Visualization of GML data using XSLT.

W.T.M.S.B.Tennakoon

February, 2003

Visualization of GML data using XSLT.

by

W.T.M.S.B.Tennakoon

Thesis submitted to the International Institute for Geo-information Science and Earth Observation in partial fulfilment of the requirements for the degree in Master of Science in *Geoinformatics*.

Degree Assessment Board

Thesis advisor	First Supervisor : Drs. R.A. Knippers Second Supervisor : Ir. R.L.G. Lemmens
Thesis examiners	Chairman : Dr. Ir. R.A. de By External Examiner : Prof. Dr. F.J. Ormeling

INTERNATIONAL INSTITUTE FOR GEO-INFORMATION SCIENCE AND EARTH OBSERVATION

ENSCHEDE, THE NETHERLANDS

Disclaimer

Abstract

The problem of geospatial data interoperability has been an issue throughout the GIS (Geographic Information Systems) industry for a long time. The Open GIS Consortium developed Extensible Markup Language (XML) based Geography Markup Language (GML) with the intention to overcome this issue. GML is the standard for transport and storage of geographic information for those who need spatial data sharing in the future. GML, being a subset of XML, separates the content from presentation. Making maps from GML data involves a transformation of GML data to a display format that can be interpreted by viewer software. Scalable Vector Graphics (SVG) is an application of XML to two-dimensional graphics that can be used to visualize GML data. The transformation of GML data into SVG can be accomplished using a Extensible Stylesheet Language Transformation (XSLT) stylesheet together with an XSLT processor. The XSLT stylesheet is an XML based document that describes how data in GML document is transformed into graphic elements in the SVG document. By using different stylesheets, the same GML data set can be visualized differently. In the same way, different data sets having a homogenous schema can use the same stylesheet. The possibility to use one stylesheet for the visualization of different data sets could be very useful in geographic data visualization. The Cascading Style Sheet mechanism and script languages can be used together with SVG to enhance the effectiveness of the visualization. This research is entirely focused on "How GML data is transformed to SVG using XSLT stylesheets to view in web browsers and how SVG display can be changed with Cascading Style Sheets".

Keywords

Visualization, XML, GML, XSL, XSLT, SVG, CSS

Contents

List of Figures

List of Tables

Acknowledgements

First of all I would like to extend my sincere gratitude to the Netherlands government and ITC for providing the financial and a remarkable educational environment for me. Special thanks go to Sri Lanka government and Survey Department of Sri Lanka for offering me this opportunity.

My heartfelt gratitude goes to my first supervisor, Drs. Richard Knippers, for clear guidance and encouraging me through out this task. I must be thankful for your interest and deep insight on my research and, the way of supervision which guided me step by step from the very beginning.

I like to thank my second supervisor Ir. R.L.G. Lemmens, for giving me valuable suggestions and guidance throughout the thesis period.

My honor goes to the ITC academic staff specially in GFM and GIM courses who enlightened me in the field of Geoinformatics. Mr. Gerrit Huurneman, and Dr. T. Bouloucos , Student Advisors Master of Science Degree courses in Geoinformatics (GFM), thanks for your guidance during my study period of 18 months in ITC. And I must thank the non academic staff including Computer Cluster Manager, Library, Student affairs and Dean's office, Book shop and Help desk for their courteous service.
And I sadly remember the unexpected departure of Mr. Allen Brown who was our GFM course coordinator with full of humanity and kindness, during the course of study.

Thanks all GFM-2 (2001), GFM-3 (2001) classmates and specially my Sri Lanken colleagues for giving me a friendly environment to enjoy my study and stay in the Netherlands.

I cannot express enough of my appreciation to my mother and late father who devoted their lives to take care of me and to make me educated, I owe you for ever. I can never forget my beloved wife Pushpa for encouraging me all the way with love and affection, and taking care of my daughter Thushanka and son Chamith during my long absence. I thank you so much for making me mentally so comfortable to concentrate on my studies.

My thanks also go to those whose names are not mentioned here but encourage me and helped me during my research in one way or another. Thank you all.

Chapter 1

Introduction

1.1 Background & Motivation

Geographic data describes the geographic features and phenomena on the earth and makes the base for Geographic Information Systems (GIS) as well as for maps. In many countries, the production of geographic data, which is inherently costly and time consuming, is considered to be one of the major National tasks. The high production cost and it's vital role in GIS and Mapping has increased the need of data sharing. At present, many countries produce and store geographic data in digital formats where sharing is much easier than ever before. Unfortunately this process is not yet straightforward, because of the incompatibility of data which use different data models. There are solutions given but all of them have it's pros and cons. Therefore problems of interoperability has been an issue for a long time in the field of GIS. Using a widely adopted, non-proprietary, neutral format for storage and transport of spatial data in an extendable and flexible way could resolve most of the issues in sharing and interoperability.

The Open GIS Consortium (OGC), an organization whose mission is to address issues of interoperability between systems that process geospatial data through the establishment of standards, developed Geography Markup Language (GML) to solve most of the issues in data interoperability. The GML is an XML (Extensible Markup Language) based encoding standard for the transport and storage of geographic information including both geometry and properties of geographic features [15]. The use of GML as a standard for describing geographic information enables data sharing and integration easier than ever before. Since GML is an XML dialect, GML upholds the principle of separating geographic content from its presentation and it does not address the visualization of the geographic features.

The separation of content from presentation offers more flexibility in data production, handling as well as in visualization. Therefore data providers need not to concern about the presentation aspects, but the importance of visualizing the content remains unchanged. In geographic data visualization graphical visual properties such as colours and symbols play an important role to make the

presentation more informative and effective.

Visualization of XML structured geographic data in SVG through XSLT stylesheets has been carried out before. In a research done by Philip A. Mansfield and Darryl W. Fuller in collaboration with National Research Council Canada [14], a stylesheet was used to visualize geographic data in SVG. Another research carried out by Gilles Taladoire at university of New Caledonia [24] has used XSLT to transform GML to SVG. The resulted map has no appropriate symbology used. Technical University of Delft, The Netherlands has also used the same procedure in visualizing GML data in a project carried out by them for the Dutch Topographic Service (TDN) [25]. They have embedded SVG in HTML pages and basic GIS functions have been programmed with JavaScript. Styling attributes for each feature layer is kept in a CSS. The main purpose of developing the viewer was in order to test the conversion from the datasets provided by the TDN into GML. They have not concentrated much on the visualization aspects such as the use of symbols.

1.2 Problem Definition

Geospatial data providers like National Mapping Organizations have already commenced supplying digital data in GML format. For instance, the Ordnance Survey in the UK has adopted GML standards for producing their Digital National Framework dataset. Many countries have launched pilot projects to convert their data to GML. One of them is the Dutch Topographic Service in The Netherlands, which has started re-engineering their the data structure of their products to meet future user requirements. One step in that process is to develop a new Digital Landscape Model and adopt GML [25].

Whatever the usage of geographic data, finally it has to be visualized in a certain media. As mentioned earlier, GML data does not contain presentation information. In order to visualize the GML content, the data has to be converted into a graphic format that a graphic viewer can interpret. This process is referred to as Map Styling and can be accomplished through technologies such as XML Transformation (XSLT). However the graphic format does not have to be XML-based, but given the increasingly wide availability of XML Transformation Language (XSLT) and the XML-based two dimensional graphic language Scalable Vector Graphics (SVG), it is highly desirable [17]. Apart from that GML, XSLT and SVG are XML based open standards of OGC and World Wide Web Consortium.

GML data visualization into SVG is not straightforward. It includes creating stylesheets that encode how to draw specific visualizations of structured data and transform it with a XSLT parser. GML and other XML based technologies used in the process of styling, transformation and visualization are new and under development.

1.3 Research Objectives

The main objective of this research is to explore how GML structured data can be styled and converted into SVG format for the visualization in web browsers. Another goal is to study the use of Cascading Style Sheets in this process.

1.4 Research Questions

Based on the objectives, the following research questions have been defined:

- How is GML data structured ?
- How can GML data be visualized ?
- How can XSLT be used to transform GML data to SVG ?
- What are the possibilities of SVG in data visualization ?
- How is GML data converted into SVG format ?
- How can Cascading Style Sheets be used in the SVG visualization process ?

1.5 Methodology

In this research, relevant literature on Extensible Markup Language (XML) and XML based technologies such as Geography Markup Language (GML), Extensible Stylesheet Language Transformation (XSLT) and Scalable Vector Graphics (SVG) have been reviewed. Thereafter an XSLT stylesheet has been developed to transform GML data into SVG graphical format. The stylesheet transforms GML data based on the Dutch Topographic Survey (TDN) application schema. Finally the use of the Cascading Style Sheet (CSS) mechanism on changing the SVG visualization and the use of JavaScript on displaying SVG attributes have been studied. A sample GML data set from the Dutch Topographic Service (TDN) is used for testing purposes.

Step 1. Study the GML application schema to identify feature collection classes and their properties.

Step 2. Decide on their presentation geometries (Point, Line or Area) and visual properties accordingly.

Step 3. Define symbols and patterns for the resulting SVG document.

Step 4. Develop the XSLT stylesheet to comply with XSLT 1.0 specification and transform GML features into corresponding SVG graphic elements. The XSLT stylesheet refers to an external XML document which contains SVG symbol and pattern definitions to be extracted during processing by the XSLT processor. The output SVG document refers to the Cascading Style Sheet that contains styling information and to the JavaScript file that provides coding for adding interactivity to the SVG output.

Step 5. Proces the XSLT stylesheet using an XSLT processor.

Step 6. Create different Cascading Style Sheets for changing appearance of SVG display and develop a JavaScript file for adding interactivity.

The expected methodology is depicted in Figure 1.1

1.6 Thesis Structure

Chapter 1. Introduction; describes the research background and motivation, problem definition, research objectives and questions, methodology and the thesis outline.

Chapter 2. XML based technologies; focused on XML and related technologies.

Chapter 3. Geographic data in GML; describes how geographic data is represented in Geography Markup Language.

Chapter 4. XSLT in GML data visualization; details what XSLT is and how to use it in transforming XML data (GML) to another graphical XML format (SVG)

Chapter 5. GML data visualization in SVG; provides information on how SVG can be used to visualize GML data and the use of CSS in changing SVG display.

Chapter 6. Implementation; describes the procedure followed in developing the XSLT stylesheet, symbol library, Cascading Style Sheets and JavaScript that results in a SVG visualization.

Chapter 7. Conclusion and Recommendations.

