MSc Thesis in Geomatics

Three-dimensional reconstruction of underground utilities for real-time visualization

Josafat Isaí Guerrero Iñiguez
Reconstructed pipes and cables from the Maasvlakte 2 area.
Red: electricity cables.
Green: telecommunication cables.
Blue: water pipes.
Yellow: others.
The brown ditches correspond to lateral cable space.
Utilities data provided by Port Rotterdam,
Buildings courtesy of Geemeente Rotterdam.
Visualization performed using X3DOM.
Three-dimensional reconstruction of underground utilities for real-time visualization

Author:
Josafat Isai GUERRERO IÑIGUEZ

Supervisors:
Dr. Sisi ZLATANOVA
Dr. Martijn MEIJERS

Co-reader:
Dr. Erik JANSEN

September 2012
To my parents: Julio and Alicia.
Abstract

This research discusses the conversion of underground utilities datasets into scene graph models for its 3D visualization using the WebGL technology, which is a Javascript interface to access the 3D graphics hardware. This conversion requires to reconstruct the volumetric appearance of the objects from its abstract representation, involving several decisions towards its mapping into a scene graph.

Each decision taken carries consequences in the rendering performance and given the nature of the underground utilities, multiple problems arise which should be addressed before achieving a fluid visualization. This research considers multiple reconstruction approaches and studies the corresponding factors affecting the performance by introducing a testing framework. With the results of the tests, the considered factors are ranked based on their performance and used to give advice towards the best approaches for the reconstruction underground utilities and its applicability to WebGL environments.
Acknowledgments

I would like to thank all the persons involved in this project, friends and colleagues from all the corners of the world who made this project possible, to whom I owe a big part of my life.

First, I express my utmost gratitude towards Sisi, a wonderful woman who gave me the opportunity to work in the MV2 project and showed me the path to a new dimension: the third. I also thank Martijn for its endless patience and timely comments, but specially for having the time to share a few words so needed in desperate times. Thanks also to Erik Jansen for contributing with his knowledge towards the completion of this work.

A special mention goes to the people involved in the MV2 project: Anne Jan Boersma, Albert Mulder and Suzana Kana from Port Rotterdam, Joris Goos from Geemente Rotterdam, Jackob Beetz form TUE, Marian de Vries and Hugo Ledoux from OTB.

I also want to say thanks to Elfriede Fendel, who get me into this master program and helped during almost all the process involved with it.

A loud mention goes to to Statoil and TU Delft who fully funded my studies, to whom I owe not money but an endless appreciation.

In addition, I also want to express my gratitude towards my colleagues: Naresh, Ravi, Daphne, Elise, Pien, Simeon, Hoe Ming, and Wouter, who made this experience something unique.

At last but not the least, I also thank my family who helped me along this years, gave me moral support and endless love.

Thanks.
Josafat
‘Voici mon secret. Il est très simple: on ne voit bien qu’avec le cœur.
L’essentiel est invisible pour les yeux.’
Antoine de Saint Exupéry
Le Petit Prince
# CONTENTS

1 **Introduction** 1  
1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1  
1.2 Technological push . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3  
1.3 Objective and Research Questions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4  
1.4 Enclosing project . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5  
1.5 Research scope . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6  
1.6 Chapters overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7  

2 **Background Information** 9  
2.1 Related work on underground utilities . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9  
2.2 State of the art in 3D visualization for the Web . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10  
2.2.1 OpenGL and OpenGL ES . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10  
2.2.2 WebGL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11  
2.2.3 WebGL frameworks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12  
2.3 3D Visualization Pipeline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15  
2.4 Rendering pipeline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17  
2.5 Visualization performance issues . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19  

3 **Reconstruction of underground utilities** 21  
3.1 General requirements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21  
3.2 Reconstruction flow . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22  
3.3 Underground objects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24  
3.3.1 Cables and Pipes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25  
3.3.2 Cables and pipes protection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25  
3.3.3 Free space trace and lateral cable space . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26  
3.3.4 Pits and signs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26  
3.4 Reconstruction methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26  
3.4.1 Split methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30  
3.4.2 Non-Split methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39  

4 **Reconstruction implementation** 45  
4.1 Target Frameworks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45  
4.1.1 X3DOM and X3D . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45  
4.1.2 JSON and SceneJS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48  
4.2 Scene Assembly . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49  
4.2.1 X3DOM . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50  
4.2.2 SceneJS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51  
4.3 Complexity analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53  
4.3.1 Primitive analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53  
4.3.2 Reconstructed objects analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55  
4.3.3 Scene analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
## Performance Testing Framework

### 5.1 Terminology

- Performance Testing Framework
- Performance factors
  - Factors impact
  - Factors combination
  - Relationship with the reconstruction methods

### 5.2 Performance factors

- Baseline performance testing
  - Performance hypothesis
  - Applicability to the studied objects

### 5.3 Baseline performance testing

- Performance Acceptance criteria

### 5.4 Performance Acceptance criteria

- Test planning
  - Key scenario
  - Modeling data and user variability
  - Collected metrics

## Performance Testing Implementation

### 6.1 Testing environment

- Test implementation

### 6.2 Test implementation

- Test execution

### 6.3 Test execution

- Test results
  - Responsiveness (FPS)
  - Triangle throughput (TPS)
  - Batch throughput (BPS)

### 6.4 Test results

- Verification

### 6.5 Verification

- Baseline conclusions

### 6.6 Baseline conclusions

- Performance-testing conclusions

### 6.7 Performance-testing conclusions

- Real data performance

## Conclusions, Recommendations and Future Work

### 7.1 Conclusions

- Main contributions

### 7.2 Main contributions

- Recommendations and future work

### 7.3 Recommendations and future work
The Port of Rotterdam, as one of the largest in the world, executes multiple processes important for its daily operations using a substantial amount of information mostly expressed as 2D line drawings. However, many tasks related to underground utilities require the use of 3D information to extract their depth, compute the available space to lay down new pipes and cables, perform checks for data overlaps on both crossing and straight segments, plan maintenance procedures based on the actual three-dimensional disposition of the assets, deduce relationships between underground utilities and assets above the ground, among many other tasks.

The availability of 3D data is a requisite to perform these tasks, but is not a sufficient condition to complete them, requiring additional analytical and visualization tools. In particular, the visualization of pipes and cables is required to comprehend the situation on three dimensions not only for planning and maintenance purposes, but for also for communication and validation of analyses and results. However, the visualization of 3D lines on the screen introduces a problem to the user as they lack of volumetric appearance, required to produce depth perception and key to understand the disposition and relationship of the objects on the screen.

To solve this problem, volume should be added to non volumetric 3D lines in a process referred during this research as reconstruction, creating the outer shell of the desired object using only triangles and making them suitable to realtime rendering using computers equipped with Graphics Processing Units (GPUs).

With the recent creation of WebGL, a technology enabling 3D content embedded on the web browser, and the interest in building 3D applications on the web displaying pipes and cables, it becomes important to know what are the capabilities of the platform for such task but also how does the reconstruction process fits within the capabilities of the technology. This reconstruction process involves elements used as building blocks with implications on the performance which should be analyzed, organized and jointly tested to establish their final impact on the rendering stage. If a reconstruction method provides a good performance, the visualization of pipes and cables is guaranteed to provide a smooth experience to the final user, enabling richer scenes but also establishing the visualization requirements in terms of hardware and software to display underground utilities.

1.1 Motivation

The visualization of three dimensional lines on computer environments poses several problems for the user to correctly interpret the three dimensional situation of a scene due to the lack of depth perception, which is the ability to perceive the world in three dimensions and judge the distance to an object.

Cutting and Vishton [1995] state that to understand with reasonable accuracy the layout of most natural environments, including cluttered well lit environments, multiple combinations of information sources or depth cues shall be used. Monocular depth cues are of interest for this research as they only require a single eye to trigger depth perception, in contrast to binocular depth cues.

The following monocular depth cues include Cutting and Vishton [1995]:

- Occlusion: the partial obstruction of one object by another from the user point of view.

Cutting and Vishton [1995] understands for reasonable an error up 15 percent, which will not have consequences on everyday life.
• Relative size: the comparison of objects which are similar in appearance and textures with respect to their projected retinal size. Under this situation, the most remote object will have the smaller size.

• Relative density: the association between the density of a group of objects and the interpretation of distance, so objects in the distance will look grouped at higher densities than closer objects.

• Height in visual field: the assumption that the difference between the bases of the objects is smaller in the distance than with closer objects.

• Aerial perspective: the introduction of blurriness as the objects are far from the observer, usually produced by environmental phenomena like fog.

• Motion perspective: is the effect produced by the difference in displacement speed between objects close to the observer and objects in the distance, where the former move fast while the latter seem static in the horizon.

Each one of these depth cues trigger different psychological and psycho-physical signals which in combination, provide the required depth information to understand the studied scenes. Figure 1.1 shows some examples combining monocular depth cues which form the basis for 3D visualization on non 3D displays.

![Figure 1.1: Examples of depth cues showing interposition and shading, aerial perspective, relative size and perspective.](image)

In contrast to the presented depth cues, lines displayed on current 3D graphics hardware cannot be shaded as do not have any volume or surface, and as consequence occlusion and relative size are impossible to achieve so their portrayal on 2D or even 3D displays cannot provide any depth information. Figure 1.2 shows this comparative and bring an a good example on how occlusion and shading depth cues are useful for to determine the position and orientation of the objects in space, comparing 3D lines drawn using only lines with they corresponding reconstructed objects where 3D cues are present. With lines only, results impossible to determine the closest objects and the understanding of the scene becomes difficult if not impossible.
1.2 Technological push

Realtime visualization on current GPU’s is achieved by processing triangles and as consequence requires the use of triangulated models describing the outer shell of the objects. This specific requirement leads the reconstruction of pipes and cables towards creating triangular models in replacement of 3D lines to get the benefits of the 3D technology and provide depth cues to initially non-dimensional objects, producing perspective, shading, occlusion and motion perspective cues. This modeling approach has the advantage of being a simple way to describe objects on three dimensions with the added benefits of providing a smooth visualization experience to the user.

The interest on mixing 3D content and web technologies to create 3D web applications has been around for several years and achieved through the use of plugins. However, plugins introduce compatibility problems due to the availability of different operating systems and browsers.

With the recent introduction of WebGL providing an Application Programming Interface (API) on Javascript for 3D graphics, the need of plugins is not longer an issue. This standardization effort enables new ways of displaying and distributing 3D content on the web, being of particular interest for this research the distribution and visualization of content based on reconstructed models of pipes and cables.

On top of WebGL, 3D engines are built to simplify the load, display and interaction with the 3D content. Modern 3D engines are based on the concept of the scene graph, which is a data structure for hierarchical modeling and manipulation of 3D scenes, object oriented and rich of facilities for interaction with the user, to perform navigation and exploration of the scene, but also to provide the ‘object picking’ functionality.

X3DOM is one of those 3D engines and is an X3D player written on Javascript. X3D is an XML-ISO based standard for displaying 3D graphics, an is the successor of VRML 97 [Web3D.org]. X3D provides a well specified syntax and semantics for delivering platform independent 3D content. X3D only consists of document and runtime specifications, requiring a 3D engine implementing the given standard, for which X3DOM is an example.

SceneJS is another 3D engine for WebGL providing a Javascript Object Notation (JSON) API to create and interact with 3D content, specifically designed to be used for engineering applications composed of multiple objects to interact with. Lacks a standardized 3D format support however is based on concepts borrowed from OpenSceneGraph[3] thus allowing a direct manipulation of the scene graph but giving freedom to the programmers to implement the required functionalities.

With the availability of 3D engines to display 3D content without the use of plugins, two ways of mixing 3D information and current web technologies can be considered:

The first way corresponds to applications extended to add 3D content without disturbing their original functionalities, reusing the existing technology stack. This extensions make sense when applications can not be replaced but the 3D content can be added to provide an additional value. Examples of this applications include catalogs where existing images can be replaced by 3D models.

---

2Virtual Reality Modeling Language.
3http://www.openscenegraph.org/projects/osg
The second type correspond to new applications created around 3D viewers which can include the existing web technologies. This applications extend the range of solutions for the Web environment, traditionally centered around 2D but now incorporating interaction and navigation schemas in 3D, producing additional ways of exploring and analyzing data on the web.

Applications to visualize underground utilities on the web can be created using the described approaches and using existing 3D engines but before implementing solutions or applications based on WebGL, research should be conducted towards studying its suitability.

WebGL is a promising new technology based on Javascript, known for its low performance, making it a candidate for slow visualization. Research is required to ensure that the reconstructed objects process can be displayed using the WebGL technology but also that the platform has enough power to achieve realtime visualization.

1.3 Objective and Research Questions

Pipes and cable are conceptually simple and narrow objects with clearly defined shapes, spanned over large geographical areas and made of multiple pieces.

The reconstruction of underground utilities can be performed in different ways and should lead to objects with similar appearance, produce visual continuity between segments, include objects depicting specific connections and even consider buffer volumes displaying the uncertainty and the security distance between objects.

The creation of visually pleasing reconstructions may require a high number elements which increase the complexity of the scene, which can be measured with the following elements:

- Number of scene graph nodes
- Number of triangles and vertices on the screen
- Transformations and
- Appearance options.

In order to achieve high frame rates, the complexity of the scene should be kept low to reduce the number of processed elements.\textsuperscript{4} To guarantee this, the underground utilities should be reconstructed using efficient\textsuperscript{5} representations with repeatable approaches for which its complexity can be measured and compared.

The representation efficiency itself is not sufficient to guarantee a high frame rate during rendering time as other factors are introduced in the reconstruction, creation and rendering of the scene affecting the performance, whose effects can only be measured during runtime.

Moreover, the availability of different reconstruction approaches produces tradeoffs between representation footprint, rendering performance, visual quality and even interactive features.

Based on the interest of constructing 3D applications using WebGL and Javascript, the rendering performance and the interactive features are the first aspects considered from this list, followed by the visual quality and finally the representation footprint.

With this background, the final goal of this thesis is to design and evaluate different approaches for reconstruction of pipes and cables, characterize the produced objects, their complexity and test the performance on the rendering stage using WebGL as the objective platform.

With this goal in mind, the objective of this thesis is to reconstruct underground utilities to assemble scenes composed of a big amount of individual pickable objects where each object can be queried for administrative information, display the results of different analyses through cartographic processes and be interactively explored by the user.

With this objective, the corresponding research question is stated as follows:

\textsuperscript{4}GPU's minimum logical unit is the triangle, and as such, the speed of the GPU can be expressed in terms of the total number of displayed or processed triangles per second.

\textsuperscript{5}Efficient representations are those who don't waste space or processing time by holding unused, invisible or redundant information.
1.4. ENCLOSING PROJECT

Which approaches for reconstructing underground utility objects improve the performance of visualization clients based on WebGL?

To help answer this question, the following sub research questions are introduced:

• Is the need to identify and select individual objects affecting the performance?
• Does the size of the scene graph in terms of nodes affects the performance?
• Can the use of levels of detail improve the performance?
• Is the visual appearance playing an important role on the performance?
• Is the size of the produced files affecting the performance?
• Is it possible to categorize the reconstruction methods and rank them based on their performance?
• What are the relevant restrictions of the platform affecting the performance?
• What is the impact of the hardware on the performance?

To give an answer to those questions, this work starts with the characterization of the underground objects, available data models for underground utilities and the actual data sources provided during the development of this work to produce archetypical representations of the objects in 3D.

Later on, a set of requirements is defined and applied for creating 3D scenes using the studied objects. Examples of such requirements include the application of volumetric appearance with realistic dimensions based on administrative information, identification of individual objects, linkage of the reconstructed objects to the datasources, application of visual differentiators and interaction with the objects on through the screen.

Given the set of archetypical shapes, the common building blocks applicable for the reconstruction methods are defined based on the target frameworks and used to classify them accordingly.

Later on, the involved building blocks are analyzed and tested during the rendering stage to point out their real impact. Finally, the factors are confronted with the reconstruction methods, allowing an assessment and producing a ranking for the target platform, required to give recommendations over a specific reconstruction approach and finally answer the research questions.

1.4 Enclosing project

The Port of Rotterdam is one of the largest in the world and the gateway to the European trade for more than 350 million consumers. It has an annual throughput of 430 million tons and is spread over an area of 105 km² and a distance of 45 km. When considering the amount of administrative objects used for its daily operations, the port has to manage 123 piers, more than 89 km of quays and more than 1500 km of pipes and cables among many other individual elements.

The construction of a second Maasvlakte started in 2008 and aims to start operations in 2013 and as soon as it gets operational, industries currently located on the old harbor will shift further to the sea increasing substantially the amount of managed administrative objects to be managed.

The development and maintenance of the Port of Rotterdam, its infrastructure, facilities, logistics and other assets requires the management and exchange of critical information coming from different sources and expressed in different formats. This information involves a large number of public and private stakeholders, including companies, environmental authorities, municipalities, various institutions and citizens.

Much of this information concerns to features with high level of spatial and infrastructural dependency, embedded in a dynamic environment in a constant state of transformation. These objects are spatially distributed above (cadastral parcels, topographic elements, buildings, transportation features), underground (cables, pipes, geological and geotechnical data, tunnels), in the air (sensors, radar, cameras) as well as in the water (quays).

6http://jaarverslag.portofrotterdam.com/jaarrekening-2010/991
CHAPTER 1. INTRODUCTION

The Port of Rotterdam has interest in exploring new technologies which can help with the analysis and visualization of the administrative objects, critical to complete tasks requiring three dimensions, but also to display and communicate results between between different departments and stakeholders. WebGL is a web technology with potential to reduce the barriers created by multiple platforms, browsers and applications, removing the need for plugins and producing a single platform. Due to its recent appearance, not many studies have been carried out to give a clear advice on its capabilities, being hard to give advise or give recommendations towards its adoption. This research aims to fill this gap and provide an assessment focused on displaying a specific administrative object: underground utilities.

1.5 Research scope

- The main focus of this work is the reconstruction of underground utility objects for realtime visualization purposes, but specifically their definition using 3D formats and the possible options to provide performance improvements.
- The main indicator for performance studied here corresponds to the rendering speed, expressed as a frame rate.
- WebGL is the platform studied during this research and the interest on using it derives from its potential integration with the rest of the web technologies.
- Two specific WebGL 3D frameworks are considered for this research, X3DOM and SceneJS. This choice is guided mostly by the interest shown from the industry and academia on those two engines, placing them ahead of other solutions with less adoption and support.
- The impact of related Earth features such as terrain, buildings, roads, among others and their representation won't be studied here, despite its unquestionable relationship with the studied objects.
- The considered working paradigm will consist on the use of clients of medium complexity on the front-end and a medium server complexity on the backend, meaning that we will only provide models or scenes declared on 3D formats, so no further work will be done to modify the model once it is on the client side.
- Despite the broad support of WebGL by many web Browsers, this research doesn't focus on the difference between browsers performance but on the responsiveness and scalability using different reconstruction methods for a given set of scenes. Under this assumption, only one browser will be considered and informally, will be the browser that informally gives the best performance.
- The use of standalone applications or heavy desktop clients is considered here only for verification purposes, as the tested platform is the Web Browser.
- The computational complexity of the reconstruction methods is not considered here, so no analysis of algorithms is presented in an exhaustive fashion.
- Server side code and implementation of Web servers will be required to deploy the objects to the clients, however the objective of this work is not to analyze or design them, so no further analysis will be presented.
- The storage solutions related to the underground information wont be elaborated here, however, as these topics are linked to this proposal some considerations were taken.
- The comparison and evaluation of modeling languages such as CityGML, VRML, X3D, COLLADA or SceneJS API is out of the scope of this work.
- The computational cost of extracting information from the data sources will not be studied here nor its transferring time to the client, however the transferring time is dependent on the network bandwidth among other factors, and it can be deduced from the actual file size, so no further work is done on this area.
1.6 Chapters overview

The structure of the thesis is presented as follows:

- Chapter 2 presents related work and background information required to understand the context of this work and related technologies. It starts with underground utilities related literature, models and applications. Next, the state of the art for 3D visualization on the Web is presented including existing 3D frameworks, followed by the 3D visualization pipeline, required to understand how different elements are converted from their data sources into images. Later, details about the rendering pipeline are introduced to understand the underlying architecture for 3D graphics. Finally, exposes a set of potential problems which may appear while visualizing underground utilities and key ideas to tackle them.

- Chapter 3 elaborates on a set of requirements for the reconstruction of underground utilities aimed at creating applications centered around 3D content. The studied objects, their encodings and graphical archetypes are defined. A reconstruction workflow is described and the set of algorithms employed on the reconstruction methods are presented.

- Chapter 4 Introduce and describe the target frameworks next to the corresponding constructive statements adopted towards implementing the reconstructions, finishing with examples for each one and characterizing the complexity of each reconstruction method.

- Chapter 5 introduces a conceptual framework for comparing the studied methods based on their factors and describes an abstract testing procedure to overcome the variabilities introduced. This framework describes what is tested, how it is tested, how it relates to the reconstruction methods and how it applies to real data.

- Chapter 6 implements the testing framework, describes testing requirements and executes the actual tests based on the considered 3D frameworks, produces results and analyzes them. The obtained results are verifies and validates using a standalone player. At the end, this chapter shows the importance of the studied factors with real data, creating a sample scene, testing its performance and giving conclusions.

- Chapter 7 concludes the obtained results, summarizes contributions of this work, finishes with recommendations based on the results and proposes future research lines.
2.1 Related work on underground utilities

Substantial research has been carried out for working with underground utilities information on different aspects. Part of this previous work concerns to computer models and storage solutions for pipes and cables. Another relevant part for this research deals with the transformation of GIS data into visual representations, both on 2D or 3D.

Research on modeling and visualization of utilities in 3D has been done by Du and Zlatanova [2006], where information is transformed into 3D objects and visualized on the fly as shapes with volumetric appearance and symbols depicting special pipeline attachments. A corresponding prototype is presented using a spatial database as storage solution, implemented with Oracle Spatial built in spatial types. Results are visualized and transformed using Microstation an the Java edition of Microstation Development environment (JDML) to convert on the fly the lines into 3D shapes. This desktop platform can be seen as a limiting factor towards the seamless distribution of information to different stakeholders.

This approach have been applied in two study cases [Du, 2006], one for the city of Yibing, China and another using an experimental dataset of the Netherlands. Data from different sources was successfully integrated and visualized accordingly.

The benefits of managing centralized utilities information using DBMS and providing 3D visualization have been shown of critical importance by Zlatanova et al. [2011], revealing better relationships between pipes and objects, making easier the visual inspection while reducing the misunderstanding to the minimum.

The management and registration of utility networks in 4D (space + time) using a spatial DBMS is showed as a promising solution to maintain centralized management and a correct registration of legal rights and obligations, facilitating the analysis and comparison with the related parcels [Döner et al., 2011], as some legal aspects can only be solved using 3D information to determine its spatial relationship with utilities above or under the ground.

Work on overlaying utilities information over panoramic images has been studied by Verbree et al. [2004], addressing the problem of understanding maps while translating their contents into reality and the other way around, closing the gap between georeferenced information and augmented reality.

The process of creating 3D visualizations form 2D geographical sources on the fly have been addressed by Schall et al. [2008] with a transcoding pipeline. This process separates the model content from the presentation, allowing to generate temporary 3D models on demand without storing them. This transcoding pipeline requires GIS data, rules for the model generation and styles for visualization, but also a ‘scene graph’ specification to represent the transcoded model.

This process have been applied by Schall et al. [2010] for modeling underground utilities on mobile devices for Augmented Reality applications, consuming geographical information encoded in Geographical Modeling Language (GML) and converted into ‘scene graphs’.

Standardization efforts towards separating content from presentation are proposed in Basanow et al. [2008] and presented via ‘Styled Layer Descriptors’. A specific description useful for visualization of underground utilities have been proposed for lines, extruding them based on radius and color information.

1Applications where the reality is augmented with computer information.
2http://www.opengeospatial.org/standards/gml
So far, most of this work have been done on the desktop and web environment using plugins to display 3D content. With the advances on the Web, WebGL have appeared as a technology for displaying 3D content without the need of plugins. Given the novelty of this technology and the interpreted nature of the Javascript language, research performed towards the suitability of this technology have not been been carried out and moreover, the implications on the transcoding procedures on performance have not been addressed so far.

### 2.2 State of the art in 3D visualization for the Web

In the computer world, programming libraries, abstractions, frameworks or API’s are created and used towards gaining access to complex features through simplified programatic calls. In the 3D world this also applies and in general terms the 3D technology is spanned into low and high level API’s for displaying 3D graphics.

Low level API’s are in charge of talking directly to the graphics hardware through programatic calls and hiding the underlying details to the programmers. Examples of low level 3D API’s include: OpenGL and the OpenGL Shading Language, OpenGL ES 3D API for embedded devices, WebGL, Direct3D as a subset of DirectX, RenderMan, RenderWare and Glide API, just to name some.

On the other hand, High level 3D APIs provide higher levels of abstraction to work with 3D graphics, simplifying the access to the Low Level API’a and hardware, providing advanced functionalities like complex data types, constructors, geometric primitives, animations, lighting, data streaming, navigation and interaction modes, etc. Several API’s are available for different purposes, including Hoorde3D, Java3D, JMonkey Engine, Mobile 3D Graphics, Nvidia Scene Graph, Open Inventor, OpenGL Performer, Open Scene Graph, Open SG, mostly developed for creating desktop applications.

This research focuses on the use of high level interfaces, but specifically on 3D frameworks built on top of WebGL. The term 3D framework used here denote the combination of a platform, a 3D engine and an API used in conjunction to construct applications.

#### 2.2.1 OpenGL and OpenGL ES

OpenGL [OpenGL.org, 2012], was introduced in 1992 to provide an API for creating 2D and 3D graphics applications. It is aimed to create interactive and portable solutions across different platforms and has become one of the industry’s most used and supported API’s. OpenGL offers a set of rich and powerful functions for visualization and has become an industry standard for visual computing applications.

