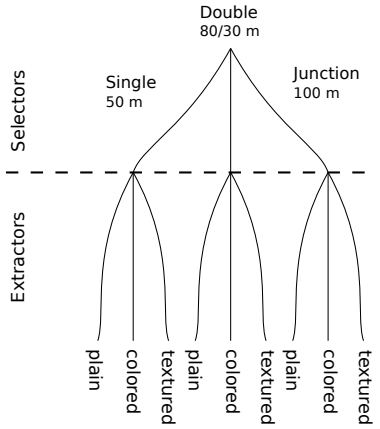


Figure 4.2: An example execution plan. Each branch resembles a single visualisation configuration. Each node in a branch signifies an IRP operation.

Using semantic and thematic building information allows for geometry loading strategies based on knowledge instead of spatial and visibility algorithms (Section 1.4).

For instance, easily recognizable buildings such as banks and fast-food restaurants may help the user orient himself more easily. A sensible choice is to display these *types* of building in full detail. This rule is formalised as

If building is close to route AND is of type restaurant THEN display it in full detail

The semantic controller translates the distance measure *close* to a numerical value and calls an IRP selector with that value. From the list of returned buildings the controller selects the buildings of type *restaurant* and invokes the *textured* IRP extractor and themer (Section 3.2.2).

Translating the rules to IRP instructions is performed by an expert system. Following is a short conceptual description of an expert system.

4.3.2.1 Expert system

An expert system consists of three parts: a knowledge base, a rule set and a ruling engine that makes inferences. The knowledge base contains statements that connect pieces of knowledge together to form facts. The rules describe how a set of facts is related and lead to a different set of facts. An expert system thus deduces new information from known information by way of rules that capture the system's behavior. The workings of an expert system are explained through family relations.

Consider the case where Thomas and Norma have two sons, Bruce and Sam. Bruce and Sam are thus brothers. An expert system can deduce the second fact from the first through a rule connecting the two facts. The facts are formalised as

```
son_of(bruce, thomas, norma)
son_of(sam, thomas, norma)
```

These rules state that Bruce is a son of Thomas and Norma. The rule relating these facts to information about brotherhood is written as

```
  then brother_of($person1, $person2)
if
  son_of($person1, $father, $mother)
  son_of($person2, $father, $mother)
  check $person1 != $person2
```

This rule checks whether two people are brothers based on information about their parents. The conclusion *brother_of* is reached only after each of the *if* conditions are met. Conditions are facts themselves which are pattern matched with known facts in the knowledge base. Say that $person1 is Sam and $person2 is Bruce. The first condition of the above rule is checked and a fact is found that matches

```
son_of(sam, thomas, norma)
```

The same goes for the second condition. The last conditions ensures that $person1 and $person2 are the same. The result of a rule is a new fact that is added to the knowledge base. In the example above the new fact is

```
brother_of(sam, bruce)
```

In other words, a new piece of knowledge is created stating that Sam and Bruce are brothers. Note that the first two clauses passed despite the lack of information about the $father and $mother variable. When no information is provided about a part of the fact variables are assigned values that belong to the matched fact i.e. since $person2 equals Bruce, and the only fact involving him states him to be the Thomas and Norma's son, they are assigned to the values of $father and $mother, respectively. Rules are thus pattern matched and assigned values from the facts whenever a value is ommited upon the invocation of a rule.

An expert system is thus able to validate rules and produce new information. Making it fit for use in the current application is done by writing the corresponding rules. The facts are provided by the building's themselves through their semantic properties. Section 5.2.2 discusses the implementation of the ruling engine for the case of buildings.

4.4 Visualisation, testing and benchmarking

The last phase of the Prototyping, Benchmarking and Testing framework is the assessment of each visualisation configuration. This process is split in two parts: visualisation and testing, and benchmarking.

4.4.1 Visualisation and testing

The visualisation and testing step is meant to visualise each IRP configuration such that the effects of the design choices can be seen. To this end a simulation is run that mimmicks the display of a car navigation device. A moving viewport is implemented that follows a shortest route around the urban environment thereby showing the effects of the IRP selection and extraction operators. Section 5.3 outlines the implementation details.

4.4.2 Benchmarking

The goal of a visualisation designer is to craft a visualisation that is cognitively effective while at the same time runs smoothly on a mobile device. Measuring the cognitive effectiveness of a visualisation is a difficult task as it entails performing live user tests. A possible set-up involves potential users looking at a running simulation and commenting on in either through questionnaires or simply by discussing their observations. Such tests are not performed in this project.

Optimizing a visualisation for speed and smoothness entails making correct choices about the number and complexity of geometries that are displayed, the number and complexity of textures, etc. Predicting beforehand what the impact of a certain configuration will be on the performance is difficult. The best way of addressing this issue is by defining a set of performance indicators that characterize the performance of a visualisation configuration, running different visualisation configuration and observing and recording the values of the indicators.

Table 4.1 lists the performance indicators used in this research.

	Performance indicator	Unit
1.	Frames per second	fps
2.	Virtual memory	MB
3.	Graphics memory	MB
4.	Drawing time	ms
5.	Culling time	ms
6.	Number of scene nodes	[-]
7.	Number of geometries	[-]
8.	Number of vertices	[-]
9.	IRP: selection time	s
10.	IRP: extraction time	s
11.	IRP: preparation time	s
12.	Storage space	MB

Table 4.1: Performance indicators

Note that the PBT is run on a desktop machine and not on a mobile device. The thus captured performance indicators will therefore characterize its performance instead of the mobile device's. The produced performance indicators

are thus meaningless in absolute terms as they capture the performance of the simulation on the hardware it is running on. They do give insights when seen in relative terms i.e. compared to one another they do allow the determination of a *best* configuration.

Chapter 5

Prototype

The previous chapters introduce the Information Reduction Pipeline that is aimed at reducing the amount of information handled and displayed by the mobile device, and the Prototyping, Benchmarking and Testing framework that is aimed at easing the task of finding and optimal IRP configuration.

This chapter presents a prototype application that implements the theories and ideas presented in the previous chapters. The prototype implements the methods set forth in the IRP: data selection, extraction and theming as well as operator chaining. The three PBT phases i.e. preparation, execution and assessment are implemented as well. Users are thus able to input a 3D city model, a route and a set of visualisation rules and the prototype will create a reduced scene, run simulations and deliver a report on the performance of each configuration.

The prototype is built with Python using an object-oriented design. The following tools and additional libraries are used:

- Linked Data sources are quiried with SPARQL using the Python *SPARQL-wrapper* (Herman et al.) library. XML files such as COLLADA and KML are parsed using the Python *lxml* (Behnel et al.) library which is also used to execute XPath queries. Coordinate transformations are performed using the *proj* library which is a Python Interface to PROJ.4 *proj4*. *Shapely* is used for spatial operations that are performed outside the database. These libraries are chosen as they are the defacto standard for performing the needed tasks.

- The spatial database SpatiaLite (Furieri) is used for storing geographical data and performing spatial analysis on it. SpatialLite is the spatial extension of the SQLite database. SpatiaLite is a lightweight serverless transaction SQL database engine that has an almost complete implementation of SQL-92 standard and is OGC-SFS complaiant. Its (2D) functionality is comparable to that of PostGIS and Oracle Spatial e.g. it supports spatial operations such as buffers, intersections, etc., and indexes, namely

R*Tree and MbrCache. SpatiaLite is self-contained; the entire database i.e. its definitions, tables, indexes, and the data itself is stored in a single cross-platform file. SpatiaLite is therefore suited for single-user embdedd applications. SpatiaLite is chosen for this project as it provides the needed functionality within the posed constraints i.e. absence of an internet connection (hence no access to a PostGIS/Oracel Spatial server), but more importantly because it is readily available on modern mobile devices. Porting the functionalities developed in this prototype is then simply a matter of copying the database file and wrapping the SQL queries in the device's native language.

- The simulation is implemented on top of a game engine. A game engine is a system designed for the development of interactive 2D and 3D computer visualisations such as video games and augmented reality installations. Game engines facilitate development by providing ready-made facilities for rendering, sound, animation, events, scripting, artificial intelligence, networking, streaming, memory management, scene graph, benchmarking facilities, etc. Such facilities greatly simplify development of interactive applications as they provide ready-made functionality that a developer can readily re-use and does not need to develop himself.

  The game engine of choice is Panda3D (Goslin and Mine, 2004) as it is entirely written in Python. Panda3D therefore integrates easily with the rest of the developed functionalities. It furthermore has a very mature documentation and a lively community. These factors are important when picking up a new technology as tremendously speeds up the learning process. In contrast, other game and render engines such as Ogre (ogr, 2012) and OpenSceneGraph that are ported to Python often lack a thorough Python documentation or a community using the port as the attention generally goes to the platform's native language (C++ in both cases). Furthermore, most ports are a work-in-progress which means that a Python developer runs the risk of needing functionalities that are not ported yet, although they are spoken of in the original documentation. The second major reason to choose for Panda3D is its extensive benchmarking facilities (Figure 5.8). Panda3D provides real-time graphical and command-line tools to check and record the performance of the running visualisation. This functionality aligns perfectly with the purpose of the PBT; compare the performance of different IRP configurations.

- The expert system of choice is *Python Knowledge Engine* (Frederiksen). The knowledge-based inference discussed there is implemented through the Python Knowledge Engine (PyKE). PyKE is an inference engine that brings Prolog-like logic programming to the Python language. PyKE supports the creation of a knowledge base and rules that link facts together to form new knowledge. It is mainly chosen for its simplicity and the fact that it is written in Python.

The prototype is a direct implementation of the Prototyping, Benchmarking

and Testing. Like the PBT, it consists of three phases as shown in Figure 4.1. The preparation phase extracts the data from the various data sources files and populates the database. The execution phase executes the information reduction process by parsing user created visualisation rules and invoking the IRP. The final PBT step produces runs the various IRP configurations in a simulation and shows their performance measures.

The IRP, as noted in the previous chapters, plays a double role in this research. On one hand it incorporates the main information reduction algorithms as a stand alone piece of functionality that is intended to run on the car navigation device, and on the other it is a core part of the PBT and the prototyping process.

This chapter discusses the prototype's implementation details. Section 5.1 discusse the prototype's preparation phase. The IRP implementation is discussed in the execution phase in Section 5.2. Section 5.3 discusses the built simulation and the results it is capable of producing.

5.1 Preparation phase

The first step of the PBT is the preparation step where geographical and semantic information is extracted from the COLLADA/KML files and Linked Data sources, and stored in the database. Three types of data are inserted in the database: 3D building geometries, 2D footprints, and semantic and thematic information obtained from Linked Data sources.

The building geometries are displayed in the simulation; the only processing performed on them is their extraction from the COLLADA files. The extraction process is trivial and is documented in Appendix A.

5.1.1 Footprints

The building footprints are central to the information reduction methods discussed in Section 5.2.3; they are used to determine which buildings are located close to the shortest route.

As discussed in section 2.1, COLLADA assets are stored in libraries. Each library contains a different type of asset e.g. reference to images, effects, scenes, materials, etc. The building geometries are stored in the *geometries_library*. Figure 5.1 shows its location in the COLLADA tree.

A building consists of three types of structures: *footprint*, *facade* and *super*. COLLADA geometries are triangulated, hence each structure is described by a collection of triangles. There are three type of triangle collections that correspond to the three structure types. Listing 5.1 shows the geometry definition of a single building. Each building has a collection of sources (lines 4 and 7) that hold, amongst other, the triangle vertex coordinates and texture coordinates. The triangles themselves are stored in typified *triangle* elements (lines 11 and 16). There are as many triangle collections as the structures of a

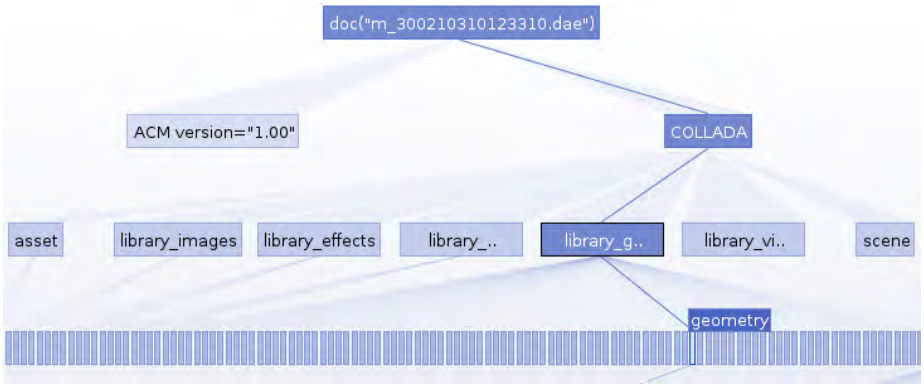


Figure 5.1: The COLLADA libraries

building. The footprints geometry is contained in the triangle element with the *material="footprint" property*. COLLADA does not define geometry explicitly, but uses a referencing scheme (Appendix A).

```
1 <library_geometries xmlns="http://www.collada.org/2005/11/
      COLLADASchema">
2     <geometry xmlns="http://www.collada.org/2005/11/
          COLLADASchema" id="ground42242-geometry" name="
          ground42242-geometry">
3         <mesh>
4             <source id="ground42242-geometry-position">
5                 <float_array id="ground42242-geometry-position
                      -array" count="24">-6.28 5.19 0.06 -11.30
                      -0.88 ...</float_array>
6             </source>
7             <source id="ground42242-geometry-uv">
8                 ...
9             </source>
10            ...
11            <triangles material="footprint" count="2">
12                <input semantic="VERTEX" source="#ground42242-
                      geometry-vertex" offset="0"/>
13                <input semantic="category" source="#
                      ground42242-geometry-category" offset
                      ="1"/>
14                <p>2 0 1 0 0 0 0 0 3 0 2 0</p>
15            </triangles>
16            <triangles material="tex_300210310123310_0001"
                  count="8">
17                ...
18            </triangles>
19                ...
20        </mesh>
21    </geometry>
22    <geometry>
23        ...
24    </geometry>
25    ...
26 </library_geometries>
```

Listing 5.1: Footprint triangle definition

Extracting the footprints boils down to retrieving the triangle vertices stored in the *float_array* on line 5 and their pointers contained in the <*p*> element on line 17. The XPath expressions that extracts these from the COLLADA file are given as:

```
/c:COLLADA/c:library_geometries/c:geometry[@id="ground42242-
    geometry"]/c:mesh/c:source[@id="ground42242-geometry-
    position"]/c:float_array/text()

/c:COLLADA/c:library_geometries/c:geometry[@id="ground42242-
    geometry"]/c:mesh/c:triangle[@material="footprint"]/c:p/
    text()
```

Where *c* is a shorthand for the COLLADA XML namespace `http://www.collada.org/2005/11/COLLADASchema`. The first line extracts vertices that make up the triangle, while the second line extracts the pointers to their values.