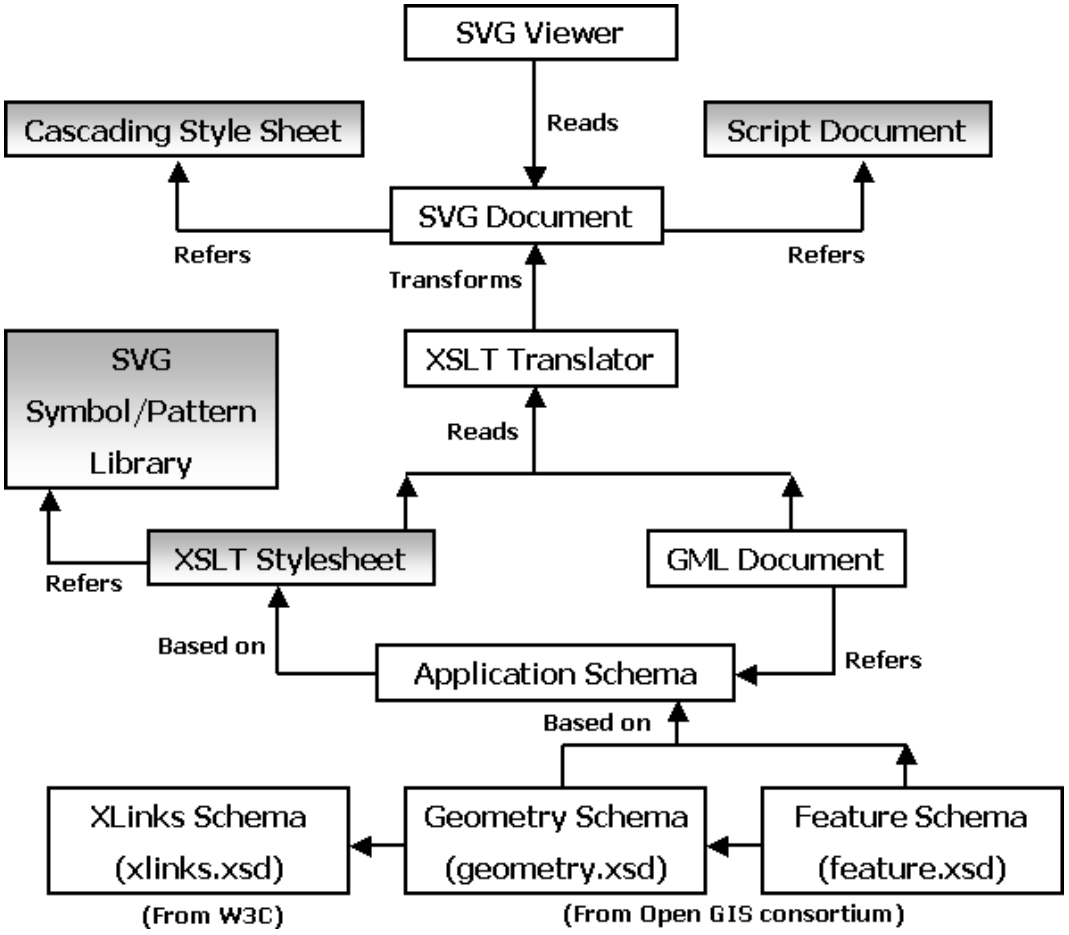

Figure 1.1: Methodology for GML data visualization.

Chapter 2

XML Based Technologies

2.1 Introduction

This chapter examines what is Extensible Markup Language, its advantages and the basics behind it. It also describes the XML structure, and the relation with Document Type Definition, Schema, Document Object Model, and stylesheets which are common in any XML application.

2.2 Evolution of XML

Information on a network, which connects many types of computers, should be usable on all of them. Information in a form that can be reused in different ways, can help to minimize waste of time and effort on reformatting.

Standard Generalized Markup Language (SGML) which is the international standard (ISO 8879:1986) for defining the structure and content of different types of electronic documents was introduced for this kind of purpose. Thereafter it has been used as the ”mother tongue” for describing thousands of different document types in many fields and Hyper Text Markup Language (HTML) is one of those document types. The problem with SGML is that it is too general and complex than Web browsers and average users can cope with. HTML was designed to provide Web authors with a relatively simple and efficient means of publishing documents for Web distribution. The introduction of HTML has largely contributed the rapid expansion of World Wide Web.

Since HTML is more concerned with information presentation rather than providing useful information about the content, it is of limited use as a way of describing information. To work efficiently with information, computers need to be told exactly what the information is, how it is related and how to deal with it. Due to its nature, HTML has failed to cope up with this need of many new applications that require a more robust and flexible infrastructure. The XML is a new meta-language designed to achieve this goal [9].

2.3 The Extensible Markup Language (XML)

"The Extensible Markup Language (XML) is the universal format for structured documents and data on the Web".

The XML 1.0 specification describes XML as a simple dialect [or 'subset'] of SGML with the goal to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with HTML. For this reason, XML has been designed for ease of implementation, and for interoperability with both SGML and HTML [23].

XML is a markup language, designed to structure, store and send information over the Web. It provides no predefined tags, as in the case of HTML, but provides standards so that the user can define his own tags and document structure. Hence XML is free and extendible. Furthermore, as XML is in plain text format, it provides a software and hardware independent way of sharing data. It enables data to be accessed by all kind of "reading machines" or processors.

Since XML documents are in plain text and structured (encoded) with user defined tags, it does not do anything without some kind of software. Therefore it has to follow some standards in encoding data to enable decoding by some other programs. For this XML adheres to the standards specified by the XML specifications. At present (October 2002) XML Specification 1.0 is the World Wide Web Consortium (W3C)'s implementation recommendation. XML documents must follow standard rules including the syntax for marking up and the meaning behind the markup. What a valid markup is defined by a Document Type Definition (DTD) or alternatively by an XML Schema.

2.3.1 XML Document Structure

There are certain rules that have to be adopted while authoring XML documents and these can be summarized as follows:

- XML documents need a declaration at the top to signal what they are;
- Every XML document must have a root element (tag) that encloses the content;
- Every start tag must have a closing tag;
- Tags must nest cleanly;
- Empty tags have a different form to make it clear that these are tags with no closing tags;
- All attribute values must be in quotation marks;
- Tags are case sensitive and must match.

The XML documents must be precise, those do not comply with these rules can not be processed by the XML parsers embedded in browsers or standalone processors. An XML document that conforms to these rules specified in XML specification, as determined by an XML parser is classified as *well-formed*. An XML Parser is a software program that creates the Document Object Model (See section 2.7) in the computer memory [12].

An XML document mainly consists of two parts; prolog and body. The prolog contains the declaration, and the body contains the actual marked up document. The content of both parts (whole document) is composed of *declarations, elements, comments, character references*, and *processing instruction*, all of which are indicated in the document by explicit markup [21].
An example XML document is given below:

```
<? xml version="1.0" encoding="ISO-8859-1" standalone="no" ?>
<!DOCTYPE Thesis SYSTEM "Thesis.dtd">
<!- - An XML Example - ->
<thesis>
    <title>Visualization of GML data using XSLT</title>
    <pub_date>2003-01-02</pub_date>
    <production id="4084" media="paper"/>
    <chapter>Introduction to XML
        <para>What is SGML</para>
        <para>What is HTML</para>
        <para>What is XML</para>
    </chapter>
    <chapter>XML syntax
        <para>Elements must have a closing tag</para>
        <para>Elements must be properly nest</para>
    </chapter>
</thesis>
```

2.3.2 Declarations

The first line of an XML document is a declaration which notifies that the document has been marked up as an XML document. The XML declaration itself is a *processing instruction* and therefore it begins with <? and ends with ?>. The *version* attribute indicates the version of XML specification that the document complies with and the *standalone* attribute specifies whether the document has any markup declarations that are defined in a separate document. Thus, value "yes" implies no markup declarations in external documents and "no" leaves the issue open. The document may or may not access external documents. The *encoding* attribute denotes the character encoding system used in the document. The DOCTYPE declaration (second line in the example) declares the name, type and location of the related Document Type Definition (DTD).

2.3.3 Elements

Elements are the basic unit of XML content. An *element* consists of a start *tag*, and end *tag*, and everything in between. Anything between a < sign and a >

sign is a *tag* except that is inside a *comment* or a *CDATA* section. Relationships in XML elements are expressed in terms of *parents* and *children*. In the given example "thesis" is the *root element*. Title, pub_date, production, and chapter are *child elements* of "thesis" element and thus it is the *parent element*. Title, pub_date, production, and chapter are *siblings* (or sister elements) because they have the same *parent*.

Elements can have different content types such as *elements* that contains other elements like "thesis" element, *mixed* that contains both text and other elements as "chapter" element, *simple* that contains only text like "para" element in the example, or *empty* that contains no information like "production" element. In case of *empty* elements no closing *tag* appears.

The name of an *element* can not contain space and must not start with a number or punctuation character or with the word "xml". XML elements can have *attributes* in name/value pairs which provide additional information about elements and the attribute values must always be enclosed in either single or double quotes.

2.3.4 Comments

The *comments* are the character data in an XML document that XML processor ignores. The comments follow the syntax of <!- - content - ->. The content of the comment should not have "-" or "–" characters that might confuse XML parser and also a comment should not be placed within a tag and cannot be nested.

2.3.5 Character References

Character data includes any legal character except < which is reserved for the start of a tag. XML provides a set of entity references that helps to avoid the ambiguity in specifying character data against markup. In XML, >, <, &, " and ' characters can be substituted by &gt, &lt, &amp, &quot, and &apos respectively. CDATA blocks in XML provides a convenience measure to include large blocks of special character data. For example, a internal Cascading Style Sheet can be defined in CDATA block.]]> is not allowed within a CDATA block as it signals the end of a CDATA block.

2.3.6 Processing Instructions

A processing instruction is a bit of information meant for the application using the XML document. That is, they are not really of interest to the XML parser. Instead, the instructions are passed intact, straight to the application using the parser. The application can then pass this on to another application or interpret itself. All processing instructions follow the generic format

of <?Name_of_application_instruction_is_for Instructions?> and all processing instructions must be in lowercase. For example, XML declaration is a processing instruction, which name is “xml”.

2.4 Document Type Definition (DTD)

In XML the definition of a valid markup is handled by a Document Type Definition (DTD). It is a file (or several files to be used together) with “dtd” extension, written in XML’s Declaration Syntax, which contains a formal description of a particular type of document. DTD sets out what names can be used for element types, where they may occur, and how they all fit together. For example the DTD of the above XML document is as follows:

```
<!ELEMENT Thesis (Chapter+ | ANY)>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Pub_date (#PCDATA)>
<!ELEMENT Production (EMPTY)>
    <!ATTLIST Production
        id  NMTOKEN #REQUIRED
        media   CDATA #IMPLIED>
<!ELEMENT Chapter (Para+ | #PCDATA)>
<!ELEMENT Para (#PCDATA)>
```

In DTDs all keywords must be in UPPERCASE, such as ELEMENT, ATTLIST, #REQUIRED etc.. However user defined elements and attributes can be any case as user choose, as long as they are consistent. A DTD can either be included as part of a *well-formed* XML document, (standalone=“yes”), or it can be referenced from an external source, (standalone=“no”). When external DTD is referenced, the SYSTEM attribute whose value indicates the location of DTD has to be added to the DOCTYPE declaration. Thus in order to reference an external DTD, both XML declaration and DOCTYPE declaration have to be changed. An XML document that conforms to the rules of a DTD is called a *Valid* document. A *Valid* document is necessarily *well-formed*. When XML document is parsed by a processor, it use first to parse the DTD and then read the document to identify where every element type comes and how each relates to each other. Using a DTD when editing documents preserve them consistent and valid [21].

2.5 XML Schema

XML Schemas is a W3C Recommendation for defining the structure, content and semantics of XML documents. It is an alternative to DTD written in Schema Definition Language, which is an XML language for describing and constraining the content of XML documents. In general terms, an XML Schema describes how data is marked up and these files are given with “xsd” extension. XML Schema offers many advantages over DTD. One of the greatest advantages is the support for data types. Since DTDs are designed for use with text, they have no mechanism for defining the content of elements in terms of data

types. Therefore a DTD cannot be used to specify numeric ranges or to define limitations or checks on the data content. The XML Schema provides a means of specifying element content in terms of data type, so that document type authors can provide criteria for validating the content of elements as well as the markup itself. Some other strengths of a Schema are; they are written in XML thus avoids the need for another processing software; supporting namespaces and extensibility to future additions [6].

XML Schema of the above example can be defined as below:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xsd:schema
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.itc.nl"
    xmlns="http://www.itc.nl"
    elementFormDefault="qualified">
<!--An XML Schema Example -->

<xsd:element name="thesis" type="thesisType"/>

<xsd:complexType name="thesisType" mixed="true">
 <xsd:sequence>
  <xsd:element name="title" type="xsd:string"/>
   <xsd:element name="pub_date" type="xsd:date"/>
   <xsd:emement name="production" type="productionType"/>
    <xsd:complexType name="productionType">
     <xsd:complexContent>
      <xsd:restriction base="xsd:integer">
       <xsd:attribute name="id" type="xsd:positiveInteger"/>
       <xsd:minInclusive value="0"/>
       <xsd:maxInclusive value="1000"/>
      </xsd:restriction>
     </xsd:complexContent>
     <xsd:attribute name="media" type="xsd:string"/>
    </xsd:complexType>
   <xsd:element name="chapter" type="chapterType"/>
    <xsd:complexType name="chapterType"/>
     <xsd:sequence>
      <xsd:element name="para" type="xsd:string"/>
     </xsd:sequence>
    </xsd:complexType>
   </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

The <`schema`> element is the root element of every XML Schema where “xsd” is the namespace prefix. The fragment `xmlns:xsd="http://www.w3.org/2001/XMLSchema"` indicates that the elements and data types used in the Schema come from the `"http://www.w3.org/2001/XMLSchema"` namespace. The *targetNamespace* attribute indicates that the elements defined by this schema are from `"http://www.itc.nl"` namespace and *xmlns* attribute gives the default namespace.
The `elementFromDefault="qualified"` fragment indicates that any element used

by the XML instance document which will declare in this schema must be namespace qualified. All namespaces used in the schema must be declared in prolog and all elements and data types must be defined in the body.