OpenGL has gone through a series of revisions adding on each one extra functionalities. Some of those functionalities are in direct relationship with the advances introduced by new hardware functionality while others related to library organization itself and its modeling paradigms. The most relevant revisions for this research comes with OpenGL 2.0, with the inclusion of an assembly-like shadow language later called OpenGL Shading Language(GLSL). This language allowed to replace the fixed function vertex and fragment pipes with shaders, traduced into less complexity required into graphics chips to support the OpenGL standard.

OpenGL ES [Kronos Group, 2012] is a lightweight API for advanced embedded graphics defined as a subset of GL, being less complex and easier to integrate on mobile and embedded platforms. OpenGL ES 2.0 is defined relative to OpenGL 2.0 with emphasis on a programmable 3D graphics pipeline, usually available on high end devices.

OpenGL/ES is based on a client/server architecture [OpenGL.org] where the program is the client issuing commands to the server, which is OpenGL. Both API’s use a rendering mode known as ‘immediate mode’, which means that for every frame displayed all the instructions required to draw on the screen should be sent to the server. OpenGL maintains all the information concerning the current state dictating how to draw objects, and the commands issued to GL are immediately applied to change this state. The typical commands issued to set the state of OpenGL include messages defining transformations, lighting, materials and geometric objects. Within the same frame, the state and objects set from previous commands can be

---

reused, allowing to reduce the number of commands sent to OpenGL and thus allowing improvements on the rendering speed.

### 2.2.2 WebGL

The evolution of the Web technologies have raised the interest in producing platform independent applications, suitable to distribute information, enable integrated visualization using remote data sources and include editing capabilities within the browser.

Advances in Web standardization have led to the implementation of HTML5 \cite{w3.org, 2011}, the fifth revision of the core language of the Web, introducing several enhancements, but specifically one that interests to this project deals with the visualization part, namely with the introduction of WebGL \cite{Kronos Group, 2011}, a low level API that allows the generation of interactive 3D graphics within the browser without the use of plugins. It is based on OpenGL ES 2.0 and provides a set of flexible primitives that can be applied to many use cases in 3D. On top of WebGL, custom 3D frameworks control the actual rendering of scenes, using specific rendering pipelines and logic to attend the use cases.

The Web nature of WebGL next to the use of JavaScript allows the use of existing third party libraries to attend additional tasks out of the 3D context and create a rich ecosystem where advanced 3D applications can be deployed.

The access to 3D graphics through WebGL have raised the interest of several software developers and companies to produce 3D applications embedded in the browser without the need of additional plugins besides a compatible graphics card. Some example applications showing of the capabilities of WebGL to produce distribute and explore information include:

- **"Zygote Body"**\footnote{http://www.zygotebody.com} formerly Google Body, was one of the first 3D HTML5 applications developed to show the capabilities of WebGL to produce online 3D applications. It is a human anatomy atlas capable of showing the human body in full 3D, presenting the body as a set of different layers which can be hidden at choice. See Figure \ref{fig:zygote-body}.

- **MapsGL**\footnote{http://support.google.com/maps/bin/answer.py?hl=en&answer=1630790}, the WebGL version of GoogleMaps that makes use of 3D rendering and hardware graphics acceleration to provide an experience that is seamless, smooth, and runs directly in the browser. Rather than loading pre-rendered image tiles from servers, vector data for the map is sent to the browser and rendered on the fly using WebGL. See Figure \ref{fig:maps-gl}.

- **The ChemDoodle Web Components library**\footnote{http://web.chemdoodle.com/demos/pdb-ribbons}, is a pure Javascript chemical graphics and cheminformatics library derived from the ChemDoodle application and produced by iChemLabs. See Figure \ref{fig:chemdoodle}.

- **"BioDigitalHuman"**\footnote{http://www.biodigitalhuman.com/}, allowing the exploration of human anatomy and conditions using over 3000 3D peer reviewed objects. Includes utilities for searching, relationships, dissection, bookmarking, cross sections, data integration and exporting images are provided. Its particularly interesting because relies on the use of additional libraries such as SceneJS, an open source 3D engine also used for other engineering applications. See Figure \ref{fig:biomedicalhuman}.

- **Quake II GWT Port**\footnote{http://code.google.com/p/quake2-gwt-port/} is Google’s impressive attempt to bring 3D gaming experience into the browser. It uses WebGL, The Canvas API, HTML audio, a local storage API and WebSockets to demonstrate the possibilities of pure web applications in modern browsers such as Safari and Chrome. A custom WebGL renderer was created, multiplayer network layer made using UDP, an asynchronous resource loading was made, a custom GWT Java bio buffer based on WebGL arrays, and a simple file system is emulated to keep persistent storage using Web Storage API.
• BIMSurfer\(^9\) Its an open source viewer for IFC models based on WebGL. This application can load
models expressed in JSON format and currently is in beta stage. This application also runs on top of
SceneJS. See Figure 2.1(e).

These examples show how an embedded Web technology like WebGL coupled with JavaScript enables
a different range of 3D applications while relying on standard web technologies. Figure 2.1 gives a clear
idea about the quality of the final product both due to the graphics capabilities and also due to the level of
interactivity achieved.

The visual quality and interactive features shown on these developments raise the interest on using the
Web as a platform for the management and distribution of BIM and GIS information. With the mixture
of WebGL and Javascript , the complexity of the visualization clients can be raised, enabling realtime 3D
graphics while providing complex interaction schemes on the browser, having direct access to features and
generating display elements, features useful to explore and interact with BIM and GIS information, but
specially with underground utilities information.

2.2.3 WebGL frameworks

WebGL is a low level API giving access to the GPU in the browser, enabling the display of 3D content without
the need of additional plugins. However, neither WebGL nor the 3D content know how to interpret the
content or being rendered respectively.

This section describes the role of WebGL frameworks as fundamental pieces required to display 3D
content using the interface provided by WebGL, defining specific interaction, visualization and rendering
pipelines.

WebGL frameworks can be defined as the combination of the Web platform, the user interface elements
provided by the browser, a 3D engine, a set of libraries for defining and interacting with the content, and
additional functionalities for managing the scenes, which in combination are used to create applications
based on 3D viewers.

Such frameworks bring access to the 3D capabilities in the browser defining a rendering engine with
specific an rendering pipeline, tools for accessing and modifying the underlying scene graph, provide inter-
action and navigation options to the user. The web platform provides additional functionalities that the 3D
frameworks can take advantage from like controls for creating user interfaces, networking capabilities, XML
processing capabilities, access to events and appearance options defined by the browser, and in general
interaction with the rest of the browser elements.

The scene graph\(^{11}\) is the main element to many 3D frameworks. It is a hierarchical data structure holding the information to represent a 3D scene. This structure is general-
arly an acyclic graph made of nodes. There are several types of nodes including: Shape nodes, Light
and Camera nodes, Group nodes, and Property nodes. The Shape nodes typically contain geometri-
cal information while the rest of the nodes are used to create logical groups. The information held by the
scene graph besides the geometry, required normals and indices include elements such as illumination, the
camera, appearance options and transformations.

Each object defined on the scene graph corresponds to a set of low level rendering operations. This
relationship establish the core for 3D visualization using GPUs, and is by far the main function of the 3D
engines, i.e. translating the scene information into low level rendering calls.

Various 3D frameworks have been developed for WebGL, and is worth to mention cases with a widespread
adoption and a rich set of features, presented in the following list:

• X3DOM\(^{10}\) Is an experimental open source framework and runtime with support to include X3D doc-
uments. Is created to show how an integration of HTML5 and declarative 3D content could look like.
Is already part of the HTML5 specification for declarative 3D content and allows including X3D ele-
ments as part of any HTML DOM tree.

---

\(^9\)http://bimsurfer.org/
\(^10\)http://www.x3dom.org/
2.2. STATE OF THE ART IN 3D VISUALIZATION FOR THE WEB

(a) Zygote Body

(b) Google Maps GL

(c) ChemDoodle

(d) BioDigitalHuman

(e) BimSurfer

Figure 2.1: WebGL Applications built using different frameworks.
• SceneJS\(^{11}\) is an open source 3D engine with a JSON-based scene graph API, built on top of WebGL. Specialized on rendering of large numbers of objects, individually pickable and articulated, as those common found on applications in engineering and medicine. Several projects are based in this engine, including BioDigitalHuman, BIMSurfer, BIMShare.

• SpiderGL\(^{12}\) is a JavaScript 3D Graphics library for real-time rendering based on WebGL. Provides data structures and algorithms for real-time rendering without forcing specific paradigms like the common usage of the "Scene Graph", however also allows low level access to the WebGL graphics layer. Is built specially for use of COLLADA files and includes mechanisms for asynchronous XML transfers.

• three.js\(^{13}\) is a rich Javascript 3D Engine with a low level of complexity. The engine can render using the `<canvas>`, `<svg>` and `<WebGL>` tags but also produces stereo, analgyp cross-eyed 3d visualizations. Includes utilities for importing from common 3D formats and includes geometry primitives like plane, sphere, torus, 3D text and tube.

• O3D, Google’s Open Source web API for creating rich, interactive 3D applications in the browsers. Initially created as a plugin but evolved into a javascript library on WebGL.

• OSG.JS is a scene graph library similar to OpenSceneGraph\(^{14}\), borrowing concepts from it. Provides toolboxes to interact with OpenGL via Javascript and provides facilities for exporting various files to the osgjs format.

• CopperLicht. Is a full JavaScript 3D engine for creating games and 3D Web applications. It includes a full world editor with supports for around 20 3D common file formats and facilities to reduce the file size of the rendered files by converting them to binary files.

• CubicVR.js is a lightweight, high-performance and implicit WebGL engine. Provides low level access, managed scenes, events support and physics. Includes support for COLLADA in both XML and BadgerFish-JSON. The physics engine included is based on the Bullet physics engine, ported to JavaScript from C/C++.

• EnergizeGL, WebGL framework focused on generative design and information visualization. Is different from other frameworks as is not specifically designed for games, but intended to build website interfaces.

• C3DL, is an open source javascript library to build web applications with WebGL. Includes support for cameras, lights, swappable shader effects, COLLADA files, mouse picking, particle system and more.

This list showed a variety of frameworks created for different purposes, ranging from games creation, data visualization and engineering visualization. Game engines focus more on creating visually stunning and complex scenes, with physics simulation and limited degree of interaction with the objects while achieving high rendering speeds. On the other hand, engineering applications focus more in displaying, interacting and navigating the contents of the scene while selecting multiple objects for which information can be retrieved.

Given this two options, frameworks suitable for building GIS applications correspond to the second category as GIS scenes are filled with multiple elements with distinctive appearance and linked to administrative data, so the possibility to identify multiple individual objects is compulsory and not always available on game engines. GIS scenes are usually composed of multiple layers and the capacity to dynamically integrate multiple elements from different sources is mandatory.

Based on this criteria, X3DOM and SceneJS frameworks are being considered on this research as they have features enabling the interaction and identification of objects, but also flexible enough to dynamically load and modify the elements of the scene.

\(^{11}\)http://scenejs.org/
\(^{12}\)http://spidergl.org/
\(^{13}\)https://github.com/mrdoob/three.js/wiki/Features
\(^{14}\)http://www.openscenegraph.org
X3DOM offers an interesting advantage over other frameworks due to its support to X3D, which is a well established standard to deliver 3D content, with multiple related ISO specifications for encoding and programming language bindings [Brutzman and Daly 2007], including DOM bindings coming directly from the World Wide Web Consortium (W3C). X3D offers different functionality levels through the use of profiles, each one offering different number of components and behavior. X3DOM also receives the benefits of all the available tools capable of creating X3D content, but also on its inclusion on the HTML5 standard.

SceneJS is targeted to create 3D scenes using a light declarative format, enabling its generation, filtering, query, storage and transport through a JSON API. The JSON API processes commands and scene graph definitions written in JSON, producing a lightweight data-interchange format and a scripting language allowing to create, query, update and destroy nodes in the scene graph. SceneJS is created to optimize the created scenes by recompiling them to recycle previous definitions, leading to less state changes and faster rendering.

2.3 3D Visualization Pipeline

With the introduction of the ‘scene’ concept in the Web3D Service (W3DS) [Shilling and Kolbe 2010] to deliver geodata, the process of integrating different 3D data sources is addressed for implementing Web Services, separating the geometry from its presentation or portrayal. Those ‘scenes’ are hierarchical structures containing display elements that can be showed and explored in Web browsers coupled with plugins, or loaded into Virtual Globe applications like Google Earth. Those 3D scenes correspond to the ‘scene graphs’ in 3D graphics. X3D, VRML, GeoVRML or similar languages used for describing scenes. X3D, VRML, GeoVRML, or similar formats used for describe scenes.

![Integration of different data sources at different levels in the Portrayal Pipeline](image)

The integration of dissimilar content coming from different sources can be modeled via the four level Portrayal Pipeline [Altmaier and Kolbe 2003], where abstract information without explicit visual characteristics is extracted from various data sources and transformed on subsequent levels, adding the presentation elements required to render images and finally present them to the user via an imaging device.

This pipeline implies that 3D scenes made of geographical elements can only be integrated on the level of ‘display elements’ or at lower levels. Given that geographical information can be expressed on different reference systems, 3D geometries should also share the same spatial reference system. Figure 2.2 shows an the Portrayal pipeline where information from four different sources (A,B,C and D) is being integrated. After extracting the corresponding information the result is transformed into display elements where appearance information is applied before being rendered.

Visualization clients can be classified according to the number of performed tasks within the Portrayal
Pipeline. Simple clients only deal with the display of images to the user while complex clients are involved in more tasks, including the rendering, the generation of display elements and even the selection of elements from the data sources. The level of complexity of the clients determines the level of required infrastructure to deploy 3D information to the user in terms of the client and related servers. It is important to note that a lower complexity client needs a counterpart on the server side to provide the missing functionality. This difference in complexity is expressed with the following categories [Altmaier and Kolbe, 2003]:

- Thick client. They can communicate with the server exchanging features, generate display elements and are free to realize complex visualization and interaction schemes.
- Medium client. The elements displayed are represented using standards like VRML, X3D or SVG and interfaces are required to provide navigation, interaction and realtime rendering capabilities.
- Thin client. They only deal with rendered images and thus, there is no need for plugins, however only static images are supported and no further interaction is possible on realtime.

![Classification of visualization clients and servers according to their complexity](image)

Following this trend, visualization of 3D scenes including underground utilities information can be achieved using clients with different capabilities and consume information provided by a variety of sources, including geographical Web services or other non geographical sources. However, the choice of 3D visualization clients is not easy as many factors are involved in its adoption, such as the compatibility with different 3D formats, portability, security, interaction features, loading mechanisms, rendering performance and image quality among other important features. Important factors that could be considered critical include the portability between platforms and the possibility to integrate the 3D visualization into an existing business logic. With this two factors, the rest can be sorted depending on its importance, however an important factor to consider is the performance, as 3D visualization is useful as long as it is kept realtime.
2.4 Rendering pipeline

WebGL is based on OpenGL ES 2.0 and as consequence its implemented over a programable pipeline, in contrast to the fixed function pipeline of OpenGL ES 1.0. This difference means that all the transformation and rasterization operations available on the fixed pipeline should be implemented as custom shaders or programs in a High Level Shading Language (HLSL), which for simple rendering functionality are always based on the same boilerplate code. Figure 2.4 shows the two pipelines side by side, where the Transform and Lighting pipes are replaced for a Vertex shader, while the Texture Environment, Color sum, Fog, and Alpha Test pipes are replaced by a Fragment Shader. These shaders are programmable units specialized on performing transformation and rendering operations at high speeds.

![Figure 2.4: Difference between OpenGL ES 1.0 fixed function pipeline and the OpenGL ES 2.0 programmable pipeline. Images from the Kronos Group website.](image)

The programable pipeline presented on Figure 2.4(b) can be further simplified as five general consecutive pipes or stages [ope, 2011], [Wei, 2005] and is a key piece to understand how is every object processed, the amount of computations involved but also how is every object displayed. This simplified pipeline consist of the following stages:

1. Vertex Shader: It has as main task the computation of the vertex position into an homogeneous coordinate system\[15\] for the following stages of the pipeline, however here is also possible to produce the attributes required for the fragment shader. A typical implementation (mimicking a fixed function pipeline) takes the input coordinates, multiplies them with the model-view projection matrix, the perspective projection and also calculates lighting parameters. The illumination parameters of a vertex is computed from the position and intensity of the light sources, eye position, vertex normal and the original color of the vertex. Is possible to perform a custom shading model within the programable shaders, in contrast to the fixed-function pipelines. Texture coordinates are passed here to other stages for basic texturing or even more advanced effects. As a programable shader, it can be used to perform almost any arbitrary operations on a per vertex basis.

2. Primitive Assembly. In this step, the processed vertices are assembled back into triangles, an operation mostly internal performed with little or no control from the user. However, the description of this step is important due to its impact on the rendering process. The three main functions performed here are Clipping, Perspective Division and Viewport Transformation. Clipping discards the primitives completely lying outside of the viewing volume or frustum, but also clipping those partially located outside. This operation is performed to save processing time by rendering only the visible primitives. The other two operations are internal and related to the rasterization process, normalizing the coordinates to the range $[-1, 1]$, and finally transforming them into window coordinates via scaling and offsets into $[0, height - 1] \times [0, width - 1]$ range.

\[15\] Also know as Clip coordinates in OpenGL due to its convenience for clipping geometries.
CHAPTER 2. BACKGROUND INFORMATION

3. Rasterization. This step creates images based on the set of primitives or triangles lying on the viewport. Three initial steps are executed before generating the actual pixels on the Fragment Shader:

- Culling: process where the back faces of a polygons are discarded if they are not visible. The user has little control on this area but can decide per drawn object the behavior of the rendered faces.
- The Depth offset is an operation used to avoid problems with polygons lying on the same plane where a ‘z-fight’ can occur and produce visual artifacts.
- The third step is called Varying Interpolation. Here, the Varying outputs produced by the Vertex Shader are a set of values interpolated between each vertex and used to draw pixels for each fragment accordingly.
- Fragment Shader. This is the second most relevant part of the programmable pipeline next to the Vertex Shader. It assigns a color to each of the produced fragments. Here is where the lighting, colors and textures are mixed with the outputs of the Vertex Shader to produce the rendered fragments. Explicit code should be included in order to achieve the desired rendered results and is executed for every pixel composing a fragment.

4. Fragment Operations, also know as Raster Operations (ROP) is the set of operations that can be performed on the rendered fragments. Some can be used to produce visual effects like shadows while others affect directly the performance of the rendering like the Depth Buffer testing, used to avoid rendering fragments occluded by the foremost objects.

- Depth-Stencil Buffer. Is the combination of a depth buffer and the stencil buffer. The depth buffer test is used tests to improve the performance, avoiding the render of non visible pixels depending of their order on the z-buffer. It The test is performed per pixel and is configurable, requires the z-buffer and affects the output buffer. The Stencil Buffer allows to perform updates on an arbitrary stencil. This is used on some systems to implement shadows using a second pass and different viewing parameters such a different camera position and specific lighting. The more realistic they become, more complex and further operations are required on additional passes.
- Colour Blending allows to mix or blend the pixel values of different fragments, typically made to achieve visual effects. This operation is performed directly on the output buffer.
- Dither or Antialiasing is used to increase the quality of the perceived output, as it may produce jaggy edges on curved or diagonal lines. This blends neighboring pixels of an object with the background to smooth the appearance of the image.

5. Rendering Target is the step where the rendered image is shown to the user. The buffer used for this purpose is the Frame Buffer, but if a ‘picking pass’ is executed a different buffer can be used and no output is displayed to the user.

From this pipeline is clear that many operations occur during rendering stage before displaying the final image to the user. These operations are mostly spread on two big areas: vertex processing and pixel processing. The amount of work performed in the Vertex Shader has a direct relationship with the geometry of the input objects in terms of triangles and vertices. On the other hand, the amount of work performed in the Fragment Shader deals mostly with the quantity of appearance options applied per declared triangle, the amount of visible pixels on the screen and finally, to the size of the rendered images. This difference will be considered again on later chapters while studying the visualization of underground utilities using WebGL clients.
2.5 Visualization performance issues

The development of advanced 3D visualization tools is mandatory to provide meaning to bidimensional drawings of underground utilities, reducing the problems when reading such cluttered drawings. However, the visualization of underground utilities produce scenes with high geometry count in terms of triangles and vertices for such narrow shapes, and moreover, require a high number of scene graph nodes to hold the composing pieces.

Typical rendering engines rely on the use of scenes graphs to maintain all the scene information. To render a frame of the scene, the scene graph is traversed and all the nodes and leafs visited. For each visited leaf, the corresponding drawing calls associated with the low level API are created and executed, fetching the geometry and setting the appearance and transformations values. Fetching the geometry implies pushing arrays of vertices, normals and triangle lists, whose transfer time depends on the size of their arrays, introducing additional latency on big models.

With an increasing number of nodes to draw, the time spent to render each frame increases in proportion to the size of the scene graph, reducing the apparent responsiveness and fading the idea of realtime visualization. In general, the time spent drawing a frame is distributed in traversing the scene graph, executing the corresponding low level API calls which have an execution cost, and fetching the geometrical information which have a direct cost associated with the amount of transferred information.

The process of rendering an object of the scene graph involves a group of API calls required to Draw a primitive where no changes in the state of the triangles occur. This process is known as ‘batch’ and includes the set of calls prior to the Draw operation, the actual drawing calls and the final calls to disable states and release variables. On every batch, no changes occur in the OpenGL state any change in the state implies a different batch. In this process, transformations matrices are set, triangles, normals and triangle lists are submitted, materials and illumination declared but also the actual program used to render the object. Figure 2.5 shows an example of an actual render batch for WebGL including the mentioned elements.

The visualization of underground utilities produces multiple objects with visual similarities, requiring multiple nodes in the scene graph and as consequence, multiple ‘batches’. This situation is triggered by the data itself as the underground features are composed of multiple pieces and segments. Underground utilities objects share typical shapes, geometric characteristics and appearance that makes them suitable to optimizations, avoiding changing states between ‘batches’. However, the capabilities of the rendering engines to foresee this optimization opportunities are dependent on the languages used for modeling the scene graphs but more important, on the actual implementation of the rendering engines. If a rendering engine does not translate and implement the optimizations provided by the 3D format, a hit in performance is expected during the rendering of the scene due to the excessive changes in state produced even if the objects do not require to change states between draws.
Figure 2.5: Example of a render batch on WebGL, composed of all the initialization steps, the actual draw call on line 100 and the release of variables after that. The yellow lines denote redundant calls, identical to previous batches. Screenshot taken using WebGL Inspector [Vanik].
The three-dimensional reconstruction of underground utilities involves an information flow between the data sources and the final visualization client passing through different formats, processes and decisions. This chapter describes this reconstruction flow, the underground objects involved and the reconstruction methods in charge of creating and mixing the geometry with appearance options.

The common building blocks used across the reconstruction methods are used on Chapter 4 to select the appropriate scene graph elements for the selected 3D frameworks. These elements constitute the basis to compute the complexity of each object and the overall scene. This complexity is a metric used to understand the differences in performance based on the amount of information encoded, but also to understand how each scene graph expands into a bigger number of low level WebGL calls, giving indication of the work performed on the GPU.

To start the description of the reconstruction process, the visualization requirements established by the enclosing project are presented and applied accordingly across the next subsections.

### 3.1 General requirements

The benefits obtained by bringing volume and distinctive shapes to non-dimensional centerlines corresponding to pipes and cables are unquestionable given the additional insight given to the scene due to the depth perception. However, providing only 3D cues is not enough to fully comprehend and recognize the elements composing a three-dimensional scene made of underground elements.

The creation of useful 3D applications for engineering purposes, impose requirements on the reconstructed scenes and composing objects. This requirements are presented as follows:

- Bring volumetric appearance and realistic dimensions based on the radius of the objects or another suitable attribute.
- Propagate the identifiers of the administrative data into the scene objects.
- Assign appearance options like colors, textures, transparency, etc, to the rendered objects based on administrative data or another set of values, aimed to apply cartographic methods and techniques.
- Assign a picking name for each identifiable object.

The creation of volumetric appearance for each the non-dimensional object require to set a series of transformation rules rules unique to each object. In the case of pipes, cylinders or extruding shapes are produced based on the radius information.

Within GIS world, the minimum logical organization unit distinguish is set at feature features level, assigning to each individual feature a unique identifier. This level of organization should be propagated from the data sources and must be mapped into the corresponding scene graph. With this mapping, typical GIS operations like selection of objects, highlighting, retrieve attribute information and even download geometrical objects can be achieved.

---

1. The questions of how to map underground utilities is not addressed on this research, but is understood that such mapping is required to produce a meaningful and usable scene.
Another typical functionality available in the GIS environments, deals with the object classification based on attribute information. This is typically performed to bring distinctive appearance to the considered entities, mapping attributes into visual appearance to ease the comprehension of the scene, explore the enclosed data repositories, produce thematic maps and in general apply cartographic methods and techniques [de By et al., 2004].

The provided datasets include certain attributes useful to apply cartographic methods including content type, owner, maintenance dates and flow capacity among many others. Given the characteristics of the 3D engines, the corresponding visual mappings or symbology can be applied through textures, colors, and transparencies associated to the geometries. With this three visual options, three of the six visual variables described by Bertin [1983] can be further applied, i.e. value, texture and color. The rest of the variables may produce problems deals with the reconstructed objects and producing collisions with the application of realistic volume.

An additional requirement in 3D visualization for engineering purposes is to identify each individual object presented on the screen through the rendered 2D image, an operation performed through a series of projections from the plane, the camera position and the selected pixel into the three-dimensional space of the scene in an attempt to recover the foremost object hit by the ray casted from the camera position and the selected pixel on the screen. This operation commonly known as ‘object picking’ imposes restrictions and requirements on the object modeling. To perform this operation, each selectable scene graph node should have a name after which such node can be identified during the ‘picking pass’. The name of each node has a different meaning from the identifiers and is used for different purposes, as the identifiers considered here are related to the administrative data records while the name is related to the pickable rendered nodes. On engineering applications, the geometry drawn on the screen is the same as the pickable geometry, however is possible to use a different picking geometry from the rendering counterpart in order to reduce the time spent on the picking pass. The actual capabilities of each 3D engine to perform this operation, the possibilities of using different geometries and the actual process are not under discussion here, but the impositions expressed on the modeling approaches, which on this research requires to assign a name to each unique identifiable object drawn, or in simple words, ‘what you draw is what you pick’.