The triangles are then dereferenced,their coordinates are transformed to the Dutch Rijksdriehoek (de Bruijne et al., 2005) coordinate system (Appendix A) and stored as Polygons in the database as:

```
INSERT INTO footprints(building_id, geom) VALUES ('ground42242
    +300210310123310', GeomFromText(triangle_geom));
```

Where a building's id is a concatenation of its geometry id and the tile id as geometry ids are not globally unique, and *triangle_geom* is a Well-Known Text representation of a footprints triangle polygon.

The *footprints* table is extensively used during the buffer/footprints intersection process described in Section 5.2.3. Its *geom* column is therefore indexed with SpatiaLite's R*Tree index as:

```
SELECT CreateSpatialIndex('footprints', 'geom');
```

This builds a R*Tree index and stores the results in four distinct tables, three of which contain the tree itself. The first table is called *indexname*_name and stores binary data corresponding to each tree node. The table *indexname*_parent stores the tree nodes hierachy i.e. the relations between the parent and child nodes. The table *indexname_rowid* associates each tree node to the row id of the indexed data. These are internal tables that cannot be queried.

The fourth table, *indexname*, contains the geometry ids along with their bounding box coordinates. This table is queried whenever a bounding box operation is needed with SpatiaLite's built-in *RTreeIntersects*, *RTreeWithin*, etc. functions as explained in Section 5.2.3.2.

5.1.2 Semantic data

Semantic data is extracted from the linkedgeodata.org datastore using SPARQL queries. LinkedGeoData's backend is a spatial database (PostGIS); one is thus able to perform well-known spatial operations on the linked data. Extracting all the information around Rotterdam is done as

```
PREFIX lgdo: <http://linkedgeodata.org/ontology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

Select ?geo ?type ?label {
  ?n a lgdo:Node .
  ?n rdfs:label ?label .
  ?n lgdo:directType ?type .
  ?n geo:geometry ?geo .
  { Select ?n {
    ?n geo:geometry ?g .
    Filter(bif:st_intersects(?g, bif:st_point(4.479166508,
        51.92166519), ...
        2.527677297573578)) .
    Filter(bif:st_x(?g) > 4.3567 && bif:st_x(?g) < 4.5750 &&
        bif:st_y(?g) ...
        > 51.8335 && bif:st_y(?g) < 51.9962) .
    }
  }
}
```

Listing 5.2: SPARQL query that fetches all objects that have a *type* property and are close to Rotterdam.

This query retrieves all objects from the data store that has a *type*, a *label* and a *geometry* property and resides close to Rotterdam's center the latitude/-longitude position 4.479, 51.92. The result is a list of points that denote the locations of all buildings alongside their OpenStreetMap and Wikipedia type and label designations as JavaScript Object Notation (Listing 5.3. This information is stored in the database as:

```
INSERT INTO landmarks_linkedgeodata(type, label, geom)
VALUES ('FastFood', 'La Luna', GeometryFromText('POINT(4.46043
    51.9265)', 4326));
```

The TomTom dataset is enriched by intersecting these points with the building footprints. Currently this is done manually with QGIS.

Figure 2.8 shows the result. The enriched dataset is used in the semantic extractor as discussed in the following sections.

```
[{
  "type": {
    "type": "uri", "value": "http://linkedgeodata.org/ontology
        /FastFood"
  },
  "geo": {
    "datatype": "http://www.openlinksw.com/schemas/virtrdf#
        Geometry",
    "type": "typed-literal", "value": "POINT(4.46043 51.9265)"
  },
  "label": {
    "type": "literal", "value": "La Luna"
  }
}, {...}]
```

Listing 5.3: Results of above SPARQL query as JSON

5.2 Execution phase

In the execution phase, the PBT interprets the rules written by visualisation experts and runs the various IRP configurations. IRP configurations differ because their *selection* and *extraction* phases are run with different operators that in turn are invoked with different parameters. An example IRP configuration is one that is run with a 50 m *single* selector coupled to a *textured* extractor, while another is one that is run with a 40/80 m *double* selector that is coupled to a *colored* extractor. To this end two controllers are developed that form the bridge between the spatial and semantic rules, and the functionality defined in the IRP.

This section first presents the two controllers and then moves to a discussion about the IRP itself.

5.2.1 Spatial controller

The spatial controller directs the IRP by way of an execution plan. It parses the plan and directly invokes the IRP operators in the encoded order. Execution plans are written by visualisation experts and are essentially recipes that tell the IRP exactly how to operate. This mode of operation is termed spatial as selection and styling decisions are solely based on the building's distances from the route. Listing 5.4 shows an example execution plan that results in three distinct IRP configurations.

This plan schedules the execution of a *single* selector with a width of 50 m that is coupled to three distinct extractors. Adding an additional selector is done by adding the element shown in Listing 5.5 to the *children* list on line 3 of the original plan. This adds a 100 m *single* selector piped to a *plain* extractor to the plan.

```
{
  "name": "Selector",
  "children": [
    {
      "name": "single",
      "options": {"distance": [50]},
      "children": [
        {
          "name": "Extractor",
          "options": [{}],
          "children": [
            {
              "name": "plain"
            },
            {
              "name": "colored"
            },
            {
              "name": "textured"
            }
          ]
        }
      ]
    }
  ]
}
```

Listing 5.4: An example execution plan.

```
{
  "name": "single",
  "options": {"distance": [100]},
  "children": [
    {
      "name": "Extractor",
      "options": [{}],
      "children": [
        {
          "name": "plain"
        }
      ]
    }
  ]
}
```

Listing 5.5: Extending the execution plan of *lst:exec_plan*.

5.2.2 Semantic controller

The semantic controller is implemented using the Python Knowlede Engine (PyKE) expert system. In this prototype, PyKE is used to decide which operators to execute in each IRP phase, and what parameters to execute them with. Currently, decisions are based on a building's *type* that is extracted from the LinkedGeoData and DBpedia data sources (5.1.2). Note that because the data provided by TomTom lacks a road network (and thus road semantics such as type, width, etc.), only styling decisions are made i.e. what extractor to use. Making selection decision require a road network.

In this prototype, the expert system fulfills the role of a decision system, more so than a reasoning engine. It separates the prototype's execution logic from the rules; the Python implementation details are abstracted to PyKE's structured language. Users can therefore perform experiments without needing to know the internal workings of the IRP. The alternative to an expert system is a collection of if-else statements that check the rules and call the appropriate IRP operators. However, this approach is not suited for complex rules; implementing and maintaining it quickly becomes a burden.

Section 4.3.2.1 introduced the conceptual workings of an expert system. The expert system is executed for each building separately. Telling PyKE that a building is of type *fast food* is done by creating a corresponding fact:

```
building(fast_food)
```

This statment creates a fact called *building* and gives it a *fast_food* value. This fact states that the building under consideration is of type *fast food*. PyKE collects facts in a knowledge base a.k.a. fact base and rules in a rule base. The above fact already forms a small knowledge base on which simple rules can be applied. The most simple visualisation rule one can come up with is *if building is of type 'fast food restaurant', style it with textures*.

This rule is written in PyKE as:

```
1 styling_colored
2   use style(colored)
3   when
4     building(fast_food)
```

The first line defines a new rule called *styling_colored*. The second line acts as the rule's *then* clause i.e. its conclusion, namely: *use a colored style*. The third line acts as the rule's *if* clause. PyKE evaluates this rule by checking all statements in the *when* clause against the knowledge base. If it finds a piece of knowledge that supports the statement it moves on to the next statement until it raches the last one. If all statement evaluate to "true" i.e. knowledge is found

that supports them, PyKE reaches the conclusion in the *use* clause and adds it to the knowledge base.

In the example case above, PyKE searches for knowledge supporting the *use* clause and finds it in the form of the *building(fast_food)* fact and concludes that the *style* must be *colored*. Invoking this rule in Python is done by calling the PyKE *prove_goal()* function as:

```
1 add_case_specific_fact('building', 'fast_food')
2 prove_goal('style($style)')
```

Here, the first line adds the building type to the knowledge base, the second line starts the decision process. The result of *prove_goal* is the string "colored", which means that this particular building should be styled with the *colored* operator. A call to the IRP's extractor is made accordingly.

A more involved rule is stated as *display all fast food restaurants in their dominant colors, except if they are near a junction.* A new rule is added to the rule base as:

```
1 styling_textured_junction
2     use style(textured)
3 when
4     building(fast_food)
5     junction(yes)
```

Listing 5.6: Styling fast food restaurants with textures when near a junction.

This rule is an extension of the first rule: the last line checks whether the building is located near a junction. Knowledge about whether a building is near a junction comes from the *junction* selector. This rule is executed like the previous one, but now the additional fact *junction(yes)*:

```
1 add_case_specific_fact('building', 'fast_food')
2 add_case_specific_fact('junction', 'yes')
3 prove_goal('style($style)')
```

PyKE automatically checks all rules that have identical *use* clauses and evaluates the correct one. In this case, since the fact *junction(yes)* is found in the knowledge base, PyKE evaluates the second rule that requires knowledge about the junction and reaches the *style(textured)* conclusion. If it fails to find a *junction(yes)* fact, it evaluates the first rule and reaches a *style(colored)* conclusion. Thus fast food buildings that are near junctions are styled with the *textured* extractor, while all other buildings are styled with the *colored* extractor.

Facts produced by rules can be used as conditions in other rules. Upon execution the latter are also tested so as to retrieve the corresponding fact. Consider the case where the driver wishes to temporarily lower the textural level of detail of the whole scene i.e. textured buildings become colored and colored buildings become plain. This is done by introducing a new rule that relies on the outcome of the previous *style()* rules as:

```
1 styling_textured_reduced
2     use style(colored)
3 when
4     style(textured)
5     reduced(yes)
```

Listing 5.7: A rule that downgrades the textural complexity of buildings.

Upon execution PyKE starts to evaluate the *when* clauses of the above rule. It searches the knowledge base for the fact *style(textured)* but does not find any matches. It does, however, find a rule with that name and sets out to evaluate it first and, upon a successful evaluation, continues with the next fact i.e. *reduced(yes)*.

Suppose PyKE is called as:

```
1 add_case_specific_fact('building', 'fast_food')
2 add_case_specific_fact('junction', 'yes')
3 add_case_specific_fact('reduced', 'yes')
4 prove_goal('style($style)')
```

This call extends the previous one by adding the *reduced* fact to the knowledge base i.e. an indication that the textured must be donwgraded. Executing this query makes PyKE find proofs for all *(*style($style)) rules. The result is an ordered list of rules that validate given the provided facts. In this case, the *styling_textured_junction* (Listing 5.6) rule evaluates and validates first as it matches the first two facts i.e. *building(fast_food* and *junction(yes)*). It thereby adds the fact *style(textured)* to the knowledge base. The rule of interest i.e. *styling_textured_reduced* (Listing 5.7) is evaluated and valiadte next as all the needed knowledge is available. The first item in the list contains the result of the *styling_textured_reduced* list as it evaluated last.

The semantic controller thus controls the IRP by deciding which extractor opertor to call based on visualisation rules that incorporate thematic, spatial, driver-derived, etc. information.

5.2.3 IRP

The main Information Reduction Pipeline functionality is selection and extraction of building objects and painting these in a certain style. The IRP defines

three operators that perform these tasks: a selector, an extractor and styler, and a visualisation preparer. It takes a 3D scene and a shortest route as input and returns a reduced scene that contains only buildings close to the road that are styled according to visualisation rules.

The IRP execution consists of the following steps:

- receive and prepare shortest route
- segment route in segments of user specified length
- for each segment: execute *selector* and *extractor* operators, and create visualisation asset i.e. a geometry file

Its output is a 3D geometry file, in this case a Wavefront OBJ file (see Section 5.12), that is visualised in Panda3D.

5.2.3.1 Route preparation

The IRP starts immediately after it receives the shortest route between the user's current location and desired destination. Conceptually, this route comes from the navigation device. However, since the prototype is implemented on a desktop computer, the shortest route is calculated with Google Maps' Direction Service [1] (Figure 5.2). The Directions Service calculates the shortest route between two points on Google's road network and returns the solution in a KML file. This file is parsed with *lxml*, the coordinates of the therein contained route are transformed to the Dutch coordinate reference system using *pyproj*.

Figure 5.2: An example of a shortest route calculation as performed by Google's Direction Service.

[1] https://developers.google.com/maps/documentation/directions/

Next, the route is segmented using SpatiLiate's *ST_Line_Substring* function as:

```
SELECT ST_Line_Substring(ST_GeomFromText(geometry, 28992),
    start_fraction, end_fraction));
```

Listing 5.8: SQL line segmentation query

where *geometry* is a Well-Known-Text representation of the route as extracted from the KML file. The resulting segments are stored in the *segments* view and are used as input for the first IRP phase. Each of the IRP phases is executed at least once on each segment.

5.2.3.2 Selector

The selector is the first IRP phase; its task is to identify buildings that are close to the shortest route. The selector is implemented as an intersection between a 2D buffer of the route and the 2D building footprints. As discussed in Section 3.2.1.1, the selector defines three operators. The *single* and *double* operators are explained first as their implementation is similar. The *junction* operator is discussed afterwards.