An XML Schema document consists of four basic constructs: declaration of elements, definition of types (simpleTypes and complexTypes), the possibility to define subtypes by extending or restricting supertypes, and use of aliases (substitutionGroup mechanism).

2.5.1 Simple Type

An XML element that contains only text(data in any type) is a simple type element. It cannot contain any other elements or attributes. But it can have a default value or fixed value set. Simple element is defined by `<xsd:element name="Name_of_Element" type="Data_Type" default="Default_Value" (or fixed="Fixed_Value")/>`. Common data types in XML Schema are string, decimal, integer, boolean, date and time. The type of a simple element is defined by `<xsd:simpleType>`.

2.5.2 Attributes

In Schema an attribute is always declared as a simple type and an element with attributes always has a complex type definition. An attribute is defined by, `<xsd:attribute name="Attribute_Name" type="Data_Type"/>`. Attributes also can have a default or a fixed value specified. Even though all attributes are optional by default, with ”use” attribute it can be explicitly specify whether it is ”optional” or ”required”.

2.5.3 Restrictions and Extensions

Defining a type for XML element or attribute imposes a restriction for the element or attribute content. With XML Schemas, it is able to add own user restrictions to user XML elements and attributes. In the example restriction has imposed on *id* attribute. The *id* can have only integer values between 0 and 1000 (including that numbers). Likewise to limit the content of an XML element to a set of accepted values, the enumeration constraint can be used. All restrictions that can be applied to data types are given in XML Schema Specification. Moreover types can be defined by extending existing types.

2.5.4 Complex Type

An XML element that contains elements, mixed content, empty content or attributes is considered to be a complex type element. For example, *thesis* element is a complex one. Its type has been defined as *thesisType* separately. The child elements of *thesis* element are surrounded by the `<xsd:sequence>` indicator. Separate defining complex types offers more flexibility, because these types can be used by other elements too.

2.6 XML Namespaces

Since XML uses no fixed element names, the same element name could be used in different documents to describe different types of elements. If such XML documents are added together, there would be an element name conflict. XML Namespaces is

a method to avoid these element name conflicts. It solves the name conflicts using a prefix and the namespace attribute (xmlns) alone with element name. The namespace attribute is placed in the start tag of an element and all child elements with the same prefix are associated with the same namespace. The W3C Namespace specification states that the namespace itself should be a Uniform Resource Identifier (URI) which is a string of character that identifies an Internet Resource. The most common URI is the Uniform Resource Locator (URL) which defines an Internet domain address.

2.7 The Document Object Model (DOM)

XML standards include specifications of how an XML document should be parsed and represented within any computer irrespective of type or operating system. This internal representation (tree representation) of an XML document, which is generated within a computer by an XML parser is called the Document Object Model (DOM). DOM allows a single document to be accessed in the same way by different applications running on different computer platforms through XML tag references. In a software context, the DOM is a programming API (Application programming Interface) for an XML document, which defines the logical structure of documents, and the way a document is accessed and manipulated. It details the characteristic properties of each element of a document, thereby detailing how these components can be manipulated and, in turn, manipulating the whole document. Therefore with the DOM, programmers can create and build documents, navigate their structure, and add, modify, or delete elements and content. As a W3C specification, one important objective for the DOM is to provide a standard programming interface that can be used in a wide variety of environments and applications. The DOM can be used with any programming language and provides precise, language-independent interfaces.

2.8 Unicode System

In HTML, a document is in one particular language, whether English, Japanese or Arabic. Applications that can not read the characters of that language cannot do anything with the document. But XML uses the new Unicode standard, which is a character-encoding system that supports intermingling of text in the world's major languages. Because of that applications that read XML can properly deal with any combination of these character sets. Thus XML will enable exchange of information in different languages. The character set used in the XML document encoding is specified in xml declaration with *encoding* attribute. Since XML is fully internationalized for both European and Asian languages, with all conforming processors required to support the Unicode character sets in UTF-8 , UTF-16 and ISO-8859-1.

2.9 Viewing XML Documents

Since XML documents are text based, they can be viewed in any text editor. But they will not view any hierarchical structure of the document elements. The browsers like Internet Explorer 5.0 (and higher), Netscape 6 or any other XML editors like XMLSpy can be used to view XML documents. Any error in an XML document, will be reported by those browsers or editors. As mentioned above, since XML elements are user defined the browser has no idea how to display the content other than just viewing whole

document as it is. Therefore XML documents are associated with stylesheets which provides GUI (Graphic User Interface) instruction for a processing application like a web browser. There are different ways to visualize the content of an XML document using these stylesheets and they are presented in Figure 2.1. In addition, JavaScript (or VBScript) can be used to import data from an XML document and display them inside an HTML page. The use of CSS and XSL in XML data visualization will be discussed in the next chapters.

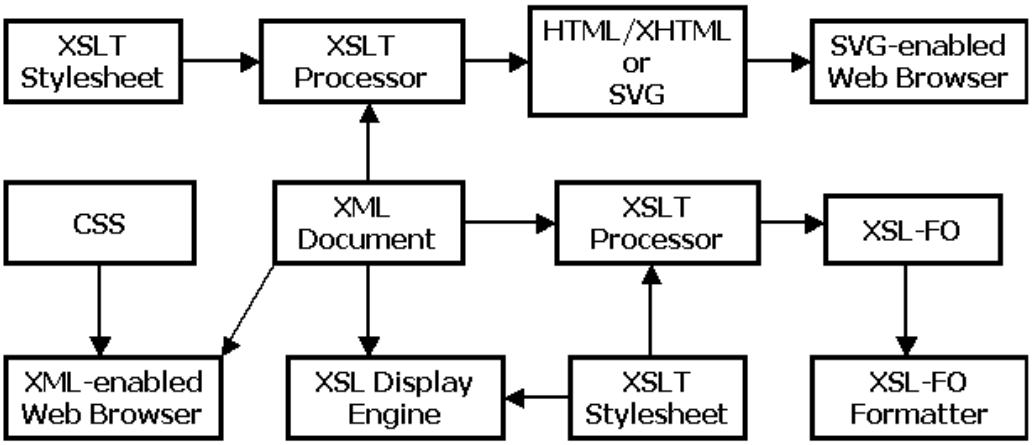

Figure 2.1: Options for displaying XML documents

2.10 Stylesheets

The stylesheets are the documents that provide information on data presentation. The international standard for stylesheet for SGML document is DSSL, the Document Style and Semantic Specification Language. This provides Schema-like languages for stylesheets and document conversion, and is implemented in the Jade formatter. Cascading Style Sheet provides a simple syntax for assigning styles to elements, and has been partly implemented in some HTML browsers. Extensible Stylesheet Language has been created for use specifically with XML. It uses XML syntax but combines formatting features from both DSSL and CSS. The stylesheet of an XML document should be declared in the prolog section of the XML document in the format of `<?xml:stylesheet href="Location_of_stylesheet" type="text/xsl"?>` incase of CSS the type="text/css".

The separation of style from the content allows for the same data to be presented in different ways and it enables:

- Reuse of fragments of data: the same content should look different in different contexts.
- Multiple output formats: different media (paper, online), different sizes (manuals, reports), different classes of output devices (workstations, hand held devices)

- Styles tailored to the reader's preferences (e.g. accessibility): print size, color, simplified layout for audio readers.
- Standardized styles: corporate stylesheets can be applied to the content at any time.
- Freedom from style issues for content authors: technical writers needn't be concerned with layout issues because the correct style can be applies later.

A stylesheet specifies the presentation of XML content using two basic categories of techniques. One is an optional transformation of the input document into another structure and the other is a description of how to present the transformed information.

2.11 XML based Markup Languages

Due to the flexibility and robustness of XML technology, a number of markup languages have been developed according to the XML standards and some of them are: Chemical Markup Language (CML),a markup language to managing molecular information; Wireless Markup Language (WML), used to markup Internet applications for handheld devices like mobile phones; MathML, an XML application for describing mathematical notation and capturing both its structure and content; Extensible Stylesheet Language (XSL),a language for expressing stylesheets; Geography Markup Language (GML), for transport and storage of geographic information; and Scalable Vector Graphics (SVG) to describe 2D vector graphics.

XSL, GML, and SVG are the markup languages that are reviewed extensively in the following chapters to lay the foundation for the process of visualizing GML data.

2.12 Conclusion

XML is a simplification of SGML. It enables easy implementation, and interoperability with both SGML and HTML. Any valid XML document must have a DTD or Schema that describes the structure. XML separates data content from presentation information. Stylesheets provide presentation information for XML documents and XML elements can be manipulated through the DOM. There are number of markup languages developed in different domains based on XML standards.

Chapter 3

Geographic Data in GML

3.1 Introduction

This chapter is focused on Geographic Markup Language and it discusses the use of GML and how it can be used to model geographic data. It also describes the possibilities of visualizing GML data. Through that it explains the structure of GML data and how to visualize them.

3.2 Background and Evolution of GML

Geographic Information is not limited to a stand-alone system or single corporate networks. It is distributed all over the world in heterogeneous environments. In most cases geographic data is collected by particular agencies for a particular purpose and therefor the data could be in different models. Applications that use diverse collection of data (e.g. disaster management, environmental protection, highway construction etc. and provision of on-line spatial data services and vendor-independent spatial client services) have to interoperate with systems not only just on the data level but also at the application level. It involves data sharing between systems as well as data structuring in a way that has meaning in the appropriate paradigm.

The introduction of XML has enabled easy creation of specialized markup languages and as a result some geographically oriented languages were created to cope up with above requirements, but none was successful as an acceptable standard for the geo-information field. Finally, the OpenGIS Consortium (OGC) proposed a standard called Geographic Markup Language (GML), for representing geographic data and it's meaning in an XML-based form. It specifies structure for encoding data and their associations and provides a means for transporting and storing data in the Internet environment [17].

"The Geography Markup Language (GML) is an XML encoding for the transport and storage of geographic information, including both the spatial and non-spatial properties of geographic features" [1].

GML 1.0 was approved by OGC in May 2000 and it is based on XML 1.0 specifications. It consists of three forms of GML profiles, which offer different levels of flexibility in data structuring. GML 1.0 makes use of Document Type Definition (DTD) and Resource Description Framework (RDF) for encoding geographic features in three profiles [1].

The Profile 1 is for those who wish to use a pure DTD based solution and are not prepared to develop application specific DTD's or wish data to be returned against a fixed set DTD's. This profile requires the use of GML Feature, and GML Geometry DTD's.

The Profile 2 is for those who wish to use pure DTD based solution but are prepared to develop their own application specific DTD's or are prepared to accept data encoded with a referenced DTD. This profile requires the user to create an application specific Feature DTD that uses the GML Geometry DTD.

The Profile 3 is for those who are prepared to make use of RDF (Resource Description Framework) and RDF Schema. The users of this profile will typically require stronger control over the geospatial typing framework (e.g. they must be able to relate a type name to an actual schema definition). This profile requires the user to create an application specific RDF Schema definition that uses the GML RDF Schema definition. Alternatively Profile 3 users may employ DTD's which are derived in some fashion from an RDF Schema or which can trace their elements to types defined in an associated RDF Schema. RDF Schema, used in profile 3 is an alternative basis for DTD, which is used in profile 1 and 2.