3.2 Reconstruction flow

The three-dimensional reconstruction of underground utilities involves an information flow between the data sources and the final visualization client passing through different formats, processes and decisions.

The proposed reconstruction flow is an abstraction created to understand and implement the reconstruction process of the studied geometrical features. This flow is basic for implementing 3D web applications and resembles the proposal of [Schall et al., 2008] to convert the 2D geospatial data into 3D models and deliver them through 3D scenes. This delivery mechanism is a concept also used by [Altmaier and Kolbe 2003], [Heinen et al.] [2005], [Basanow et al.] [2008] where W3DS and other related WebServices encapsulate the server side functionality and deliver geographical scenes.

1. The reconstruction flow starts with the information stored into a spatial database, from which an arbitrary user request triggers a query (spatial or not) to the datastore, producing a series of results organized on tables where each row or record returned corresponds to an feature in the database, composed of a geometry definition and a set of attributes.

The geometrical information is stored with spatial types such as Points, LineStrings and Polygons and a choice of format options exist to encode the retrieved information. The data extracted is encoded using "Extended Well Known Text" (EWKT) [Ramsey et al., 2005], which is an extension of the OGC Well Know Text (WKT) format [Open GIS Consortium Inc 1999] used to support three or more dimensions.

---

2 Lightness or color intensity as a function of the mapped value.
3 This procedure can be accomplished in many ways, usually depending on the z-buffer, the clipping space and the precision and data types used on such operations.
4 The term name is inherited from the OpenGL API, assigned to each identifiable object.
3.2. RECONSTRUCTION FLOW

Figure 3.1: Schema of the reconstruction flow the different steps described.

on the used database

The rest of the data is held using typical DB data types and contain at least an identifier per object, an type expressed as with a numerical code or a String, radius information and related administrative information. The type field is used to assign visual characteristics based on a predefined mapping of types into appearance parameters. This mapping is the result of a classification performed a priori and can be done on different basis, but that discussion is not pertinent to this work.

2. Given the geometry, the attributes and appearance mapping, the actual creation of the 3D objects starts by choosing an reconstruction approach. The chosen approach transforms the geometric object with lower dimensionality into a 3D object based on its own conversion rules. Depending on the method used, primitives can be defined to construct the objects an reused along the procedure or custom 3D meshes used in replacement. The Appearance mapping is also used to adjust the colors applied to the object and its additional attributes into the corresponding 3D format. In the context of this thesis, starting from this step the actual decisions taken to reconstruct objects start affecting the performance at the rendering stage as the corresponding scene graphs nodes are created with different elements and organized on different ways.

3. After all the objects are reconstructed, the scene is assembled by linking the produced 3D shapes to their appearance, including identifiers, actions and names for object picking.

4. When a scene is finally assembled, it is made available to the 3D engine which parses the information to produce an internal representation suitable for rendering. In this step, the declared objects, materials, identifiers, names and additional information are converted into a scene graph.

5. After creating the scene graph, the objects are displayed to the user as sequences of two-dimensional images producing the illusion of movement. If the user interacts with the scene and requires a differ-

5Later versions of the Implementation Standard add support for 3D encoding, however PostGIS does not support it on the used version.
ent elements than those present on the scene, the hosting 3D application redirects that request into a database query, starting a new reconstruction flow and displaying the new elements to the user.

Figure 3.1 presents schematically the reconstruction flow. On the scheme, a GeoDBMS is the data source where records are extracted from. The extracted data is mixed with appearance options and reconstructed into 3D objects on the server side. After the scene is created, it is delivered to the client and presented to the user through the 3D client. Further user interactions with the scene requiring additional data are sent back to the server which translates the requests into a new query, starting again the reconstruction flow.

3.3 Underground objects

The underground objects considered on this research correspond with the objects defined on the data sources provided by the enclosing protect.

The mentioned dataset consist of pipes, cables, assets protections, the free space around current cables, pits and signs is stored into OracleSpatial. A subset corresponding to the study area is delivered into shape file format [ESRI, 1998] and later imported into PostgreSQL database with the PostGIS spatial extension. This procedure respects the structure of the delivered data to ease the management and extraction of the data. The spatial types used to encode the underground utilities are LineStrings, Point, and Polygon.

The datasets provided to carry out this research are defined only on 2D and some transformations are made to add the z-coordinate. This conversion routines are only intended for demonstration of the reconstruction approaches and the corresponding details are presented while describing each underground object. Moreover, only pipes and cables are considered for the reconstruction as the rest of the objects require additional information which was not provided. Despite the missing information, the rest of the objects are introduced and discussed towards its reconstruction, pointing out the missing requirements in terms of attributes.

Each of the underground objects is associated to an archetype, required to guide the transformation of the objects from its abstract form into 3D shapes. These archetypes correspond to cylinders for cables, pipes and pipes protection; squared sections for cable protection; extruded surfaces for the metallic sheet protection and lateral cable space. Table 3.1 presents an overview of the mentioned objects.

<table>
<thead>
<tr>
<th>DB object</th>
<th>DB Geometry</th>
<th>3D Shape</th>
<th>Attribute requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipe</td>
<td>LineString</td>
<td>Cylinder</td>
<td>radius</td>
</tr>
<tr>
<td>Cable</td>
<td>LineString</td>
<td>Cylinder</td>
<td>radius</td>
</tr>
<tr>
<td>Pipe Protection</td>
<td>Polygon</td>
<td>Cylinder</td>
<td>depth, radius</td>
</tr>
<tr>
<td>Cable Protection</td>
<td>Polygon</td>
<td>Parallelepiped</td>
<td>depth, width</td>
</tr>
<tr>
<td>Pipe Protection</td>
<td>Polygon</td>
<td>3D Face</td>
<td>depth, thickness</td>
</tr>
<tr>
<td>Pit</td>
<td>Point</td>
<td>Truncated Cone</td>
<td>depth and radius</td>
</tr>
<tr>
<td>Pit Area</td>
<td>Polygon</td>
<td>Box</td>
<td>depth</td>
</tr>
<tr>
<td>Signs</td>
<td>Point</td>
<td>Custom Object</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Underground objects present on the data source and their respective 3D shape. The DB object corresponds to the conceptual object stored in the database, the DB Geometry denotes encoding data type, the 3D shape correspond to the archetype of the reconstructed object, and the Requirements column specify which information is required for the geometrical reconstruction.
3.3. UNDERGROUND OBJECTS

3.3.1 Cables and Pipes

Pipes and cables within this research are the main studied object. The pipes and cables considered on this research are stored with LineStrings on its 2D version and the stored values correspond to their centerlines on the horizontal case. The z-coordinate is added by offsetting the values with respect to the ground level, which on this case is equal to zero. The assigned value for the z-coordinate corresponds to the vertical centerline of the pipe. Table 3.2 present the list of considered offsets applied for different pipes and cables based on their content.

<table>
<thead>
<tr>
<th>Object</th>
<th>Content Type</th>
<th>Offset (m)</th>
<th>PMKL color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipe</td>
<td>Gas</td>
<td>0.80</td>
<td>Yellow</td>
</tr>
<tr>
<td></td>
<td>Water</td>
<td>1.0</td>
<td>Blue</td>
</tr>
<tr>
<td></td>
<td>Heating</td>
<td>1.2</td>
<td>Orange</td>
</tr>
<tr>
<td></td>
<td>Drainage</td>
<td>1.5</td>
<td>Blue</td>
</tr>
<tr>
<td></td>
<td>Others</td>
<td>2.0</td>
<td>Black</td>
</tr>
<tr>
<td>Cable</td>
<td>Telecommunication</td>
<td>0.30</td>
<td>Green</td>
</tr>
<tr>
<td></td>
<td>Television</td>
<td>0.40</td>
<td>Green</td>
</tr>
<tr>
<td></td>
<td>Electricity</td>
<td>0.50</td>
<td>Red</td>
</tr>
</tbody>
</table>

Table 3.2: Offset values applied across the pipes and cables to assign a z-value. This values are applied to demonstrate the reconstruction approaches and provide visualization examples. A color is assigned to each object for presentation purposes, taking as reference a presentation model from IMKL [Kadaster, 2008].

Information about the radius of the pipe is mandatory to create a realistic reconstruction and such information is not directly available as an attribute on the dataset. In order to proceed with the reconstruction, an attribute with a textual annotation of the characteristics of the pipe is used to fill this gap. The annotation provides the corresponding external and internal diameter, but also the material. For demonstration purposes, this textual description is interpreted taking the first numerical value, but in most cases is not clear how to derive the proper radius or the obtained values are not within the usual size of the pipes. Due to the lack of complete documentation not all the tags can be processed to extract the radius value. Unprocessed tags are assigned a default diameter of 25 cm.

Additional information such as the content of the pipe or the type of ‘product’ transported by cable, owner, unique identifier and additional administrative information was provided and used accordingly during reconstruction to create assign visual appearance. Given this elements, the 3D object supporting those pipes and cables conceptually correspond to a set of chained cylinders, one for each line segment. Figure 3.2 shows the actual data before and after its reconstruction, contrasting the 2D map with a 3D scene, where pipes are classified accordingly to the transported product using a presentation model (PMKL) defined for IMKL [Kadaster, 2008].

3.3.2 Cables and pipes protection.

Protection for cables and pipes is expressed as areas on the map and recorded as polygons into the database. The corresponding 3D object representing the protection elements differs between pipes and cables. Pipe protection can be made of a big concrete pipes enclosing one or more tubes leading to a cylindrical shape, while cable protection can consist of protection jackets or squared metallic cases, requiring to reconstruct a cylinder or a rectangular shape. Another protection element consists of metallic sheets placed above the protected elements. Given the category overlaps in the dataset, the lack of attribute information, documentation, and the missing z-coordinate required to discriminate between various cases, this reconstruction is not implemented and no further work is elaborated on this research.
3.3.3 Free space trace and lateral cable space.

The free space traces are indication of routes where future cable or pipe elements can be laid out, modeled through ‘lines strokes’ on the original maps and encoded with Line Strings on the database.

The corresponding archetype for free space corresponds to ditches parallel to the ground level down to a certain depth, however on the provided database, depth and width information was not available for a complete reconstruction, but for the sake of demonstration, a predefined 1 meters depth was considered and a width of half meter.

In the case of available space at each side of the cable beds, the same modeling approach is used in the database, storing them as LineStrings and reconstructed as 3D ditches centered around the centerline. Ideally, the line drawn over the map should be related to a triangulated terrain, so the actual reconstruction could follow on detail the terrain profile. However that level of quality requires additional algorithms to compute the intersection and triangulate accordingly. These details are not considered on this research.

Figure 3.3 shows an example of the mentioned elements before and after reconstruction, bringing additional insight on the disposition of the elements and the interaction on the considered elements in the 3D scene in comparison with the map.

As stated at the beginning of this section, this elements are only discussed but not considered for the development of the research.

3.3.4 Pits and signs

The pit objects in the database are modeled in two different ways, the first corresponding to point features which are reconstructed as truncated cones while the second corresponds to areas stored as Polygons which can be reconstructed as extruded polygons. On the other hand, signs are only stored as points representing the horizontal position on the ground where objects are located. Specifics of every objects such as type, height, actual text content of the signs and even material are not present on the provided data, however the purpose of the reconstruction is to attach existing 3D objects to the locations. Further symbolization require more attributes and specific symbols which may enhance the final representation but that discussion is not discussed on this work.

Figure 3.4 shows the proposed reconstructions. A pit is depicted as a gray cone while a sign in present in red. Both elements are anchored to the ground. The rest of the elements previously mentioned are left on the scene for reference.

3.4 Reconstruction methods

The reconstruction flow described before requires the choice of a reconstruction method to convert the 2D(3D) abstract underground objects into 3D shapes suitable for visualization and interaction. The reconstruction methods set rules and procedures for interpreting the 2D data, but also define how to create a set of scene graph nodes representing the objects.

The basic scene graph nodes considered for this purpose are adapted from [Strauss and Carey, 1992] and include:

**Shape Nodes** Represent geometric or physical objects. The Shape nodes are leafs on the graph and are associated to low level representations of triangulated models. Example of of those include Indexed Face Sets, Triangle Strips, Triangle Fans, Indexed Triangle Meshes, Line Sets, among many others. This nodes can also include definitions for common objects like Cone, Sphere, Cylinder, etc. For the reconstruction purposes, only indexed representations are considered but specifically, the Indexed Triangle Meshes, which holds a list of vertices, normals and list of triangles representing the 3D object.

**Group Nodes** Used to connect other nodes into graphs or subgraphs. Examples include the Switch node, useful to implement libraries of objects and materials. The Group node is also useful to aggregate multiple nodes and share common attributes.

---

6The name of this node change across different 3D engines and scene graph implementations, so here is used only as reference.
3.4. RECONSTRUCTION METHODS

Figure 3.2: Pipes and cables before and after reconstruction, where shading, perspective and relative size cues are present. A cylindrical representation is used to assign volume and objects classified according to content type, assigning color and diameter based on administrative information.

(a) Free space on the map.  (b) Free space as 3D shapes.

Figure 3.3:  a) Free space traces are shown on the map as as dotted red lines and lateral cable space is displayed with dotted green lines. The corresponding pipes and cables are kept to show the relationship between them. b) The reconstructed elements are shown as ditches extruded at certain depth.
CHAPTER 3. RECONSTRUCTION OF UNDERGROUND UTILITIES

Figure 3.4: Abstracted and archetypical representation of the pits and signs as 3D shapes. Occlusion, perspective and shading cues are present.

**Property Nodes** Describe attributes of the objects related to the appearance and of the objects, necessary to provide distinctive appearance. Examples of the used classes include *BaseColor*, *Material*, *Normal*, *Texture2*, *Transform*, among many more. The *Transform* node is of special importance as it allows to apply affine transformations to the objects.

In addition to those, each node have provisions to attach identifiers and associate picking names, useful for the picking of objects.

With this elements in consideration, four basic decisions are taken towards reconstructing the objects:

- **Geometry** The first choice corresponds to the geometry used to encode the objects, which in this case can be based on primitives like the *Sphere* or *Cylinder* or arbitrarily defined using *Indexed Triangle Meshes*.

- **Transformations** *Transformations* are a property node and one of the basic operations used to scale, rotate or translate siblings of that node. Geometric primitives and templates need to be transformed every time to place them on the desired positions.

- **Object Reuse** *Scene graphs* allow to create objects and reuse their definitions, acting as templates for the creation of other similar objects. Such objects should be defined on local coordinates so transformations parameters can be provided to transform them during runtime. An example of template could be a traffic sign used, which can be further reused by providing only its transformations parameters.

The reconstruction based on primitives or templates can reduce the number of definitions required to assemble them. However, is not everything can be constructed using primitives, having consequences on the creation of the scene. When objects cannot be reused, they should be defined as unique geometries and expressed in world coordinates. This uniqueness increase the size of the *scene graph* but avoids transformations. Sharing nodes is a feature that can trigger hints in the 3D engines, reducing the number of definitions and avoiding change in the state of OpenGL. These saving are traduced into less representation space and less processing time.
3.4. RECONSTRUCTION METHODS

**Material Reuse** Finally, every scene graph node requires the definition of appearance in order to be rendered. The hierarchical structure of the scene graphs allows to share appearance definitions among its siblings, but some 3D engines also allow to share definitions across different nodes. Similar to the object reuse, sharing material definitions reduce representation space but also can trigger optimizations in the rendering engine to avoid state changes and improve rendering performance.

Another decision of higher level involves the modeling approach of the underground objects. On this regard, two approaches are considered: The first considers that an underground object is modeled with simple independent objects, producing multiple scene graph nodes. This is a $1:n$ relationship between the input and the output. The second approach considers that the smaller parts composing underground utilities are dependent on each other and should produce a bigger single entity. This is a $1:1$ relationship between input and the output. Reconstruction approaches following the first category are termed here as *Split* methods while the others are termed *Non-split* methods.

Figure 3.5 shows possible reconstruction paths obtained when considering all the five decisions. When a reconstruction is possible following certain path is indicated with blue, red otherwise.

![Figure 3.5: Taxonomy of the reconstruction methods considering the five main decisions. The blue lines denote possible reconstruction paths while the red lines are not possible and finish earlier. Two main classes are produced: Split and non-split methods. Split methods take a single object and produce multiple scene graph nodes. Non-split methods take one object and produce one single node.](image-url)
3.4.1 Split methods

The Split methods can be applied when the reconstructed object can be divided into smaller pieces and as consequence modeled with several scene graph nodes. In the case of a pipe modeled as LineString, each pair of consecutive points define by themselves a line conceptually corresponding to a cylinder, while each point corresponding to a change in direction. The change in direction can be modeled on different ways but here a Sphere is used. Figure 3.6 shows a reconstructed pipe made of cylinders and spheres.

![Pipe reconstruction with cylinders and spheres](image)

Figure 3.6: A pipe made of multiple objects, using primitives as building blocks: cylinders and spheres. All the objects are visually coherent but retain logical independence, as they are modeled as independent scene graph nodes, as shown by the selected cylinder in red.

After considering all the line segments of a LineString, a reconstructed object will consist of multiple single pieces visually coherent assigned to a single identifier. Under this approach every object will correspond to one or more scene graph nodes. The main advantage of this modeling approach is its simplicity for modeling pipes and cables using simpler geometries. However, a disadvantage is the increase in number of scene graph nodes created per reconstructed object, which on the current case doubles the number of segments per LineString. Figure 3.7 shows the this situation, which corresponds to the approach taken by Du and Zlatanova [2006] to reconstruct objects based on spheres and cylinders. In general, three main variations are presented to reconstruct pipes and cables and presented next. Figure 3.7 shows this three options considering material reuse.
3.4. RECONSTRUCTION METHODS

Figure 3.7: Decision tree for the split methods. The preferred decision paths are marked on blue while Orange paths denote not considered decisions. Red are impossible paths.

Split method 1: Primitive based

This method involves the use and reuse of the Sphere and Cylinder primitives and the definition of their transformation parameters to model each element of the pipe with a primitive. The transformation parameters take the geometry of the primitives and transform them to its final configuration. This transformation modify the length, width, rotation and translation of the primitives to accomplish the modeling purposes. To understand how such parameters are computed, both primitives are defined and the procedures to compute its parameters presented as follows:

**Cylinder** The Cylinder primitive definition under consideration is best described by X3D specification [Web 3D Consortium, 2008] and presented next:

"The Cylinder node specifies a capped cylinder centered at (0,0,0) in the local coordinate system and with a central axis oriented along the local Y-axis. By default, the cylinder is sized at "+1" to "+1" in all three dimensions. The radius field specifies the radius of the cylinder and the height field specifies the height of the cylinder along the central axis. Both radius and height shall be greater than zero...",

"The cylinder has three parts: the side, the top (\(Y = +height/2\)) and the bottom (\(Y = -height/2\))...".

The cylinder described before is an idealization and its display on a GPU requires its discretization. On this way, the cylinder shell is approximated with triangles. Figure 3.8(a) illustrates the Cylinder node definition used all along this document as the reference primitive while Figure 3.8(b) its corresponding discretization with the top and bottom parts removed.

Given a line segment defined by two points, \(\text{begin}\) and \(\text{end}\), the transformation parameters of the Cylinder are computed with the following procedure:

- The middle point \(p_m = \frac{\text{line.begin} + \text{line.end}}{2}\) is obtained.
- The direction vector \(\vec{a} = p_m - \text{line.end}\) is computed.
- The normal vector \(\vec{n}\) to the plane defined by \(\vec{a}\) and \(\vec{b} = (0,1,0)\) is computed using the cross product, i.e. \(\vec{n} = \vec{a} \times \vec{b}\). This normal vector \(\vec{n}\) corresponds to the rotation vector parameter.
CHAPTER 3. RECONSTRUCTION OF UNDERGROUND UTILITIES

(a) Cylinder definition

(b) Cylinder discretized

Figure 3.8: The Cylinder definition is an abstraction of the primitive and provides no guidelines for its discretization. The discrete cylinder is an approximation of its shell made with triangles. Image from the definition taken from the X3D specification [Web 3D Consortium, 2008].

• The angle $\theta$ between $\vec{a}$ and $\vec{b}$ is computed based on the dot product formula:

$$\theta = \arccos \frac{\vec{a} \cdot \vec{b}}{||\vec{a}|| \cdot ||\vec{b}||}, \theta \in [0, \Pi]$$

• The translation parameter is defined by $p_m$, moving the center of the cylinder to the final location.

• The scale parameter is expressed on the three axis, with the $x$ and $z$ axis corresponding to the radius of the pipe while the $z$ factor is half of the line segment.

Figure 3.9 shows this transformation on graphical terms, where the corresponding parameters transform the primitive defined on local coordinates to its final size and position.

**Sphere** The Sphere definition used in this document is also taken from the X3D specification [Web 3D Consortium, 2008], so is quoted next:

"The Sphere node specifies a sphere centred at (0, 0, 0) in the local coordinate system. The radius field specifies the radius of the sphere and shall be greater than zero."

This abstract definition of the sphere is not suitable for its display using GPU’s, and similar to the Cylinder primitive, an approximation of the sphere should be done with triangles. Figure 3.10 illustrates the X3D Sphere node definition used in this document while Figure 3.10(b) presents a discretized version made of triangles.

In the case of the sphere, the transformation requires only a translation from the origin to the final position, as the main assumption is that spheres are smooth enough to ignore rotations parameters.

After computing all the transformation parameters for the corresponding line segments of the pipe, the scene graph nodes created are grouped to conform the same logical object, sharing the identifier of the administrative record and the picking name. Following this procedure, the elements required to describe a pipe are reduced to the set of parameters required to transform each of its pieces based on the primitives, and the primitives themselves. An advantage of this approach is the compact scene graphs produced when the primitives are reused, as no additional geometries need to be defined for each node. However, this
3.4. RECONSTRUCTION METHODS

Figure 3.9: Parameters for the transformation of the Cylinder Primitive. All parameters are computed with respect to the center of the Cylinder, which is anchored to the origin on the local coordinate system. Rotation is computed with respect to the vector $\vec{y} = (0, 1, 0)$ and the middle of the pipe centerline.

Figure 3.10: The abstract definition of the sphere includes radius, $\theta$ and $\phi$ angles, required to apply wrapped textures over the primitive. Image taken from the X3D specification [Web 3D Consortium, 2008]. On the other side, a discretized version of the sphere is presented.
solution is dependent on the speed of the GPU to perform all the required transformations and the 3D engine to interpret the provided hints on the scene and traduce them into optimizations. Moreover, the little control on the built-in primitives complexity can increase the total workload required to render the object. This complexity is expressed in terms of the number of vertices, triangles, low-level drawing calls and vertex ordering.

Algorithm 1 shows a high level view of the creation of the transformation parameters for a pipe defined as a LineString.

<table>
<thead>
<tr>
<th>Data: LineString</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result: SceneGraph node</td>
</tr>
<tr>
<td>fileFormat ← choice(X3D, SCENEJS);</td>
</tr>
<tr>
<td>radius ← 1;</td>
</tr>
<tr>
<td>Vec3 cylinderBegin ← [0, −1, 0];</td>
</tr>
<tr>
<td>Vec3 cylinderEnd ← [0, 1, 0];</td>
</tr>
<tr>
<td>Cylinder cylinder ← new Cylinder(cylinderBegin, cylinderEnd, radius);</td>
</tr>
<tr>
<td>Vec3 sphereOrigin ← [0, 0, 0];</td>
</tr>
<tr>
<td>Shpere sphere ← new Sphere(sphereOrigin, radius);</td>
</tr>
<tr>
<td>GeometryParametersBag geometryParametersBag ← new GeometryParametersBag();</td>
</tr>
<tr>
<td>forall the line ∈ LineString do</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>lastLine ← LineString.lastLine();</td>
</tr>
<tr>
<td>sphereParameters ← computeTranslationParameters([0, 0, 0], lastLine.end);</td>
</tr>
<tr>
<td>bag.add(sphereParameters);</td>
</tr>
<tr>
<td>bag.id ← LineString.id;</td>
</tr>
<tr>
<td>bag.appearance ← LineString.appearance;</td>
</tr>
<tr>
<td>forall the parameters ∈ ParametersBag do</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

**Algorithm 1:** Creation of the transformation parameters from a high level view.

Split method 2: Custom Primitive based

The second approach for creating pipes and cables follows from the previous method. The only difference is the replacement of the built-in primitive with a custom geometry to control the quality of the objects.

Besides the use of different primitives based on the same definition, the rest of the procedures are identical to the previous method so no further details are provided to compute the transformation parameters, however the details of the actual reconstruction of the objects are important to understand the differences on final performance and the size of the objects.

The creation of the custom objects is presented from an mathematical point of view as follows:

**Cylinder:** To create a cylinder, the following parametric equation is used to describe a circle on any arbitrary plane and position:

Let be \( \vec{u}, \vec{v} \) the ortonormal vectors defining a plane anchored at point \( \vec{C} \), then the following parametric equation describes a circle with center \( \vec{C} \) and defined on the plane \( P \).
3.4. RECONSTRUCTION METHODS

\[ \vec{P} = \vec{C} + r \cos(\theta) \vec{u} + r \sin(\theta) \vec{v}, \theta \in [0, 2\pi], r > 0 \]

where \( r \) is the radius of the circle, \( \vec{C} \) circle’s center point and \( t \) is the angle parameter at which a point \( p \in P \) in a circle is defined, expressed in radians.