Single and Double operators The selector receives a route segment as input in the form of a LineString object. The selection process consists of four steps:

1. buffer route segment
2. create buffer bounding box
3. intersect buffer bounding box with building footprints
4. intersect result of 3 with the buffered segment

The first step is calculating a buffer around the segments using *ST_Buffer* as:

```
SELECT ST_Buffer(geom, ?) FROM segments WHERE id=1;
```

Listing 5.9: Buffering of a route segment

The so calculated buffer is stored in the *buffers* view. Directly intersecting the calculated buffer with the building footprints is not efficient as *ST_Intersect* does not use the spatial index. The spatial index is used only when performing bounding box operations. SpatiaLite supports four bounding box search operations, namely *within*, *contain*, *intersects* and *distance within*. Each has a corresponding function that takes the two extreme bounding box coordinates that form the search area. Upon execution, the function searches through the

index for all geometries that intersect it, and returns a list of their ids. The following query retrieves all buildings that intersect the buffer's bounding box.

```
SELECT DISTINCT id, footprints.geom, FROM footprints
WHERE id IN (SELECT pkid FROM idx_footprints_geom, buffers
WHERE buffers.id=1 AND idx_footprints_geom.pkid
MATCH RTreeIntersects(MbrMinX(buffers.geom), MbrMinY(buffers.
    geom), MbrMaxX(buffers.geom), MbrMaxY(buffers.geom)));
```

Listing 5.10: Boundig box intersection using SpatiaLite's R*Tree index

The result of this query is stored in a view called *footprints_selected*. Here, *idx_footrprints_geom* is the queryable spatial index table mentioned in Section 5.1.1 that contains the footprint's bounding boxes (Figure 5.3.

	pkid	xmin	xmax	ymin	ymax
1	68139	86283.703125	86305.617188	434992.125000	435026.062500
2	68140	86287.718750	86324.601562	434992.125000	435028.531250
3	68145	86287.718750	86354.460938	434992.125000	435028.531250
4	68138	86287.718750	86357.687500	434967.125000	434992.125000
5	68143	86287.718750	86357.687500	434981.968750	435006.968750
6	68137	86290.953125	86316.101562	434942.125000	434967.125000
7	68141	86290.953125	86335.070312	434944.937500	434967.125000
8	68146	86290.953125	86357.687500	434947.406250	434981.968750
9	68144	86324.601562	86354.460938	435006.968750	435031.968750
10	68142	86335.070312	86361.710938	434947.406250	434981.968750

Figure 5.3: The COLLADA libraries

In the final step, the footprints that came from the previous query are intersected with the buffer itself as:

```
SELECT building_id, ST_Intersects(buffers.geom,
    footprints_selected.geom) AS intersects
FROM buffers, footprints_selected
WHERE buffers.id = 1 AND intersects = 1 AND building_id
NOT IN (SELECT id FROM footprints_result)
```

Listing 5.11: SQL query intersecting the building footrpints with a buffer selector

This is the final result i.e. the buildings that are within the specified distance from the route. These results are returned to Python and passed on to the next IRP phase.

The buffers of two consecutive segments overlap; some buildings will therefore be selected twice (Figure 5.4. To prevent this, a list of already selected buildings is kept in the *footprints_result* table. Newly selected buildings are that are contained in this list ared dropped.

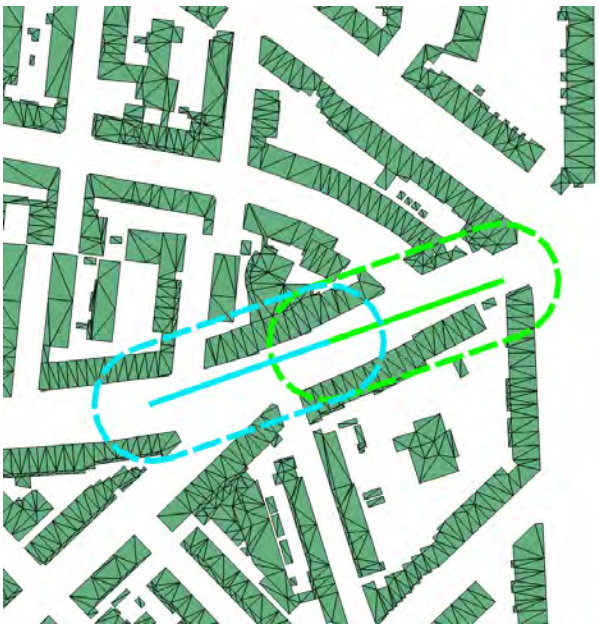


Figure 5.4: A *sized* selector with two distinct themes.

The *double* selector is implemented much like the *single*. The only difference is that the last step is executed twice i.e. once for the inner buffer and another for the outer buffer. Note that in this arrangement, the ordering of the buffers is important: applying the large buffer first will prevent the smaller buffer from selecting anything since the *footrpints_result* table will be populated with the larger buffer's results. This is automatically taken care of in the Python implementation.

Junction operator The *junction* operator is implemented as a proof-of-concept as there the provided dataset lacked a road network. Junctions locations are therefore handpicked from OpenStreetMap and inserted into a temporary table called *junctions* as points.

Processing junctions is five step process:

1. find junctions near the route with bounding box queries
2. apply a buffer of 5 m to junction
3. intersect junction buffer with route to determine correspondence

4. apply buffer of desired size

5. intersect with building footprints

The first step finds potential junctions by intersecting all junctions with a bounding box of the route, thereby using the database's spatial index.

```
SELECT id, geom from junctions
WHERE id IN (SELECT pkid FROM idx_junctions_geom, route
WHERE pkid MATCH RTreeIntersects(MbrMinX(route.geom), MbrMinY(
    route.geom), MbrMaxX(route.geom), MbrMaxY(route.geom)));
```

This result is stored in a view called *nearby_junctions*. The second step buffers the junction points in order to account for mistmatches between the junctions dataset and the road network. The third step intersects the buffer with the route as:

```
SELECT id, ST_Intersects(ST_Buffer(nearby_junctions.geom, 5),
    segments.route) as intersects from nearby_junctions,
    segments
WHERE intersects=1 AND nearby_junctions.id AND id NOT IN (
    SELECT id from processed_junctions);
```

Since buffers may overlap, the danger exist that a junction intersects two buffers, thereby selecting buildings twice. To prevent this, processed junctions are stored in the temporary *processed_junctions* table. The result of this query is stored in a view called *selected_junctions* Once the junctions that intersect the route are found, the desired buffer size is applied and intersected with the building footprints that fall within the route's bounding box as:

```
SELECT building_id, ST_Intersects(ST_Buffer(selected_junctions
    .geom, <size>), footprints_geom) as intersects from
    selected_junctions, footprints
WHERE intersects=1
AND footprints_id NOT IN (SELECT id FROM footprints_result);
```

where <*size*> specifies the desired buffer size. The results of this query are added to the *footprints_result* table.

When combining the *junction* selector with other selectors care should be taken to combine them in the correct order. *Junction* selectors need to be applied first or they will not achieved the desired result. For instance, applying a *single* buffer before a *junction* buffer will leave a stroke of *single* buildings running right through the *junction* selector.

5.2.3.3 Extractor and themer

The extractor's task is to retrieve the 3D geometries of buildings that are selected by the selector and theme them. The theming consists of fetching corresponding pieces of information from the database such that the visualiser described in the next section can prepare the geometries for visualisation. Section 3.2.2 defined three operators that effectively result in three themes, namely *plain*, *colored* and *textured*.

The three operators first extract the building's 3D geometry and, in the case of the *colored* and *textured* operators, augment it with other properties.

```
SELECT type, ST_AsText(xyz) FROM triangles join coordinates ON
    triangles.id = triangle_id
WHERE building_id=<id> AND tile_id=<id>;
```

Note that the *tile_id* is passed to this query as building ids are not unique across tiles. The *plain* operator is the most simple of all: it simply fetches the geometry by executing the above query and returns the result. The *colored* operator returns, next to a building's geometry, also a color for the roof and facade (Figure 5.5a). This information is extracted from the *extra* field of each building as:

```
SELECT id, wall_color, roof_color FROM buildings
WHERE id = <id> AND tile_id=<id>;
```

The *textured* extractor also extracts information about the building's texturing (Figure 5.5b). The textured extractor consists of two statements: an alternative geometry extraction one and one filling the buildings geometry.

```
SELECT texture FROM buildings WHERE id=<id> AND tile_id=<id>;

SELECT type, ST_AsText(xyz), uv FROM coordinates JOIN
    triangles ON triangle_id = triangles.id where building_id
    = <id>;
```

5.2.3.4 Preparation for visualisation

The last phase of the IRP is responsible for encoding the information retrieved from the database into a format that is readable by Panda3D. The format of choice is Wavefront as it is a lightweight plain-text format. It is easy to create and is recognized by many visualisation softwares. Wavefront supports storage of materials and their properties. These are stored in an auxillary file with an .mtl exension. Materials are defined by specyfing a set of properties such as

ambient color, diffuse color, specular color, etc. The location of texture libraries are also stored in the materials file.

Listing 5.12 shows an excerpt from a Wavefront .obj file. A Wavefront file first lists the geometry's vertices. Each vertex is preceded with a *v*, followed by a coordinate triplet. Vertex normals are stored next and are preceded by a *vn*. Faces are formed by reffering to individual vertices, normals and texture coordinates by their location in each of the three lists. For instance, line 18 creates a face that consits of three vertices; the first vertex' coordinates/normals/texture coordinates are located in the 10th/1st/1st location in each of the lists.

```
1  mtllib /home/simeon/tomtom/data/tile_300210310310130.mtl
2
3  v 91407.056566 437048.018263 -1.1
4  v 91406.336566 437041.448263 11.9
5  v 91406.336566 437041.448263 -1.1
6
7  ...
8
9  vn -0.3300 -0.9440 0.0
10 vn -0.3300 -0.9440 0.0
11 vn -0.3300 -0.9440 0.0
12 ...
13
14 vt 0.7676 0.4443
15 vt 0.7637 0.4404
16 vt 0.7549 0.4951
17 ...
18
19 usemtl tex_300210310311013_0023
20 f 10/1/1 48/3/1 11/2/1
21 f 10/1/2 47/4/2 48/3/2
22 f 11/5/3 49/7/3 12/6/3
```

Listing 5.12: An example Wavefront file

The *mtlib* command on line 1 specifies the location of the auxillary material file. The *usemtl* command on line 17 specifies the material of following faces. This material is used until a new *usemtl* directive is found.

This IRP phase produces a Wavefront file for each individual segment and IRP configuration. A route of seven segments with two IRP configurations thus produces 14 files.

5.2.4 Example IRP results

This section shows some example IRP configurations.

Figure 5.5 displays the result of a *colored* and *textured* extractor.

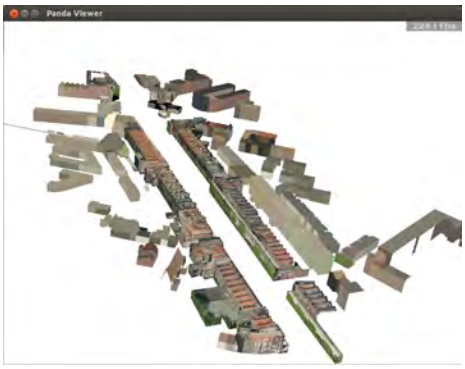
(a) The result of a *colored* extractor.

(b) The result of a *textured* extractor.

Figure 5.5: Raw *colored* and *textured* extractors.

Figure 5.6a displays the result of a 100/30 m *sized* selector piped to the three types of extractors. The result of these configurations is that buildings that are closer than 30 meters to the road are are styled in a richer theme than buildings further away. The second tier of buildings is 100 m away form the road.

(a) The result of a 100/30 m *sized* selector piped to a *colored* and *plain* extractor (white buildings).



(b) The result of a 100/30 m *sized* selector piped to a *textured* and *colored* extractor (outer buildings)

Figure 5.6: The results of a *sized* selector.

Figure 5.7 displays the result of coupling a 100 m *junction* and 30 *single* selector to the three types of extractors. The result of these configurations is that buildings close to junctions are styled in a richer theme than buildings not within 100 m of a junction.

(a) The result of a 100 m *junction* selector and a 30 *single* selector piped to a *colored* and *plain* extractor.

(b) The result of a 100 m *junction* selector and a 30 *single* selector piped to a *textured* and *colored* extractor.

Figure 5.7: Results of the *junction selector*.

5.3 Visualisation, simulation and benchmarking

The PBT framework's goal is to enable visualisation designers to quickly experiment with, and test and evaluate different ways of visualizing 3D data for navigation purposes. To this end a simulation is built that visualises the numerous IRP configurations and collects performance measures. It also facilitates the assesment of a visualisation's cognitive effectivenes by mimicking the display of a real world navigation device: a moving viewport is implemented that creates an impression of a vehicle that travels through the digital city. Vehicle motion is simulated by making the camera drive over i.e. move on top of, the shortest route as calculated by the device and is used to select the buildings. While the simulation runs it collects the afore mentioned performance indicators discussed in Section 4.4.2. Performance metrics are recorded by Panda3D's PStats 5.8 facility and graphed using the Python matplotlib graphing library.

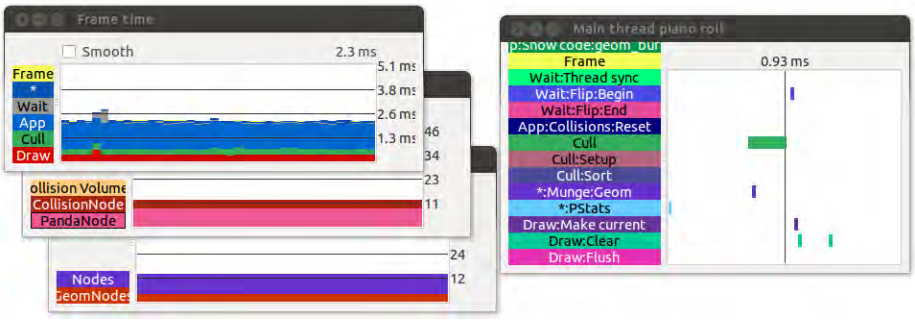


Figure 5.8: A selection of Panda3D's benchmark information streams.