GML 2.0, was developed and passed in March 2001 by OGC as the successor to the GML 1.0. It provides a single encoding method (XML Schema) and a single approach for creation of GML schemas. XML Schema adoption in GML 2.0 avoids the awkward combination of DTD and RDF in GML 1.0 and it incorporates support for type inheritance, distributed schema integration, and namespaces. Moreover it provides a rich set of primitive data types (e.g. string, boolean, float, month), and allows the creation of built-in and user-defined datatypes. The use of XML Schema in GML, provides validation mechanism defined in XML 1.0 specification [20].

GML 2.0 specification defines three base schemas; Geometry schema (geometry.xsd), Feature schema (feature.xsd), and XLink schema (xlinks.xsd) for encoding geospatial information. These three basic meta-schemas provide a set of foundation classes from which an application schema can be constructed.

3.3 GML Features

GML models the world, being based on a common model of geography, called the OpenGIS Abstract Specification, which defines a geographic feature as;

" A feature is an abstraction of a real world phenomenon; it is a geographic feature if it is associated with a location relative to the Earth." [20]

Therefore the real world can be digitally represented by *features*. The state of a *feature* is defined by a set of *properties*, where each *property* has a *name, type,* and *value* triplet. The *type* is the *class* and properties are *associations*, or *attributes* of the *feature class* in object modelling terminology. Geographic features are those with properties whose values may be a geometry. A feature collection is a collection of features that can itself be regarded as a feature. Consequently a feature collection has a feature type and thus may have properties of its own, in addition to the features it contains.
The definition of features types depends on the domain of application and the number

of properties a feature may have is determined by feature type.

3.4 Simple Features

GML specifications 1.0 and 2.0 are concerned with what the OGC calls *Simple Features*. These are features whose geometry properties are restricted to two-dimensional geometry. Even though GML 2.0 does permit coordinates to be specified in three dimensions, it provides no direct support for three-dimensional geometry construct. The *Simple Feature model* is a simplification of the general model described by OGC "Abstract Specification". This has resulted two major simplifications, one is; the features are assumed to have either Simple Properties or Geometric Properties; and the other is, geometries are assumed to be defined in two-dimensional SRS (Spatial Reference System) and use linear interpolation between coordinates.

Furthermore, *Simple Feature model* provides geometry elements corresponding to the geometry classes of Point, LineString, LinearRing, Polygon, MultiPoint, MultiLineString, MultiPolygon, MultiGeometry and Box. Simple properties are those that may be given by basic data formats such as strings, integers, real numbers, Boolean etc.

3.5 Geometry Property

The OGC simple feature model defines a set of basic geometries (mentioned above), and GML defines a set of geometric property elements to associate these geometries with features. A feature type can have more than one geometry property describing its different geospatial properties. GML names Geometry properties in three levels; Formal name, denotes geometry properties allowed as a property value based on the type of geometry; Descriptive names, a set of standardized synonyms for the formal names that provides a user-friendly set of terms; and Application specific names, chosen by user and defined in application schemas based on GML.

The formal and descriptive names for the basic geometric properties are listed in Table 3.1.

Table 3.1: Basic geometric properties in GML 2.0. (Source: GML 2.0 specifications)

Formal name	Descriptive name	Geometry type
boundedBy	-	Box
pointProperty	location, position, centerOf	Point
lineStringProperty	centerLineOf, edgeOf	LineString
polygonProperty	extentOf, coverage	Polygon
geometryProperty	-	*any*
multiPointProperty	multiLocation, multiPosition, multiCenterOf	MultiPoint
multiLineStringProperty	multiCenterLineOf, multiEdgeOf	MultiLineString
multiPolygonProperty	multiExtentOf, multiCoverage	MultiPolygon
multiGeometryProperty	-	MultiGeometry

3.6 Core GML Schemas

GML 2.0 defines three base schemas for encoding spatial information. These schemas provide the building blocks for constructing GML application schemas. In other words, the base GML schemas provide a meta-schema or set of foundation classes, from which an application schema can be constructed.

3.6.1 The Feature Schema

The Feature schema defines the general feature-property model which describe both abstract and concrete elements and types. It supports feature collection (as feature type) and includes common feature properties such as *fid* (a feature identifier), *name* and *description*. The `<include>` element in the Feature schema brings in the definitions and declarations contained in the Geometry schema for use in defining feature types [20]. The UML model of Feature schema is shown in Appendix A.

3.6.2 The Geometry Schema

The GML Geometry schema includes the detailed geometry components, that is type definitions for both abstract geometry elements, concrete(multi) point, line and polygon geometry elements, as well as complex type definitions for the underlying geometry types. The Geometry schema targets the 'gml' namespace. The `<import>` element in the Geometry schema brings in the definitions and declarations contained in XLink schema. The UML model of Geometry schema is shown in Appendix B.

3.6.3 The XLink Schema

As W3C states, the XLink allows elements to be inserted into XML documents in order to create and describe links between resources. It uses XML syntax to create structures that can describe links similar to the simple unidirectional hyperlinks of HTML, as well as more sophisticated links. The XLink schema in GML provides the XLink attributes to support linking functionality.

3.7 Encoding Geographic Information with GML

Using the three base XML schemas described above, it is possible to encode a wide variety of geospatial information such as; features with or without geometry, geometry, collection of features, and associations between features. GML 2.0 specification explains how these encodings are performed.

As described in section 3.3, a geographic feature in GML is a list of simple and geometric properties. The features are captured as element names in GML and the feature class definition prescribes the named properties that a particular feature type should have. GML follows some conventions in encoding these. Type definitions take the corresponding class name and append the "Type" suffix. Type names are in mixed case with leading capital (e.g. A Road feature is coded as `<element name="Road" type="RoadType"/>`, the names of geometric properties and attributes are in mixed case with leading lower case character (e.g. pointProperty, "familyName"). The name of abstract elements are in mixed case with leading underscore (e.g. _Feature, _FeatureCollection).

3.8 GML Application Schemas

The three core XML Schemas (Feature Schema, Geometry Schema and XLink Schema) alone do not provide a schema suitable for constraining data instances; rather, they provide base types and structures which may be used by an *application schema*.
A GML application schema is an XML Schema document constructed with the components provided by the base GML schemas. It declares the actual feature types and property types of interest for particular domain, using components from GML in standard ways. Defining an application schema from components in base GML schemas, benefits from standardized constructs and guaranteed to conform to the OGC Feature Model.

Any GML application schema must adhere to the schema development rules described in GML 2.0 specification and must not change the name, definition, or data type of mandatory GML elements. But in application schemas, the abstract type definitions can be freely extended or restricted. Furthermore an application schema must target a namespace and that must not be 'gml' namespace. And such an application schema must be made available to anyone receiving data structured according to that schema.

3.9 Structure of an Application Schema

As any other XML document, a GML application schema also consists of header (prolog) and body.

3.9.1 Header

The header starts with a xml declaration which consists *version* and *encoding* attributes. The value of *encoding* attribute indicates the Unicode character set used in encoding GML.

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

The <schema> open tag must contain the namespaces that are used for the schema definition. The *targetNamespace* attribute defines the user namespace, and its value is a unique identifier for the GML namespace. For an example:

```
<schema targetNamespace="http://www.itc.nl/gfm"
    xmlns:gfm="http://www.itc.nl/gfm"
    xmlns:gml="http://www.opengis.net/gml"
    xmlns:xlink=http://www.w3.org/1999/xlink
    xmlns=http://www.w3.org/2000/10/XMLSchema
    elementFormDefault="qualified" version="1.0">

<annotation>
  <appinfo>gfm.xsd v1.0 </appinfo>
   <documentation xsl:lang="en">
    GML application schema for Topographic data
   </documentation>
</annotation>
<!--import constructs from the GML Feature and
  Geometry schemas-->
<import namespace=http://www.opengis.net/gml
  schemaLocation="feature.xsd" />
```

The <annotation> element provides the information about the application schema and the <import> element imports other schema definitions that are used in this schema. In this case the GML feature schema definition (feature.xsd) is imported and it includes geometry and xlinks schemas. The *schemaLocation* attribute describes the location path of the importing schema.

3.9.2 Root Element

The first <element> in the application schema defines the root element and becomes the open tag of the GML file. In schema definition, the root element is a *substitutionGroup* for the *glm:_FeatureCollection* and a *extension* based on *gml:AbstractFeature CollectionType*. For example, if the topographic information of a village is modelled in GML, it may consist of road, river, terrain, building, utility and boundary feature classes. The collection of these feature classes is the root element and if it is named as *VillageTopoModel*, it can be defined in the schema as follows.

```
<element name="VillageTopoModel" type="gfm:VillageTopoModelType"
   substitutionGroup="gml:_FeatureCollection/>

<complexType name="VillageTopoModelType">
 <complexContent>
  <extension base="gml:AbstractFeatureCollectionType">
   <sequence>
    <element name="dateCreated" type="date"/>
   </sequence>
  </extension>
 </complexContent>
</complexType>
```

3.9.3 Shared Definitions

There are some properties that are shared by many objects in a schema definition. Such shared definitions are modelled with group definitions in XML Schema. For example, temporal and metadata information will be related to every feature in the model. To inherit these properties in all features, a *complexType* can be defined that refers to group definitions and all feature types have to be defined as extensions of that type.

```
<group name="TemporalData">
 <sequence>
  <element name="begindate" type="string" />
  <element name="enddate" type="string" />
 </sequence>
</group>

<group name="MetaData">
 <sequence>
  <element name="source_type" type="string"/>
  <element name="source_description" type="string"/>
  <element name="accuracy" type="string"/>
  <element name="actuality" type="string"/>
  <element name="code_num" type="integer"/>
 </sequence>
</group>
```

```
<complexType name="TopoDataType" abstract="true">
 <complexContent>
  <extension base="gml:AbstractFeatureType">
   <sequence>
    <element name="code_id" type="integer"/>
     <group ref="gfm:TemporalData"/>
     <group ref="gfm:MetaData"/>
   </sequence>
  </extension>
 </complexContent>
</complexType>
```

3.9.4 Feature Definitions

Features are the objects that are visible on the map. In the example below, building is a one feature type and the collection of building features are represented by *BuildingLayer* type. All such feature layers can be defined as a *TopoDataLayerType* which is an extension of *gml:AbstractFeatureCollectionType*.

```
<complexType name="TopoDataLayerType">
 <complexContent>
  <extension base="gml:AbstractFeatureCollectionType"/>
 </complexContent>
</complexType>

<element name="BuildingLayer" type="TopoDataLayerType"
   substitutionGroup="gml:_FeatureCollection"/>

<element name="Building" type="gfm:BuildingType"
   substitutionGroup="gml:_Feature"/>

<complexType name="BuildingType" >
 <complexContent>
  <extention base="gfm:TopoDataType>
   <sequence>
    <element name="type" type="string"/>
    <element name="function" type="string"/>
    <element name="height_category" type="string"/>
    <element name="height" type="string"/>
    <element name="status" type="string"/>
    <element ref="gml:geometryProperty"/>
    <element name="heightlevel" type="integer" minOccurs="0" />
    <element name="name" type="string" minOccurs="0" />
   </sequence>
  </extension>
 </complexContent>
</complexType>
```

All other feature layers can also be defined in the same way.