This equation can be rewritten for its use on non-vectorial programming languages\(^7\) as follows:

\[
\begin{align*}
P_x &= C_x + r \cdot \cos(t) \cdot u_x + r \cdot \sin(t) \cdot v_x \\
P_y &= C_y + r \cdot \cos(t) \cdot u_y + r \cdot \sin(t) \cdot v_y \\
P_z &= C_z + r \cdot \cos(t) \cdot u_z + r \cdot \sin(t) \cdot v_z
\end{align*}
\]

Based on this equation, the Cylinder primitive can be described with the mentioned equation as the triangulation created between two circles of radius \( r = 1 \) positioned on \( p_1 = (0, 1, 0) \) and \( p_2 = (0, -1, 0) \) respectively, and defined on the plane \( P = p_i + a \vec{u} + b \vec{v} \), where \( a = (1, 0, 0) \) and \( b = (0, 0, 1) \). This orientation follows the computer graphics coordinate axis, where the \( Y \) axis runs from bottom of the screen to the top of it and the \( Z \) axis runs positive towards the user, in contrast to the GIS coordinate systems where the \( Z \) axis points to the zenith and the \( y \) axis runs positive facing away of the user. Figure 3.11 explains better this difference.

Figure 3.11: Axis orientation difference between Computer Graphics and GIS worlds. Both are right handed coordinate systems and one can be transformed to the other rotating 90° around the X axis.

To start the triangulation, the circle is discretized by defining a number equally spaced points lying on its circumference. The number of points is equal to the number of \( faces \) defined per cylinder. Figure 3.12 shows the how triangulation of the cylinder starts, creating triangles using vertices \((0,1,8), (8,1,9), (1,2,9), (9,2,1)\) and so on, until all the points for the circle approximation are visited. The creation of each face be generalized with the following expression:

\[
[i, i + 1, faces + i], [faces + i, i + 1, faces + i + 1] \quad \forall \ i \in [0, faces), i \in \mathbb{N}_0
\]

**Sphere**: The Sphere considered on this research is constructed by sweeping along two angles, \( \theta \) and \( \phi \), creating bands along its trajectory, further subdivided into triangles and composing the sphere mesh. The angles follows from the sphere definition.

Formally, the sphere is created by the three parameters: the radius, \( \theta \) and \( \phi \), proceeding as follows:

- The arrays \( \theta[i], \phi[j], i \in [0, 2\pi], j \in [0, \pi], i, j \in \mathbb{N}_0 \), are created to store the value of the angle at certain band or division.

\(^7\)Programming languages where operations are generalized to apply transparently to vectors, matrices and higher dimensional arrays.
CHAPTER 3. RECONSTRUCTION OF UNDERGROUND UTILITIES

Figure 3.12: Triangulation of the cylinder shell creating triangles between the top and bottom circle.

- For each $i, j$, the vertex $(\text{radius}, \theta[i], \phi[j])$ is created.

- Normals are attached to each vertex with values identical to the vertex coordinates, important to give a smooth appearance to the sphere, but also to improve the rendering speed.

- For each two adjacent bands, $\phi[i], \phi[i+1]$ and a triangle strip is created along $\theta$ using adjacent vertices and the following expression for the triangles:

$$
(r \text{adius}, \theta[i], \phi[j]), (r \text{adius}, \theta[i+1], \phi[j]), (r \text{adius}, \theta[i], \phi[j+1])
$$

and

$$
(r \text{adius}, \theta[i+1], \phi[j]), (r \text{adius}, \theta[i+1], \phi[j+1]), (r \text{adius}, \theta[i], \phi[j+1])
$$

Figure 3.13 shows the sphere using different rendering options. Figure 3.13(a) reveals the mesh required to draw the sphere, while Figure 3.13(b) is the back culled version showing how each band is created and the corresponding triangles created between two consecutive bands. Figure 3.13(c) shows the model with hard faces as result of using normals per face, while 3.13(d) shows the model using shared normals per vertex, producing a roundish look and smooth appearance.

Rendering flat faces require unique vertices and normals per triangle, creating duplicated vertices on the shared positions and thus increasing the rendering workload.
3.4. RECONSTRUCTION METHODS

(a) Wireframe model

(b) Wireframe model with culled back faces

(c) Shaded hard faces

(d) Smooth shading

Figure 3.13: Triangulation of the sphere based on a given number of bands and a radius. In this case, eight divisions are used for $\theta$ and $\phi$
Algorithm 2 shows the code used to reconstruct the sphere adapted from X3DOM.

```java
Data: double radius; int longBands, longBands;
Result: ShapeNode node;

class List
    array[];
    append(List a, List b);
    add(item);

class ShapeNode
    List vertice, indices, normals;

class Vector3d
double x, y, z = 0.0;

Procedure createSphere(double radius, int longBands, int longBands)
ShapeNode node ← new ShapeNode();
int latNumber, longNumber;
int latitudeBands ← longBands;
int longitudeBands ← longBands;
double theta, sinTheta, cosTheta;
double phi, sinPhi, cosPhi;
double x, y, z;
for latNumber ← 0; latNumber <= latitudeBands; latNumber ++ do
    theta ← (latNumber * Math.PI)/latitudeBands;
    sinTheta ← Math.sin(theta);
    cosTheta ← Math.cos(theta);
    for longNumber = 0; longNumber <= longitudeBands; longNumber ++ do
        phi ← (longNumber * 2.0 * Math.PI)/longitudeBands;
        sinPhi ← Math.sin(phi);
        cosPhi ← Math.cos(phi);
        x ← -cosPhi * sinTheta;
        y ← -cosTheta;
        z ← -sinPhi * sinTheta;
        node.vertice.add(newVector3d(radius * x, radius * y, radius * z));
        node.normals.add(newVector3d(x, y, z));
    end
end
int first, second;
for latNumber ← 0; latNumber < latitudeBands; latNumber ++ do
    for longNumber ← 0; longNumber < longitudeBands; longNumber ++ do
        first ← (latNumber * (longitudeBands + 1)) + longNumber;
        second ← first + longitudeBands + 1;
        node.indices.add(first); /* first triangle */
        node.indices.add(second);
        node.indices.add(first + 1);
        node.indices.add(second + 1); /* Second triangle */
        node.indices.add(first + 1);
    end
end
return node;
```

Algorithm 2: Procedure for creating a sphere on the origin.

https://github.com/x3dom/x3dom/blob/master/src/nodes/Geometry3D.js
3.4. RECONSTRUCTION METHODS

Split method 3: World based primitives.

This time each cylinder and sphere are created in their final position, i.e. expressed in world coordinates instead of local coordinates. The reason to follow this approach is to skip transformations during runtime in order to save processing time. A drawback of this method is the increase in storage to define each unique object instead of reusing primitives along the scene.

Two ways are conceived to produce the set of transformed points defining the Cylinder and Sphere. The first involves to compute the transformation matrix which may transform the original object vertices to their final position while the second method implies the use of a generative approach defining the object in its final position.

The technique considered here involves the generative approach and uses the procedures described on Method 2 to create primitives. In the case of the Cylinder, the start and end points correspond to the actual begin and end coordinates of each line segment, requiring only to compute the normal plane for each line using the following routine:

For each line ∈ LineString

- Compute \( \vec{l} \), the direction vector between line.begin and line.end points, i.e.
  \[
  \vec{l} = \text{line.begin} - \text{line.end} \\
  = (\text{line.begin}.x - \text{line.end}.x, \text{line.begin}.y - \text{line.end}.y, \text{line.begin}.z - \text{line.end}.z)
  \]

- Create a random vector \( \vec{r} \) = \( \text{random}(\), \text{random}(\), \text{random}()\), statistically guaranteed to be non-collinear to the direction vector.

- Compute the first vector on the normal plane: \( \vec{a} = \vec{r} \times \vec{l} \),

- Compute the second vector on the normal plane: \( \vec{b} = \vec{l} \times \vec{a} \).

- Finally, the center point of each plane corresponds to the begin and end points.

Once all the required vectors are computed, the cylinder shell is created as described on Method 2, by creating two triangles for each consecutive set of vertices between the two circles, requiring only to define the corresponding indices.

Figure 3.14(a) shows how the cylinder is created by defining vectors \( \vec{a} \) and \( \vec{b} \) on the normal plane to the line segment. With those vectors, the equation of the circle is used to compute the corresponding points at a given angle.

In the case of the sphere, a transformation is applied to translate the vertices of the sphere to its final position with respect to the center of the original Sphere. Figure 3.14(b) shows the corresponding translation. Written in a formal notation, the transformation required is performed as follows. Given \( \vec{C}_{x,y,z} \) the center of the Sphere in world coordinates and \( \vec{v} \) a vertex on the sphere on Local Coordinated, the transformed vertices \( \vec{v}' \) are computed as follows:

\[
\vec{v}'_{x,y,z} = (\vec{v}_{x,y,z} + \vec{C}_{x,y,z}), \forall \vec{v}_{x,y,z} \in \text{Sphere.vertices},
\]

3.4.2 Non-Split methods

The second main branch on the reconstruction taxonomy correspond to approaches where the reconstructed objects are modeled visually and logically as a single scene graph node. The difference imposes a stricter control on the modeling of objects involving more restrictions in order to reduce the number of scene graph nodes representing the object. A first simple and naive approach just appends all the objects modeled with triangles into a single list before rendering, reducing the final node count but not the triangle and vertex count. A further refinement referred here as ’stitching’, require to compute the actual intersection points between the composing objects, avoiding unseen triangles, allowing vertex recycling and storage savings.
CHAPTER 3. RECONSTRUCTION OF UNDERGROUND UTILITIES

(a) Cylinder creation in world coordinates. (b) Sphere creation in world coordinates based on translation of the vertices.

Figure 3.14: Creation of primitives in world coordinates: a) The cylinder is created by rotating around the circle anchored at $c$ over the plane defined by $\vec{a}$ and $\vec{b}$. The procedure is applied for the bottom and top part. b) The sphere is created by translating the vertices to the corresponding destination with respect to the sphere center in the origin.

Figure 3.15 shows the corresponding branch of non–split methods considering two approaches and the available decisions for reconstructing the objects. Is clear that the only options available is the choice of reusing materials, which is preferred over non using.

Figure 3.15: Considered non split methods: Append and Stitch. There are not many options available on this reconstruction approaches except for reusing materials.
3.4. RECONSTRUCTION METHODS

Split method 4: Appending world based geometries

One of the main drawbacks of the ‘split’ methods is the count increase of scene graph nodes, traduced into additional render calls or batches per modeled object. To deal with this situation, an important fact about the scene graphs can be used: every node is an explicit list of independent triangles. These independent triangle lists can be ‘appended’ at the end of each other and drawn instead within a single render call or ‘batch’, as long as they belong to the same administrative object and share appearance options.

This technique is typically used in the computer graphics world to render multiple objects using the least amount of render calls, packing as much objects as possible within the allowed array size. A disadvantage of this method is the amount of nodes that can be packed, limited by the identity of the objects. If two nodes with different identities are packed, they lose their identity and the picking name, violating the requirements established at the beginning of this chapter. To overcome this limitation, additional logic should be added to the 3D engine. However, this work is specifically aimed at modeling underground scenes from a high level point of view, so the access to such features is not considered. Without access to low-level features, the one to one relationship between the rendered geometry and picking geometry cannot be broken.

Getting back to the specifics to this approach, appending lists of triangles require geometries expressed on world coordinates, similar identifiers and common appearance options. Given that the lists are expressed using indexed triangulations, an offset should be applied on the triangle indices before appending them. With this elements, appended lists create nodes with larger geometries without the need to create groups. In terms of implementation, this approach takes the output created by the previous ‘split’ method and appends the world custom primitives into a single node. This procedure append each individual component, i.e. the vertices, the normals and the indices at the end of the desired node.

Algorithm 3 presents an implementation to append two nodes, used to append all the corresponding nodes form the previous reconstruction method:

```
Data: ShapeNode targetNode, additionalNode;
Result: ShapeNode targetNode;
Procedure appendShapeNodes( targetNode, additionalNode )
    targetNode.vertices.append( additionalNode.vertices );
    targetNode.normals.append( additionalNode.normals );
    /* Computing the offset before appending */
    offset ← targetNode.indices.size;
    forall the i ∈ [0, targetNode.indices.size) do
        additionalNode.indices[i] += offset;
    end
    targetNode.indices.append( additionalNode.indices );
    return targetNode;

class List
    array[];
    append (List a, List b);

class ShapeNode
    List vertices;
    List indices;
    List normals;
```

**Algorithm 3**: Procedure for appending a scene graph nodes into another. Related class definitions are also included.
CHAPTER 3. RECONSTRUCTION OF UNDERGROUND UTILITIES

Split method 5: Stitching world based geometries

The final approach considered in this research is an improvement over previous methods aimed to reduce the vertex count, triangle count and scene graph node count by sharing vertices, avoiding spheres and modeling the pipe as a single object without breaks and ruptures. The reduction in the number of nodes has a direct impact on the performance, reducing the number of processed elements and the corresponding number of 'batches'.

The basis of this method is the removal of the spheres placed between consecutive cylinders and the adjustment of their length to match both ends. To accomplish this, the pipes are extended and cut at the bisecting plane between two consecutive centerlines.

The bisecting plane is defined by two vectors \( \vec{c} \) and \( \vec{n} \), which are computed in the following way:

- The direction vectors \( \vec{a}, \vec{b} \) corresponding to the consecutive lines are computed, facing opposite directions anchored in the common point.
- The first vector lying on the bisecting plane is computed with as sum of the director vectors, i.e.
  \[ \vec{c} = \vec{a} + \vec{b} \]
- The second orthogonal vector lying on the plane is computed using the cross product between previous vectors, i.e. \( \vec{n} = \vec{a} \times \vec{c} \), so is guaranteed to lie on the desired plane.

Three cases are considered here with respect to the bisecting angle between consecutive lines, having further implications on the stitching:

1. If the angle \( \alpha \) is almost \( \pi \), then the norm of \( \vec{c} \) is almost zero, i.e. \( ||\vec{c}|| \approx 0 \), introducing numerical issues with further usage of this vector, so a different approach should be taken to compute such plane.
2. \( \frac{\pi}{2} < \alpha \ll \pi \), the vector \( \vec{c} \) is numerically stable so the plane can be computed with this vector.
3. If the angle is smaller than \( \pi/4 \), the vector is also numerically stable but the extension of pipes will produce visual glitches as the segments will grow and be joined at distances bigger than the radius with respect to the joining point. In this case, another approach should be considered, which under this method is simply fixed by adding a Sphere. Under this assumption is expected that the odd cases fallen on this situation are scarce, otherwise the benefits of having a low vertex and triangle count will be neglected. Another solution not adopted here require to add additional segments to solve this problem but introducing deviations from the recorded data.

In addition to the bisecting plane, the normal plane of the first line segment is computed to create the first cylinder. The normal plane is obtained using the procedure expressed on the Method 3.

Once both the bisecting planes and normal plane are found, the extension and cutting of the pipes can proceed as follows.

1. Given the normal planes of the first segment, the first Cylinder points are computed, both for the start and the end, designated as \( C_{start}[i = 0][j] \) and \( C_{end}[i = 0][j], j \in [0, faces) \)
2. Given the line defined between \( C_{start}[i][j] \) and \( C_{end}[i][j] \) and the plane \( B[i] \), the intersection point \( B[i][j] \) between them is computed. In addition, the end point of the cylinder is updated, i.e. \( C_{end}[i][j] = B[i][j] \).
3. Given the direction vector \( \vec{a} \) and the point \( B[i][j] \), the start and endpoints of the next cylinder are computed, i.e. \( C_{start}[i' = i + 1][j] = B[i][j] \) and \( C_{end}[i' = i + 1][j] = B[i][j] + \vec{a} \). This way, lines required to create the next cylinder are defined using vectors parallel to the centerline.
4. The procedure is repeated from the second step until all the segments have been traversed and until all the faces \( j \) have been considered.
3.4. RECONSTRUCTION METHODS

Figure 3.16 shows this process for a single line, where it is extended an the intersection with the bisecting plane used as the stitch point. Later this point is projected into the normal plane of the cylinder to create the following cylinder.

Figure 3.16: Stitching procedure based on the bisecting plane between two consecutive line segments and the direction vectors \( \vec{a} \) and \( \vec{b} \). The cylinder is extended and cut with a bisecting plane \( B[i] \) while the direction vector \( \vec{a} \) parallel to the center line is used to generate the following cylinder starting from the intersection points \( B[i][j] \).

Once all the start and end points for each cylinder are computed on the bisecting plane, a triangulation is created between those points using the expressions defined on previous method for each cylinder. Figure 3.17 shows the triangulation created between each set of points with different rendering options.

Figure 3.17: Four different renderings for the stitched Geometries. Cylinders are extended and cut to share the intersection points. A cylinder is highlighted to show this cuts. Eight faces per cylinder were used to simplify the visualization and also to show the round appearance obtained with that quality using shared normals.
The reconstruction methods presented on the last section have for objective the transformation of abstract information representing pipes and cables into 3D objects with unique identity, suitable for applying cartographic operators and interaction through the ‘picking’ functionality. To achieve this, the objects are transformed into scene graph elements, assuming the existence of specific nodes to store the geometry, appearance, transformations and additional parameters.

Section 4.1 examines the WebGL frameworks previously selected, X3DOM and SceneJS, and present the actual nodes and parameters used to implement the reconstruction procedures from Chapter 3. Section 4.2 shows the assembled scenes with selected examples for both frameworks. Finally, Section 4.3 presents the analysis on the reconstruction and assembly process for the reconstruction methods at three levels: primitive level, reconstructed object level and scene level.

4.1 Target Frameworks

The selection of 3D frameworks brings a series of possibilities and imposes restrictions to express 3D objects and scenes with respect to its appearance, relationships, actions and identity. Based on them, possibilities, restrictions and the reconstruction methods, the corresponding elements of each framework are selected and presented.

Before starting directly with the descriptions, relevant information about each framework is presented about its purpose and objectives, but also about the corresponding scene graph structure, useful to compare them and understand what could be the differences between them.

4.1.1 X3DOM and X3D

X3DOM is a framework and a runtime created to integrate 3D content the web. Is implemented entirely on JavaScript and based on WebGL technology.

X3DOM is a minimal X3D runtime system with support for animations and events, mapping the X3D scene content into DOM, thus making use of CSS and DOM events the main update mechanism. This situation turns X3DOM into a non standard X3D runtime, where the Sensor nodes alike are dropped [Behr et al., 2011].

X3D is an ISO standard for delivering 3D content composed of both a format and runtime specification for which X3DOM is a runtime example. On this regard, X3DOM depends on the X3D specification for the definition of its content. X3D is a declarative language allowing to construct simple objects based on predefined geometric primitives like the Sphere, Cube, Cylinder and Cone, but also have provisions to declare complex geometries using planar faces or polygons, arbitrary triangular meshes constructed with triangle strips, triangle sets or triangle fans in both indexed and non indexed versions. The complete list of geometric nodes in X3D contains more elements than the ones exposed here and is split on several profiles, which are the mechanisms of X3D to provide additional capabilities and functionality. Those geometric nodes simplify the construction of objects independently of the rendering engine and its low level implementation, allowing high level representations without having to specify low level details.

The primitives considered relevant for the reconstruction include the Sphere, Cylinder and IndexedTri-
angleSet node. This last node is used to specify arbitrary 3D objects defining their vertices, triangle indices, normals, colors and textures. Listing 4.1 shows examples of the mentioned X3D primitive nodes encoded into XML.

```
...<Shape>
  <Cylinder DEF='CYL_8' bottom='false' top='false' solid='true' radius='1.0'/>
</Shape>

<Shape>
  <Sphere DEF='SPH_8' solid='true' radius='1.0'/>
</Shape>

<Shape>
  <IndexedTriangleSet DEF='CUSTOM_CYL_8' ccw='true'
    normalPerVertex='true' colorPerVertex='false'
    index='0 1 8 8 1 9
          1 2 9 9 2 10
          2 3 10 10 3 11 ...
'>
    <Coordinate point='1.0 0.0 -1.0
                   0.707 0.707 -1.0
                   0.0 1.0 -1.0
                   -0.707 0.707
                   -1.0 -1.0 0.0 -1.0 -0.707 -0.707 ...
'>
    <Normal vector='1.0 0.0 0.0
                  0.707 0.707 0.0
                  0.0 1.0 0.0 ...'/>
  </IndexedTriangleSet>
</Shape>
Listing 4.1: X3D Primitives considered for the reconstruction: Cylinder, Sphere and IndexedTriangleSet.

X3D has a provision to assign unique identifiers to every element through the DEF attribute and reference it later through a USE tag, allowing to construct scenes made of a high number of similar objects without the burden of duplicated definitions. Due to the mapping of the X3D content into the DOM tree, the identifier of each node is specified via the id attribute. X3DOM automatically creates the id attribute for each node based on the DEF value.

The appearance of each object is declared via the Appearance node which has space to define textures and materials and can be reused similarly via the DEF and USE attributes.

The Transform node declares a set of transformations to rotate, scale and translate the enclosed elements via parameters, leaving to the user the computation of the actual values.

Listing 4.2 shows the described nodes, defining a Transformation, Appearance node and reusing the geometry declared on Listing 4.1.

```
...<Transform translation='-5.924 -32.828 20.676'>
  <Shape DEF="id_10" onclick='clickAction(this);'>
    <Appearance DEF='MAT_0'>
      <Material diffuseColor='0.8 0.8 0.8' emissiveColor='0.0 0.0 0.0'
                  ambientIntensity='0.2' specularColor='0.0 0.0 0.0'/>
    </Appearance>
    <IndexedTriangleSet USE='CUSTOM_CYL_8'/>
  </Shape>
</Transform>
..."}

Listing 4.2: X3D transformation and appearance nodes.

In X3D, Sensor nodes are used to interact with the contents of the scene, executing actions when the sensor is triggered. However, the integration of X3D into the web architecture requires modifications to

---

1 On the introduced Scene Graph, those nodes correspond to a Sphere Kit, a Cylinder Kit and an IndexedTriangleMesh
the X3D content, dropping their support on the runtime [Behr et al., 2011]. In replacement, browser events shall be used making reference to Javascript functions. The specifics of the actions triggered by the browser events are left out of the discussion, but of interest for this research is the ‘onClick’ event, bound to the ‘picking’ functionality. This allows to retrieve the object under the selected position and execute a user defined function.

The “object picking” functionality implemented in X3DOM allows to select up to \(2^{16} - 1\) individual nodes. However, without relying on the rendered image it is possible to query individual objects given their identifiers using the DOM API.

To avoid hardcoding functions, a simple approach is taken towards processing multiple actions. It consists of declaring a function to process all the requests having as parameter the identifier of the node or the object reference itself. This decouples the scene declaration from the taken actions, relying then on external code to define interaction modes, something out of scope of X3D.

This approach increases the modularity of the scene and reduces the need to rewrite the scenes to attach specific actions. Examples of such modes include the hypothetical ‘query’ mode, where administrative information should be retrieved for a given identifier, result of ‘picking’ the object below the mouse. Another example is the ”measure” mode, where given two selected objects a function is used to measure the distance between them.

Listing 4.11 shows a snippet of code implementing the mentioned function for processing different actions, decoupling declaration from implementation for the ‘onClick’ event. The parameter of the function is the ‘picked’ object below the selected position in the rendered image.

```javascript
var mode='query';

function clickAction(object){
    var id = object.DEF;
    if(mode == 'query'){
        var results = query(id);
        showResults(results,'query');
        return;
    }
    if(mode == 'measure'){
        if(firstID == null){
            firstID=id;
            return;
        }else{
            secondID=id;
            var distance = measure(firstID,secondID);
            showResults(distance,'measure');
            firstID=null;
            secondID=null;
            return;
        }
    }
}
```

Listing 4.3: Javascript function showing how to decouple the node actions from the node declaration via a common processing function.

In its current state, X3DOM is still in a experimental stage with a large number of implemented X3D nodes, dropping the more complicated nodes due to the limited processing speed on JavaScript. However, to reconstruct the desired objects, the required primitives, the appearance information, transformations, the use of identifiers and reuse of nodes are available, making the reconstruction of the scenes possible within the limitations of the runtime.

X3D and X3DOM do not provide direct support for linking administrative information via special nodes
nor support for any special format. However, given the nature of the web platform, it is possible to create Javascript code to retrieve the appropriate information from the data sources using the identifier, which, as described in the beginning of this chapter, follows from the administrative data.

### 4.1.2 JSON and SceneJS

SceneJS is a 3D framework for WebGL composed of a 3D engine and a JSON API, enabling to generate, filter, query, store and transport 3D content. This API is in constant evolution, and new characteristics are being added on successive iterations. This process is flexible and can add additional functionalities without going through the long standardization process like X3D.

SceneJS JSON API defines 3D content in a **scene graph** structure. The scene graph nodes consist of two main parts:

- Generic node information like children's, parent, type, identifier, among others.
- Type-specific information like geometry and material attributes.

The part regarding specific information is termed **core** and can be reused among nodes of the same type, using the **coreId** attribute which is a pointer to the defined object. This allows share vertex arrays to reduce the number of WebGL calls and the redundant vertex use. The attribute **coreId** is different from the **id** attribute. The id makes reference to the whole object while used as a reference to the same object.

The **geometry** node is used to draw arbitrary objects and support various low level rendering modes via points, lines and triangles. The only primitive available for the reconstruction is the **Sphere** node, as no Cylinder definition exists. To solve that issue, a **Cylinder** node is created following the definition from Chapter 3.4.1 and reused via the **coreId** attribute.