The simulation is also aimed at exploring various data loading strategies. This is an interesting issue as mobile devices are limited in available virtual memory. This imposes limits on the amount of data the device can visualise at once. Seeing that TomTom's data exceeds 7 GB it becomes evident that it cannot be loaded all at once, but needs to be sliced and visualised in pieces. A sensible approach is to load and display data only when it is needed and unload it immediately when it becomes useless.

In the current case this means loading and displaying information that is "in front of the driver" and unloading it once the driver passes it. Reading information from a hard disk is, however, a slow operation that will disturb the visualisation if it takes too long to complete. A balance must thus be found between the number of disk reads, the amount of information contained in each read and the available virtual memory.

To explore these issues a "on-the-fly" data loading strategy is implemented i.e. geometries are loaded as soon as they are needed, not before. Data loading events are triggered by Panda3D's collision detection system. The collision system allows one to create so-called collision objects that fire collision events whenever they intersect.

In the current implementation one such collision object, namely a sphere, is placed on the virtual moving vehicle and another at the center of each geometry region or tile. The first sphere has a diameter of 10 units, whereas the second is set to 800-1000. The second diameter is chosen to be much larger than the extent of the geometry region itself. As the vehicle moves around the three-dimensional world it will collide with the spheres of the tiles thereby triggering collision events, but since the spheres are much larger than the tiles the collisions will occur well outside the visibile range. The scene thus appears continous to the user, when in reality it is not.

The geometry loading strategy is defined as

```
1 load tile 0
2 current_tile = 0
3 next_tile = current_tile + 1
4
5 load geometry of current_tile
6
7 add collision sphere at location of next_tile
8 start motion
9     upon collision with next_tile
10    load geometry of next_tile
11    current_tile += 1
12    next_tile += 1
13    place a collision sphere at the location of next_tile
```

Listing 5.13: Geometry loading using Panda3D's collision spheres

Early experiments revealed that loading geometry in memory and immedi-

Figure 5.9: A top-down display of the the collision spheres used for loading information in the game engine. The larger of the two sphere is the one triggering the loading of a tile, while the smaller triggers its visualisation.

ately displaying it costs too much resources. A better solution is to first load the geometry in memory and display it a couple of frames later. To facilitate this an extra sphere is introduced: one that triggers the loading of data, and one that triggers the display of data. Figure 5.9 shows an example of this set up.

Unloading tiles is performed on the same principle: the collision system triggers a new collision event when the camera leaves the tile's collision sphere.

Controlling the size of the collision spheres as well as the tiles allows one to control the amount of information that is transported around the system. Bigger spheres result in more information as more tiles are loaded at once which directly translates into a larger virtual memory usage. The tile size determines the amount of information that is loaded per load action. Bigger tiles result in less loads, but longer load times.

The simulation ends when the virtual vehicle arrives at the end of the route.

Chapter 6

Use case and PBT results

This chapter demonstrates the workings of the IRP and its effects in a concise and graphical manner by going through the execution steps of several visualization configurations in a step-by-step manner. The effects of the exemplified IRP configurations are visualised and explained by applying them a simple route and comparing their outcomes. The content of this chapter sits in between the conceptual discussion of Chapters 3 and 4, and the technical implementation discussion of Chapter 5.

The chapter starts with a step-by-step explanation of the workings of an IRP configuration which uses a *single* selection buffer with varying parameters, and two different extractors, namely *colored* and *textured*. Section 6.2 presents the results of running this IRP configurations in the simulator.

6.1 IRP operation

This section aims to explain the workings of each IRP stage in a graphical and step-by-step manner. The technical implementation details are discussed in the previous chapter, the purpose of this section is to show the effects of invoking the IRP phases with different parameters. The section begins with an example IRP configuration that uses a *single* selector with width 50 m together with a *colored* and *textured* extractor, and ends by comparing it to configurations which have *single* selector widhts of 10, 25 and 80 m.

The Information Reduction Pipeline starts operating immediately after the car navigation device calculates the shortest route between the driver's current position and his desired destination. For illustration purposes, this example runs the different IRP configurations on a simple, short and straight route that has a length of 870 m. Figure 6.1 shows the test route (blue) overlayed on top of OpenStreetMap (black lines) and the COLLADA footprints (green). The OpenStreetMap data is retrieved through a QGIS plugin and is here used as a visualisation aid. In this case the driver travels from the right edge of the image

to left.

Figure 6.1: The test route on top of OpenStreetMap and the COLLADA footprints.

The first IRP phase is selection. In this example configuration a *single* selector that has a width of 50 m is used.

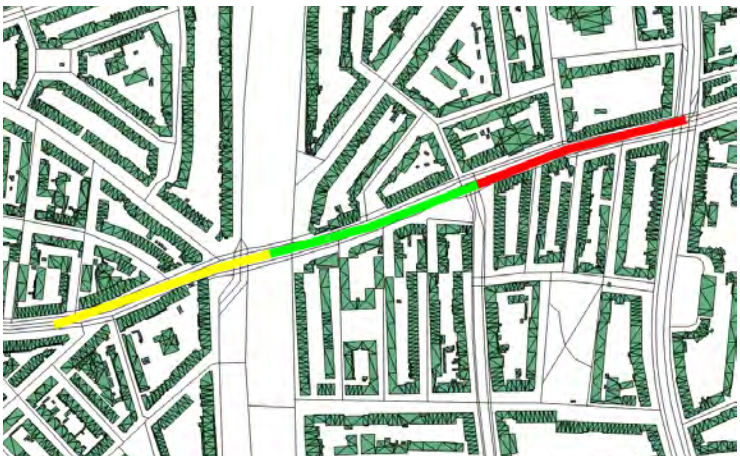


Figure 6.2: Segmented route

The selector's first action is to segment the route in segments of a certain length. In this first example, the route is split in three segments. The first two having a length of 300 meter, while the third has a length of 270 m. Figure 6.2 shows the segmented route. The segment length is a design parameter that is chosen by the map maker. Note that at this point it is unknown what the effect of this length will be on the performance of the simulation. A long segment

contains more information that a short one which may stall the simulation during the geometry loading process. Section 6.2 investigates this issue further.

The selector then proceeds to find which buildings are within 50 meters of the route. As discussed in previous chapters, this is performed by calculating a 2D buffer around the route and intersecting it with the building footprints. The sheer number of building footprints prevents us from directly intersecting them with the buffer as that would take too much time. Therefore a preselection of buildings is made that are located in the vicinity of the segment and its buffer.

The buffer/building intersection process consists of four steps. First, a 50 m buffer is calculated around the first segment (Figure 6.3a). Next, a boundig box is calculated around the buffer as shown in Figure 6.3b. The third step intersects the buffer's bounding box with the building footprints. This uses the database's spatial index and is therefore fast and efficient. The result is a selection of footprints that fall within the buffer's bounding box ((Figure 6.4a). The number of footprints that have to be intersected with the buffer is reduced from 258093 (total) to 802.

(a) The first segment (yellow) is buffered (red dashed line) with a 50 m buffer.

(b) The black rectangle represents the buffer's minimal bounding box.

Figure 6.3: Preparing the buffer for intersection calculations with the building footprints.

In the last step the buffer itself is intersected with the remaining footprints, shown in yellow in Figure 6.4b. The first segment is processed. The building id that belongs to each footprints is stored in a list and passed to the next IRP phase.

The selection procedure is repeated for all the segments of a route. Figure 6.5 shows the buffer of the second segment in green. Note that buffer segments overlap, which means that buildings will be selected twice. This is prevented by keeping a list of buildings that are selected in the previous segment and discarding them from the current selection.

The result of the total selection operation is a list of buildings that are within 50 meter of the buffer (Figure 6.6).

(a) The result of the bounding boxes intersection: the blue footprints intersect the bufer's bounding box.

(b) The yellow buildings intersect the buffer, they are thus within 50 meters of the route.

Figure 6.4: The last two steps of the building selection process.

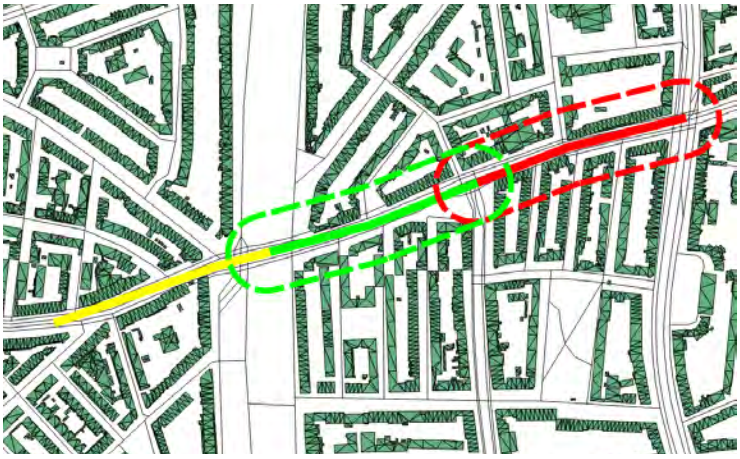


Figure 6.5: The second segment's buffer.

Figure 6.6: The result of the selection phase: all buildings that are within 50 meters of the route are shown in yellow.

The next IRP phase is extraction. In this phase the 3D building geometries are extracted from the database and styled with either a pre-defined simple color scheme, their dominant color or their textures. The modularity of the IRP allows one to run one selector type and visualise it different ways by piping the result of the first IRP phase to different and distinct extraction phases.

In this example, the *single* 50 m selector above is piped to a *colored* and *textured* extractor. In the first case, the extractor retrieves a building's 3D geometry and its facade and roof color as stored in the database. In the second case, the extractor retrieves a buildings' 3D geometry, the location of its texture file and the u, v coordinates that map the texture to the geometry. The extractor pases this information to the last IRP phase, Visualisation preparation, which encodes it in a render-friendly format, which in this case is Wavefront OBJ file. Since the IRP operates on segments, each segment is stored in a separate file. This example produces three files as the route is split in three segments.

Choosing for a 50 meter wide selector is at this point an arbitrary decision. At the same time, choosing an optimal selector size is key to the whole process. A selector that is too small will not select enough buildings making the visualisation useless to the driver as it will contain few navigation cues. A selector that is too big is a waste as it may select too much information that unnecessarily burdens the device, or worse, the driver.

Finding an optimal buffer size is done by comparing several different sizes to one another and inspecting their cognitive and technical effectivity. Figure 6.7 shows a collection of selectors all having different sizes. To make the figure more legible, the buffers are applied to the whole route and not its segments.

Clearly, the 10 m selector (shown in red) is not useful as it selects too

Figure 6.7: A number of *single* selectors with different sizes: 10 m (red), 25 m (yellow), 50 m (light-blue), 80 m (green). The blue line represents the route.

little information. The 25 meter selection (shown in yellow) is somewhat better although it fails to select enough buildings in the vicinity of junctions. Especially the large junction left from the center of the figure is completely ignored. The 80 meter selector (shown in green) on the other hand selects too much buildings; it selects buildings that sit behind the first row of structures. The buildings are theoretically invisible to the driver, or if visible, add little to his ability to navigate. The 50 meter selector (shown in blue) strikes a good balance: it selects only the buildings directly adjacent to the road and also manages to capture capture junctions.

Figures 6.8 - 6.10 show the result of these selectors as seen from the driver's perspective in the simulator. They confirm what is also observed from Figure 6.7: the 25 m selector is clearly not fit as it selects too few buildings, the 50 meter selector gives a good impression of the situation, and although the 80 meter selector selects more information than the 50 meter selector, most of it falls outside of the driver's viewing field and adds little to the visible part of the scene and, by extension, to the driver's situational awareness.

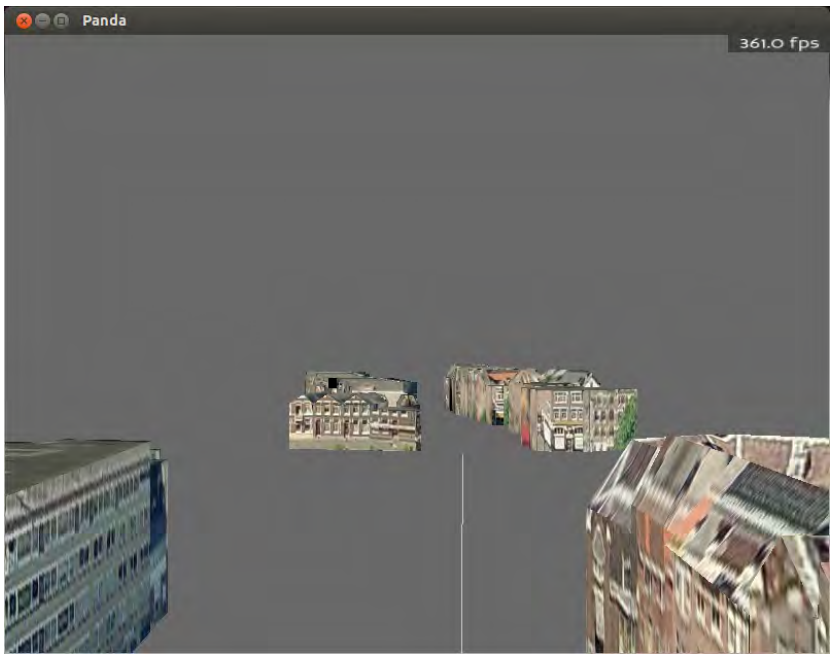


Figure 6.8: The result of a 25 meter selector.