3.9.5 Constrains on Property Values

A property of feature can have a predefined number of allowed values. When such a property is *string* type, it is possible to make an enumeration type for those properties that lists allowed entries. The *function* property of above building feature can be altered as follows:

```
<element name="function" type="functionType"/>

<simpleType name="functionType">
 <restriction base="string">
  <enumeration value="Municipality"/>
  <enumeration value="Police office"/>
  <enumeration value="Post office"/>
  <enumeration value="Church"/>
  <enumeration value="Hospital"/>
  <enumeration value="Station"/>
  <enumeration value="Storage tank"/>
  <enumeration value="Other"/>
 </restriction>
</simpleType>
```

3.10 Structure of GML Documents

Any GML document starts with the standard XML header in which the character *encoding* and xml *version* are mentioned. Then the root element of the GML document is appeared with all *namespace* definitions used in the GML document and *schema location*. The <gml:boundedBy> element contains the bounding box of all the features in the GML document. The *srsName* attribute in the <Box> element indicates the spatial reference system where coordinates of features are based on. After the bounding box of all feature collections, each data layer (feature collection) is described alone with related bounding box elements. A GML document fragment based on the above application schema fragment is given below.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="no" ?>
<!--File: VillageModel.gml-->
<gfm:VillageTopoModel
    xmlns:gfm="http://www.itc.nl/gfm"
    xmlns:gml="http://www.opengis.net/gml"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance"
    xsi:schemaLocation="http://www.itc.nl/gfm villageModel.xsd">

<gfm:dateCreated>January 2003</gfm:dateCreated>

<gml:boundedBy>
 <gml:Box srsName="EPSG:7408">
  <gml:coordinates>
   190000,446000 193000,449000
  </gml:coordinates>
 </gml:Box>
</gml:boundedBy>
```

```
<gfm:featureMember>
 <gfm:BuildingLayer>

  <gml:boundedBy>
   <gml:Box srsName="EPSG:7408">
    <gml:coordinates>
     190000,446000 193000,449000
    </gml:coordinates>
   </gml:Box>
  </gml:boundedBy>

 <gfm:featureMember>
  <gfm:Building fid="GFM.Bld260">
  <gfm:code_id>Bld260</gfm:code_id>
  <gfm:begindate>10 Janu 2003</gfm:begindate>
  <gfm:enddate/>
  <gfm:source_type/>
  <gfm:source_description/>
  <gfm:accuracy/>
  <gfm:actuality/>
  <gfm:code_num>8885</gfm:code_num>
  <gfm:type>Tower</gfm:type>
  <gfm:function>Municipality</gfm:function>
  <gfm:height_category>3</gfm:height_category>
  <gfm:height>40</gfm:height>
  <gfm:status>Occupied</gfm:status>
  <gml:geometryProperty>
   <gml:Polygon srsName="ESPG:7408">
    <gml:outerBoundaryIs>
     <gml:LinearRing>
      <gml:coordinates>
       191100,447200 191150,447200 191150,
       447225 191000,447225 191100,447200
      </gml:coordinates>
     </gml:LinearRing>
    </gml:outerBoundaryIs>
   </gml:Polygon>
  </gml:geometryProperty>
 <gfm:heightlevel>0</gfm:heightlevel>
 <gfm:name>Enschede</gfm:name>
 </gfm:Building>
</gfm:featureMember>

 <gfm:featureMember>
  <gfm:Building fid="GFM.Bld261">
  ....
  ....

  </gfm:Building>
 </gfm:featureMember>
   .....
   .....
```

```
 </gfm:BuildingLayer>
</gfm:featureMember>

<gfm:featureMember>
 <!-- Other Feature types goes here in the same way-->

</gfm:featureMember>

</gfm:VillageTopoModel>
```

3.11 Validation of GML Documents

The correctness of a GML document has to be checked through validation. First the document must be *well formed* (that is the document should comply with XML syntax). If it is well formed the next step is to check if the GML document is *valid* according to its Schema definition. It can be performed with packages like XMLSpy, TurboXML.

3.12 Viewing GML Data

According to the principles of XML technology and GML development goals, GML should not contain presentation data. As a consequence, GML data files do not contain styling information. But there are ways to add styles to the GML features. One method is, when GML data is imported into GIS or other CAD application, styles can be added through that software. Another method is developing a viewer that can read GML data and generate cartographic view. The viewer software needs to have an interactive module to change the graphical properties of the features. It is also possible to provided a separate document that contains styling information along with GML data that viewer software can read both and generate a graphical display. But still this kind of viewers are not commonly available. There is another possibility, that is transforming GML into another XML graphic format SVG. That is the main focus of this thesis.

3.13 Conclusion

GML is an XML based standerd for describing geographic data. GML 2.0 is the latest specification in implementation level and it also supports only the OGC's simple feature model. GML models Geographic data as feature, and feature collections which are defined in application schemas using the basic constructs provided core GML schemas. A GML document that based on such a schema is a valide XML document that can be processed by any XML parser. GML provides no information about presentation and therefore data content has to be converted into another graphical format.

Chapter 4

XSLT in GML Data Visualization

4.1 Introduction

The main focus of this chapter is examining the ways of visualizing XML data and how Extensible Stylesheet Transformation (XSLT) contributes in that process. At the same time it explains how XSLT is used to transform GML data into Scalable Vector Graphics.

4.2 The Extensible Stylesheet Language (XSL)

Since XML does not use predefined tags or include formatting information, a generic XML processor that reads an XML document has no idea what is ”meant” by the document and the form which is desired to present it. Therefore, there must be an additional document that provides information on how to present or otherwise process the XML, and that is XSL.

XSL is a specification being developed within World Wide Web Consortium (W3C) for applying formatting to XML documents in a standard way. The specification defines, XSL is a language for expressing stylesheets. Stylesheets are used to describe how the content of a given structured document should be presented; that is how the source content should be styled, laid out, and paginated onto some presentation medium such as a window in a Web browser or a hand-held device, or a set of physical pages in a catalog, report, pamphlet or book [10].

The Extensible Stylesheet Language consists of three component languages which are described by three W3C recommendations. These are XSL Transformation (XSLT), XSL Formatting Objects (XSL-FO), and XML Path Language (Xpath). The XSL Transformation and XSL Formation Objects can function independently of each other [19].

4.2.1 XSL Transformation (XSLT)

XSLT is the most important part of the XSL Standards. It provides elements that define rules for how one XML document is transformed into another XML, HTML or text document. If the transformed document is in XML, it may use the markup and DTD of the original document, or it may use a completely different set of elements. XSLT can

add new elements into the output file, remove existing elements, rearrange and sort elements, test and make decisions, and a lot more through appropriate stylesheets.

The transformation can be performed in three primary ways. First, XML document and associated stylesheet both can be served to the browser (formatter), which then transforms the document as specified by the stylesheet. Otherwise a server can apply the XSLT stylesheet to the XML document and send the transformed document to the user. And the other possibility is, an XSLT processor transforms the original XML document into specified format according to the stylesheet before the document is placed on the server. Here both server and user only deal with the transformed document.
Each of these three approaches uses different software, although they all use the same XML document and XSLT stylesheet. This thesis emphasizes on the third approach that is more suitable for achieving the research objectives.

4.2.2 XSL Formatting Objects (XSL-FO)

The XSL-FO is an XML application that describes how pages will look when presented to a reader on screen or paper. It describes a rendering vocabulary capturing the semantics of formatting information for paginated presentation. An XSLT stylesheet can be used to transform XML document in semantic vocabulary into a new XML document that uses the XSL-FO presentational vocabulary [3].

4.2.3 XML Path Language (Xpath)

The Xpath is a language for referencing specific parts of an XML document, essentially for cases where it is needed to say exactly which of a document are to be transformed by XSLT. Xpath is designed to be used by both XSLT and XPointer which defines an addressing scheme for individual parts of an XML document. Xpath has an extensible string-based syntax that describe the ”location path” between parts of a document or documents using common ”path/file” file system syntax.

4.3 Tree and Nodes

A tree is a data structure composed of connected nodes beginning with a top node called the root. Therefore every well-formed XML document is a tree. The root is connected to its child nodes, each of which is connected to zero or more children of its own, and so forth. The most useful property of a tree is that each node and its children also form a tree. Thus, a tree is a hierarchical structure of trees in which each tree is built out of smaller trees. XSLT models an XML document as a tree that contains seven kinds of nodes: The root, Elements, Text, Attributes, Namespaces, Processing instructions, and Comments. The Document Type Definition (DTD) and document type declaration are specifically not included in this tree [8].

4.4 XSLT Stylesheet

An XSLT stylesheet is basically a set of rules expressed in Extensible Stylesheet Language for transforming XML documents. In case of data visualization, the role of XSLT stylesheet is to transform the XML data content into a presentation format such as HTML/XHTML, Scalable Vector Graphics (SVG), XSL-FO, text or any other structured format. However, traditional stylesheets encode information about the appearance of text and the layout of content. But in the context of GIS, XML data found in GML has

to be presented in a graphical format like Scalable Vector Graphics which is an XML based language for describing 2D graphics.

4.5 XSLT Stylesheet Structure and Elements

An XSLT stylesheet consists of a set of rules called templates. A template contains a set of template rules which has two parts; a pattern which is matched against nodes in the source tree and a template which can be instantiated to form part of the result tree. This allows a stylesheet to be applicable to a wide class of documents that have similar source tree structures.

As any other XML documents, an XSLT stylesheet begins with an *xml* declaration. The next line is either `<xsl:stylesheet>` element or `<xsl:transform>` element which are completely synonymous defines the start of stylesheet and the root element and declares the document to be an XSLT stylesheet. This element must have a *version* attribute to indicate the version of XSLT in which the stylesheet is based. The XSLT namespace attribute is given by `xmlns:xsl="http://www.w3.org/1999/XSL/Transform` (by convention "xsl" prefix is used to map the XSLT namespace). The elements that occur as a child of an `<xsl:stylesheet>` element are called top level elements. The basic structure of a stylesheet is as follows.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <!- templates go  here -- >

</xsl:stylesheet>
```

4.5.1 Templates

Template rules defined by `<xsl:template>` elements are the most important part of an XSLT stylesheet. Each `<xsl:template>` element contains rules to apply when a specified node is matched in source document. These rules describe the contribution that the matched elements make to the output document. The rules may contain both text that will appear literally in the output document and XSLT instructions that copy from the input XML document to the result.

Following template works on gfm:Building nodes in the input document and extract the content of gfm:code_id, and gfm:type child nodes and create a `<g>` element in output document with *id* and *class* attributes. It uses another template to work on gml:Polygon node.

```
<xsl:template match="//gfm:Building">
 <xsl:variable name="ID" select="gfm:code_id"/>
 <xsl:variable name="GfmType" select="gfm:type"/>
  <xsl:element name="g">
    <xsl:attribute name="id">
        <xsl:value-of select="$ID"/>
    </xsl:attribute>
    <xsl:attribute name="class">
        <xsl:value-of select="$GfmType"/>
    </xsl:attribute>
```

```
    <xsl:apply-templates select="..//gml:Polygon"/>
  </xsl:element>
</xsl:template>
```

The *match* attribute in <xsl:template> element specifies which node of the input document the template is instantiated for. It can also be used to define a template for a whole xml document. (i.e. match="/" defined the whole document). When the XSLT processor reads the input document, the root is the first node it finds and then the rules match that root node are carried out.

To get beyond the root, <xsl:apply-template> element have to be used. By including this element, the formatter is instructed to compare each child element of the matched source element against the templates in the stylesheet, and if a match is found, output the template for the matched node. The xsl:apply-template is supplied with *select* attribute to designate the children to be selected.

4.5.2 Matching Nodes

The *match* attribute of the <xsl:template> element supports a complex syntax that allows to express exactly which nodes are needed and which are not needed to match. The match patterns enable to match nodes by element name, child elements, descendants, attributes, element id, comments, processing-instruction, text, and or operator and, as well as by making simple tests on some of these items.

4.5.3 Selecting Nodes

The *select* attribute is used in xsl:apply-templates, xsl:value-of, xsl:for-each, xsl:copy-of, xsl:variable, xsl:param and xsl:sort to specify exactly which nodes are operated on. The value of this attribute is an expression written in the XPath language. The XPath language provides a means of identifying a particular element, group of elements, text fragment, or other part of an XML document.

The expressions are a superset of the match patterns mentioned above. They are not limited to specifying the children and descendants of the current node. XPath provides a number of axes that can be used to select from different parts of the tree relatives to some particular node in the tree called context node. In XSLT, the context node is normally initialized to the current node that the template matches, though there are ways to change this. The Figure 4.1 demonstrate the axes provided by XPath.