Listing 4.4 shows the JSON scene graph example with a **Sphere** node and a set of indexed triangles, inside a **geometry** node. The example makes reference to a **material** node defined in the Listing 4.5.

```json
{
  type: "material",
  coreId: "MAT_54",
  nodes: [
    {
      type: "sphere",
      id: "sphere1",
      slices: 6,
      rings: 12
    },
    {type: "geometry",
     id: "29389",
     coreId: "29389"
     primitive: "triangles",
     positions: [196.848, -2.0, 76.1, 196.834, -1.903, 76.062, 196.802, -1.863, 75.972, 196.769, -1.903, 75.881],
     normals: [0.046, 0.0, 0.128, 0.032, 0.096, 0.09, 0.0, 0.136, 0.0, -0.032, 0.096, -0.09],
     indices: [0, 1, 2, 2, 1, 3]
    }]
}
```

Listing 4.4: Javascript function showing how to decouple the node actions from the node declaration via a common processing function.
4.2. SCENE ASSEMBLY

Appearance information is defined with material node, which can be grouped into a library node for further reuse, allowing to reduce the footprint and apply avoid changing states in OpenGL ES, traduced as optimizations. Listing 4.5 shows a library of material making use of the coreId attribute.

```
{ type: "library",
  nodes: [
    {type: "material",
     coreId: "MAT_54",
     id: "MAT_54",
     baseColor: { r:1.0, g:0.843, b:0.0},
     specularColor: { r:0.0, g:0.0, b:0.0},
     emit:1.0,
     specular:1.0,
     shine:0.8
    },
    ...
  ]
}
```

Listing 4.5: SceneJS materials library created using JSON scene graph API. The coreId attribute allows to reference this nodes for further usage.

The ‘object picking’ functionality is enabled for nodes placed below a name node, allowing to select more than one geometry at the same time with the same picking name. Listing 4.6 shows an example enabling ‘picking’ functionality for a geometry node named wall_1, which geometry was previously defined on Listing 4.4 and is referenced using the coreId attribute.

```
{ type: "name",
  name: "wall_1",
  nodes:[{
    type: "geometry",
    coreId: "29389"
  }]
}
```

Listing 4.6: Javascript function showing how to decouple the node actions from the node declaration via a common processing function.

4.2 Scene Assembly

In order to understand how the scenes are created with the mentioned elements, selected examples are presented showing the basic structure of a scene for both the target frameworks with the following structure:

- Header
- Materials library (if materials are being reused).
- Objects library (if objects are being reused).
- Nodes definition

Within the header section, details related to the scene are located, including camera, lighting, and other parameters. The header part is not relevant for the development of this research, so no details are provided. In contrast, libraries of materials and objects are presented here as they are considered on the construction of the objects. Finally, the nodes are presented containing specific details like identifiers, appearance used and display options.
CHAPTER 4. RECONSTRUCTION IMPLEMENTATION

This structure allows to represent all the possibilities expressed on Section 3.4 and graphically expressed in Figure 3.5. Using this template, examples for both frameworks are built, presented and explained accordingly in the following sub sections.

4.2.1 X3DOM

The X3D scenes constructed here use the presented template and focus on the structure rather than the actual content due to formatting constraints of this document. The X3D header contains the required XML and X3D tags to define the document and prepare the scene.

```xml
<?xml ?>
<!DOCTYPE X3D PUBLIC "ISO//Web3D//DTD X3D 3.2//EN"
"http://www.web3d.org/specifications/x3d-3.2.dtd">
<X3D xmlns:xsd='http://www.w3.org/2001/XMLSchema-instance'
profile='Full' version='3.0'
...>
  <Scene>
    <Background DEF='Background'
     groundColor='0.7 0.4 0.3'
skyColor='1 1 1'/>
  </Scene>
</X3D>
```

Listing 4.7: Javascript function showing how to decouple the node actions from the node declaration via a common processing function.

The appearance and object libraries are constructed using `Switch` nodes, allowing to define the objects and reference them without being rendered. On all the considered reconstruction approaches, material reuse is recommended to reduce the size of the scene and the number of submitted operations. However, object reuse is not always possible to use, specially when the reconstructed objects are expressed in world coordinates. Listing 4.8 shows a material library with only one appearance node, followed by a an object library containing a unitary `Sphere` and `Cylinder`.

```xml
<Switch DEF='materialLibrary' whichChoice='-1'><!-- -1 means no choice -->
  <Shape>
    <Appearance DEF='MAT_0'><!-- -1 Material Identifier -->
      <Material diffuseColor='1.0 0.843 0.0'
        emissiveColor='0.0 0.0 0.0'
        ambientIntensity='1.0'
        specularColor='0.0 0.0 0.0'
        shininess='0.8' />
    </Appearance>
    <Box/><!-- dummy geometry -->
  </Shape>
</Switch>

<Switch DEF='objectLibrary' whichChoice='-1'>
  <Shape>
    <Cylinder DEF='CYL_8' bottom='false' top='false' solid='true' />
  </Shape>
  <Shape>
    <Sphere DEF='SPH_8' solid='true' />
  </Shape>
</Switch>
```

Listing 4.8: X3D example showing how to create a material and objects library.

Finally, the displayed objects are presented and grouped accordingly. In this example, an `IndexedTriangleSet` is drawn in world coordinates reusing the `appearance` defined on the previous library, followed by a `Sphere`
inside a Transform node, reusing also materials and the Sphere object. Each created shape is assigned a function into the on\textit{Click} browser event, relaying on the before mentioned function to separate declaration of the scene from the implementation.

```xml
<Group id='29389' DEF='29389' onclick='clickAction(this);' >
  <Shape >
    <Appearance USE='MAT_0' />
    <IndexedTriangleSet ccw='true'
      normalPerVertex='true'
      solid='true'
      colorPerVertex='false'
      index='0 1 8 8 1 9
        1 2 9 9 2 10 ...' >
      <Coordinate point='196.848 -2.0 76.1
        196.834 -1.903 76.062
        196.802 -1.863 75.972 ...' />
      <Normal vector='0.046 0.0 0.128
        0.032 0.096 0.09
        0.0 0.136 0.0 -0.032 ...' />
    </IndexedTriangleSet>
  </Shape>
</Group>

<Transform translation='-5.924 -32.828 20.676' >
  <Shape DEF="test10" onclick='clickAction(this);' >
    <Appearance USE='MAT_0' />
    <Sphere solid='true' USE='SPH_8'/>
  </Shape>
</Transform>
```

Listing 4.9: X3D example showing the geometry nodes built reusing materials and objects from the library.

### 4.2.2 SceneJS

The corresponding SceneJS example following the mentioned structure is presented next and highlights the differences and similarities between 3D formats. Libraries in SceneJS are defined with a \texttt{library} node and the affected elements are defined within the \texttt{nodes} node. Listing 4.10 shows a library with only one material and two geometry nodes, one for a custom Cylinder defined as \texttt{triangles} and another object using the built-in \texttt{Sphere}. In both cases, the reusability is achieved via the \texttt{coreId} attribute. SceneJS allows to define \texttt{material} nodes without geometric content, being different than the approach taken by X3D, where the appearance needs to include a geometry.

```json
{type:"node",
 nodes: [
  {type:"library", nodes: [
    { type:"material",
      coreId:"MAT_0",
      id:"MAT_0",
      baseColor: { r:0.8, g:0.0, b:0.0},
      specularColor: { r:0.0, g:0.0, b:0.0},
      emit:0.2,
      specular:1.0,
      shine:0.2
    },
    {type:"geometry",
      coreId:"CYL_8",
      primitive:"triangles",
      positions:[1.0, 0.0, -1.0,
```
CHAPTER 4. RECONSTRUCTION IMPLEMENTATION

Listing 4.10: SceneJS library example declaring one material definition and two geometry nodes, one for a custom cylinder and another for the sphere. All the elements in the library include the \texttt{coreId} attribute for its further reuse.

Once all the required libraries are declared, the renderable nodes with geometry can be declared. On this example a \textit{sphere} is assigned a ‘picking’ name, a \textit{material} reused from the library, transformed and finally reused from the library. A clear difference with X3DOM is the order of transformations is explicitly declared from the top to the bottom, applying first a scaling, then rotation, finally a translation.

Listing 4.11: SceneJS example showing the transformation of the sphere, scaling, rotating and translating it to its final position.
4.3 Complexity analysis

The reconstruction of objects based on the exposed approaches produce different complexity numbers which should be measured in order to analyze and quantify the performance impact of the assembled scenes. On this regard no distinction is made between the output of specific 3D formats as complexities are applicable to both 3D formats, being the only difference the encoding size for the different XML and JSON tags used next to the materials definition. To relate the complexity of the used primitives with the complexity of the complete scene, the description is made on three levels:

1. Primitive
2. Reconstructed Object
3. Scene

With this separation, the numbers obtained on each level are computed and used on further levels, as complexity propagates all along the reconstruction process from the data sources up to the assembled scenes.

4.3.1 Primitive analysis

To analyze the primitives complexity, the number of divisions or approximations used per object is used here as a parameter to compute the vertex count of the objects. A higher number of divisions per object produce higher visual quality. The visual quality relates to the aesthetics of the rendered objects and its evaluation is a subjective task out of the scope of this research.

The number of divisions is used here as parameter for the construction of the objects and is referred here as the ‘quality’ parameter, abbreviated with the letter $q$ and used later for the complexity computations.

The default numbers used in X3DOM are used here as the reference for high visual quality. The Cylinder in X3DOM is made of 32 patches or faces to approximate its round appearance without considering the top and bottom part, which are not used for the reconstruction. On the other hand, the Sphere is constructed using 24 bands per axis to produce 576 faces.

Figure 4.1 shows pipes made with Cylinders and Spheres, rendered with different vertex count and thus producing different visual quality. The lower quality objects can be considered acceptable despite they showing some flat appearance for the Cylinder while producing artifacts at the unions with the Spheres. This situation is less apparent when increasing the vertex count.

In both cases, Cylinder and Sphere nodes shares vertices and normals between consecutive faces, useful to reduce the vertex count but also to force the round appearance of the objects as the corresponding normals are ‘averaged’ between faces.

With the quality of the objects, concrete numbers are obtained but also the analytical expressions required to understand how the numbers relate. Table 4.1 shows the complexity of the Cylinder at different qualities while Table 4.2 shows the corresponding numbers for the Sphere.

<table>
<thead>
<tr>
<th>Quality $(q)$</th>
<th>Vertices $(2q)$</th>
<th>Normals $(2q)$</th>
<th>Triangles $(2q)$</th>
<th>Indices $3(2q)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>192</td>
</tr>
<tr>
<td>24</td>
<td>48</td>
<td>48</td>
<td>48</td>
<td>144</td>
</tr>
<tr>
<td>16</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>96</td>
</tr>
<tr>
<td>8</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>48</td>
</tr>
</tbody>
</table>

Table 4.1: Complexity numbers obtained for the modeled Cylinder primitive
Figure 4.1: Examples of rendered pipes at different qualities using normals per vertex to produce a round appearance. On top, $q=8$ shows how the round appearance persists at acceptable levels with low quality. In the middle, $q=16$, produces a higher vertex count and shows improvements while $q=24$ at the bottom is close to the default appearance, producing good results.
4.3. COMPLEXITY ANALYSIS

### Table 4.2: Complexity numbers obtained for the Sphere primitive.

<table>
<thead>
<tr>
<th>Quality (q)</th>
<th>Vertices (q^2)</th>
<th>Normals (q^2)</th>
<th>Triangles (2(q^2))</th>
<th>Indices (3(2(q^2)))</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>576</td>
<td>576</td>
<td>1152</td>
<td>3456</td>
</tr>
<tr>
<td>16</td>
<td>256</td>
<td>256</td>
<td>512</td>
<td>1536</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
<td>64</td>
<td>128</td>
<td>384</td>
</tr>
</tbody>
</table>

4.3.2 Reconstructed objects analysis

The next step considers the complexity of the reconstructed objects after integration of their pieces, being the pipe the main object under study. As each pipe has a variable number of segments, an average number is taken as an estimation of the general complexity per reconstructed object. Considering the provided dataset, the average value \( k = 8 \) is obtained with a standard deviation \( \sigma = 18 \).

The numbers used to quantify the complexity include:

- Number of input objects from the database and the corresponding produced scene graph nodes.
- Number of declared vertices, normals and triangles
- Number of materials applied
- Number of transformed nodes denoted as matrix changes
- Memory and encoding footprint in bytes
- Number of processed vertices and triangles
- Number of transformed vertices

With this numbers, different tables are presented, each one composed of three sections:

- The first section denotes the number of input elements and output nodes
- The second, denotes the complexity of the total declared nodes considering object reuse. This numbers correspond to the size of the files created representing the scene.
- The third correspond to the number of processed elements per rendered frame, which is different than the number of declared elements.

Table 4.3 shows the corresponding complexity per pipe in abstract terms, taking into consideration that some methods reduce the total footprint by reusing declarations, having as consequence a reduced memory footprint on the client. However, the number of processed vertices, triangles and material changes remain the same. The memory footprint is computed considering 4 bytes to represent a floating number, used for both vertices and normals, while the triangle indices require three entries per triangle and each vertex index is encoded using 2 byte integers. For the encoding part, an average of eight bytes are considered to encode the float data while the index part is approximated using the average number of characters required to encode all the indices for a sphere and cylinder. To encode the sphere at using 24 bands, 3456 indices are required and approximately 3.5 characters. With the sphere, 192 indices are used and 2.8 characters are needed to encode them.

---

2Local to world coordinate transformations.
<table>
<thead>
<tr>
<th>Factor</th>
<th>Method 1 (M1)</th>
<th>Method 2</th>
<th>Method 3 (M3)</th>
<th>Method 4</th>
<th>Method 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB Objects</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3D nodes</td>
<td>2k</td>
<td>2k</td>
<td>2k</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Declared

<table>
<thead>
<tr>
<th></th>
<th>Method 1 (M1)</th>
<th>Method 2</th>
<th>Method 3 (M3)</th>
<th>Method 4</th>
<th>Method 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>$2q_{cyl} + q_{sph}^2$</td>
<td>idem M1</td>
<td>$k(2q_{cyl} + q_{sph}^2)$</td>
<td>idem M3</td>
<td>idem M3</td>
</tr>
<tr>
<td>Normals</td>
<td>$2q_{cyl} + q_{sph}^2$</td>
<td>idem M1</td>
<td>$k(2q_{cyl} + q_{sph}^2)$</td>
<td>idem M3</td>
<td>idem M3</td>
</tr>
<tr>
<td>Triangles</td>
<td>$2q_{cyl} + 2q_{sph}^2$</td>
<td>idem M1</td>
<td>$k(2q_{cyl} + 2q_{sph}^2)$</td>
<td>idem M3</td>
<td>idem M3</td>
</tr>
<tr>
<td>Matrix Changes</td>
<td>2k</td>
<td>2k</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Materials</td>
<td>2k</td>
<td>2k</td>
<td>2k</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Memory Footprint (bytes)</td>
<td>$2(\text{vertices})4$</td>
<td>+</td>
<td>idem</td>
<td>idem</td>
<td>idem</td>
</tr>
<tr>
<td>Encoding Footprint (bytes)</td>
<td>$2(\text{vertices})8$</td>
<td>+</td>
<td>$2(\text{vertices})8$</td>
<td>+</td>
<td>idem M3</td>
</tr>
<tr>
<td></td>
<td>$3(2q_{cyl})2.8$</td>
<td>+</td>
<td>$k(3(2q_{cyl}))2.8$</td>
<td>+</td>
<td>idem M3</td>
</tr>
<tr>
<td></td>
<td>$3(2q_{sph})3.5$</td>
<td>+</td>
<td>$k(3(2q_{sph}))3.5$</td>
<td>+</td>
<td>idem M3</td>
</tr>
</tbody>
</table>

### Processed

<table>
<thead>
<tr>
<th></th>
<th>Method 1 (M1)</th>
<th>Method 2</th>
<th>Method 3 (M3)</th>
<th>Method 4</th>
<th>Method 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>$k(2q_{cyl} + q_{sph}^2)$</td>
<td>idem M1</td>
<td>idem M1</td>
<td>idem M1</td>
<td>$k(2q_{cyl})$</td>
</tr>
<tr>
<td>Triangles*</td>
<td>$k(2q_{cyl} + 2q_{sph}^2)$</td>
<td>idem M1</td>
<td>idem M1</td>
<td>idem M1</td>
<td>$k(2q_{cyl})$</td>
</tr>
<tr>
<td>Transformed Vertices</td>
<td>$3(\text{Triangles}^*)$</td>
<td>Indices</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.3: Abstract complexity of the reconstructed objects given an average number $k$ of segments per pipe.

Given this abstract table, specific numbers for the considered average number of segments and quality of the primitives are obtained. Table 4.4 shows the numbers obtained when default quality is used. This table shows how methods using spheres for the reconstruction have an overall higher complexity than the ‘stitch’ method which don’t use them. Spheres require a quadratic number of vertices, normals and triangles in terms of the quality of the objects, while Cylinders complexity grow linearly. Tables 4.5, 4.6 and 4.7 show the corresponding numbers obtained with reduced quality models.
4.3 Complex Analysis

<table>
<thead>
<tr>
<th>Factor</th>
<th>Method 1</th>
<th>Method 2</th>
<th>Method 3</th>
<th>Method 4</th>
<th>Method 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB Objects</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3D nodes</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

| Declared       |          |          |          |          |          |
| Vertices       | 640      | 640      | 5120     | 5,120    | 512      |
| Normals        | 640      | 640      | 5120     | 5,120    | 512      |
| Triangles      | 1,216    | 1,216    | 9728     | 9,728    | 512      |
| Materials      | 16       | 16       | 16       | 1        | 1        |
| Matrix Changes | 16       | 16       | 0        | 0        | 0        |
| Memory Footprint | 5,931  | 5,931    | 47,445   | 47,445   | 4,437    |
| Encoding Footprint* | 33,114 | 33,114   | 176,474  | 176,474  | 16,922   |

| Processed      |          |          |          |          |          |
| Vertices       | 5,120    | 5,120    | 5,120    | 5,120    | 512      |
| Triangles      | 9,728    | 9,728    | 9,728    | 9,728    | 512      |
| Transformed    | All      | All      | None     | None     | None     |

Table 4.4: Complexity of the average object with parameters \(k = 8, q_{sphere} = 24, q_{cylinder} = 32\)

<table>
<thead>
<tr>
<th>Factor</th>
<th>Method 1</th>
<th>Method 2</th>
<th>Method 3</th>
<th>Method 4</th>
<th>Method 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB objects</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3D nodes</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

| Declared       |          |          |          |          |          |
| Vertices       | 624      | 624      | 4,992    | 4,992    | 384      |
| Normals        | 624      | 624      | 4,992    | 4,992    | 384      |
| Triangles      | 1,200    | 1,200    | 9,600    | 9,600    | 384      |
| Matrix Changes | 16       | 16       | 0        | 0        | 0        |
| Materials      | 16       | 16       | 16       | 1        | 1        |
| Memory Footprint | 5,792  | 5,792    | 46,336   | 46,336   | 3,328    |
| Encoding Footprint* | 32,467 | 32,467   | 172,243  | 172,243  | 12,691   |

| Processed      |          |          |          |          |          |
| Vertices       | 4,992    | 4,992    | 4,992    | 4,992    | 384      |
| Triangles      | 9,600    | 9,600    | 9,600    | 9,600    | 384      |
| Transformed Vertices | 3(Triangles) | 3(Triangles) | None     | None     | None     |

Table 4.5: Complexity of the average object with parameters \(k = 8, q_{sphere} = 24, q_{cylinder} = 24\)

4.3.3 Scene Analysis

Based on previous numbers and the restrictions imposed by the different 3D frameworks, static numbers are obtained when assembling a 3D scene. The numbers presented denote the complexity of the scene before the rendering phase and are numbers required to compare the performance of each reconstruction.
These numbers are computed considering average sizes for the input objects, so when applied to real scenes the actual numbers may differ. Table 4.8 shows the relationships between the parameters of the objects and the estimated numbers obtained.

Next, examples changing different quality factors are used to show the actual values on the theoretical reconstructed scene made from 161 objects, equal to the actual number of records from the study area within the provided dataset. This numbers are useful to give a concrete picture of the complexity based on the different reconstruction methods.
4.3. COMPLEXITY ANALYSIS

<table>
<thead>
<tr>
<th>Factor</th>
<th>Method 1</th>
<th>Method 2</th>
<th>Method 3</th>
<th>Method 4</th>
<th>Method 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB Objects</td>
<td>161</td>
<td>161</td>
<td>161</td>
<td>161</td>
<td>161</td>
</tr>
<tr>
<td>3D nodes</td>
<td>2nk</td>
<td>2nk</td>
<td>2nk</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>Declared</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vertices</td>
<td>$2q_{cyl} + q_{sph}^2$</td>
<td>idem</td>
<td>$nk(2q_{cyl} + q_{sph}^2)$</td>
<td>idem</td>
<td>$k(2q_{cyl})$</td>
</tr>
<tr>
<td>Normals</td>
<td>$2q_{cyl} + q_{sph}^2$</td>
<td>idem</td>
<td>$nk(2q_{cyl} + q_{sph}^2)$</td>
<td>idem</td>
<td>$k(2q_{cyl})$</td>
</tr>
<tr>
<td>Triangles</td>
<td>$2q_{cyl} + 2q_{sph}^2$</td>
<td>idem</td>
<td>$nk(2q_{cyl} + 2q_{sph}^2)$</td>
<td>idem</td>
<td>$k(2q_{cyl})$</td>
</tr>
<tr>
<td>Matrix Changes</td>
<td>2nk</td>
<td>2nk</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Material Changes</td>
<td>2nk</td>
<td>2nk</td>
<td>n</td>
<td>n</td>
<td></td>
</tr>
<tr>
<td>Memory Footprint</td>
<td>2$(vertices)4$</td>
<td>+</td>
<td>idem</td>
<td>idem</td>
<td>idem</td>
</tr>
<tr>
<td>Encoding Footprint</td>
<td>3$(Triangles)2$</td>
<td>+</td>
<td>idem</td>
<td>idem</td>
<td>idem</td>
</tr>
<tr>
<td>(bytes)</td>
<td>2$(vertices)8$</td>
<td>+</td>
<td>idem</td>
<td>idem</td>
<td>idem</td>
</tr>
<tr>
<td>(bytes)</td>
<td>3nk$(2q_{cyl})2.8 + 3*$</td>
<td>+</td>
<td>$nk(3(2q_{cyl}))2.8 +$</td>
<td>$nk(3(2q_{cyl}))3.5$</td>
<td>idem</td>
</tr>
<tr>
<td></td>
<td>$2nk(q_{sph}^2)3.5$</td>
<td></td>
<td>$nk(3(2q_{sph}))3.5$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pushed Vertices</td>
<td>$nk(2q_{cyl} + q_{sph}^2)$</td>
<td>idem M1</td>
<td>idem M1</td>
<td>idem M1</td>
<td>idem M1</td>
</tr>
<tr>
<td>Processed Vertices</td>
<td>3nk$(2q_{cyl} + 2q_{sph}^2)$</td>
<td>idem M1</td>
<td>idem M1</td>
<td>idem M1</td>
<td>idem M1</td>
</tr>
<tr>
<td>Triangles</td>
<td>$nk(2q_{cyl} + 2q_{sph}^2)$</td>
<td>idem M1</td>
<td>idem M1</td>
<td>idem M1</td>
<td>idem M1</td>
</tr>
<tr>
<td>Transformed</td>
<td>$3nk(2q_{cyl} + 2q_{sph}^2)$</td>
<td>idem M1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Vertices</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.8: Analytical complexity of the reconstructed scene given an average number $k$ of segments per LineString and $n$ objects reconstructed.

Table 4.9 shows the numbers obtained using the default quality objects and a clear differences is visible on several aspects. One aspect is the memory and encoding footprint of methods using transformations against those not requiring them, where the object reuse give a clear advantage. The second big difference is visible on the amount of pushed vertices versus the amount of processed vertices, which is dependent on the number of triangles. The ‘Stitch’ method offers a good balance between the number of declared vertices and both memory and encoding footprint, but also have the lowest complexity compared with the rest methods. This advantage is obtained because this method does not require spheres, which require an quadratic amount of vertices with respect to the quality of the objects. Tables 4.10, 4.11 and 4.12 show the corresponding numbers obtained for lower quality numbers for all the discussed methods.