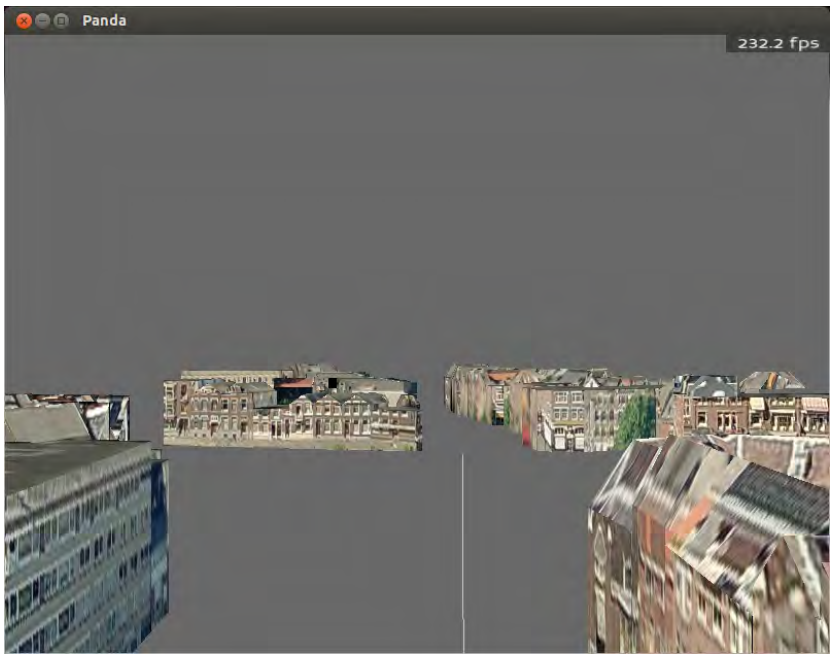


Figure 6.9: The result of a 50 meter selector.

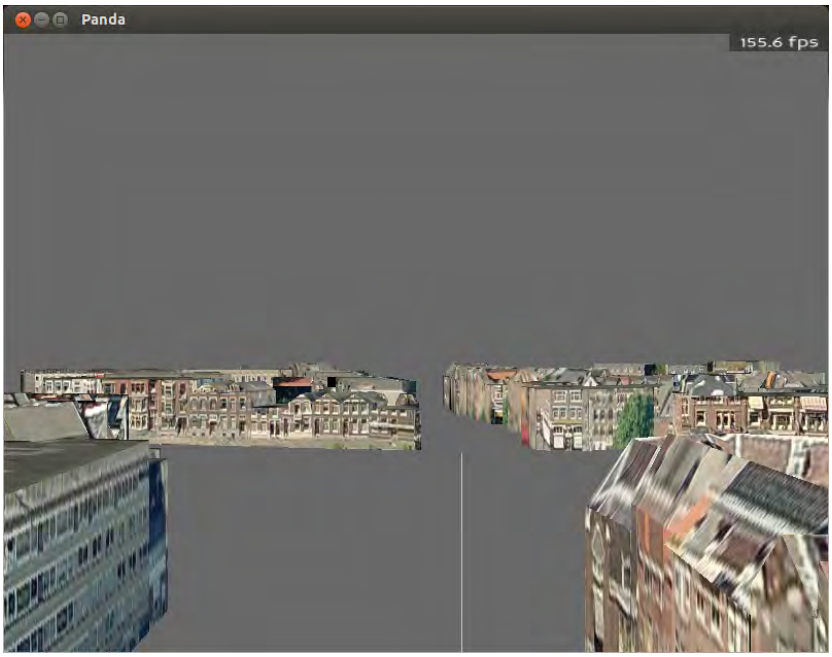


Figure 6.10: The result of a 80 meter selector.

A slightly more realistic route that involves a turn is shown in Figure 6.11. The first part of the route now visits smaller streets and it becomes clear that the 50 meter selector is too large as it selects buildings that are not directly adjacent to the road. In this case it is the 25 m selector that suffices. This need for different buffer sizes points at linking the selector size to the type and size of the road it is applied to; a bigger road warrans a larger selector. An alternative to this method is running a preprocessing method which counts the number of intersected building for each road stretch. If the number of selected buildings is below a certain threshold the selector size is probably too small and a bigger one should be selected.

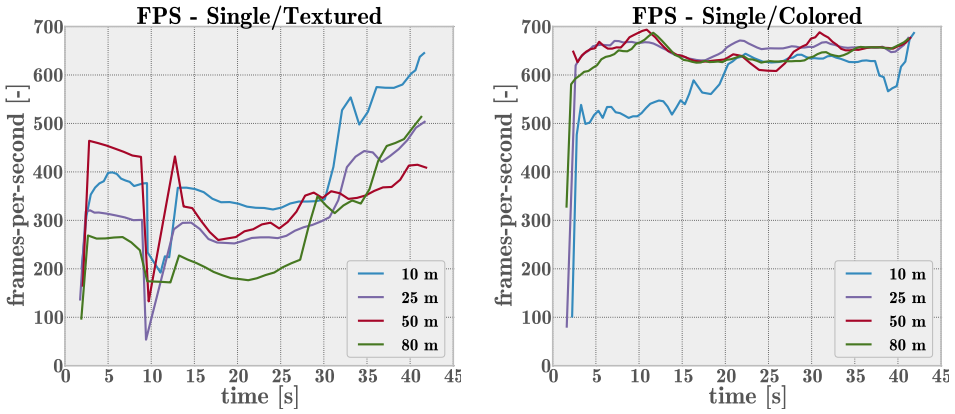
But in the end, all depends on the desires and goals of the visualisation designers, and the experiences of the drivers themselves. For instance, the case may be such that choosing the second row of buildings is desirable; that will make the 80 meter selector fit for purpose instead of the 50 meter version. These considerations have to be linked to the technical performance of each visualisation to make the visualisations effective on a cognitive and performance level.

Figure 6.11: the same situation as Figure 6.7 now including a turn.

6.2 Performance results

This section presents and discusses the performance measures of the selector/extractor combinations discussed in the previous section. The presented results showcase only results obtained by simulating a drive over the straight route. The route with a turn is not presented as it is expected that the performance indicators will be similar.

Figure 6.12 shows the frames-per-second measurements for the *textured* and *colored* styles. Figure 6.12a shows an expected pattern: the highest frame rate is obtained with the 10 meter selector when there are few objects to display. The frame rate gradually drops as the selectors increase in size: the 25 meter selector runs at an average FPS of 300, the 50 meter slightly above 300 and the 80 a little below that at 200. The graphs increase towards the end as the number of visualised buildings gradually decreases.



(a) Frames-per-seconds measurements for the *textured* extractor.

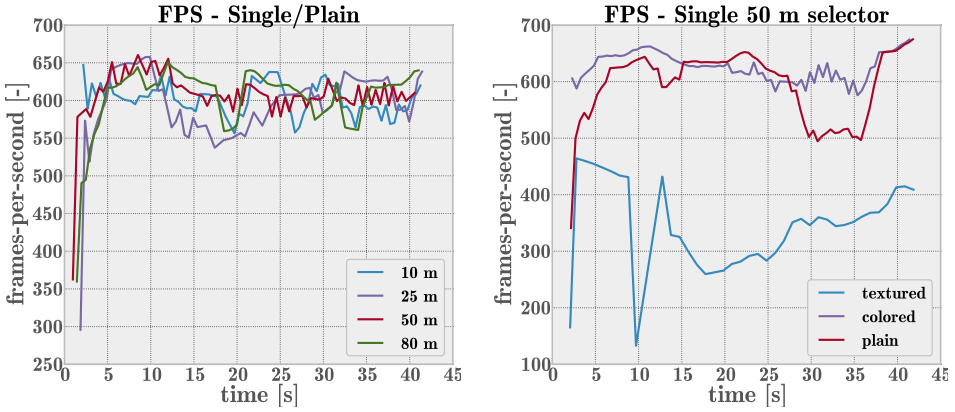
(b) Frames-per-seconds measurements for the *colored* extractor.

Figure 6.12: Frames-per-seconds measurements for the two extractors.

After ten seconds into the drive, the virtual vehicle nears the second segment. A load event is triggered and the simulation starts loading the geometry from disk. This is reflected in the drop of frame rate. Avoiding this drop in frame rate is achieved by reducing the amount of information that has to be loaded at once. This either means using a smaller selector width (observe that the 10 m case does not exhibit such a severe drop) or by reducing the segment length.

Figure 6.12b shows the frame rates of the *colored* extractor. These are, as expected, considerably higher than the *textured* case. All of them except the 10 m case hover at around 650 FPS. The 10 m graph is slightly lower in the beginning due to a temporary external load on the test machine. These

graphs suggest that reducing the amount of displayed geometry does not have a significant influence on the frame rate. To confirm this, the *colored* results are compared to the lightest and most simple extractor that styles all buildings in the same color i.e. *plain* shown in Figure 6.13a. Here too the graphs do not differ much and hover around 630 FPS; it is therefore safe to conclude that reducing the number of geometries has less impact on the frame rate than reducing the amount of textures.



(a) Frames-per-seconds measurements for the *plain* extractor.

(b) Frames-per-seconds measurements for the *colored* extractor.

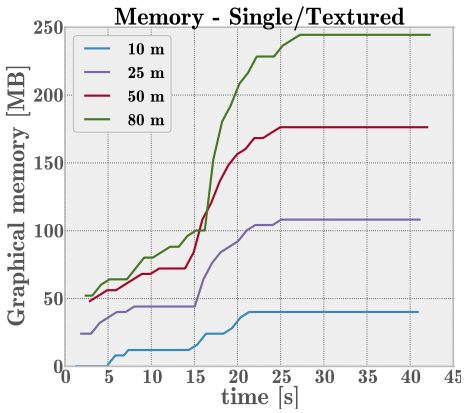
Figure 6.13

Figure 6.13b compares all three extractors to each other using a 50 meter selector. It is evident that the *textured* case is the heaviest to render as it has the lowest frame rate from all 50 meter extractors. These results show that textures do have a significant impact on the frame rate. In search of better performance, efforts should therefore be directed towards reducing the amount of textures and/or their complexity instead of the geometry.

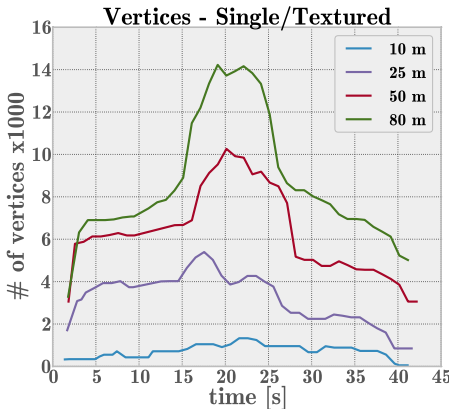
Figures 6.14 and B.4 show graphs of the graphics memory usage and the number of vertices for the *textured* and *colored* cases. Figure 6.14a shows the real impact of using a textured scene: the used memory reaches a maximum of 250 MB, compared to 0.35 MB in the non-textured case (Figure 6.15a). Reducing the selector size to 50 m yields a maximum memory usage of 180 MB. A decrease of 70 MB is significant and underlines the point made in the previous section that it is important to choose the right buffer size. These values can further be reduced by reducing the occurrence of textured buildings by showing them only when needed.

The geometry load event at the tenth second is reflected in these figures as the increase in memory usage at the 18th second as that is the point at which the loaded geometry is displayed i.e. sent to the graphics card.

The cause of the difference in maximum reported vertices between the *colored*

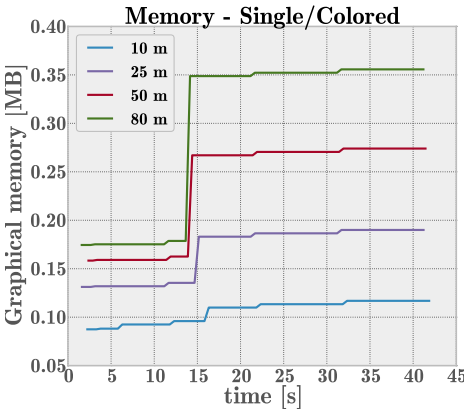


(a) Graphics memory usage of the *textured* extractor.

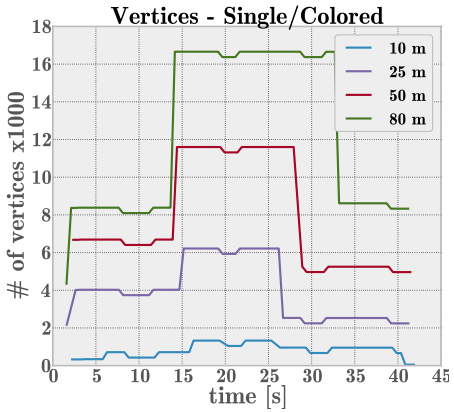


(b) Number of vertices.

Figure 6.14: Performance metrics for the *textured* extractor.



(a) Graphics memory usage of the *colored* extractor.

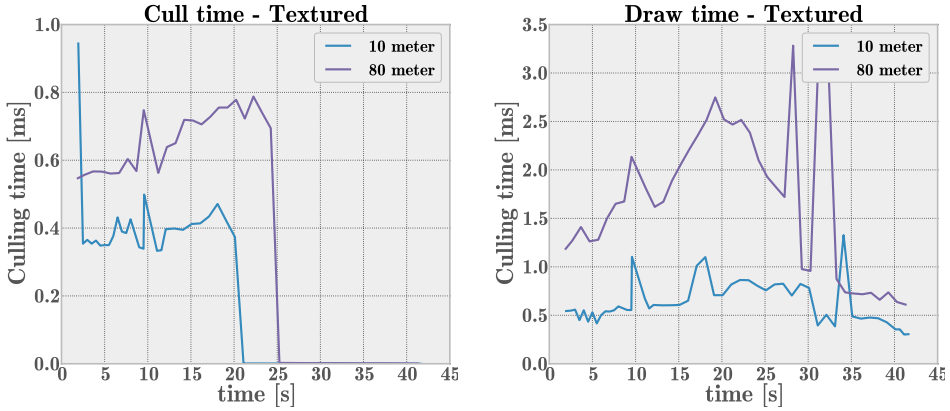


(b) Number of vertices.