4.5.4 Named Templates

The <xsl:template> element can have a *name* attribute by which it can be explicitly invoked, even when it isn't applied directly. Such templates are called named templates. Named templates are used to repeat a template rule inside other template rules and they enable to include data from the place where the template is applied. The <xsl:call-template> element is used to call a *named template* and the value of its *name* attribute provides the name of the named template.

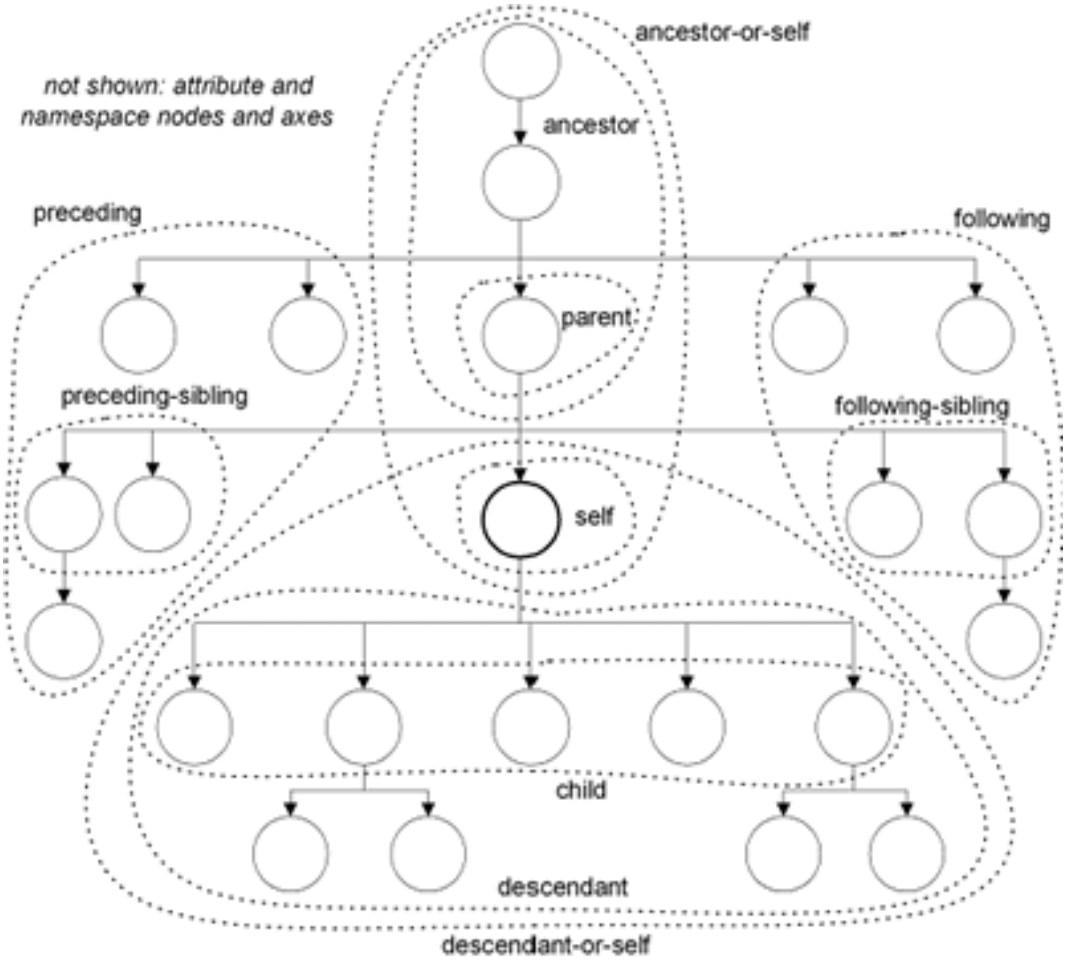

Figure 4.1: Graphical presentation of axes provided by XPath. (Source: Sun Microsystems)

4.5.5 Content of Output

In an XML document transformation, it is often necessary to include new elements, attributes, processing instruction, comments, etc. in the output document in order to conform with desired output structure. For instance, the output of a XSLT stylesheet designed to transform GML content into SVG, should comply with SVG specifications. This is accomplished with the corresponding xsl elements such as `xsl:element`, `xsl:attribute`, `xsl:processing-instruction`, `xsl:comment`, and `xsl:text` elements and attribute value templates.

Attribute value templates copy data from the input document to attribute values in the output. The <`xsl:element`> element inserts an element into the output document. The name of the inserting element is given by the value of *name* attribute and the content by the content of the <`xsl:element`> element. The <`xsl:attribute`> element defines an attribute name and value and inserts them to the elements in output document. Therefore this must appears as a child of either an <`xsl:element`> or a literal element, before any other content in those elements. When the same group of attributes are applied in many different elements, such an attribute set can be defined

as a top level element with <xsl:attribute-set> and inserted wherever necessary with <xsl:use-attribute-sets>.
The <xsl:processing-instruction> element places a processing instruction in the output document. The target of the processing instruction is specified by *name* attribute and the content of the output <xsl:processing-instruction> element become the contents of the processing instruction itself. The <xsl:comment> and <xsl:text> elements insert comments and text respectively to output document.

4.5.6 Output Methods

Most of the XSLT processors support three types of output methods XML, HTML and Text. The XSLT processor behaves differently depending on which of these output methods stylesheet uses. The output method is defined by the top-level <xsl:output> element and it's *method* attribute specifies the output method which is "xml" by default. It also has a number of attributes that allow changing the prolog, indenting, and CDATA sections in the output document as well.

The following four attributes in <xsl:output> element format the XML declaration in the output document in case the output method is "xml". The *omit-xml-declaration* attribute can have the value "yes" or "no" and when the value is "yes", no xml declaration is included in the output document. At present (January 2003), the default version of the XML declaration is 1.0 and it's the only value allowed. The *version* attribute of <xsl:output> element allows to change the version used in XML declaration accordingly in the future. The *encoding* attribute sets the encoding system in output document and it's value can be any encoding name registered with the Internet Assigned Numbers Authority. The *standalone* attribute can set the standalone attribute and the value "yes" or "no" in XML declaration of the output document.

The XSLT provides no elements for building an internal DTD subset for the output document. However, it provides two attributes of <xsl:output> element that can be used to include a DOCTYPE declaration that points to an external DTD. These are *doctype-system* and *doctype-public*. The first inserts a SYSTEM identifier for DTD and the second a PUBLIC identifier.

The *indent* attribute of <xsl:output> element has two values "yes" and "no". When the attribute has the value "yes", then the processor is allowed to insert extra white space into the output to make the document printable and more readable.

The standard XSLT does not allow inserting CDATA selections at arbitrary locations in XML documents produced by XSL transformations. However it can be specified that the text content of a particular element in input document to be placed as a CDATA section in output document by placing the name of the element whose text content should be wrapped in CDATA delimiters in the *cdata-section-elements* attribute of the <xsl:output> element. For example <xsl:output cdata-section-element ="SCRIPT"/> says that the content of the SCRIPT element in input document should be wrapped in a CDATA section in output document.

The attribute *media-type* of <xsl:output> element specifies the MIME media type of the output document. Mostly this will have the value text/xsl, but could be text/html or text/plain for the HTML or text output methods. This is important to the environment in which the XML document exists, but not so to the XML document itself.

4.6 Combining Stylesheets

XSLT provides two mechanisms to combine stylesheets; an inclusion mechanism that allows stylesheets to be combined without changing the semantics of the stylesheets being combined and an import mechanism that allows stylesheets to override each other.

4.6.1 Importing

The <xsl:import> element is a top level element whose *href* attribute provides the URI of a stylesheet to import. All <xsl:import> elements must appear before any other top-level element in the <xsl:stylesheet> root element. Rules in the imported stylesheet may conflict with rules in the importing stylesheet. If so, rules in the importing stylesheet take precedence.

4.6.2 Inclusion

The <xsl:include> is a top-level element that copies another stylesheet into the current stylesheet at the point where it occurs. Its *href* attribute provides the URI of the stylesheet to include. Unlike in above case, rules included by <xsl:include> elements have the same precedence in the including stylesheet.

4.7 Embedding Stylesheets

An XSLT stylesheet can directly be embedded in the XML document it applies to. In such a case, the <xsl:stylesheet> element must appear as a child of the XSLT document element, rather than a root element itself and have an *id* attribute giving it a unique name. This *id* attribute would appear as the value of *href* attribute in the xml-stylesheet processing instruction following the fragment identifier separator # in the XML document.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xml" href="#mystyle"?>
<Root_Element>
  <xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    id="mystyle">
    <xsl:template match="/">
     <!--Template content goes here-->
    </xsl:template>

    <!--Other templates go here-->

    <!--Don't display the style sheet itself
       or its descendants-->
    <xsl:template match="xsl:stylesheet"/>
  </xsl:stylesheet>
  <!--rest of xml data elements go here-->
   .....
   .....
</Root_Element>
```

4.8 Creating an XSLT Stylesheet

An XSLT stylesheet must be a well-formed XML document and should comply with XSLT specification which describes allowed syntax and vocabulary. The content of the stylesheet entirely depends on the input document structure (Schema) and the required output structure. The following general steps in XSLT stylesheet creation are based on the assumption that input is a GML document and output target is SVG.

1. XML declaration.

```
<?xml version="1.0" encoding="UTF-8"?>
```

2. Root element of Stylesheet including version and namespace attributes (including all namespaces found in input document).

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:gml="http://www.opengis.net/gml"
    xmlns:gfm="http://www.itc.nl/gfm"
    xmlns:xlink="http://www.w3.org/1999/xlink">
```

3. Declaring top-level elements.

```
<xsl:output method="xml" encoding="UTF-8" standalone="no"
indent="yes" omit-xml-declaration="no" media-type="text/xml"
doctype-public="-//W3C//DTD SVG 20010904//EN"
doctype-system="http://www.w3.org/TR/2001/REC-SVG-20010904
                /DTD/svg-10.dtd"/>
```

This stylesheet fragment outputs the following declarations in the output document.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
 "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg-10.dtd">
```

4. Template rule matching the root element of the input document. This may include many computations required to determine attribute values for the root element of the output document and other templates in the stylesheet.

```
<xsl:template match="/">
```

5. Processing instruction for output document should be included here. This refers to the external stylesheet.

```
<xsl:processing-instruction name="xml-stylesheet">
   href="Style.css"  type="text/css"
</xsl:processing-instruction>
```

The outcome will be as follows in the output document.

```
<?xml-stylesheet  href="Style.css"  type="text/css" ?>
```

6. Create Root element and connected attributes in output document.

```
<xsl:element name="svg">
 <xsl:attribute name="xml:space">preserve</xsl:attribute>
 <xsl:attribute name="width">100%</xsl:attribute>
 ...
 ...
```

An example for the output of this stylesheet fragment is;

```
<svg xml:space="preserve" width="100%" height="100%"
    viewBox="158000 -434000 2000 2000"
    id="svgAll" onmousemove=" showCoords(evt)"
    onzoom="resetCoords()" onscroll="resetCoords()"
    onload="startMap(evt)">
```

7. Create Top-level elements in output document.

```
<xsl:element name="script">
 <xsl:attribute name="xlink:href">External.js</xsl:attribute>
 <xsl:attribute name="type">text/ecmascript</xsl:attribute>
</xsl:element>
```

It crates the following <script> element in the output document which refers to the external Display.js script file.

```
<script xlink:href="Display_Data.js" type="text/ecmascript"/>
```

8. Access require child nodes in input document according to the order they should be occurred in output document.

```
<xsl:apply-templates select="//tdn:AdminstratiefGebied"/>
<xsl:apply-templates select="//tdn:GeografischGebied"/>
....
....
```

These selected nodes will be worked on by the matching templates and create relevant elements on the output document.

9. Apply any other template rules matching root element and specific named templates that contribute to form output document. For example in visualizing geographic data, adding grid lines , text, and other map cosmetics will make the presentation more informative. The following fragment calles a template to draw grid lines.