In general, is clear that several variables are involved in the rendering of the pipes and cables. However, is not clear which of those variables have a relevant impacting on the performance. In general, a clear difference is visible between the amount of declared and the processed information when using objects defined on local coordinates, but also are several differences are visible between the values in terms of their order of magnitude. Even with this information is not clear the rendering performance of the different reconstruction methods and the relevance the variables should be know. Some variables should have a bigger impact on the performance but conclusions can only be obtained during the rendering phase. This requirement leads to the next section, where a testing framework is developed and further used to compare the affecting variables.
### Table 4.9: Complexity of the average scene with parameters $k = 8$, $q_{sph} = 24$, $q_{cyl} = 32$

<table>
<thead>
<tr>
<th>Factor</th>
<th>Method 1</th>
<th>Method 2</th>
<th>Method 3</th>
<th>Method 4</th>
<th>Method 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB Objects</td>
<td>161</td>
<td>161</td>
<td>161</td>
<td>161</td>
<td>161</td>
</tr>
<tr>
<td>3D nodes</td>
<td>2,737</td>
<td>2,737</td>
<td>2,737</td>
<td>161</td>
<td>161</td>
</tr>
</tbody>
</table>

#### Declared

<table>
<thead>
<tr>
<th>Factor</th>
<th>Method 1</th>
<th>Method 2</th>
<th>Method 3</th>
<th>Method 4</th>
<th>Method 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>640</td>
<td>640</td>
<td>824,320</td>
<td>824,320</td>
<td>82,432</td>
</tr>
<tr>
<td>Normals</td>
<td>640</td>
<td>640</td>
<td>824,320</td>
<td>824,320</td>
<td>82,432</td>
</tr>
<tr>
<td>Triangles</td>
<td>1,216</td>
<td>1,216</td>
<td>1,566,208</td>
<td>1,566,208</td>
<td>82,432</td>
</tr>
<tr>
<td>Matrix Changes</td>
<td>2,737</td>
<td>2,737</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Material changes</td>
<td>2,737</td>
<td>2,737</td>
<td>2,737</td>
<td>161</td>
<td>161</td>
</tr>
<tr>
<td>Memory Footprint</td>
<td>5,931</td>
<td>5,931</td>
<td>7,638,699</td>
<td>7,638,699</td>
<td>714,411</td>
</tr>
<tr>
<td>Encoding Footprint*</td>
<td>33,114</td>
<td>33,114</td>
<td>26,390,874</td>
<td>26,390,874</td>
<td>2,638,362</td>
</tr>
</tbody>
</table>

#### Processed

<table>
<thead>
<tr>
<th>Factor</th>
<th>Method 1</th>
<th>Method 2</th>
<th>Method 3</th>
<th>Method 4</th>
<th>Method 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pushed Vertices</td>
<td>824,320</td>
<td>824,320</td>
<td>824,320</td>
<td>824,320</td>
<td>82,432</td>
</tr>
<tr>
<td>Processed Vertices</td>
<td>4,698,624</td>
<td>4,698,624</td>
<td>4,698,624</td>
<td>4,698,624</td>
<td>247,296</td>
</tr>
<tr>
<td>Triangles</td>
<td>1,566,208</td>
<td>1,566,208</td>
<td>1,566,208</td>
<td>1,566,208</td>
<td>82,432</td>
</tr>
<tr>
<td>Transformed Vertices</td>
<td>4,698,624</td>
<td>4,698,624</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Table 4.10: Complexity of the average scene with parameters $k = 8$, $q_{sph} = 24$, $q_{cyl} = 24$

<table>
<thead>
<tr>
<th>Factor</th>
<th>Method 1</th>
<th>Method 2</th>
<th>Method 3</th>
<th>Method 4</th>
<th>Method 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB objects</td>
<td>161</td>
<td>161</td>
<td>161</td>
<td>161</td>
<td>161</td>
</tr>
<tr>
<td>3D nodes</td>
<td>2,737</td>
<td>2,737</td>
<td>2,737</td>
<td>161</td>
<td>161</td>
</tr>
</tbody>
</table>

#### Declared

<table>
<thead>
<tr>
<th>Factor</th>
<th>Method 1</th>
<th>Method 2</th>
<th>Method 3</th>
<th>Method 4</th>
<th>Method 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>624</td>
<td>624</td>
<td>803,712</td>
<td>803,712</td>
<td>61,824</td>
</tr>
<tr>
<td>Normals</td>
<td>624</td>
<td>624</td>
<td>803,712</td>
<td>803,712</td>
<td>61,824</td>
</tr>
<tr>
<td>Triangles</td>
<td>1,200</td>
<td>1,200</td>
<td>1,545,600</td>
<td>1,545,600</td>
<td>61,824</td>
</tr>
<tr>
<td>Matrix Changes</td>
<td>2,737</td>
<td>2,737</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Material Changes</td>
<td>2,737</td>
<td>2,737</td>
<td>2,737</td>
<td>161</td>
<td>161</td>
</tr>
<tr>
<td>Memory Footprint</td>
<td>11,066,496</td>
<td>11,066,496</td>
<td>11,066,496</td>
<td>11,066,496</td>
<td>680,064</td>
</tr>
<tr>
<td>Encoding Footprint*</td>
<td>32,467</td>
<td>32,467</td>
<td>25,731,283</td>
<td>25,731,283</td>
<td>1,978,771</td>
</tr>
</tbody>
</table>

#### Processed

<table>
<thead>
<tr>
<th>Factor</th>
<th>Method 1</th>
<th>Method 2</th>
<th>Method 3</th>
<th>Method 4</th>
<th>Method 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pushed Vertices</td>
<td>803,712</td>
<td>803,712</td>
<td>803,712</td>
<td>803,712</td>
<td>61,824</td>
</tr>
<tr>
<td>Processed Vertices</td>
<td>4,636,800</td>
<td>4,636,800</td>
<td>4,636,800</td>
<td>4,636,800</td>
<td>185,472</td>
</tr>
<tr>
<td>Triangles</td>
<td>1,545,600</td>
<td>1,545,600</td>
<td>1,545,600</td>
<td>1,545,600</td>
<td>61,824</td>
</tr>
<tr>
<td>Transformed Vertices</td>
<td>4,636,800</td>
<td>4,636,800</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
### 4.3. Complexity Analysis

<table>
<thead>
<tr>
<th>Factor</th>
<th>Method 1</th>
<th>Method 2</th>
<th>Method 3</th>
<th>Method 4</th>
<th>Method 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB objects</td>
<td>161</td>
<td>161</td>
<td>161</td>
<td>161</td>
<td>161</td>
</tr>
<tr>
<td>3D nodes</td>
<td>2,737</td>
<td>2,737</td>
<td>2,737</td>
<td>161</td>
<td>161</td>
</tr>
</tbody>
</table>

**Declared**

<table>
<thead>
<tr>
<th>Factor</th>
<th>Method 1</th>
<th>Method 2</th>
<th>Method 3</th>
<th>Method 4</th>
<th>Method 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>288</td>
<td>288</td>
<td>370,944</td>
<td>370,944</td>
<td>41,216</td>
</tr>
<tr>
<td>Normals</td>
<td>288</td>
<td>288</td>
<td>370,944</td>
<td>370,944</td>
<td>41,216</td>
</tr>
<tr>
<td>Triangles</td>
<td>544</td>
<td>544</td>
<td>700,672</td>
<td>700,672</td>
<td>41,216</td>
</tr>
<tr>
<td>Matrix Changes</td>
<td>2,737</td>
<td>2,737</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Material Changes</td>
<td>2,737</td>
<td>2,737</td>
<td>2,737</td>
<td>161</td>
<td>161</td>
</tr>
<tr>
<td>Memory Footprint</td>
<td>2,667</td>
<td>2,667</td>
<td>3,434,667</td>
<td>3,434,667</td>
<td>357,205</td>
</tr>
<tr>
<td>Encoding Footprint*</td>
<td>14,861</td>
<td>14,861</td>
<td>11,875,853</td>
<td>11,875,853</td>
<td>1,319,181</td>
</tr>
</tbody>
</table>

**Processed**

<table>
<thead>
<tr>
<th>Factor</th>
<th>Method 1</th>
<th>Method 2</th>
<th>Method 3</th>
<th>Method 4</th>
<th>Method 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pushed Vertices</td>
<td>370,944</td>
<td>370,944</td>
<td>370,944</td>
<td>370,944</td>
<td>41,216</td>
</tr>
<tr>
<td>Processed Vertices</td>
<td>2,102,016</td>
<td>2,102,016</td>
<td>2,102,016</td>
<td>2,102,016</td>
<td>123,648</td>
</tr>
<tr>
<td>Triangles</td>
<td>700,672</td>
<td>700,672</td>
<td>700,672</td>
<td>700,672</td>
<td>41,216</td>
</tr>
<tr>
<td>Transformed Vertices</td>
<td>2,102,016</td>
<td>2,102,016</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.11: Complexity of the average scene with parameters $k = 8$, $q_{sph} = 16$, $q_{cyl} = 16$

<table>
<thead>
<tr>
<th>Factor</th>
<th>Units</th>
<th>Method 1</th>
<th>Method 2</th>
<th>Method 3</th>
<th>Method 4</th>
<th>Method 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB objects</td>
<td>number</td>
<td>161</td>
<td>161</td>
<td>161</td>
<td>161</td>
<td>161</td>
</tr>
<tr>
<td>3D nodes</td>
<td>number</td>
<td>2,737</td>
<td>2,737</td>
<td>2,737</td>
<td>161</td>
<td>161</td>
</tr>
</tbody>
</table>

**Declared**

<table>
<thead>
<tr>
<th>Factor</th>
<th>Units</th>
<th>Method 1</th>
<th>Method 2</th>
<th>Method 3</th>
<th>Method 4</th>
<th>Method 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>number</td>
<td>80</td>
<td>80</td>
<td>103,040</td>
<td>103,040</td>
<td>20,608</td>
</tr>
<tr>
<td>Normals</td>
<td>number</td>
<td>80</td>
<td>80</td>
<td>103,040</td>
<td>103,040</td>
<td>20,608</td>
</tr>
<tr>
<td>Triangles</td>
<td>number</td>
<td>144</td>
<td>144</td>
<td>185,472</td>
<td>185,472</td>
<td>20,608</td>
</tr>
<tr>
<td>Matrix Changes</td>
<td>number</td>
<td>2,737</td>
<td>2,737</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Material Changes</td>
<td>number</td>
<td>2,737</td>
<td>2,737</td>
<td>2,737</td>
<td>161</td>
<td>161</td>
</tr>
<tr>
<td>Memory Footprint</td>
<td>bytes</td>
<td>736</td>
<td>736</td>
<td>947,968</td>
<td>947,968</td>
<td>178,603</td>
</tr>
<tr>
<td>Encoding Footprint*</td>
<td>bytes</td>
<td>4,038</td>
<td>4,038</td>
<td>3,298,758</td>
<td>3,298,758</td>
<td>659,590</td>
</tr>
</tbody>
</table>

**Processed**

<table>
<thead>
<tr>
<th>Factor</th>
<th>Units</th>
<th>Method 1</th>
<th>Method 2</th>
<th>Method 3</th>
<th>Method 4</th>
<th>Method 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pushed Vertices</td>
<td>number</td>
<td>103,040</td>
<td>103,040</td>
<td>103,040</td>
<td>103,040</td>
<td>20,608</td>
</tr>
<tr>
<td>Processed Vertices</td>
<td>number</td>
<td>556,416</td>
<td>556,416</td>
<td>556,416</td>
<td>556,416</td>
<td>61,824</td>
</tr>
<tr>
<td>Triangles</td>
<td>number</td>
<td>185,472</td>
<td>185,472</td>
<td>185,472</td>
<td>185,472</td>
<td>20,608</td>
</tr>
<tr>
<td>Transformed Vertices</td>
<td>number</td>
<td>556,416</td>
<td>556,416</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.12: Complexity of the average scene with parameters $k = 8$, $q_{sph} = 8$, $q_{cyl} = 8$
The availability of different reconstruction approaches is useful to address scenarios where different restrictions apply in terms of system memory, storage capacity, transmission bandwidth and processing power. However, despite the immediate complexity numbers characterizing each reconstructed object and assembled scene, the rendering performance is still unknown.

Without information about the actual performance is impossible to have a complete assessment of the reconstruction approaches and give a clear advise on which one is more appropriate for which situation. To fill this gap, a testing framework is designed and developed focused on performance of the scenes created with the presented reconstruction approaches.

This framework extracts from the presented methods the factors directly affecting the performance having implications during rendering stage. In this sense, the framework is not aimed towards evaluating reconstructed scenes directly but towards measuring the impact of specific declarative building blocks used during the reconstruction of the underground objects. This separation produces an abstract test which is used to answer the following performance-testing questions:

- What are the factors with largest negative impact on performance?
- What is the degree of influence of each factor?
- What is the base line for performance? Or in other words, what is the worst combination of factors?
- What is the best combination of factors?

In brief, to answer the mentioned questions the introduced testing framework considers a combination of factors to create automated tests, reduce the variability induced by the user due to interaction with the scene, reduce the variability of the data sets, but also showing the applicability to real data, requisites to produce repeatable results and give an objective assessment using this framework.

### 5.1 Terminology

Before proceeding with the development of the tests, some definitions are introduced as they conform the basis for the developed testing framework.

- **Performance testing**: is a type of testing intended to determine the responsiveness, throughput, reliability, and/or scalability of a system under a given workload. [Meier et al., 2007].
- **Responsiveness**: is the ability of a system to complete a task within a given time.
- **Frame rate**: a.k.a frame frequency, is the frequency at which an imaging device produces unique consecutive images or frames, also expressed in frames per second (FPS).
- **Triangles per second (TPS)**: is the total number of rendered triangles for time unit. Its value is the product of the rendered FPS and the number of triangles per frame (TPF), i.e.

\[
TPS = \frac{triangles}{s} = FPS \times TPF = \frac{frames}{s} \times \frac{triangles}{frame}
\]
• **Batches per second (BPS):** Is the total number of rendered batches for time unit. Its value is the product of the rendered FPS and the number of rendered nodes or batches per frame (BPF), i.e.

\[ BPS = \text{batches/s} = \text{FPS} \times \text{BPF} = \text{frames/s} \times \text{batches/frame} \]

• **Baseline:** is the starting point for comparison for benchmarking and is used to "evaluate the effectiveness of subsequent performance-improving changes to the system or application" [Meier et al., 2007]. This baseline is useful as long as the characteristics of the test remain the same except for those explicitly changing.

## 5.2 Performance factors.

Based on the reconstruction approaches described in the previous chapter, it is hard to have a clear idea of the performance of a scene based only on their description and the obtained complexity numbers. With subtle differences occurring between methods, it is difficult to understand how different construction choices affect the final performance of the assembled scenes. To overcome this situation, the constructive building blocks used for each reconstruction and affecting the performance at the rendering stage are considered as testing variables, presented in the following list:

• The use of Primitives. In most of the cases, pipes can be constructed using Primitives, based on the same *Sphere* and *Cylinder* definition. This objects can be reused if the parameters used on their creation are identical, so no need to create additional copies besides the differences on appearance. Furthermore, Primitives can be created using optimized representations where the number of processed vertices are minimized by making an efficient use of the Post Transform Vertex Cache\(^1\), reducing process time. In cases where the 3D framework Primitives are not being used, custom objects are created and used instead.

• The use of Transformations, required to convert objects expressed in local coordinates to world coordinates. Its use introduces a matrix change per render batch and also imposes the transformation of all the required vertices accordingly. If an object is made of \(n\) triangles, then \(3n\) vertices should be transformed at most for every frame.

• The reuse of scene graph nodes\(^2\). This feature has implications in the client side, allowing to reuse geometry definitions, but also triggering rendering optimizations by saving some buffer bindings\(^3\) only if the previous rendered object was based on the same geometry, reducing the size of the batch but also and avoiding the expensive binding.

• The reuse of materials. Similar to previous case, helps to reduce the memory usage in the client by referencing existing materials, but also helps to reduce the batch size if the previous rendered object had the same definition.

• Number of scene graph nodes. This number denotes the amount of rendered objects per frame and corresponds to the number of batches submitted per frame or BPF. The more nodes on the scene graph, the more batches submitted. This number is one of the variables used during tests to determine the scalability of the considered Frameworks.

• Number of triangles. This number correspond to the number of processed triangles and also to the number of processed vertices, as every triangle is made of three vertices. This number indicates the actual GPU workload in the Vertex Shader, but also the workload in other parts of the rendering pipeline including the Primitive Assembly stage and the Fragment Shader.

---

1. A small Cache memory where the processed vertices on the VertexShader are stored for it further reuse
2. Also known as instancing in CityGML and many 3D formats, prototyping in X3D, and Node referencing in SceneJS.
3. The process of copying a buffer from the client memory into the server memory or GPU memory.
5.2. PERFORMANCE FACTORS.

Each one of the mentioned factors affects different parts of the rendering pipeline making it difficult to compare them simultaneously, however strong interactions occur in some cases, making easier to construct comparison criteria and reduce the total number of combinations for the involved variables.

The use of transformations has a strong relationship with the node reuse, as only nodes defined on local coordinates can be reused. If the nodes can be reused, some memory can be saved and data do not need to be pushed again into the graphics card, saving transfer calls\( ^4 \). In this situation, transformations need to be applied for every processed vertex, requiring a graphics card capable of handling the additional workload. In the opposite situation, whenever transformations are not used, nodes cannot be reused. Despite this drawback, the omitted transformations save processing time at the expense of storing and transferring extra data into additional vertex and index buffers.

Materials should be defined for every rendered node to produce an image, but defining them for every node produces longer batches overall. If the rendered node can share appearance with the previously rendered node, the same OpenGL state can be used and some operations can be saved per batch and as consequence processing time. Under this assumption, using materials give optimization hints to the rendering engine in order to avoid setting appearance options again for every frame. However, the reuse of materials has a strong interaction with the reuse of objects, as their combination may break the optimizations introduced by each other.

Table 5.1 shows this situation rendering two different shapes, Sphere and Cube, using two different appearance options, Red and Blue, producing four different batches in total. In the example, rendering the nodes in different order produces different state savings, ranging from 1 to 3. This example shows the importance to know which of the two considered factor produces more benefits, required to establish the best rendering order.

<table>
<thead>
<tr>
<th>Batch rendering order</th>
<th>State savings</th>
<th>Objects Reused</th>
<th>Materials Reused</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>Second</td>
<td>Third</td>
<td>Fourth</td>
</tr>
<tr>
<td>Sphere Red</td>
<td>Sphere Blue</td>
<td>Cube Red</td>
<td>Cube Blue</td>
</tr>
<tr>
<td>Sphere Red</td>
<td>Sphere Blue</td>
<td>Cube Blue</td>
<td>Cube Red</td>
</tr>
<tr>
<td>Sphere Red</td>
<td>Cube Red</td>
<td>Sphere Blue</td>
<td>Cube Blue</td>
</tr>
<tr>
<td>Sphere Red</td>
<td>Cube Red</td>
<td>Cube Blue</td>
<td>Sphere Blue</td>
</tr>
<tr>
<td>Sphere Red</td>
<td>Cube Blue</td>
<td>Cube Red</td>
<td>Sphere Blue</td>
</tr>
<tr>
<td>Sphere Red</td>
<td>Cube Blue</td>
<td>Sphere Blue</td>
<td>Cube Red</td>
</tr>
</tbody>
</table>

Table 5.1: Example showing the impact of rendering order to avoid changing states between frames. Two different shapes are defined, Sphere and Cube using two different materials: red and blue. Rendering the nodes in different order produces different state savings, ranging from 1 to 3. This example show the importance to know which of the two considered factor produces more benefits, required to establish the best rendering order.

Finally, the availability of primitives and its characteristics is dependent on the 3D engine implementation where internal features can be implicitly exploited and avoid creating duplicated data, duplicated data pushing, and even some object reordering performed to minimize the total calls. Despite the potential benefits of using primitives, is not always possible to control the quality of the objects and then, maybe the use of a simpler custom geometry can save transformations and representation space at the expense of not triggering the right optimizations.

\( ^4 \)In the OpenGL jargon, this is known as buffer binding.
5.2.1 Factors impact

Up to this point, the use of Primitives versus custom objects, the use of transformations, object reuse, and materials reuse, are factors applied among different reconstruction methods.

Under this assumptions, changes in these factors affect the performance on three main areas:

- Producing larger batches, which should be pushed by the CPU
- Increasing the amount of the transferred data
- Increasing the amount of processed data

According to this, the selected factors affect mostly the size of the batches, the transfer stage where Vertex and Index data are pushed into the memory and finally, the Vertex Shader where the majority of the operations will occur. Under this assumption, no big changes in the workload are being introduced to the FragmentShader as no special lighting or textures are being considered, or special code modified per object, thus simplifying the testing. Table 5.2 briefly shows the impact of each factor on the three considered locations during rendering and described in rough terms.

<table>
<thead>
<tr>
<th>Factor</th>
<th>Batch Size</th>
<th>Data Transferring</th>
<th>Data Processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitives</td>
<td>Smaller Batches</td>
<td>Less Transferred Data</td>
<td>Some optimizations may apply</td>
</tr>
<tr>
<td>Transformations</td>
<td>Bigger Batches</td>
<td>Non Significant</td>
<td>Additional processing per Vertex</td>
</tr>
<tr>
<td>Node Reuse</td>
<td>Smaller Batches</td>
<td>Less Transferred Data</td>
<td>No changes</td>
</tr>
<tr>
<td>Materials Reuse</td>
<td>Smaller Batches</td>
<td>Non Significant</td>
<td>No changes</td>
</tr>
</tbody>
</table>

Table 5.2: Effect of the considered factors on three different stages of the rendering process.

5.2.2 Factors combination

In sum, the four factors expressed above can be compared against each other producing a number of combinations which should be tested, characterizing their influence based on their performance impact and introducing an order relationship. This ordering helps to find a ‘Worst Case scenario’ or baseline where further improvements can be measured.

Table 5.3 shows the combination of the four considered factors in different combinations. Each combination produces a test used to find the performance baseline. The total number of combinations using four factors adds up to sixteen, however when objects are already transformed, the reuse of objects and primitives cannot be realized, reducing the number of combinations to ten.

<table>
<thead>
<tr>
<th>Test Abbreviation</th>
<th>Primitives</th>
<th>Transformation</th>
<th>Object Reuse</th>
<th>Material Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_T_O_M</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>P_T_O_nM</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>P_T_nO_M</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>P_T_nO_nM</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>nP_T_O_M</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>nP_T_O_nM</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>nP_T_nO_M</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>nP_T_nO_nM</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>nP_nT_nO_M</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>nP_nT_nO_nM</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 5.3: Combination of factors considered for the testings.

Besides the mentioned elements, the number of triangles and vertices are considered a function of a given reconstruction method, the reconstructed objects, its number of segments and a the given visual
quality parameter. The total number of produced scene graph nodes is dependent on the method followed and used here as testing variable to produce different tests with the mentioned combination.

### 5.2.3 Relationship with the reconstruction methods

The relationship between the reconstruction methods and the expressed factors is fundamental to understand how to map the results obtained from the tests into each of the reconstructions methods independently of the underlying data. Table 5.4 shows for each reconstruction method, the applicable combinations of factors. In general, more than one combination of factors is applicable to each reconstruction method. However, the combinations providing more benefits may be preferred over the others.

<table>
<thead>
<tr>
<th>Test Abbreviation</th>
<th>Reconstruction Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_T_O_M</td>
<td>Local Primitives</td>
</tr>
<tr>
<td>P_T_O_nM</td>
<td></td>
</tr>
<tr>
<td>P_T_nO_M</td>
<td></td>
</tr>
<tr>
<td>P_T_nO_nM</td>
<td></td>
</tr>
<tr>
<td>nP_T_O_M</td>
<td>Custom Local Primitives</td>
</tr>
<tr>
<td>nP_T_O_nM</td>
<td></td>
</tr>
<tr>
<td>nP_T_nO_M</td>
<td></td>
</tr>
<tr>
<td>nP_T_nO_nM</td>
<td></td>
</tr>
<tr>
<td>nP_nT_nO_M</td>
<td>Custom World Primitives</td>
</tr>
<tr>
<td>nP_nT_nO_nM</td>
<td>Append World geometries</td>
</tr>
<tr>
<td></td>
<td>Stitch World geometries</td>
</tr>
</tbody>
</table>

Table 5.4: Relationship between the reconstruction methods and the proposed factors.

### 5.3 Baseline performance testing

The definition of a performance baseline is required to compare and understand the different factors involved in the reconstruction of underground utilities. This baseline is produced within a controlled environment, producing results directly comparable and required to assess and understand their interactions and impact at rendering stage.

Without such line is hard to evaluate if a reconstruction approach will benefit by using the considered factors or even if they are producing an inferior performance. This information is handy to justify the time spent implementing specific reconstruction methods and provide expectations on the final performance.

To produce this baseline, an abstract test should be conducted to remove variations introduced by specific scenes made of different types of elements with variable size and spatial extent. This test is to be conducted with only with one object type but varying 1) the number of considered nodes in the scene graph and 2) the quality of the mentioned objects. This two axis help to understand how does a reconstruction method behaves on the tested framework but specifically, how it scales with an increasing number of scene graph nodes, and second, how does the reduction on the visual quality of a given reconstructed object influences the performance, expressed also as the triangle count.
5.3.1 Performance Hypothesis

This testing framework allows to compare between different hypothesis and rank their influence directly linked to the reconstruction methods, useful to give a performance expectation on how the mentioned reconstruction approaches will behave.

In this regard, the following hypothesis are considered:

1. The reduction of triangles translates into an increased performance as less information is pushed and processed.
2. Larger batches (more calls) require more time to commit them, thus translating into lower performance.
3. Transformations introduce additional operations per batch, but also during the rendering, so is expected they will lower the performance.
4. Object reuse provides performance improvements over not doing it.
5. Using geometric primitives may provide better performance over not using them, as they may induce specific hints and optimizations into the 3D engines.
6. The reuse of materials introduces savings in the batch sizes, being the worst case where every node redefines again its appearance options, and the best case where all the objects share the same material, thus setting them only once.

Based on this list, a direct comparison between hypothesis is not possible as each one is independent. However, once results are obtained for the different tests, hypothesis can be compared based on the common denominator which is the performance impact.

5.3.2 Applicability to the studied objects.

The choice of using a single type of object to execute the tests may seem to obscure the applicability of the results to real scenes made with various types of objects with different triangle count. However, the definition of the sphere geometry contains enough triangles to hold a cylinder using similar visual quality.

Table 5.5 presents the equivalence between a single Sphere defined at certain quality \( q = 8, 16, 24 \) and the number of Cylinders and Spheres at other qualities. This translation is not attached to any specific reconstruction method but helps to give a rough idea on how this may translate to real data. For example, one sphere drawn with 24 bands per axis can hold up to 72 cylinders made of eight faces per segment. In particular for the 'Stitch' method, one default Sphere can represent a pipe made of 72 stitched segments.

<table>
<thead>
<tr>
<th>Sphere quality:</th>
<th>( q = 8 )</th>
<th>( q = 16 )</th>
<th>( q = 24 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangles:</td>
<td>128</td>
<td>512</td>
<td>1152</td>
</tr>
<tr>
<td>Spheres ( q = 8 )</td>
<td>1</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>Cylinders ( q = 8 )</td>
<td>8</td>
<td>32</td>
<td>72</td>
</tr>
<tr>
<td>Spheres ( q = 16 )</td>
<td>0.25</td>
<td>1</td>
<td>2.25</td>
</tr>
<tr>
<td>Cylinders ( q = 16 )</td>
<td>4</td>
<td>16</td>
<td>36</td>
</tr>
<tr>
<td>Spheres ( q = 24 )</td>
<td>0.111</td>
<td>0.444</td>
<td>1</td>
</tr>
<tr>
<td>Cylinder ( q = 24 )</td>
<td>2.667</td>
<td>10.666</td>
<td>24</td>
</tr>
</tbody>
</table>

Table 5.5: Relationship between a single sphere at a given quality and cylinders and spheres at different qualities.
5.4 Performance Acceptance Criteria

The testing framework is expected to measure the performance impact of the different variables involved on the reconstruction of underground utilities. The performance is understood here as the rendering responsiveness, being the frame rate the main metric.