Figure 6.15: Performance metrics for the *colored* extractor.

and *textured* case is difficult to pinpoint. Panda3D transforms the Wavefront files to an internal binary format and performs a number of undocumented optimizations that may explain the differences.

Figure 6.16 show graphs depicting the time the renderer requires to draw a frame and determine which geometries are visible from the driver's position (this is known as culling). These results rhyme with the previous findings. Drawing more geometries in a textured scene takes longer to cull (Figure 6.16a) and draw (Figure 6.16b) a scene.



(a) Time required to determine which geometries are visible.

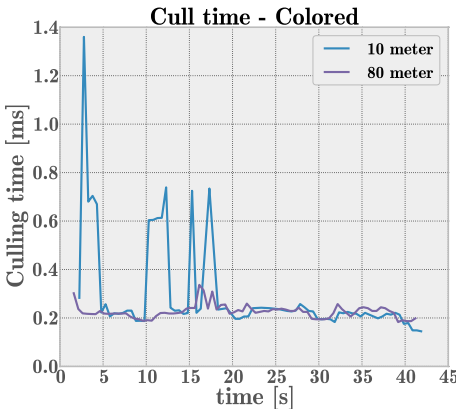
(b) Time required to draw a scene.

Figure 6.16: Performance metrics for the *textured* extractor.

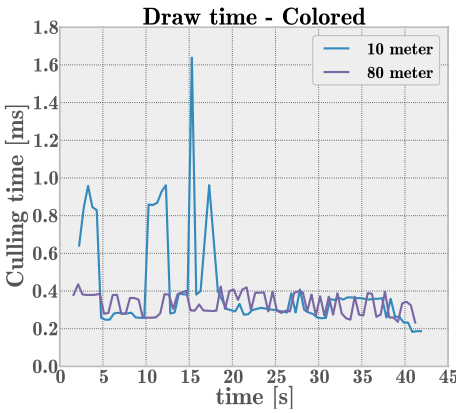
The *colored* extractor cull and draw time measurements are shown in Figure 6.17. The spikes at the beginning of the graph are caused by the engine starting up and displaying the first geometries. The graphs near the 10 s mark are caused by the loading of the second segment. It is, however, unclear why the 10 m case which contains less geometries has higher peaks. The cull and draw times stabilize after the 20 s mark and hover around 0.2-0.3 ms which is, despite the spikes, lower than the *textured* case.

Decreasing the segment size also has the anticipated effect on the performance of the simulation. Figure 6.18 shows the performance indicators of a *textured* extractor combined with the 300 m selector shown earlier (Figure 6.12a), and a *single* selector with a width of 100 m. The 100 meter case (Figure 6.18b) does not exhibit the huge frame drop of the 300 meter case. The cost of this is, however, a lower average frame rate as the engine is constantly reading data from disk which is a slow operation.

The remaining 100 m results are quite similiar to their 300 m and can therefore be found in Appendix B.

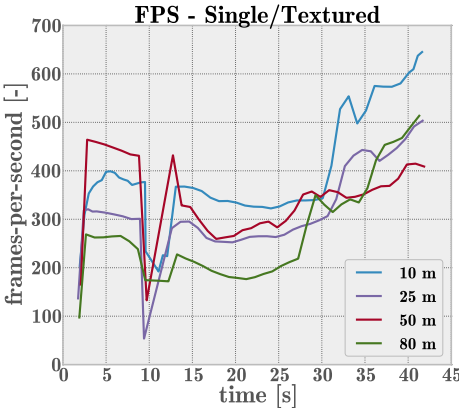


(a) Time required to determine which geometries are visible.

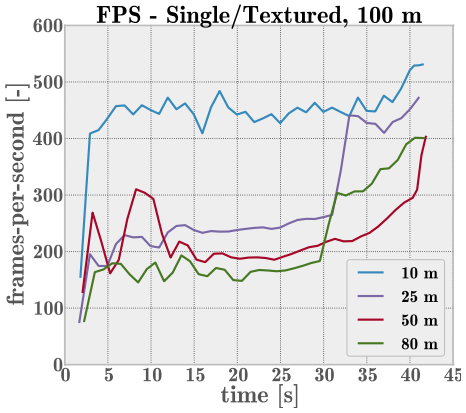


(b) Time required to draw a scene.

Figure 6.17: Performance metrics for the *textured* extractor.



(a) Frames-per-seconds measurements fot the *textured* extractor using 300 m segments.



(b) Frames-per-seconds measurements fot the *colored* extractor using 100 m segments.

Figure 6.18: Frames-per-seonds measurements for the two extractors.

Chapter 7

Conclusion

This thesis presents an information reduction and prototyping framework that reduces the information contained in three-dimensional city models so a to enable their loading and display on car navigation devices where they aid drivers to navigate better. The research is subdivided in two parts. In the first part a method is designed and implement that reduces the information contained in three-dimensional city models by extracting only information that is valuable to drivers. In the second part a prototyping framework is designed and implemented that automates and smartens the developed information reduction method such that it can be used to explore different information selection schemes and find an optimal one.

The main information reduction method consists of two steps. First, a selection of buildings is made that are located close to the driver's route. The rationale at play is that drivers are only aided by 3D information that is directly related to their current navigation task. Second, the selected building's textural level of detail is adapted to reflect their significance in the navigation process. Buildings that do no aid the navigation process are styled in a lower textural level of detail.

The prototyping framework automates and smartens the information reduction method's execution thereby turning it into a prototyping tool that is used to find an optimal information selection scheme. Different selection and styling schemes are run in a simulator that captures their performance characteristics.

In this research, the information reduction method is applied to a three-dimensional model of Rotterdam. Four selection distances (10 m, 25 m, 50 m and 80 m) combined with textured and non-textured styling of buildings are applied to a short test route. These selections are run in the simulator, performance measures are collected and the 8 cases are compared. The obtained results are promising; reducing the amount of selected and displayed information results in overall improvement of visualisation performance. Decreasing the selection size from 80 to 50 m in the textured case results in an increase of 80 FPS and a reduction of used memory equal to 80 MB.

The tests clearly reveal that textures are resource hungry. They consume a large amount of processing power, virtual memory and storage space. The biggest gains in overall performance i.e. used memory and frame rate, are observed when the amount of textured structures is reduced. The effect of reducing the number of untextured geometries, on the other hand, is less drastic: while the used memory decreases significantly, the change in the frame rate is negligible.

It is therefore concluded that in the quest to reduce the information contained in 3D city models efforts should be directed at minimizing the amount and/or complexity of textures rather than implementing geometry optimization techniques.

Completely removing textures, however, is not an option as they are deemed valuable by users that use 3D maps to navigate the real-world. Fortunately, not all structures need to be textured, and not all textures are equally valuable. Non-salient buildings such as identical rows of residential homes contain less navigational cues than salient buildings such as fast food restaurants, gas stations, banks, etc. Styling the former with simple textures while keeping the latter with complex textures reduces a scene's total texturing footprint. This research implements a simple texture simplification strategy that replaces textures with a building's dominant facade and roof colors. The results look sensible and are a better solution than textureless buildings as the former style produces easily distinguishable buildings.

Reducing the amount of textures can also be achieved by removing low quality textures from the dataset. For instance, a large number of the data set's 3D facades are obstructed by trees thereby making the whole building green during visualisation. These textures have zero navigational value and should not be shown during visualisation, or better yet, be removed from the model during a preprocessing step. Deciding which textures to use is sometimes a technical question, and sometimes a cognitive one.

This research addresses the cognitive question by implementing an expert system that decides on the appropriate textural level of detail based on a building's thematic and semantic properties, as well the cognitive aspects discussed above. The saliency of a building depends on more than its visibility from afar. The structure of a fast food restaurant might not be salient in the geographical sense, but it may nonetheless act as a landmark because people recognize it from far since it is their favourite restaurant, because it is a well known brand, it has a memorable logo, etc. This kind of saliency is captured and linked to a building's external representation by encoding it in visualisation rules. Rules can be complex or simple, they act on a single building or on many buildings at once and they take geographical, thematical and user-generated data into account. The expert system combined with the visualisation rules allow for powerful decision making about when to use which texturing scheme on what building. This flexibility and power allows designers to experiment with different configurations and fine tune their 3D scenes An expert system abstracts the implementation details by hiding them behind the relatively simple visualisation

rules.

However, rule based decision systems have their drawbacks. The rule base of an expert system quickly becomes verbose as the number and complexity of rules increase. A set of complex rules might undermine the purpose of the prototyping framework i.e. the complexity of the rules might endanger the quickness and agility of the experimentation process.

Crafting sensible rules requires additional, non-geographical information. This is currently lacking in the COLLADA data set; as an experiment, information about a building's type is obtained from the LinkedGeoData data set. LinkedGeoData contains information from Wikipedia and OpenStreetMap and is theoretically a good source of urban semantic and thematic information. The nature of the two parent data sources is such that LinkedGeoData itself is incomplete; it contains relatively few buildings. Enriching every building in the COLLADA dataset is thus not feasible; LinkedGeodata should rather be used as a source of semantically salient buildings by the rationale that if it is in Wikipedia, it must be noteworthy.

To fully leverage the expert system's potential, TomTom needs to craft a high-quality semantic data set that contains information about most buildings and write rules that link the therein contained information to external representations.

In general, the developed information reduction methods can also be used in other navigation applications where a route is being followed such as walking or cycling. It less suitable for exploration or orientation types of use as these generally lack routes, but functionality can still be derived by taking the user's current location and using it as the location of a virtual junction by using the *junction* selector i.e. load geometry that is within a certain distance of his current position. Loading new geometry can either be triggered by the user or automatically by keeping track of users' displacement.

Chapter 8

Future work

Currently, the information reduction process operates on the whole route; all road segments and buildings are processed and their results stored as files prior to their visualisation in the simulation. This results in a slow visalisation startup time and makes the system inflexible to changes in the route. If a driver misses a turn the device has to drop the previously calculated segments, calculate a new route and process all of its segments priot to visualising them. This is wasteful and annoying for the driver as the whole process become sluggish, the long wait after a wrong turn being especailly confusing as the driver needs to receive a new route and navigation aids as soon as possible.

A "select and render as you go" approach i.e. streaming, of segments is much more desirable as it makes the system process the minimum amount of information needed to create a sensible visualisation and it is able to quickly produce new results in the case of a detour from the computed route. Streaming can be implemented by first making the IRP process route segments in batches of 3-4 and feed them "just in time" to the game engine. Batch processing has to be triggered by the game engine itself; a third collision sphere can be implemented that, upon intersection with the vehicle, starts an IRP batch processing cycle. The size of these spheres must be large enough so that the segments are processed and available by the time the driver arrives near them.

The sphere size can be a function of distance or speed of the vehicle. The first option is the safest but may also result in tiles being loaded too soon if the driver's speed is low, or too late if the driver's speed is high. The second option may prove difficult to implement robustly as it is difficult to predict how long each segment takes to process.

The vehicle speed can also be used as input for deciding which buildings to select and how to style them. For instance, when the speed is high it makes sense to reduce the textural complexity of buildings as the driver is probably on a high-speed road such as a highway and is not navigating.

Currently, route/footprint intersection are performed using triangles. While

this works for small datasets such as single cities, the number of triangles may slow down the search when the number of cities increases. An initial step in dealing with this is to convert the footprint triangles into polygons and index the latter. This reduces the number of bounding boxes that need to be searched and will keep lookup speeds high as the index grows.

The data used in this research lacks a road network. While this does not hamper the operation of the spatial controller, it does prevent the semantic controller i.e. the expert system, from controlling the IRP selector phase as it lacks information about, for instance, the road type. This problem can be alleviated in two ways 1) include a semantically rich road network to the available data set or 2) provide semantic information about the road type through the shortest route that is provided to the IRP. The first option is preferred as it opens the door to new type of selectors altogether and it will enable the use of the expert system and its visualisation rules in the selection phase of the IRP. It will then be possible to use base the selector type and size choices on the road's thematic and semantic properties and thereby address the problem described in Section 6 where the 50 m selector is sufficiently large for one road type, but is too big for another. Furthermore, rules that act on a road/building combination become possible e.g. style commercial buildings next to a highway in a lower level of detail than buildings next to a residential road.

A road network furthermore enables the automatic extraction of junctions (this is currently done manually) and creating rules based on their complexity. For instance, buildings in the vicinity of simple T-shaped junctions my be rendered with low textural quality, whereas buildings next to complex junctions may be rendered in a higher level of detail.

As discussed in the Conclusions, a large amount of the textures contained in the COLLADA model are not sufficient for navigation purposes as they are covered by trees. These textures should be removed from the dataset by either replacing them with generic textures or marking the building as having bad texturing and treat it accordingly by, for instance, not displaying the textured model upon visualisation.

The COLLADA texture files contain texture information about several buildings. When the game engine loads textured geometry it reads the whole texture file even when it is referenced by a single building. Further reducing the peak load on the game engine during geometry loading can therefore be achieved by processing the texture images such that they contain only textures that are actually referenced by a geometry. Two processing options are proposed: color all non-referenced texture pixels black and let the PNG comperssion algorithm reduce the size of the texture, or cut out the referenced pixels and create a new smaller PNG image and pass it to the game engine instead of the original 1.5 MB texture.

A major drawback of the main information reduction method is its inability

to select highly visible buildings that are located outside the buffer. Altgough not an an issue in densely constructed areas, this shortcoming may be the source of confusion at locations with less dense construction, or locations with large open spaces such as parks, rivers, bridges, etc. This issue is solveable in several different ways: use predefined landmarks, find landmarks based on a building's geometric, semantic and thematic properties or deploy visibility algorithms that determine which buildings are visible from a certain position.

The first approach uses already existing manually created landmark databases that on most devices are implemented as Points of Interest. The drawback of these landmarks is that they are chosen primarily for their use and tourist attraction value e.g. gas stations, hotels, restaurants, sights such as The Louvre, Euromast, etc., and not for their saliency i.e. fitness for navigation. Hence, these sets do not contain tall non-tourist buildings such as high-rise offices and residential towers.