```
<xsl:call-template name="Grids">
 <xsl:with-parem name="Max_X" select="$Max_X"/>
 ....
 ....
</xsl:call-template>
```

10. Close root element in output document.

```
</xsl:element>
```

11. Close Template rule matching root element of the input document.

```
</xsl:template>
```

12. Declare template rules matching selected child nodes. These are the actual templates that work on selected node to output relevant elements mentioned in 8th item above.

```
<xsl:template match="tdn:AdminstratiefGebied">
<!--template rules go here -->
</xsl:template>

<xsl:template match="tdn:GeografischGebied">
<!--template rules go here -->
</xsl:template>

...
...
```

13. Declare Named templates.

```
<xsl:template name="Grids">
<!--template rules go here -->
</xsl:template>
```

14. Close root element of stylesheet.

```
</xsl:stylesheet>
```

4.9 XSLT Processors

In order to perform the transformations in an XSLT stylesheet, another software program called XSLT processor has to be employed, because the source XML document and the stylesheet both are plain text documents. XSLT processor takes as input an XML document and style sheet to convert the XML document to whatever XML, HTML or Text format. In the transformation, the processor walks through the XML document tree, looking at each node in turn, compares it with the pattern of each template rule in the style sheet. When the processor finds a node that matches a template rule's pattern, it outputs the rule's template. At present, many XSLT processors conform to XSLT 1.0 specification have been developed, and Instant SAXON, iXSLT, and XML SPy are among them.

4.10 Conclusion

XML content can be presented in many ways. Extensible Stylesheet Language Transformation is the W3C specification for reformatting XML documents. XSLT stylesheet consisting transformation rules in templates accomplishes this transformation through an XSLT processor. GML data can be transformed into SVG by this method. The structure of the XSLT stylesheet entirely depends on the structure of input and output documents.

Chapter 5

GML Data Visualization in SVG

5.1 Introduction

Scalable Vector Graphics is an XML application for two-dimensional graphics. It has many advantages over other graphics formats. This chapter describes it's advantages, capabilities and suitability for viewing geographic data. It also examines how symbols and patterns are created to suit geographic data visualization and the use of Cascading Style Sheets in visualization process.

5.2 Visualization of Geographic Data

Geographic data provides information of man-made and natural features on earth's surface such as roads, hydrology, buildings, land cover, terrain relief, and boundaries etc. In the visualization, process these features have to be graphically represented. The basic graphic elements points, lines and areas can be used to create the visual designs irrespective of the medium on which it is displayed. Point elements convey a sense of position and are the most fundamental of these three types. Lines are linear array of points and exhibits direction as well as position. Area elements exhibit extent, direction, and position and are two-dimensional array of points [18].

According to the Bertin's introduction the shape, size, orientation, color (hue), value (tone) and texture are the graphic variables that can be used to make one symbol different from another [7].

In order to make the visualization more meaningful, one should identify and analyse the data to be presented before to symbolize. Text is used in visualization to transfer the information that is not possible to symbolize.

In this research the target presentation media is a Web browser and therefor the graphical format used for the visualization must suit the Web as well as the data content (geographic data).

5.3 Evolution of SVG

Most of the graphics on the Web consist of images represented as a sequence of colored pixels. GIF (Graphics Interchange Format) , JPEG (Joint Photographic Experts Group) and PNG (Portable Network Graphics) are examples of bit-mapped graphic formats which are based on this principle. An alternative approach for sending pixel values down the Web is sending instructions for drawing features like lines or curves (vectors) and filling these shapes, which offers great advantages over pixel based formats. By now, a number of vector formats are being used on the Web; e.g. Flash, Precision Graphics Markup Language (PGML), Web Compute Graphic Metafile (WebCGM) and Portable Document Format (PDF).

Their implementation through plugins and difficulty in integration with the rest of the Web, prevented them of being used in all the places that natively supported raster graphics could be. Moreover, no single format was widely and well supported by the tools for creating Web pages, and in general there was a lack of cross-platform support and of accessibility and well internationalized solutions [4].

W3C therefore developed a new standard format for vector graphics, Scalable Vector Graphics (SVG), that matches the needs of content providers and browsers like.

5.4 Scalable Vector Graphics (SVG)

SVG is a language for describing two-dimensional graphics in XML and is being developed by the SVG working group of the World Wide Web Consortium that consists of world leading graphic software development companies.

SVG 1.0 Recommendation was issued by the W3C in September 2001 and as it describes, SVG allows for three main types of graphic objects; vector graphic shapes (e.g. paths consisting of straight lines and curves), images and text. In addition it also supports gradient fills, filters and reusable components such as symbols and markers. In SVG graphical objects can be grouped, styled, transformed and composites into previously rendered objects. Furthermore, the feature set may include nested transformations, clipping paths, alpha masks, filter effects, template objects and both procedural and declarative animation. SVG drawings can also be dynamic and interactive.

SVG is a bridge between design and programming, because unlike traditional methods of creating graphics, graphics in SVG are created through a XML based programming language and consequently integrates well with other W3C standards such as the DOM. SVG is much like a vector based graphics program, with the exception of a associated graphical program interface. Instead, vector images are created through text based commands that are formatted to comply with XML specifications [5].

SVG offers many important advantages over bitmaps or raster formats [22].

- Zoomable: Images can be magnified without sacrificing sharpness, detail of clarity.
- Text stays text: Text in SVG images remains editable and searchable. There are no font limitations and users will always see the image the same way it was created.

- Small file size: On average, SVG files are smaller than other raster Web-graphic formats and are quick to download.
- Display independence: Images are always crisp on screen and print out at the resolution of the printer.
- Color control: SVG offers a palette of 16 million colors, support for ICC color profiles, sRGB, gradients and masking.
- Interactivity and intelligence: Since SVG is XML based it offers high dynamic interactive graphics far more sophisticated than bitmapped or even Flash images. Moreover SVG images can respond to user actions with highlighting , tool tips, special effects, audio, and animation.
- Open standard: SVG is an open recommendation developed by a W3C and unlike some other graphics formats, SVG is not proprietary.
- True XML based: Since SVG is an XML grammar, it offers all the advantages of XML. Interoperability, Internationalization (Unicode support), Wide tool support, Easy manipulations through standard APIs such as DOM API. Easy transformation through XML Stylesheet Language Transformation (XSLT).

5.4.1 Creating Scalable Vector Graphics

A SVG documents can be Hand-coded using any simple text editor or sophisticated XML editors like TextPad, Vim, or XML-Spy. XML-Spy offers syntax highlighting, elements and attributes completion, validation of SVG content against the SVG DTD. This method is feasible only for simple and small documents.

Another means of creating SVG is using SVG graphic packages. Corel Draw was the first large vendor to support SVG, offering an export plugin to its Draw program. The current version 10 of Corel Draw also supports partial import of SVG documents. WebDraw is a vector graphic package that offers a three-view interface. The first is the WYSIWYG view where graphics objects can be directly drawn with the drawing toolbar, change their properties, import raster images etc. The second view is a text-editor view, with syntax highlighting, where editing can be done by hand if required. The third view is the browser view where the content of the edited document is rendered as a final document and where scripting and animation are applied. All three views are synchronized at all times. Another advanced SVG tool is Adobe Illustrator 10. It supports SVG as a native format, seamlessly reading and writing SVG and SVGZ (gzipped SVG), symbols functionality saving to SVG `<symbol>` and `<use>` output, an interactive panel to add event listeners and corresponding actions to objects, and the application of filters as native SVG filters.

A more flexible and advanced method of generating SVG is through XSLT stylesheets. As discussed in the previous chapter, XSLT is designed for XML document transformation. Since SVG is XML based, application of XSLT to generate SVG from XML data is straightforward. This research emphasizes on this approach for visualizing GML data.

5.4.2 Viewing Scalable Vector Graphics

There are two main kinds of SVG viewing tools; standalone SVG viewers and SVG-enabled browsers.

SVG Viewers The most advanced and widely deployed SVG implementation is the Adobe SVG viewer. Currently (January 2003) in version 3, the plugin is available for different platforms and works in all major browsers. This viewer supports most of the SVG specification and includes advanced SVG DOMsupport, great scripting capabilities with the inclusion of an ECMAScript engine and complete animation support.
The other main standalone SVG viewer available is the Java-based and open source Batik project. Batik 1.1 supports all static features of the SVG 1.0 specification. Batik is more than a viewer, it is a toolkit, providing an SVG-to-raster converter, server side generation through DOM and APIs, a font converter, and a printer. Because of those features, it is convenient to use Batik in a Java-based web publication framework.

SVG Browsers The Mozilla is a open source implementation, that supports a good deal of SVG specification, allows for a useful scripting facility, and is available for Windows, Macintosh, and Linux. W3C's Amaya offers a considerable subset implementation of SVG as part of its XML support. It is not just a browser, it is also an authoring tool that complies with and provides proof-of-concept for W3C standards. Using this browser, multi-XML-namespaced documents can be edited in a WYSIWYG manner. In addition to these open source browsers, Internet Explorer 6.0, Netscape Navigator 6.0 support SVG 1.0 specification through plugins for the time being.

5.5 Features in Scalable Vector Graphics

SVG is a graphic format developed to cater the requirements in the Web environment. It offers many powerful features described in SVG specification. Features that are relevant to geographic data visualization are discussed here.

5.5.1 SVG Rendering Order

SVG viewers simulate the "painters algorithm", which means that overlapping areas are "painted over" or according to transparency values, underlying monitor pixels are shining through. Painters algorithm can represent Objects up to 2.5D (which applies to terrain models) correctly. The rendering sequence is established according to object sequence within the document. Objects that are featured on top in the file are rendered first, the next objects painted over the underlying ones [16].

5.5.2 Visible Area - Viewport and Viewbox

The viewport is the area into which the SVG is to be rendered. The *width* and *height* attributes of `<svg>` element are used to define its dimension. When a `<svg>` element is nested within an SVG document, a new viewport is created, with an associated new viewport coordinate system and a new user coordinate system. The boundaries of the new viewport are defined by the value of the *x*,*y*, *width*, and *height* attributes on the nested svg element.

The *viewBox* attribute defines the area to be viewed in the viewport on screen by creating a new coordinate system. Therefore it provides a way of scaling content within an `<svg>` element and the value of the viewBox attribute takes the form of viewBox="originX , originY, width, height". The *originX* and *originY* are the coordinates at the top-left corner of the viewport. The values *width* and *height* indicate the value in user units of the viewport [26].

5.5.3 Coordinate Systems and Transformations

SVG is using a Cartesian coordinate system, the origin of which is located in the upper left corner of the canvas. The x-axis increases in the normal way and y-axis increases downwards. Because of that when using SVG for map representation, y-axis have to be inverted either by multiplying all y-values with minus one (-1) or create a group around the objects to be transformed and assign an inversion to it by a matrix operation. With the help of 3x3 matrix operation, the elementary geometrical transformations (translate, scale, rotate) can be applied individually or combined, to the groups of choice, or to the whole SVG document.

5.5.4 Basic Geometric Elements

Similar to most vector drawing packages, SVG also has some predefined basic geometric shapes that can be utilized by document authors. These basic geometrical elements are; rectangle, circle, ellipse, line, polyline, polygon, and path objects. The `<path>` element is the most important and most complex shape in SVG. It can draw open or closed line objects and polygons, consisting a list of coordinates, equally working with absolute and relative values.
The following is some code that creates a *path* element.
`<path d="M x1 y1 L x2 y2 L x3 y3 Lx4 x4.....Lxn yn"/>`.
The *d* attribute of the `<path>` allows for the description of several types of drawings. The `M x1 y1` means to move to coordinates `x1, y1`. The `L x2 y2` means draw a line from the preceding point to coordinates `x2, y2` on the drawing area. The `<path>` element uses a more different form of syntax to allow complex shapes to be created with a single element. The possible values for *d* attribute are; M = moveto, L = lineto, H = horizontal lineto, V = vertical lineto, C = curveto, S = smooth curveto, Q = quadratic bezier curveto, T = smooth quadratic bezier curveto, A = elliptical arc, and Z = closepath. Here the capital letters for absolutely positioned (e.g. M), and lower case for relative positioned (e.g. m) can be used [13].

5.5.5 Styling Graphics

The styling properties in SVG defines how the graphic elements in a SVG document are to be rendered. SVG uses those styling properties for parameters such as *fill*, *stroke colors*, *line widths* and *dash styles* which are visual in nature, parameters related with text styling such as *font-family* and *font-size* and for the parameters which impact the way that graphical elements are rendered, such as specifying *clipping paths*, *masks*, *arrowheads*, *markers* and *filter* effects. In addition to static styling information, SVG allows styling SVG elements by means of declarative animation or scripting.

SVG shares many of its styling properties with Cascading Style Sheet and XSL except for any additional SVG-specific rules explicitly mentioned in SVG specification. Styles can be assigned to SVG elements in two ways. One is using SVG's presentation attributes. For example , SVG has a *fill* property that defines how to paint the interior of a shape. There is a corresponding presentation attribute with the same name (i.e., fill) that can be used to specify a value for the *fill* property on a given element. This method offers the advantages of broad support, simplicity, restyling and convenient generation using XSLT. The other is, with Cascading Style Sheets which is discussed in section 5.6.

5.5.6 Symbols and Patterns

Symbols and patterns are widely used in geographic data visualization to represent point, linear and area features. A pattern is a repetition of symbols. In SVG, symbols and patterns can be defined as reusable entities with an *id* attribute and referenced with it whenever they are applied.

The <symbol> element is used to define graphical template objects (symbols) which can be instantiated by a <use> element. They can only be rendered through a reference by a <use> element. The <symbol> element has attributes *viewbox* and *preserveAspectRatio* which allows a symbol to scale-to-fit within a rectangular *viewport* defined by the referencing <use> element. In practice, the symbols should be created to fit a certain display scale and in a way their point of insertion is at (0,0) origin. Then while inserting these symbols, the point at (0,0) in design stage will be used by the <use> element.

The <pattern> element provides a technique to create repeating visual patterns and defines patterns within a <pattern> element nested within a <defs> element. The <pattern> element has an *id* attribute to reference the pattern when used in a *fill* or *stroke* property in the document. In addition to the necessary *id* attribute, it has x, y, *width*, and *height* attributes, which define how the pattern will be laid out on screen and *viewBox* attribute, which can be used to adjust the scaling of the pattern.

SVG can group related graphic elements together using <g> element. A <g> element can contain other <g> elements nested within it, to an arbitrary depth. A group of elements, as well as individual objects, can be named using the *id* attribute and the named groups are used for several purposes such as reusable objects and animation.

The <defs> element is used in SVG to define referenced or reusable elements. It is same as the <g> element and thus any element that can be a child of a <g> can also be a child of a <defs> and vice versa. But the elements that are descendants of <defs> elements are not rendered directly and can always be referenced by other elements.

The <use> element references another element and indicates that the graphical contents of that element is included/drawn at the given point in the document. The <use> element has x, y, *width* and *height* attributes which are used to map the graphical contents of the referenced element onto a rectangular region within the current coordinate system. However the <use> element can only reference internal elements with a given *id* [5].

The use of these elements is explained in the following example.