To guarantee that the metrics are correctly collected, the tests are executed under fixed conditions, but they are also the subject to systematic errors which should be considered before discarding individual results.

For the conducted experiments, the following conditions are assumed:

- The characteristics of each render frame are kept static, including the node count, triangle count and the appearance of the objects.

- The time spent between displaying two consecutive frames is divided in two general parts: 1) the first considering the time spent on traversing the node graph, updating the involved variables and additional logic, and 2) the second is spent on pushing data into the graphics card and rendering the geometry contained in the scene graph.

- Given that all variables are kept static during the tests and a continuous amount of data is pushed into the GPU, it is expected that the time spent to render each frame is constant and as consequence, the number of rendered frames per second is constant too.

In addition, the following situations are expected to introduce variations in the results.

- Additional processes are running simultaneously on the host computer with different priorities and duration. These processes introduce small interruptions during the tests, increasing or decreasing the rendering time for each frame. However, such variations are systematic, and no additional processes are created or destroyed during the execution of the tests. To overcome this, the mean of the obtained results is computed to smooth the variations, while the standard deviation will be used to discard noisy results. As discussed before, the workload of each test is constant so similar results are expected, centered around a mean along successive repetitions.

- A quantization error is present along the tests caused by the precision of the `date` function in Javascript, used to measure the time elapsed between frames. This function measures the time with a resolution of 0.001s, and is expected to introduce errors in the same proportion, traduced into variations on the measured FPS and increasing with high FPS.

With these conditions, it is possible to know in advance the behavior of the experiments, but also under what conditions the obtained results can be considered as reliable experiments given the variations induced by the system and not by the experiment itself.

With respect to the number of rendered nodes, number of vertices used and number of triangles, no specific reference numbers are taken as the testing framework is not created to fine tune an existing scene, but to find out what are the factors affecting the performance and give advise accordingly.

With respect to the frame rate, two quantities are considered as reference [Apteker et al., 1995]:

1. 30 fps, corresponding to a quality level of 100 percent and producing smooth results

2. 10 fps, corresponding to a ‘less than just acceptable’ quality for video sources and perceived as ‘slug-gish’.

Table 5.6 shows a slightly extended rating of the perceived quality at different frame rates, where the used quantities are always on the ‘acceptable’ side.
Frame rate   Rating
30           Default quality for acceptable
15           Just acceptable
10           Much less than acceptable
5            Unacceptable

Table 5.6: Rating of different frame rates in terms of its ‘watchability’ or visual quality for video sources [Apteker et al., 1995].

5.5 Test planning

The test planning is an important step where the typical workload or key scenario under test is defined, but also where the involved variables are established and modeled, so they can be simulated and controlled. In addition, the user actions affecting the tested scenarios are modeled by defining the different paths involved and decisions taken by the user. Finally, the collected metrics obtained for each test are defined, creating a basis for future comparison and analysis.

This modeling step is important to guarantee that correct scenes are being created, the variables are being properly modeled, the tests can be executed, results collected systematically, but also to ensure that the obtained results are repeatable, all important steps to reach the performance-testing objectives.

5.5.1 Key scenario

The reconstruction of underground utilities is performed to assemble scenes composed of a big amount of individual pickable objects where each object can be queried for administrative information, display the results of different analyses through cartographic processes and be interactively explored by the user.

With this scenario, it is desirable to create scenes made of a big number of individual objects within the limits of the tested 3D frameworks. In terms of file size, during preliminary tests a 200 MB limit was reached with X3DOM for loading a scene using XML encoding, while SceneJS managed to load them. In both cases, files bigger than the mentioned size will be discarded.

The reconstructed scenes must also have a realistic appearance but different compromises exist between quality and representation size, so the considered quality levels are in relationship with the quality of the predefined objects and the minimum visual quality accepted by the user.

The considered quality levels range between the standard quality sphere of X3DOM and two lower quality values, i.e. $q = [8, 16, 24]$. Examples of the rendered objects are presented in Figure 4.1 in previous chapter while reconstructing pipes made of cylinders and spheres. This figures show how the lower quality versions are still perceived as ‘acceptable’.

With respect to the total number of rendered nodes, an arbitrary maximum number is settled down at 2050 nodes per frame based on informal tests where X3DOM produced less than six frames per second, a frame rate considered below the threshold of responsiveness, which is 10.

5.5.2 Modeling data and user variability

To ensure that no specific pattern in the data or the user interaction is affecting the result and guarantee a static workload across every test and rendered frame, the following provisions are taken to achieve the desired results:

- The tested scenes are made of multiple nodes but based on an identical geometry. This decision removes the patterns introduced when using pipes composed of different number of segments. As explained on a previous section, the Sphere geometry count has enough space to hold the geometry count of the Cylinder. In more general terms, the only difference between objects is the actual geometry count as all the objects are built using triangles and indexed representations. Indeed, both Sphere and the Cylinder primitives follow a similar vertex ordering pattern, so at least two vertices are reused for every triangle.
The objects composing the scene are randomly located within a bounding cube to guarantee that most of the objects are drawn. This situation is created to guarantee a similar workload during the test execution by reducing the effectiveness of the ‘z-buffer test’ to discard occluded objects.

On a similar note, the scene camera is located and oriented to make visible the whole scene within the viewing frustum. This action is performed to guarantee that no object is discarded due to frustum culling, in case the functionality were present on the 3D engine.

No changes are made to the objects or materials between frames, so during the execution of an individual test the scene graph is kept static, so the same number of object and materials are processed frame after frame.

An animation is created to rotate the camera around the viewing vector for every frame, while ensuring that no object is taken out of the viewing frustum. This animation simulates the user navigation of the scene, introducing changes in the model-view-projection matrix for every frame.

Client server architectures are susceptible to multiple situations which can interrupt the information flow and related processes. In particular, when retrieving the scenes from the database through a web server, multiple errors can occur like server crashes, network failures, timeouts or even security problems due to cross domain requests. To avoid this problems, no connections are made to the database to extract any geometry. Instead, the tested scenes are generated beforehand and made available through the web server as static resources, minimizing the potential problems and reducing the variability of the testing environment.

5.5.3 Collected metrics

For each executed test, a set of basic metrics is collected and stored for further analysis. Derived metrics like the number of batches per second are not collected as they can be computed during the analysis phase. The basic collected metrics are described as follows:

- Frames per Second: This is the main performance metric representing how much time is spent drawing each individual frame and indirectly, how much work is performed for every frame. There are two ways to obtain this metric: 1) measuring the time elapsed between consecutive frames and 2) measuring the number of frames drawn over a period of time. The first approach introduces quantization errors, is highly sensitive to the variations during the tests, but also produce null readings for high frame rates due to the precision of the time function. Due to the problems induced by the first approach, the metrics are collected using the second approach over periods above a second for the whole duration of the test.

The next formula is used to collect this metric:

\[
    f_{ps}[i] = \frac{\text{drawn frames}}{\text{time}_{end} - \text{time}_{start}} (1000), \text{time}_{end} - \text{time}_{start} \geq 1s
\]

where \(i\) is the \(i^{th}\) sample collected since the beginning of the test. Metrics are collected for 30 seconds and at the end of the test, the mean \(f_{ps}\) value is computed using the last 90% obtained samples. This procedure removes the initial noise introduced by the cold startup, where initial values can be erroneous due to the memory and processor allocation. The obtained value is assigned as final result for the test, producing the desired final metric, i.e.

\[
    f_{ps} = \frac{1}{n-k} \sum_{i \in [k, n]} f_{ps}[i], \forall i \in [k, n], k = 0.1(n), k \in \mathbb{N}
\]

where \(n\) is the number of samples collected during the execution of test.

This approach has the benefit of smoothing the results, but also allows to measure the variability during the test, which requires a set of values to be computed. This variability is the next metric and corresponds to the standard deviation.
• Standard deviation: describes the spread of the FPS measurements with respect to the mean. The tests are designed to produce a constant workload every frame, so no variations should be present in the results. However, the results are subject to variations deviating the obtained values around the mean. This information is required to classify the results as ‘noisy’ of ‘clean’, depending on the magnitude of this value. ‘Clean’ results can be used to characterizing performance of the tested factors while ‘Noisy’ results can be discarded from the final computations.

In addition, the following information is also associated with each collected metric, useful for further analysis, but also required to characterize the executed test:

• Test type: The combination of performance factors used during the test.
• Number of Triangles: The number of produced triangles on the scene.
• Number of Vertices: The amount of produced vertices on the scene.
• Number of Nodes: Number of rendered scene graph nodes.
• Visual quality parameter: parameter used to determine the number of approximations used per object and estimate its triangle count.
Performance Testing Implementation

The implementation and execution of the testing framework introduced on Chapter 5 requires the description of the testing environment, system architecture and testing preconditions before carrying out the tests. Once the tests are finished, results can be analyzed, compared and used to draw conclusions.

This chapter starts on Section 6.1 with the description of the relevant hardware and software used during the tests, including the CPU, its number of cores, system memory, GPU and its underlying connection bus. Within the same description, the system architecture used to carry out the tests is also presented.

Later, Section 6.2 present the preconditions for the execution of the tests, ensuring that the required files and systems are loaded and running, so no interventions are required in the middle of the tests which can disturb the execution and the obtained results.

Section 6.3 describes the execution of the tests, provide details on how are considered the tested factors, but also on the collection of results. During this process, the obtained results are validated against the acceptance criteria, ensuring that results are within the expectations of the designed test.

Section 6.4 presents the obtained results from the tests in a series of graphs, describes them and analyze the results for the tested 3D frameworks. The results are presented on two main categories: responsiveness (FPS) and throughput (TPS and BPS).

Section 6.5 verifies the obtained results using a standalone X3D player. This is done to ensure that the tests are correctly analyzing the impact of the tested factors given the set of hypothesis.

Section 6.6 presents the conclusions for the baseline test based on the results from the 3D frameworks and the standalone player, comparing between hypothesis and ranking the factors influence accordingly.

Section 6.7 answers the performance-testing questions introduced during the development of the testing framework using the conclusions from the baseline tests. These questions are a fundamental part of the testing process, but also relevant to the research question of this thesis.

Finally, Section 6.8 bridges the gap between the abstract tests and real data, showing how the factors influence persist on the reconstructed scenes.

6.1 Testing environment

The testing environment is described in two parts: the logical and the physical environment. The logical includes all the relevant software used and developed required to carry out the tests while the physical side involves the hardware affecting the tests, including details about the components, speed, number of cores, amount of memory and in particular, the GPU used to display the 3D graphics. With this elements, the system architecture used for the testing process is described and used later as reference.
Logical Environment

Several elements are required to execute the testing framework described on the previous chapter. Due to the nature of the Web environment, the tests and instrumentation are created in a client-server architecture.

The used elements are split into two parts: client and server side. Figure 6.2 shows the considered elements described on the following list:

- a) Web Browser
- b) Operating System
- c) Web server and servlet container
- d) Javascript libraries and custom code.
- e) Java servlet collecting test results
- f) Postgres Database to store results.

The web browser used for the tests should have support for HTML5 and WebGL technologies. Table 6.1 shows the gamut of available options offered by different vendors on various operating systems. Using an informal test, the fastest browser was selected to carry out the tests. Google Chrome was chosen as the fastest browser due to its fast JavaScript engine V8, its support to HTML5 and WebGL technologies and its availability on different platforms. The browser version used for the tests is 20.0.1132.47, a beta release.

The operating system used to host the client and server software is Mac OSX version 10.7.4 for Intel computers with the latest updates and patches.

The server used to host the HTML pages, 3D scenes, Javascript libraries and servlets is Jetty, a lightweight HTTP server and servlet container written in Java.

---

6.1. TESTING ENVIRONMENT

<table>
<thead>
<tr>
<th>Browser</th>
<th>Operating system</th>
<th>WebGL support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet Explorer</td>
<td>Windows</td>
<td>No/Through Google Plugin</td>
</tr>
<tr>
<td><strong>Google Chrome</strong></td>
<td>Windows</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Linux</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Mac OSX</td>
<td>Yes</td>
</tr>
<tr>
<td>Mozilla Firefox</td>
<td>Windows</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Linux</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Mac OSX</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Android</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Opera 12</strong></td>
<td>Mac OSX</td>
<td>Experimental</td>
</tr>
<tr>
<td></td>
<td>Windows</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Linux</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Safari</strong></td>
<td>Mac OSX</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Windows</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 6.1: List of popular Web browsers and their WebGL support across different operating systems.

The libraries used for the instrumentation include jQuery \(^4\) and jsanalysis \(^5\), a plugin for jQuery to add mathematical analysis functions. Those libraries are used to keep the browser information, send the result of the tests and compute the mean and standard deviation.

Most of the code created for the tests was developed to automate the loading of the scenes, collect metrics and submit the results after a defined period of time.

The metrics are collected using custom Javascript code and are obtained as described on Section 5.5.3 in periods above a second, storing each sample in order to know the variability during the execution of the test. At the end of each individual test, the average FPS and standard deviation are computed using the mentioned libraries. Finally, results and additional attributes characterizing the test are sent to the server to be stored on the database.

Table 6.2 shows details about the information sent to the server with a corresponding example.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>machine_id</td>
<td>Unique identifier assigned to the browser</td>
<td>4326</td>
</tr>
<tr>
<td>machine_specs</td>
<td>Description of the computer</td>
<td>Windows 7 Core2Duo 2.4 Ghz 8GB RAM, Chrome</td>
</tr>
<tr>
<td>num_bands</td>
<td>Sphere quality parameter: 8,16 or 24</td>
<td>16</td>
</tr>
<tr>
<td>num_objects</td>
<td>Rendered scene graph nodes</td>
<td>2050</td>
</tr>
<tr>
<td>num_triangles</td>
<td>Triangles rendered per frame</td>
<td>1049600</td>
</tr>
<tr>
<td>num_vertices</td>
<td>Number of points used in the scene</td>
<td>596550</td>
</tr>
<tr>
<td>fps</td>
<td>Collected metric</td>
<td>15.625</td>
</tr>
<tr>
<td>stddev</td>
<td>Standard deviation of the executed test</td>
<td>1.233</td>
</tr>
<tr>
<td>date</td>
<td>Time and date of the submitted result</td>
<td>‘7-7-2012-17-6-14’</td>
</tr>
</tbody>
</table>

Table 6.2: Information sent to the web server after each test, including the collected metrics and details characterizing every execution. A corresponding table with the mentioned attributes is available on the database to store the results.

On the server side, a servlet was developed with the Java programming language to receive and extract the results sent via an HTTP GET request, where the information is encoded in the URL query string. The received values are stored later into a relational database using the before mentioned attributes.

The database storing the results is implemented with PostgreSQL version 9.1 for OSX, simplifying the

---

\(^4\) [http://jquery.com/]
\(^5\) [http://code.google.com/p/jsanalysis/]
storage solution and analysis of the tests. The received information is stored into a table with the same attributes presented on Table 6.2, plus an additional identifier which is created automatically for each record.

Additional details about the web server code and the database collecting and storing the data are left out of this report, as they are just a supporting technology for the analysis and do not affect the obtained results.

Physical Environment

The hardware used to perform the tests consists of a portable computer acting as as a client and server, namely an Apple MacBook Pro 17”. The computer CPU is an Intel i7 quad core processor running at 2.4 GHz, with ‘turbo mode’ disabled but ‘hyper threading’ enabled. The system has installed 16GB of DDR3 RAM running at 1600MHz with a bandwidth of 25 GB/s. The computer contains an ATI 6770M GPU equipped with 1GB of RAM and connected with a PCI-E 2.0 8x bus, offering a bandwidth of 4GB/s. The network details of the equipment are not considered here, as the metrics are not collected until the complete scene is loaded into memory and ready to be rendered.

6.2 Test implementation

To start the tests, the following preconditions should be fulfilled to execute them without interruptions and collect results automatically:

1. The database server used to collect the metrics is loaded, ensuring that is accepting connections, the corresponding table exists and the required database user is defined and granted permissions to access the mentioned resources.
2. The Web Server receiving the results is loaded, ensuring that results are being received and processed.
3. For each of the considered combination of factors, including the quality parameter $q \in \{8, 16, 24\}$ and a number $n \in \{50, 300, 550, 800, 1050, 1300, 1550, 1850, 2050\}$ of nodes, the geometry of $n$ spheres is created placed and the nodes randomly distributed within a bounding cube. For each node an identifier is assigned, the picking functionality is enabled and the appearance parameters applied. In sum, this information is used to produce the set of files required to feed the tests.
4. After creating the files, they are placed into the a folder accessible to the web server. Each file has a different name which correspond to the tested factors, sphere quality and number of nodes.
5. The HTML files, the required Javascript libraries and the custom code are also placed on the web server so they can be loaded.
6. Once all the required files for the tests are created and accessible to the web server, the Web Server started.

Once this prerequisites are fulfilled, the actual tests can start and metrics be collected.

6.3 Test execution

The execution of the tests proceeds in the following way:

1. For a specific 3D framework, an HTML page is loaded into the WebBrowser based on the corresponding testing factors and parameters. The default parameters are $q = 8$ and $n = 50$ while the default combination of factors corresponds to ‘P_T_O_M’. The required libraries are loaded and the 3D framework executed.
2. The test scene corresponding to the parameters is requested to the server, i.e. $P_{-}T_{-}O_{-}M_{-}8_{-}50.x3d$
6.3. TEST EXECUTION

3. The server returns the requested scene to the client, which parses the file and makes it ready for its use.

4. With the scene loaded, the test is executed for the designated period of time, collecting the corresponding metrics. At the end of the test, the average FPS and the standard deviation are computed.

5. With the test results and the characteristics of the test, a HTTP Get request is used to send the results to the server, encoded into a query string, i.e.

```
http://localhost:8080/DBConnector?
machineId=12334&
machine_Specs=Windows7_Core2Duo...&
num_bands=16&
num_objects=2050&
num_triangles=1049600&
num_vertices=596550&
fps=15.625&
stddev=1.23&
date='7-7-2012-17-6-14'
```

6. The servlet receives the HTTP Get request with the encoded information and extracts the data from the query string. With this information, creates an SQL statement to insert the results into the corresponding table in the database.

7. The servlet submits the SQL statement to the database which stores the results.

8. The HTML page prepares the parameters for the next test, increasing the number of objects, increasing the visual quality parameter or change the tested factors. After updating the parameters, the page is reloaded, getting back to the first step. With this procedure, all the corresponding variables are tested, changing one at a time i.e. changing the number of nodes, the visual quality parameter, and the combination of factors of the scene graph. Figure 6.3 shows the order followed by the automated tests, scrolling through an increasing number of nodes, the visual quality parameter and the combinations of factors into a single framework.

To ensure that the produced results contain valid numbers, each individual test is executed at least ten times or until relatively low standard deviations are obtained. Let's recall that the standard deviation is measuring the variability of the system and not of the test itself.

Figure 6.4 presents a screenshot of the X3DOM tests, giving an idea of how the test progress with each successive step.
Figure 6.3: Diagram of the execution order of the tests, moving along all the variables for every test: number of nodes, visual quality parameter and the combination of construction factors.

Figure 6.4: Screenshots of the actual tests performed with X3DOM. On the left 50 spheres are being rendered while on the right, 2050 spheres are included.
6.4 Test results

After collecting enough data from each test, information is extracted and grouped accordingly to produce three type of graphs: FPS, TPS and BPS.

The FPS graphs shows the measured frames per second with respect to the number of rendered nodes per frame and the corresponding triangle count per frame. The TPS graph is derived from the collected metrics and shows the number of triangles per second processed with respect to the number of nodes per frame at certain visual quality. The BPS graph is also derived from the collected metrics and shows the corresponding number of batches per second delivered with respect to the number of nodes per frame.

6.4.1 Responsiveness (FPS)

The first set of results correspond to the number of frames per second obtained while rendering a variable amount of nodes using with a given vertex count per node. Figure 6.5 shows the results obtained results for X3DOM into three graphs, each one with different triangle count per sphere. Figure 6.6 shows the corresponding results for SceneJS. On each graph, lines are drawn at 30 FPS and 10 FPS, where the former corresponds to the default frame rate on a fluid system, and the latter to the limit where the system is perceived as not fluid. Figure 6.7 combines the best results of both 3D frameworks to understand usability of the frameworks based on their responsiveness.

In addition, Figure 6.8 presents the obtained FPS with respect to the triangle count per frame, helping to understand how a given amount of triangles distributed on fewer nodes can be used to improve the performance.

Given the graphs and following its analysis, the following conclusions are jointly drawn and presented as follows:

- In X3DOM, the performance difference between the tested combinations is within one standard deviation ($\sigma = 5$), ranging between 1 FPS and hardly above 5 FPS. With this information, is considered that no benefits are appreciable when using the built in Primitives or replacing them with custom Primitives, reusing nodes and materials, but also by saving transformations.

- In the case of SceneJS, the difference between the higher and lowest score for a given number of nodes is around 12 FPS across all the tests, and bigger than the difference obtained on X3DOM but this time bigger than one averaged standard deviation ($\sigma = 7.5$). The best performing combinations include ‘P_T_O_M’, ‘P_T_O_nM’ which use transformations and ‘nP_T_nO_M’ which do not use them. From the results, is clear that avoiding transformations give an advantage over most of the presented cases, locating this combination within the top results. The lowest performance is obtained by ‘P_T_nO_nM’, and ‘P_T_nO_M’, which don't reuse objects.

- The lack of differences in the X3DOM means a deficiency in the 3D engine to apply the optimization hints given through the scene graph definition. In SceneJS, the scene graph is recompiled and transformed into a display list to improve the rendering speed, recycling previous states and reducing the number of call per rendering batch. These optimizations are visible on the SceneJS graphs, giving not only higher FPS overall, but also showing differences between the tested combination of factors.

- The usable lower limit defined at 10 FPS is reached with X3DOM at around 1050 nodes in the scene, while SceneJS cannot reach that line within the limits of the test. For the 30 FPS line, X3DOM reaches it after rendering 300 nodes while SceneJS reaches it at the end of the test with 2050 nodes. Figure 6.7 shows this situation using the best two combinations of factors obtained per 3D framework.

- In both frameworks, the change or reduction in triangle count for a given number of rendered nodes shows no significative differences on every test. However, a fixed amount of triangles distributed on a smaller number of nodes provide higher FPS. Figure 6.8 shows this situation with two examples. The first example shows the how 166,400 triangles distributed over 1300 nodes is delivered at 35 FPS while the same amount of triangles distributed in 325 nodes is delivered at 35 FPS. The second
example shows the how one million triangles distributed in 2050 nodes produces around 5 FPS while the same amount of triangles distributed in 911 nodes delivers around 15 FPS.

- In general, the performance of SceneJS is around five times better than X3DOM for similarly constructed scenes.

- Given that the browser and the operating system are automatically adjusted to display only 60 FPS, values obtained above that line are not considered reliable. Throttling down the system to display only 60 FPS is not immediate, so errors can be introduced in the obtained metrics. Based on this information, valid numbers are only obtained below that line.

Based solely on the FPS obtained, SceneJS seems like a better 3D framework compared to X3DOM. This difference is visible on most of the tested factors except when reducing the triangle count. The analysis of the triangle count is presented next.

6.4.2 Triangle throughput (TPS)

The number of processed triangles per second is an indicator of the amount of work done by the GPU. The GPU receives vertices as input and creates the corresponding triangles, which are the basic elements for the following rendering stages. The information obtained with this results is important to understand the role in the triangle count per node into the visualization performance. Figure 6.9 shows the number of triangles processed per second using three different triangle counts. On each graph, only one combination of factors is used as reference, reducing the clutter visible on previous graphs due to the similar results obtained. The selected combination is the one providing the higher scores and corresponds to the ‘P_T_O_M’ combination.

Based on the produced graphs, the following conclusions are obtained:

- The number of triangles processed by X3DOM and SceneJS at a given triangle count per node stabilizes after certain point. However, the stabilization is not dependent on the triangle count per node. Figure 6.9 shows how scenes with a higher triangle count per node produce a higher triangle throughput.

- The stabilization on the graphs shows that a bottleneck is present on the system. However, given that an increasing vertex count per node increases the total number of processed triangles per second, the GPU can be discarded as the bottleneck.

- When comparing the triangle throughput obtained with X3DOM with the corresponding SceneJS throughput, the difference makes clear that the GPU is not the bottleneck as even more triangles can be processed per second. Figure 6.9(a) shows the obtained TPS for X3DOM and while Figure 6.9(b) the corresponding TPS for SceneJS. In those graphs, X3DOM delivers up to 12 million triangles per second using 1152 triangles per node while SceneJS reaches around 70 million triangles per second with the same triangle count per node.

6.4.3 Batch throughput (BPS)

The number of batches per second is a metric derived from the obtained FPS which indirectly denotes the amount of work executed by the CPU to render a scene. Each render batch corresponds to a block of low level calls executed by the CPU, setting the appropriate states prior drawing the geometry, executing the draw call and releasing used variables. Each low level call within a batch has an associated cost and as consequence, a batch made of a large number of calls require more processing time. In the implemented tests, all the rendered nodes are created identical so each batch is identical and has the same cost.

Based on the FPS results, the graphs showing the number of batches per second are derived and presented. Figure 6.10 shows the result for X3DOM while Figure 6.11 shows the corresponding results for SceneJS. Based on the results, the following concussions are drawn:
Figure 6.5: Frames per second obtained in X3DOM rendering spheres using different triangle count, expressed with 8, 16 and bands per axis. Standard deviation bars omitted to improve the visibility. $1 < \sigma < 5$
Figure 6.6: Frames per second obtained rendering spheres on SceneJS using 8, 16 and 24 bands per axis.
Figure 6.7: Comparative between X3DOM and SceneJS using 24 bands per axis and only two combination of tested factors.