The second approach entails finding buildings that act as landmarks based on properties such as height, color, volume, type, purpose, age, etc. Using this approach it is possible to analyse a city model and label buildings that are higher than, say, 30 meter as landmarks. Another approach is to base the saliency of a building on its volume e.g. in the case of shopping malls, theaters, cinemas, etc. with the idea of displaying they are big objects that are recognizable across a large open space. Another approach is to select popular building types such as fast food restaurants, popular shops, banks, etc. Once a building is found and labeled as landmark it can be displayed to the driver by searching for it using a bounding box that is placed in the driver's looking direction outside the buffer. The problem with these landmarks is, however, that they may not be visible from the driver's current position. This is for instance the case when traveling in an urban canyon. Loading buildings that are not visible is pointless and defies the purpose.

Performing visibility calculations is the best approach to selecting far off buildings. Visibility calculations are expensive and are therefore not fit for mobile devices. An implementation strategy therefore is to run them as a preprocessing step off the device and upload their results to the device as annotations on the street network i.e. divide the street network in nodes, perform a visibility analysis at each node and save the building ids that are visible from that node as an extra piece of information. With this information the IRP is able to load and display landmarks even though they do not intersect the buffers.

Showing 3D geometry that is far off is, however, a waste of processing power and virtual memory as the buildings will probably be too small for the driver to appreciate their three-dimensionality. A better solution is to render far off structures (and even whole cityscapes) as billboards. These can be generated on the fly on the 3D model. These billboards are created by placing the renderer's virtual camera in front of the desired scene and performing a render-to-texture action i.e. the desired 3D scene is rendered and turned into a 2D raster image. A billboard is created by assigning the texture to a 3D plane and placing it in the scene. Panda3D, for instance, has a native billboard node.

In the ideal case, the built framework acts as a very high level experimentation tool in which users are able to graphically build the execution plan and semantic rules by dragging and visually connecting the various operators together. To facilitate this the PBT may be packed in a Graphical User Interface that, for instance, uses a single sample building to display the effect of a decision in real-time prior to unleashing it on the whole model.

Appendix A

Geometry extraction

A.1 Buildings

The building geometries are stored in the COLLADA files. The COLLADA data model defines so-called libraries that store the numerous assets that form a scene (see section 2.1). The concept *building* is stored in the *visual scenes* library where it acts as a placeholder for all elements that constitute a building. Each building is composed of structures and meta information that is applicable to the building as a whole.

A building consists of three types of structures: *footprint*, *facade* and *super*. *facades* structures contain the facades of buildings, but also simple i.e. flat roof shapes. *Super* structures contain complex roof shape that sit on top of a building. COLLADA geometries are triangulated, hence each structure is a collection of triangles, which in turn are defined as collections of vertices. *Facades* and *super* structures have a material attached to them.

Each geometry contains three or more triangle sets, called *triangles*, that make up the structure. Most triangles consist of three parts: vertices, texture coordinates and category viz:

Here, the first line defines a triangle set with a certain material and 13 triangles. The following three lines define the sources of information for the triangles' vertices, texture coordinates and category values, respectively. The information itself is stored in data sources defined elsewhere in the structure's geometry definition (see next paragraph). Access to these sources is handled by the indices defined in the $< p >$ element. Each index points to a piece of information for a single vertex; a triangle is made of three vertices, hence the first 9 indices define a single triangle. Consider the first three indices 5, 12 and 6. The first index points to the location of the triangle's first vertex coordinate triplet in the vertices source. The second index points to the location of the texture coordinates for the triangle's first vertex, while the third points to the location of the triangle's cateogry value. The vertex source is defined as

```
1 <triangles material="tex_300210310311013_0023" count="13">
2   <input semantic="VERTEX" source="#ground35730-geometry-
        vertex" offset="0" />
3   <input semantic="TEXCOORD" source="#ground35730-geometry-uv"
         offset="1" set="0" />
4   <input semantic="category" source="#ground35730-geometry-
        category" offset="2" />
5
6   <p>5 12 6 19 14 6 6 13 6 5 12 6 18 15 6 19 14 6 22 45 14 21
        44 14 20 43 ... </p>
7 </triangles>
```

Listing A.1: An example of a triangle definition

```
1 <source xmlns="http://www.collada.org/2005/11/COLLADASchema"
      ...
2     id="ground35730-geometry-position">
3   <float_array xmlns="http://www.collada.org/2005/11/
        COLLADASchema" ...
4       id="ground42242-geometry-position-array" count="24">
5     -6.28 5.19 0.06 -11.30 -0.88 0.09 -13.26 0.74 0.11 -8.24
          6.81 0.07 -11.30 ...
6   -0.88 4.10 -6.28 5.19 4.10 -13.26 0.74 4.10 -8.24 6.81 4.10
7   </float_array>
8   <technique_common>
9       <accessor source="\#ground42241-geometry-position-array"
            count="16" stride="3">
10        <param name="X" type="float" />
11        <param name="Y" type="float" />
12        <param name="Z" type="float" />
13      </accessor>
14  </technique_common>
15 </source>
```

The vertex coordinates are stored in the *float_array* element as a sequence of floats. The *technique_common* defines what the string of numbers represents.

An index equal to 5 thus refers to the fifth coordinate triplet defined in the source, which is located at the 5*13 = 15th position in the *float_array* element. The first coordinate in this case equals x=-6.28, y=5.19, z=4.10. Texture coordinates and categories are accessed int the same way.

The Python *Facades* class is responsible for extracting building geometry from the COLLADA/XML files. Figure A.1 shows its UML definition.

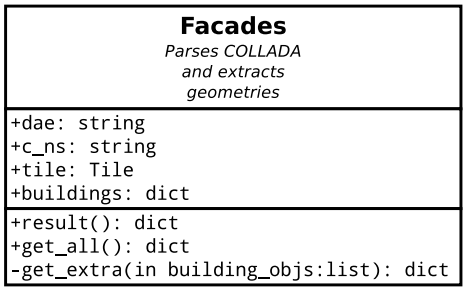


Figure A.1: The Facade class' UML diagram.

The *get_all()* method contains all the functionality. It selects all buildings nodes with this XPath epxression

```
geometry = /c:COLLADA/C:library_geometries/c:geometry[@id="<building_id>"]
```

The triangles are unpacked i.e. de-indexed and the vertices are saved in the *Vertex* table.

The result of this function is a dictionary that stores the buildings, faces, vertices, etc. as

```
dict[building_id][face_id] = {vertices, verices_indices, uv_coordinates,
uv_coordinate_indices, triangle_category, /i}
```

A.2 Tiles

Information about the tiles is stored in the top-level KML file. The extent of each tile is stored as a KML LatLonAltBox of which the north, south, east and west corners are known. The COLLADA models are embedded in the KML file as a Placemark viz.

The *Placemark* element defines a latitude/longitude center point for the model and a link to the COLLADA file that contains the model. Knowing the tile's center coordinates is important as the COLLADA files are not georeferenced.

The Python *Tile* class contains the functionality that parses the KMl file and extracts all the relevant information. Figure A.2 shows a UML definition of the class.

```
1 <Placemark id="tile\_300210310123310">\\
2   <name>tile\_300210310123310</name>\\
3   <Region>\\
4     <LatLonAltBox>\\
5       <north>51.94176632</north>\\
6       <south>51.94009723</south>\\
7       <east>4.39512567</east>\\
8       <west>4.39044874</west>\\
9     </LatLonAltBox>\\
10  </Region>\\
11  <Model id="m\_300210310123310">\\
12    <Location>\\
13      <longitude>4.39316759</longitude>\\
14      <latitude>51.94191261</latitude>\\
15      <altitude>0.00</altitude>\\
16    </Location>\\
17    <Link>\\
18      <href>300210310123310/m\_300210310123310.dae</href>\\
19    </Link>\\
20  </Model>\\
21 </Placemark>
22 }
```

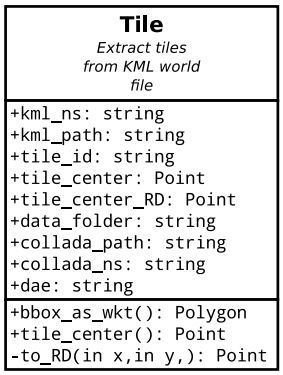
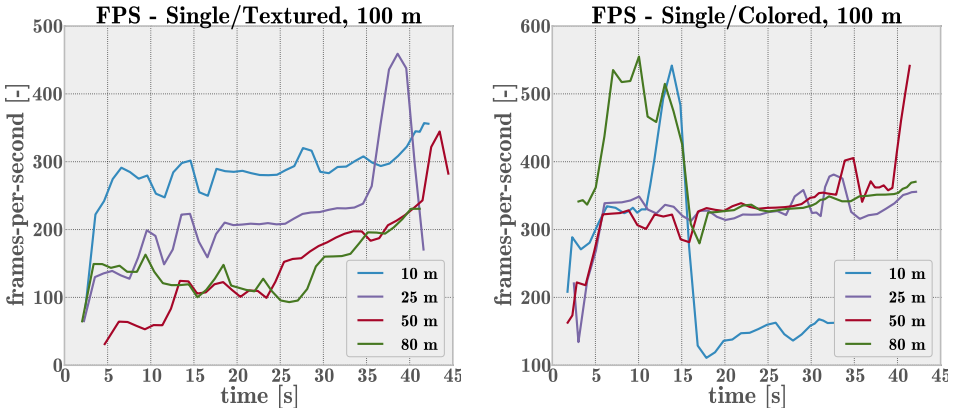


Figure A.2: The Tile class' UML diagram.

Appendix B

Performance results

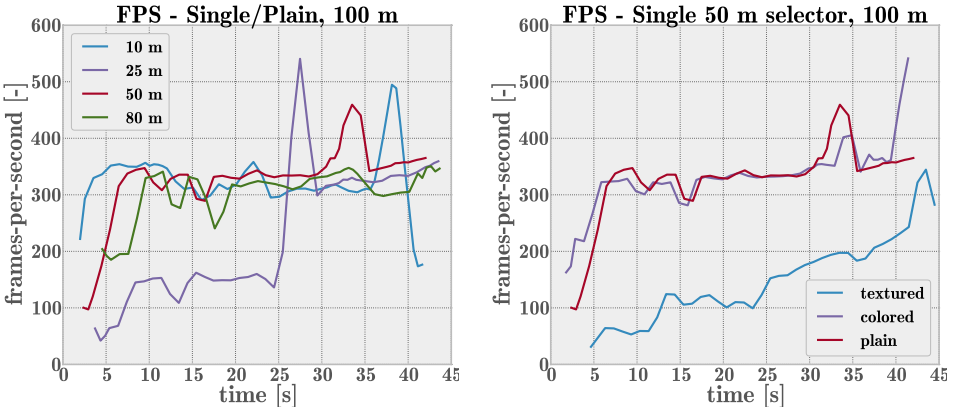
This appendix lists the 100 meter performance measurements of the test cases desribed in Chapter 6.



(a) Frames-per-seconds measurements fot the *textured* extractor.

(b) Frames-per-seconds measurements fot the *colored* extractor.

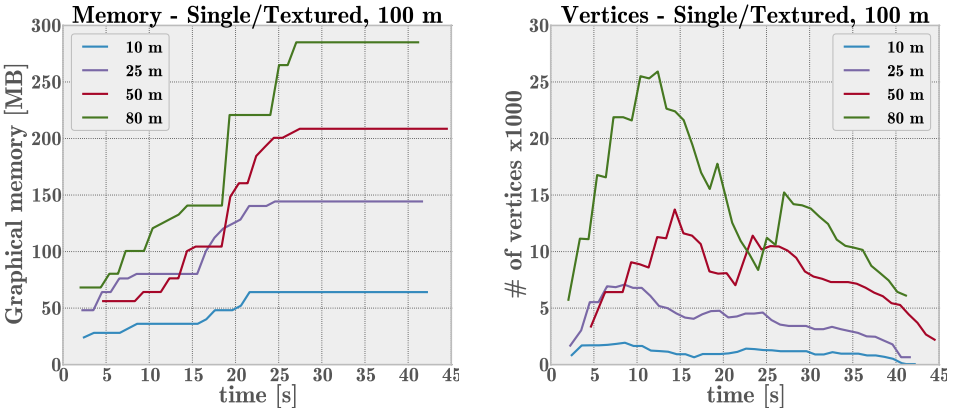
Figure B.1: Frames-per-seonds measurements for the two extractors.



(a) Frames-per-seconds measurements fot the *textured* extractor.

(b) Frames-per-seconds measurements fot the *colored* extractor.

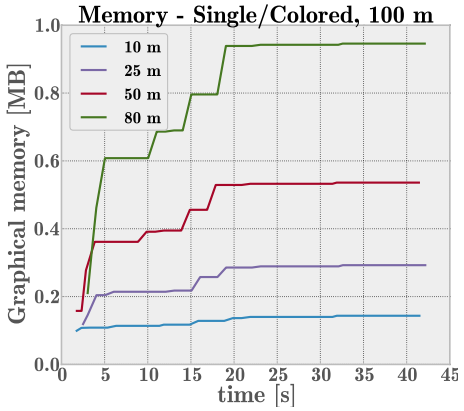
Figure B.2: Performance metrics for the *textured* extractor



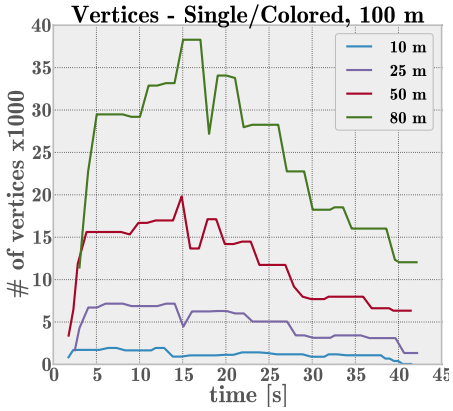
(a) Graphics memory usage of the *textured* extractor.

(b) Number of vertices.

Figure B.3: Performance metrics for the *textured* extractor.

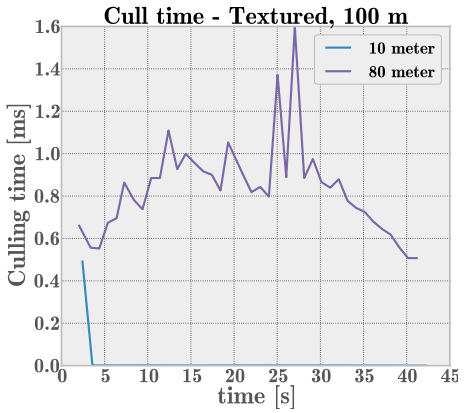


(a) Graphics memory usage of the *colored* extractor.

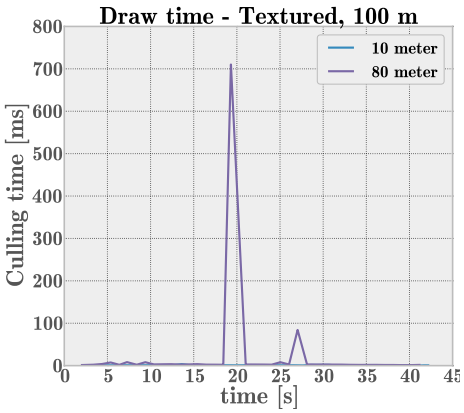


(b) Number of vertices.

Figure B.4: Performance metrics for the *colored* extractor.



(a) Time required to determine which geometries are visible.



(b) Time required to draw a scene.

Figure B.5: Performance metrics for the *textured* extractor.

Bibliography

OGRE - Open Source 3D Graphics Engine. 2012.

Ivin Amri Musliman, Behnam Alizadehashrafi, Tet-Khuan Chen, and Alias Abdul-Rahman. Modeling Visibility through Visual Landmarks in 3D Navigation using Geo-DBMS Developments in 3D Geo-Information Sciences. Lecture Notes in Geoinformation and Cartography, chapter 9, pages 157–180. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-04790-9. doi: 10.1007/978-3-642-04791-6_9. URL http://dx.doi.org/10.1007/978-3-642-04791-6_9.

Sören Auer, Jens Lehmann, and Sebastian Hellmann. LinkedGeoData: Adding a Spatial Dimension to the Web of Data The Semantic Web - ISWC 2009. volume 5823 of *Lecture Notes in Computer Science*, chapter 46, pages 731–746. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-04929-3. doi: 10.1007/978-3-642-04930-9_46. URL http://dx.doi.org/10.1007/978-3-642-04930-9_46.

M. Banes and E. L. Finch. *COLLADA-Digital Asset Schema Release 1.5.0 specification*, April 2008.

Stefan Behnel, Martijn Faassen, and Ian Bicking. lxml - XML and HTML with Python. http://lxml.de/ Accessed on 13 August 2012.

Maximino Bessa, António Coelho, and Alan Chalmers. Alternate feature location for rapid navigation using a 3D map on a mobile device. In *Proceedings of the 3rd international conference on Mobile and ubiquitous multimedia*, MUM '04, pages 5–9, New York, NY, USA, 2004. ACM. ISBN 1-58113-981-0. doi: 10.1145/1052380.1052382. URL http://dx.doi.org/10.1145/1052380.1052382.

Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked Data - The Story So Far. *International Journal on Semantic Web and Information Systems*, 5(3): 1–22, MarMar 2009a. ISSN 1552-6283. doi: 10.4018/jswis.2009081901. URL http://dx.doi.org/10.4018/jswis.2009081901.

Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. DBpedia - A crystallization point for the Web of Data. *Web Semantics: Science, Services and Agents*

on the World Wide Web, 7(3):154–165, September 2009b. ISSN 15708268. doi: 10.1016/j.websem.2009.07.002. URL http://dx.doi.org/10.1016/j.websem.2009.07.002.

Stefano Burigat and Luca Chittaro. Location-aware visualization of VRML models in GPS-based mobile guides. In *Proceedings of the tenth international conference on 3D Web technology*, Web3D '05, pages 57–64, New York, NY, USA, 2005. ACM. ISBN 1-59593-012-4. doi: 10.1145/1050491.1050499. URL http://dx.doi.org/10.1145/1050491.1050499.

T. Capin, K. Pulli, and T. Akenine-Moller. The State of the Art in Mobile Graphics Research. *Computer Graphics and Applications, IEEE*, 28(4):74–84, July 2008. ISSN 0272-1716. doi: 10.1109/MCG.2008.83. URL http://dx.doi.org/10.1109/MCG.2008.83.

Volker Coors and Er Zipf. Mona 3d– mobile navigation using 3d city models. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.90.1269.

J. Crampton. A cognitive analysis of wayfinding expertise. *Cartographica*, 29 (3):46–65, 1992.

Arnoud de Bruijne, Joop van Buren, Anton Kösters, and Hans van der Marel. *Geodetic reference frames in the Netherlands*. Netherlands Geodetic Commission, March 2005. ISBN 90 6132 291 X.

Jürgen Döllner and Jan E. Kyprianidis. Approaches to Image Abstraction for Photorealistic Depictions of Virtual 3D Models Cartography in Central and Eastern Europe. Lecture Notes in Geoinformation and Cartography, chapter 17, pages 263–277. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-03293-6. doi: 10.1007/978-3-642-03294-3_17. URL http://dx.doi.org/10.1007/978-3-642-03294-3_17.

Peter Fisher, Birgit Elias, and Claus Brenner. Automatic Generation and Application of Landmarks in Navigation Data Sets Developments in Spatial Data Handling. In *Developments in Spatial Data Handling*, chapter 36, pages 469–480. Springer Berlin Heidelberg, Berlin/Heidelberg, 2005. ISBN 3-540-22610-9. doi: 10.1007/3-540-26772-7_36. URL http://dx.doi.org/10.1007/3-540-26772-7_36.

Bruce Frederiksen. Python Knowledge Engine. http://pyke.sourceforge.net/ Accessed on 16 August 2012.

Alessandro Furieri. SpatiaLite. https://www.gaia-gis.it/fossil/libspatialite/index Accessed on 13 August 2012.

Tassilo Glander and Jürgen Döllner. Techniques for Generalizing Building Geometry of Complex Virtual 3D City Models Advances in 3D Geoinformation Systems. In Peter Oosterom, Sisi Zlatanova, Friso Penninga, and

Elfriede M. Fendel, editors, *Advances in 3D Geoinformation Systems*, Lecture Notes in Geoinformation and Cartography, chapter 21, pages 381–400. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-72134-5. doi: 10.1007/978-3-540-72135-2_21. URL http://dx.doi.org/10.1007/978-3-540-72135-2_21.

Tassilo Glander and Jürgen Döllner. Abstract representations for interactive visualization of virtual 3D city models. *Computers, Environment and Urban Systems*, 33(5):375–387, September 2009. ISSN 01989715. doi: 10.1016/j.compenvurbsys.2009.07.003. URL http://dx.doi.org/10.1016/j.compenvurbsys.2009.07.003.

M. Goslin and M. R. Mine. The Panda3D graphics engine. *Computer*, 37(10): 112–114, October 2004. ISSN 0018-9162. doi: 10.1109/MC.2004.180. URL http://dx.doi.org/10.1109/MC.2004.180.

Ivan Herman, Sergio Fernández, and Carlos Tejo. SPARQL Endpoint Interface for Python. http://sparql-wrapper.sourceforge.net/ Accessed on 13 August 2012.

Dieter Hildebrandt, Jan Klimke, Benjamin Hagedorn, and Jürgen Döllner. Service-oriented interactive 3D visualization of massive 3D city models on thin clients. In *Proceedings of the 2nd International Conference on Computing for Geospatial Research & Applications*, COM.Geo '11, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0681-2. doi: 10.1145/1999320.1999326. URL http://dx.doi.org/10.1145/1999320.1999326.

Seok-Jae Jeong and Arie E. Kaufman. Interactive wireless virtual colonoscopy. *The Visual Computer*, 23(8):545–557, August 2007. ISSN 0178-2789. doi: 10.1007/s00371-007-0117-8. URL http://dx.doi.org/10.1007/s00371-007-0117-8.

Markus Jobst and Timothy Germanchis. The Employment of 3D in Cartography An Overview Multimedia Cartography. In William Cartwright, Michael P. Peterson, and Georg Gartner, editors, *Multimedia Cartography*, chapter 15, pages 217–228. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-36650-8. doi: 10.1007/978-3-540-36651-5_15. URL http://dx.doi.org/10.1007/978-3-540-36651-5_15.

Alexander Klippel, Kai-Florian Richter, and Stefan Hansen. Wayfinding Choreme Maps Visual Information and Information Systems. volume 3736 of *Lecture Notes in Computer Science*, chapter 9, pages 94–108. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-30488-3. doi: 10.1007/11590064_9. URL http://dx.doi.org/10.1007/11590064_9.

Fabrizio Lamberti and Andrea Sanna. A solution for displaying medical data models on mobile devices. In *Proceedings of the 4th WSEAS International Conference on Software Engineering, Parallel & Distributed Systems*, Stevens Point, Wisconsin, USA, 2005. World Scientific and Engineering Academy and

Society (WSEAS). ISBN 960-8457-09-2. URL http://portal.acm.org/citation.cfm?id=1365774.1365790.

Mark Lutz. *Programming Python*. O'Reilly Media, Inc., 2006. ISBN 0596009259. URL http://portal.acm.org/citation.cfm?id=1199154.

Frank Manola and Eric Miller. RDF Primer. http://www.w3.org/TR/rdf-primer/ Accessed on 21 August 2012, 2004.

Jean E. Marvie and Kadi Bouatouch. A VRML97-X3D extension for massive scenery management in virtual worlds. In *Proceedings of the ninth international conference on 3D Web technology*, Web3D '04, pages 145–153, New York, NY, USA, 2004. ACM. ISBN 1-58113-845-8. doi: 10.1145/985040.985062. URL http://dx.doi.org/10.1145/985040.985062.

José M. Noguera, Rafael J. Segura, Carlos J. Ogáyar, and Robert Joan-Arinyo. Navigating large terrains using commodity mobile devices. *Computers & Geosciences*, 37(9):1218–1233, September 2011. ISSN 00983004. doi: 10.1016/j.cageo.2010.08.007. URL http://dx.doi.org/10.1016/j.cageo.2010.08.007.

A. Nurminen. Mobile 3D City Maps. *Computer Graphics and Applications, IEEE*, 28(4):20–31, July 2008. ISSN 0272-1716. doi: 10.1109/MCG.2008.75. URL http://dx.doi.org/10.1109/MCG.2008.75.

Antti Nurminen. m-LOMA - a mobile 3D city map. In *Proceedings of the eleventh international conference on 3D web technology*, Web3D '06, pages 7–18, New York, NY, USA, 2006. ACM. ISBN 1-59593-336-0. doi: 10.1145/1122591.1122593. URL http://dx.doi.org/10.1145/1122591.1122593.

Antti Nurminen. Mobile, hardware-accelerated urban 3D maps in 3G networks. In *Proceedings of the twelfth international conference on 3D web technology*, Web3D '07, pages 7–16, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-652-3. doi: 10.1145/1229390.1229392. URL http://dx.doi.org/10.1145/1229390.1229392.

Antti Oulasvirta, Sara Estlander, and Antti Nurminen. Embodied interaction with a 3D versus 2D mobile map. *Personal Ubiquitous Comput.*, 13(4):303–320, May 2009. ISSN 1617-4909. doi: 10.1007/s00779-008-0209-0. URL http://dx.doi.org/10.1007/s00779-008-0209-0.

J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, May 2008. ISSN 0018-9219. doi: 10.1109/JPROC.2008.917757. URL http://dx.doi.org/10.1109/JPROC.2008.917757.

Bastian Quilitz and Ulf Leser. Querying Distributed RDF Data Sources with SPARQL The Semantic Web: Research and Applications. In Sean Bechhofer, Manfred Hauswirth, Jörg Hoffmann, and Manolis Koubarakis,

editors, *The Semantic Web: Research and Applications*, volume 5021 of *Lecture Notes in Computer Science*, chapter 39, pages 524–538. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-68233-2. doi: 10.1007/978-3-540-68234-9_39. URL http://dx.doi.org/10.1007/978-3-540-68234-9_39.

Arne Schilling, Volker Coors, and Katri Laakso. Dynamic 3D Maps for Mobile Tourism Applications Map-based Mobile Services. In L. Meng, T. Reichenbacher, and A. Zipf, editors, *Map-based Mobile Services*, chapter 15, pages 227–239. Springer Berlin Heidelberg, Berlin/Heidelberg, 2005. ISBN 3-540-23055-6. doi: 10.1007/3-540-26982-7_15. URL http://dx.doi.org/10.1007/3-540-26982-7_15.

Arne Schilling, Sandra Lanig, Pascal Neis, and Alexander Zipf. Integrating Terrain Surface and Street Network for 3D Routing 3D Geo-Information Sciences. In Jiyeong Lee and Sisi Zlatanova, editors, *3D Geo-Information Sciences*, Lecture Notes in Geoinformation and Cartography, chapter 8, pages 109–126. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-540-87394-5. doi: 10.1007/978-3-540-87395-2_8. URL http://dx.doi.org/10.1007/978-3-540-87395-2_8.

Claus Stadler, Jens Lehmann, Konrad Höffner, and Sören Auer. LinkedGeoData: A core for a web of spatial open data. *Semantic Web*, 2012. doi: 10.3233/SW-2011-0052. URL http://dx.doi.org/10.3233/SW-2011-0052.

Jantien Stoter, Joris Goos, Rick Klooster, Marcel Reuvers, Edward Verbree, Gebrand Vestjens, and George Vosselman. 3D Pilot Managementsamenvatting. Technical report, Geonovum, June 2011. URL http://www.geonovum.nl/sites/default/files/standaarden/managementsamenvatting3Dpilot.pdf.