Figure 6.8: FPS obtained with a fixed amount of triangles distributed over a different number of nodes.
Figure 6.9: Triangles per second obtained with different triangle count per node. a) With X3DOM up to 12 million triangles are processed while b) SceneJS allows to process up to 70 million. The stable results indicate a bottleneck. However, the difference in throughput obtained with with different triangle count per node indicates that the GPU is not the bottleneck.
6.4. TEST RESULTS

- The difference in triangle count per node does not affect the number of batches delivered per second in X3DOM or SceneJS, where similar results are obtained across tests. Based on this, it is clear that the triangle count is not adding significant cost to the batches.

- With an increase in the node count, the number of batches processed per second increases up to a certain point and stabilizes. This situation shows that a bottleneck is found and based on the previous point and the TPS results, it is clear that the GPU is not the bottleneck. In particular, the graphs stabilize below the 60 FPS line, which is the refresh rate limit of the screen set by most computers and the browser. This line is showed on the graphs with a black vertical line.

- Given that no textures or complex lighting are being used, it can be concluded that the Fragment Shader is neither the bottleneck of this test.

- The total throughput in terms of batches per second is constant with a tendency to decrease with an increasing number of nodes per frame. X3DOM shows this behavior when comparing the BPS at 550 nodes against the BPS obtained with 2050 nodes. In the SceneJS case, bigger scenes would be required before showing this behavior, however at the end of the tests, the number of delivered batches seem to stabilize, so it is assumed the performance will start decreasing after this point.

- Avoiding transformations leads to a slightly higher throughput against most of the tested combination of factors, visible on the graphs with the red line. The gain is explained by the omission of the 'model' matrix per batch, but also due to the reduced vertex processing per frame.

When comparing results from both graphs, some conclusions can be drawn and used to bring explanations of the obtained numbers. The limit found on the maximum number of batches delivered per second indicates that a bottleneck is present, however as the triangle count is not playing a determinant role in the performance and no complex appearance options are being applied while rendering, it is clear that the GPU is not the bottleneck on this tests.

With the GPU discarded as the bottleneck, three options remain as the root of the problem: the system bus bandwidth, the memory bandwidth and the CPU. To discard the first two, the amount of information transferred per second is compared with the system bus and memory bandwidth. Table 6.3 shows the required bandwidth for three scenes using SceneJS with the corresponding bandwidth of the system. Based on the number differences, it is clear that the bottleneck is not caused by the amount of transferred information and then, those two elements are not playing an important role here.

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes per frame</td>
<td>2050</td>
</tr>
<tr>
<td>Vertices per node</td>
<td>64</td>
</tr>
<tr>
<td>Normals per node</td>
<td>64</td>
</tr>
<tr>
<td>Triangles per node</td>
<td>128</td>
</tr>
<tr>
<td>Indices per node</td>
<td>384</td>
</tr>
<tr>
<td>Frame Memory Footprint</td>
<td>2.6MB</td>
</tr>
<tr>
<td>FPS</td>
<td>30</td>
</tr>
<tr>
<td>Transferred Information</td>
<td>78 MB/s</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>25 GB/s</td>
</tr>
<tr>
<td>System Bus Bandwidth</td>
<td>4GB/s</td>
</tr>
</tbody>
</table>

Table 6.3: Difference between used bandwidth with SceneJS and available bandwidth.

After discarding those two elements, is clear that the CPU is the bottleneck. This result agrees with Wolka [2003] and Fernando [2004] who showed the relationship of the Graphics Pipeline performance with the number of batches, and how this number is limited by the CPU speed. Given that the number of batches
Figure 6.10: Batch performance for X3DOM with nodes with different triangle count. The vertical line denotes the 60 FPS limit, which is higher to the left and lower to the right.
Figure 6.11: Batch performance for SceneJS with nodes with different triangle count. The vertical line denotes the 60 FPS limit, which is higher to the left and lower to the right.
delivered per second is CPU limited and the triangle count seems to have no significant effect on the performance, an option left to increase the FPS while drawing the same amount of information in the scene is to pack as much triangles as possible within a single batch. Figure 6.8 on Section 6.4.1 shows the improvements obtained by packing a fixed amount of triangles per frame into less nodes, doubling at least the FPS in most cases.

6.5 Verification

Based on the results obtained in the previous section, it is clear that the use of different factors known to improve the performance, like the reduction in triangle count per node introduced little or no difference on the tests.

To verify if the proposed hypothesis hold for the factors showing no improvement, additional tests are carried out on the same physical conditions but using a different logical environment. The tests replace X3DOM with Instant Player, which is a standalone X3D player capable of using the same X3D files created for the previous tests.

The X3D specification does not allow modifications to the quality of the Sphere primitive, so the tests are carried out using the default triangle count, which in X3DOM is 1152 triangles per sphere while for InstantPlayer is 1280 triangles. In the case of custom primitives, changes in the vertex count are possible and used for the tests, presenting the corresponding results.

Due to the lack of instrumentation and access to the source code of InstantPlayer, metrics cannot be systematically collected. The FPS readings presented in the screen are used to obtain the desired metrics, which on the given platform are computed for every frame. The situation makes hard to read them and introduces errors due to quantization and interactions with the processes simultaneously running on the host system, a situation also present with the previous tests.

To reduce the errors in the collected readings, the values are rounded discarding the unreadable numbers, starting from the least significant digit and advancing to the most significant digit until no changes are perceived. If a reading fluctuates between two values, the average is taken for that digit until no changes are perceived.

To proceed with the tests, the same files created before are manually loaded in the player and the metrics collected accordingly. Figure 6.12 shows the resulting graphs for FPS, Figure 6.13 the corresponding graphs with TPS and Figure 6.14 the graphs with BPS.

From this figures, the following conclusions are drawn:

- Using primitives indeed provide the 3D frameworks with enough hints to apply optimizations while rendering the scene. Figure 6.12(c) shows how the worst results using primitives almost double the frame rate obtained with the best case using custom primitives with a similar triangle count.

- Reusing materials and nodes provides a clear advantage over not reusing any previous definition, visible across all the graphs, i.e. ‘P_T_O_M’ versus ‘P_T_nO_nM’.

- Reusing materials always obtain better scores than combinations not reusing them, i.e. ‘nP_T_O_M’ versus ‘nP_T_O_nM’.

- Reusing only materials provides higher results over combinations reusing only nodes, i.e. ‘P_T_nO_M’ versus ‘P_T_O_nM’.

- The triangle count is an important factor affecting the performance, and reducing the number of triangles per node increases the delivered FPS, TPS and BPS. The difference is appreciable by comparing Figure 6.12(a) with Figure 6.12(c) where the custom primitives made of fewer triangles almost double the FPS compared with the custom spheres at the default quality.

- The savings introduced by avoiding transformations offer bigger improvements with lower triangle count, and even surpasses other approaches at low triangle counts. Figure 6.13(a) shows how
Figure 6.12: Frames per second obtained with Instant Player using 1216 triangles per sphere.
Figure 6.13: Triangles per second obtained with Instant Player using 1216 triangles per sphere.
Figure 6.14: Batches per second obtained with Instant Player using 1216 triangles per sphere.
by avoiding transformations and using a lower triangle count, the obtained 400 million TPS almost reaches the GPU fill rate, which is 500 million TPS.

- The stable behavior obtained in the BPS and TPS graphs with an increasing number of nodes show again that the CPU is being the main bottleneck in the visualization. The GPU is again discarded as the main source of problems based on the performance increase with reduced triangle counts.

- The difference in FPS between the 3D frameworks based on WebGL is huge compared with InstantPlayer running over OpenGL and implemented in C++, being at least 10 times faster with 2050 nodes.

- InstantPlayer is not limited at rendering only 60 FPS, visible on the BPS and TPS graphs where the graphs are constant across all the number of nodes. However, that limitation in the operating system was disabled to show the results in the obtained ranges.

Based on these additional results, the complete set of hypothesis introduced earlier proved to be true, showing with the obtained results how the testing framework is correctly built and how it is correctly measuring the influence of the considered factors.

6.6 Baseline conclusions

From the results obtained on two different 3D engines and complemented by the validation tests, the introduced hypothesis are confirmed, information about the interaction between hypothesis obtained, and conclusions about the tested combinations drawn and presented as follows:

- Using primitives brings higher performance over custom objects as indeed they trigger specific optimizations. However, this is only visible when using InstantPlayer and not in the tested WebGL 3D frameworks.

- Reducing the triangle count indeed increases the performance, but given the limited processing speed of Javascript and WebGL, the benefits are not visible.

- The extensive node reuse on the constructed scenes provides performance improvements over not reusing them, however, given the limitations of JavaScript and WebGL, the impact of this optimization is limited.

- The increase in the number of rendered nodes provides a direct hit in the performance, as double the objects decrease the performance in half or the other way around, rendering half of the nodes doubles the performance.

- An increase in the number of objects hits faster the performance than an increase in the number of triangles.

- Scenes will not scale good when increasing the number of objects, compared to an increase in the triangle count.

- Reusing materials seems more important than reusing objects to reduce the cost of every batch.

- Using primitives and reusing materials seems like the best option overall, however avoiding transformations and reducing triangle count can beat the performance of that combination.

In conclusion, the worst combination of factors occurs when objects and materials are not reused and transformations need to be applied. This combination produces the mentioned baseline, and based on the presented conclusions, any other combination of factors will produce an increase in performance.

6http://www.amd.com/us/products/notebook/graphics/amd-radeon-6000m/amd-radeon-6700m-6600m/Pages/amd-radeon-6700m-6600m.aspx
As discussed earlier, given the bottlenecks found and the relative small improvements introduced by the other construction options, the reduction in the number of nodes gives the best improvements overall. After this, the reduction in triangle count can introduce additional benefits in performance depending on the target platform, but in particular on the size of the files.

Table 6.5 shows the relative impact on performance of the studied factors for the different tested targets based on the obtained conclusions.

<table>
<thead>
<tr>
<th>Factor</th>
<th>Influence Rank</th>
<th>Influence Rank</th>
<th>Influence Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive Use</td>
<td>Negligible</td>
<td>4</td>
<td>Minor</td>
</tr>
<tr>
<td>Transformations</td>
<td>Minor</td>
<td>2</td>
<td>Minor</td>
</tr>
<tr>
<td>Node reuse</td>
<td>Negligible</td>
<td>3</td>
<td>Minor</td>
</tr>
<tr>
<td>Material reuse</td>
<td>Minor</td>
<td>2</td>
<td>Minor</td>
</tr>
<tr>
<td>Triangle count</td>
<td>Negligible</td>
<td>3</td>
<td>Negligible</td>
</tr>
<tr>
<td>Node count</td>
<td>Major</td>
<td>1</td>
<td>Major</td>
</tr>
</tbody>
</table>

Table 6.4: Influence on the tested factors in the performance for the tested Frameworks. A factor of mayor influence means that its application provides visible improvements in the responsiveness of the system, while one of minor influence may not produce visible benefits. A factor of negligible influence means that its application does not justify the additional effort required to implement such feature.

6.7 Performance-testing conclusions

The testing framework introduced in the Chapter was created to analyze the performance impact of various declarative building blocks used during the reconstruction of underground utilities. In order to know if the mentioned framework is addressing its objective, a set of performance-testing questions were introduced. Based on the results obtained in this chapter, each question can be answered accordingly.

To conclude this chapter, the mentioned questions are reintroduced and answered based on the results obtained along this chapter:

- What are the factors with largest negative impact on performance? The increase in the number of rendered nodes consistently proved to decrease the performance, being in direct relationship with the CPU and limited by its speed.

- What is the degree of influence of each factor? For the WebGL platform, the number of nodes proved to be the most important factor for responsiveness, followed with a relative low importance of the triangle count, the use of transformations, reuse of objects and reuse materials. These last factors do not influence drastically the responsiveness of the scenes, however these factors can be considered to reduce the size of the scenes.

- What is the baseline for performance? or on other words, what is the worst combination of factors? The worst combination of factors include a high number of rendered nodes, the use of transformations, not reusing materials and not reusing objects.

- What is the best combination of factors? Two combinations offer the best performance: the first includes the use of primitives, transformations, objects and materials reuse; while the second combination corresponds to the use of custom objects with no transformations and materials reuse.

- What would be the recommended reconstruction method based in the obtained results? Giving that the most important factor is the node count, the reconstruction method with the best combination of factors, but with a low number of produced nodes will be the winner, and under the tested conditions, the ‘stitch’ method is the clear winner as it produces a low number of additional nodes, avoids transformations, has a low triangle count and allows material reuse.
6.8 Real data performance

Following from the results obtained while testing the considered factors, it is clear that some of them have larger influence than others, but also that some can be directly changed with an increase in performance, like the number of nodes and the triangle count.

When reflecting back this results into the reconstruction methods, it is expected that approaches that reduce the number of produced nodes will bring higher performance. Based on this, the methods splitting the pipes and producing additional nodes will perform slower than their non-splitting counterparts. If the target platforms were not based on WebGL, other optimizations should also provide improvements by reducing the number of triangles, vertices and even after reordering vertices. However, even if WebGL gets no significant benefits from the triangle reduction, the size of the files will do.

To confirm the implications of this results, the reconstruction methods are applied to create scenes based on real data involving the studied factors. For some reconstruction methods, the best combination of factors is chosen given the analysis presented before and applied accordingly.

The dataset used to assemble a scene consists of 162 pipes encoded with LineStrings with an average length of 18.7 segments and a standard deviation of 39.09. Figure 6.15(a) shows the corresponding histogram, and gives an idea of the variability of the data. For each pipe, appearance is applied accordingly to an internal list and in total 11 material definitions are used. Figure 6.15(b) shows the corresponding extent of the dataset, spanned over an area of $6 \text{ km}^2$.

Table 6.5 shows the obtained FPS for the created scenes when rendered on the X3DOM. Based on the presented results, big differences exist between the presented approaches, not only in the number of nodes and file size, but also in the responsiveness.

As stated before, scenes will not scale well with an increasing number of nodes. In the presented example, a scene reconstructed using a split method required 5092 nodes, producing around 2 FPS when reusing objects, while the same scene without object reuse dropped to almost zero FPS.

The equivalent version made with the ‘append’ method required around three nodes per object, delivering around 20 FPS, ten times more than the ‘split’ method. The same version of the appended scene but made with a lower triangle count dropped the object count close to the original numbers and raised the responsiveness to 30 FPS.

When the same scene is created using the ‘stitch’ approach, the object count remains equal to the source data, requiring only one node per object and delivering 50 FPS.

The results obtained with real data show how the reduction in nodes produce higher benefits in WebGL than the triangle reduction. Indeed this factor is exploited by 3D engines, ‘culling’ the objects outside the
6.8. REAL DATA PERFORMANCE

<table>
<thead>
<tr>
<th>Reconstruction Method</th>
<th>Parameters</th>
<th>Objects Quality</th>
<th>Scene Nodes</th>
<th>Triangles count</th>
<th>File Size</th>
<th>X3DOM FPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Primitives</td>
<td>P_T_O_M</td>
<td>24</td>
<td>5902</td>
<td>3,676,544</td>
<td>1.5 MB</td>
<td>2.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>5902</td>
<td>571,776</td>
<td>1.5 MB</td>
<td>2.3</td>
</tr>
<tr>
<td>Custom Local Primitives</td>
<td>nP_T_O_M</td>
<td>24</td>
<td>5092</td>
<td>4,064,640</td>
<td>1.5 MB</td>
<td>2.3</td>
</tr>
<tr>
<td>Custom World Primitives</td>
<td>nP_nT_nO_M</td>
<td>8</td>
<td>5092</td>
<td>434,016</td>
<td>1.5 MB</td>
<td>2.7</td>
</tr>
<tr>
<td>Append World Geometries</td>
<td>24</td>
<td>682</td>
<td>3,630,624</td>
<td>148 MB</td>
<td>20*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>189</td>
<td>434,016</td>
<td>18.4 MB</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>Stitched World Geometries</td>
<td>24</td>
<td>162</td>
<td>137,760</td>
<td>4.9 MB</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>162</td>
<td>45,920</td>
<td>1.6 MB</td>
<td>50</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.5: X3DOM Performance using the reconstruction methods with real data on X3DOM.

viewing frustum and reducing the number of rendered nodes and the number of batches per frame. GPU’s apply similar techniques to avoid rendering objects outside the frustum view, however this approach still requires to submit a batch with the geometry, reducing the effectiveness of the approach. Given that no other ‘improvement’ provide substantial performance benefits for WebGL and the performance is CPU bounded, opportunities for improvement are limited to reduce the number of nodes in the screen.

According to this, the reconstruction methods based on ‘non-split’ approaches have a clear advantage over the ‘split’ methods as no additional nodes are created unless necessary. In this sense, the ‘stitching’ and ‘append’ approaches are optimizations over previous methods, relying on the most effective parameter for increasing the performance, while keeping the number of produced nodes low. In addition, the ‘stitching’ method allows to reduce the size of the produced files by discarding the spheres whenever is possible.

Given our baseline for performance, the rest of the mentioned factors like material reuse introduces small benefits not only for performance but also for reducing the memory footprint and the encoding size.

Finally, from the characteristics of the tested equipment which may be considered high end, is clear that scenes containing more than 1000 nodes will be hard to visualize smoothly. Based on this limit, if a ‘split’ approach were used to reconstruct the pipes, and each pipe requires 16 nodes, then the number of the effective reconstructed object would be limited to only 62, i.e.

\[ 62 \text{ objects} \times (8 \text{ cylinders} + 8 \text{ spheres}) = 992 \text{ nodes} \]

This number is three times smaller than amount of pipes contained in the study area and small compared to typical GIS datasets, without even including the additional reference objects used along the scene, like buildings, terrain, vegetation, signs, pits, ditches, etc.
7

CONCLUSIONS, RECOMMENDATIONS AND FUTURE WORK

The 3D visualization of underground utilities help to reduce the problems with the interpretation of 2D drawings, where cluttered objects with complex layouts are hard to decipher.

To achieve the 3D visualization, the abstract information encoded with lines is reconstructed to bring volumetric appearance, using scene graphs as the modeling structure. With a scene graph, 3D frameworks can be used to display the reconstructed objects using WebGL, and distribute the information in the browser to the corresponding stakeholders.

An existing approach to reconstruct and model the underground objects include the use of cylinders and spheres, which despite its simplicity, creates complex scene graphs to model the relatively simple objects.

This reconstruction can be achieved in different ways within the scene graph. However, each decision taken involves a different modeling approach which has a different impact on the rendering performance. In addition, underground utilities datasets typically consist of multiple records spanned over large geographical areas and consisting of multiple segments, increasing the complexity of the created scenes and potentially reducing the rendering performance. Given this situation, is clear that multiple issues should be addressed before achieving a realtime visualization. In order to understand which approaches for reconstructing underground utilities can improve the performance of visualization clients based on WebGL, the research was conducted in two main lines:

• First, studying the reconstruction of pipes with cylindrical appearance starting from line abstractions, trying to categorize the reconstruction procedures and their mapping into scene graphs, study their complexity and implementing the actual methods.

• Second, highlighting the factors affecting the performance, developing a testing framework to assess the their impact on the rendering stage and finally, linking the obtained results with the reconstruction methods.

With the results obtained along this work, conclusions are drawn and presented towards solving the main objective, which is assembling scenes made of underground utilities where each individual object can be picked, queried and displayed using specific appearance options. In addition, contributions to the field are presented accordingly, and also directions towards future work.

7.1 Conclusions

The different reconstruction approaches presented and their mapping into the corresponding scene graph showed a clear distinction between its produced complexity and the rendering performance. This difference showed how reconstruction methods with an efficient encoding not necessarily translate into an efficient rendering, but also shows how common techniques for improving the rendering performance not necessarily deliver the desired results in WebGL.

In order to assemble the desired scenes and achieve realtime visualization using WebGL, the following conclusions should be considered when reconstructing underground objects.

• The CPU is the bottleneck. According to the presented results, applying some common optimizations like reducing the triangle count do not bring tangible benefits on the rendering speed, however the
reason behind it was caused by another factor minimizing the advantages. In the studied case based on WebGL, the CPU is a bottleneck limiting the number of rendered nodes per frame and vanishing the benefits of other improvements. Given this result, the first decision taken towards improving the performance with significant margins should reduce the number of nodes.

- **Aggregating nodes is a good solution.** The reconstructed objects are composed of multiple pieces sharing identical appearance, identifiers and administrative information, making them suitable for being aggregated or packed into a single node. Not every reconstruction approach allows to pack geometries and reduce the number of nodes. Fortunately, the described family of ‘non-split’ methods allows such packing and in particular, the ‘stitch’ approach shines by reducing the triangle count while, discarding most of the spheres, reducing the number of faces per cylinder, removing the duplicated vertices on the shared boundaries, but also keeping the visual quality.

- **‘Stitching’ is an optimization over previous approaches.** The ‘split’ method is considered an optimization compared to other approaches, reducing the size of the declared scene and increasing the performance at the rendering stage. In addition, the introduced optimizations are independent of specific scene graph implementations and rendering pipelines, being able to run on both programmable or fixed pipelines, guaranteeing its applicability on different platforms.

- **WebGL weakness is the performance of Javascript.** WebGL has proven to be an interesting technology to display 3D graphics embedded in the browser. However, the inherent characteristics of reconstructed underground objects produce scenes which exploit one of the weaknesses of the technology: the speed at which the can be submitted to the GPU. This weakness is originated by the Javascript language, which is an interpreted language producing additional overhead on the CPU and therefore reducing the processing power to submit work to the GPU.

In comparison, standalone players are usually developed using compiled languages and a have closer integration with the underlying hardware and graphic drivers. This difference allows to achieve a higher performance, which on the tested case produced twice to ten times more frames per second than its Javascript counterpart.

Based on this set of conclusions, the main research question is answered accordingly and rephrased here as follows:

The reconstruction approaches for underground utilities that improve the performance on WebGL clients are those who produce a small number of nodes within the scene graph. Once the number of nodes have been reduced, other improvements are welcome and proved to bring performance benefits, being the most relevant the reduction in the size of the encoded files. Proof to this statement is the “stitching” approach developed in this research, improving the performance over other approaches while reducing the size of the produced files.

### 7.2 Main contributions

The main contribution of this thesis is the analysis and development of a reconstruction method for underground utilities. The ‘stitching’ of cylinders allows to keep the visual quality of the objects while reducing their overall complexity and improving the rendering performance. In addition to this, other contributions were made, being the most relevant:

- **Classification of reconstruction methods:** The identification of the main building blocks and parameters used during the reconstruction of underground utilities allows to classify the various approaches for reconstruction, giving structure and order to the multitude of available options.

- **Testing Framework:** With the separation between the reconstruction methods and their building blocks, a testing framework was introduced to analyze the interaction between the building elements and parameters of the objects, but also to know their impact on the rendering stage. With this framework, the assessment of the reconstruction method was carried out on a formal basis, ranking the tested performance factors and giving answer to the performance-testing questions in Section 6.7.
7.3. **RECOMMENDATIONS AND FUTURE WORK**

- **Baseline results** Based on the testing framework and the ranking of the factors, a baseline for performance was obtained. This baseline allows to quantify the difference in performance between the tested factors, but also to understand the limits of each specific reconstruction approach.

- **Optimization path** The reduction of nodes in the scene is the best way to improve the performance on the tested platform, which can be achieved packing several objects in a static way before rendering. Under this assumption, common suggested practices like the use of ‘Levels of Detail’ to reduce only the triangle count should be discarded given their small to null improvements, as they are not attacking the main problem of WebGL: the speed of Javascript and its direct impact on submitting nodes for rendering. In this regard, the study of procedures for reducing the number of rendered nodes is guaranteed to introduce benefits for the reasons exposed before.

### 7.3 Recommendations and future work

The following suggestions are a starting point for improving the obtained results and continue working in the reconstruction and visualization of underground utilities.

- **Usage of topological models.** The datasets studied did not included any connectivity information useful to aggregate the objects under study. With such information, more improvements can be obtained as disconnected elements can be grouped to reduce the visual gaps.

- **Extensions to 3D formats:** The reduction in the number of nodes proved to be a key for performance improvement and with the results of this research, the foundations for two ‘use cases’ are presented as suggestions to be incorporated not only in X3DOM and SceneJS, but in many 3D file formats to include optimizations at the declarative level within the format specifications. Those use cases deal with scenes made of a large number of objects with visual similarities, but where individual objects need to be picked.

  The first ‘use case’ introduces the ‘levels of aggregation’, packing nodes with respect to the distance to the camera. When the objects are far from the camera, a single node may pack similar geometries based on appearance and ignore their picking elements, while at closer distance the node can unpacking its content and enable individual picking.

  The second ‘use case’ tackles the limitation on the level of aggregation possible, bounded by the requirements to perform object picking. On typical scenarios, one geometry defines one picking object, producing a one-to-one relationship between them and introducing a limit for aggregation. From the distinction between ‘rendering pass’ and ‘picking pass’, it possible to separate the rendering geometry from the picking geometry without duplicating content while keeping the picking behavior for individual components. This recommendation suggests that multiple nodes should be packed into a single array but the indices delimiting individual objects should be preserved. With this approach, the ‘rendering pass’ draws the complete array, where the held nodes share visual attributes. In the ‘picking pass’, the stored indices are used to draw segments of the array, corresponding to each individual object using the exact geometry. Making independent draws is a requirement to implement object picking on the GPU, while drawing the exact geometry is requirement for engineering applications where the exact geometry should be picked and not some approximation.

- **Native implementations:** With all the bottlenecks introduced by Javascript and all the layers between the browser, sandboxed environments, translation libraries, graphics libraries and the GPU, the wish to have a WebGL implementation closer to the hardware definitely would improve the performance of the platform. In this regard, the most expensive operations should be reviewed and prioritized towards having native implementations. As proven during this research, the GPU is mostly waiting for the CPU and based on the results obtained with the standalone player, at least twice the performance can be obtained with a native implementation, extending the scalability of the system on the same proportion.
BIBLIOGRAPHY


‘A man can be destroyed but not defeated.’

Ernest HEMINGWAY,
The Old Man and The Sea