```
<defs>
 <symbol id="Symbol_1" viewBox="-10 -10 20 20">
  <circle xc="0" yc="0" rc="10" style="stroke:red;
        fill:yellow; stroke-width:0.5"/>
 </symbol>

 <g id="Symbol_2">
  <rect  x="-10" y="-10" width="20" height="20" fill="red"/>
 </g>

 <pattern id="Pattern_1" patternUnits="userSpaceOnUse" x="0"
```

```
        y="0" width="10" height="10" viewBox="0 0 10 10">
  <path d="M 0 2 L 1.5 2 " stroke="Blue" stroke-width="0.25"/>
 </pattern>
</defs>

<use x="45" y="10" width="5" height="5" xlink:href="#Symbol_1"/>
<use x="25" y="20" xlink:href="#Symbol_2"/>
<rect x="50" y="50" width="50" height="50" fill="url(#Pattern_1)"/>
```

5.5.7 Text in SVG

SVG allows all parameters known by cartography in generating text elements, such as *font family*, *font types*, *size*, *position*, *width*, *direction*, *inclination*. It also facilitates text aligning along path elements, all CSS rules regarding text formatting, and supports Unicode character system. Another outstanding feature in text implementation is that text elements are indexable by search engines, since the whole document is based on XML.

5.5.8 Interaction and Animation

Apart from the basic zooming and panning utilities offered by the SVG viewers, SVG content can be interactive (i.e., responsive to user-initiated events) by adding hyperlinks, animation or scripting events. SVG 1.0 supports two event categories mouse events, and state change events. Event handeling is performed through script languages such as JavaScript and EcmaScript by linking events to the functions range from switching on and off of layers, changing graphical attributes, displaying object data etc. to very complex applications [11].

The *contentScriptType* attribute on `<svg>` element specifies the default scripting language for the given document. It is also possible to specify the scripting language for each individual `<script>` element by specifying the *type* attribute on the `<script>` element. Any functions defined within `<script>` element have a global scope across the entire document.

The scripts can be embedded in SVG or referenced in several ways; short scripts can be written directly within the event-handler of an SVG element. More complex constructs may be written in `<script>` element as CDATA block or in most cases as external script files. The following example explains how to assign an event to an element to make it interactive. The script changes the color of circle from red to yellow on mouse click.

```
<!-- SVG document prolog comes here -->

<svg width="6cm" height="5cm" viewBox="0 0 600 500"
  xmlns="http://www.w3.org/2000/svg">
<script type="text/ecmascript">
 <![CDATA[
 function circle_click(evt) {
 var circle = evt.target;
 var currentColor = circle.getAttribute("fill");
 if (currentColor == 'red') circle.setAttribute("fill",'yellow');
 else circle.setAttribute("fill",'red');
 }
```

```
 ]]>
</script>

<circle onclick="circle_click(evt)" cx="300" cy="225"
   r="100" fill="red"/>
</svg>
```

An external script can be referenced by the <script> element through xlink, for example the following fragment references an external Ecmascript file named Display_Data.js .

```
<script xlink:href="Display_Data.js" type="text/ecmascript"/>
```

Animation components and their syntax in SVG is based on SMIL (Synchronized Multimedia Integration Language) and it is SMIL compatible. Amongst others, the following animation variable can be determined; start time of animation (usually relative to an event), duration and end time of an animation, repetition, number of repetition, etc. Depending on object type, various parameters can be animated such as color value, position, position along a path, rotation, scale, and many more.

5.6 Styling with Cascading Style Sheets

Cascading Style Sheet (CSS) is a styling mechanism that has been developed by W3C specially to add styles (e.g. color, font-family, font-size, spacing etc.) to Web documents. It is a breakthrough in Web design and most of the major browsers support Cascading Style Sheets. As shown in Figure 5.1, the difference between CSS and XSL stylesheets are; that CSS can be used to style HTML and XML documents while XSL is able to transform documents. Both stylesheets can be used to style XML documents.

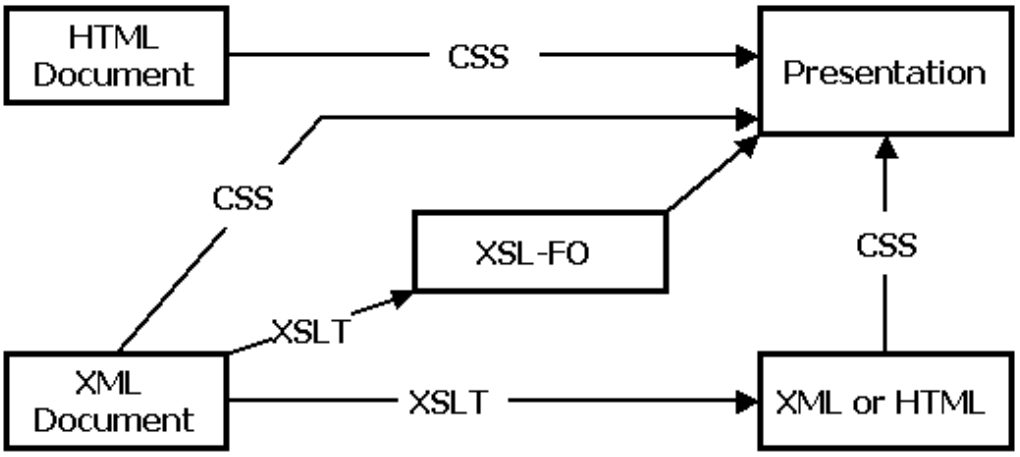

Figure 5.1: The use of CSS and XSLT in presentation process.

When an XML document is styled with a CSS, the CSS engine visits each node of the XML document hierarchy and when a node is found, it tries to match this node to a CSS rule. The matching is based on a selector expression, explained in section 5.6.2

below. However CSS cannot be used to modify the XML document structure. Therefore the rendered document is very dependent on the XML document structure. Another major difference is that CSS allows user to associate only one visual object with each element or attribute node matched. On the other hand, XSLT allows user to associate a group of visual objects to any node [2].

5.6.1 CSS in Scalable Vector Graphics

Most SVG elements represent visual objects and their visual characteristics are encoded as a set of properties. The CSS is used to set these properties. Using CSS for styling SVG content offers many advantages over styling with presentation attributes which was explained above. Some of them are; It maintains the advantage of separating presentation from content and offers ability to define named collection of properties which are applied to particular categories of elements. The use of CSS reduces the SVG file size considerably by avoiding repetition of styling properties. And also reduce the complexity in case of SVG content is embedded in to another XML, and desired to style all aspects of the compound document.

SVG implementation supports three ways of inserting CSS styles to a SVG document. External CSS, internal CSS and inline CSS styles.

External CSS

An external CSS is ideal when the style is applied to many documents. They enable to change the appearance of all documents site, just by editing a single CSS. The external CSS is referenced in stylesheet declaration, given by the syntax of

```
<?xml-stylesheet href="style.css" type="text/css"?>
```

The *href* attribute gives the name and location of external CSS. In case of XSL, the value of *type* attribute will be ”text/xsl”. The browser will read the style definitions from *style.css* file and format the document according to it. CSS can be written in any text editor, and should be saved with a *css* extension.

Internal CSS

Internal stylesheets are used when a single document has a unique style. The SVG’s `<style>` element allows stylesheet (CSS or XSL) to be embedded within the SVG content. The style data is placed within a CDATA construct as follows.

```
<defs>
 <style type="text/css">
    <! [CDATA [--------css goes here----]]>
 </style>
</defs>
```

Inline Styles

Inline styles loses many of the advantages of stylesheets by mixing content with presentation. This method should only be used when a style is to be applied to a single occurrence of an element. The *style* attribute is used to apply a particular style inline to an individual SVG element or to a group through `<g>` element. The *style* attribute allows to group several CSS styling properties within a single attribute. For example,

```
<circle cx="50" cy="50" r="100" style="stroke:#FF0000;
   stroke-width:1; fill:#FF00FF"/>

<g id="Buildings" style="stroke:red;stroke-width:0.5;fill:green">
 <path d="M0,0 L10,0 L10,10 L0,10 Z"/>
 <path d="M30,30 L50,30 L50,50 L30,50 Z"/>
</g>
```

5.6.2 Cascading Style Sheet Structure

The CSS syntax is made up of three parts; *selector*, *property* and *value*. The *selector* is normally the element/tag that is wished to define (to style), the *property* is the attribute to be changed and *value* is the content in that attribute. The property and value are separated by a colon (:) and surrounded by curly braces. If the value is multiple words, quotes are put around the value. When, more properties are specified to a element, each property has to be separated with a semi-colon.

In the following example, P is the *selector* or element to be styled; text-align, color, font-family are the *properties* and center, red and ”sans serif” are the *values* of each attribute.

```
p { text-align: center; color: red; font-family: "sans serif"}
```

The *selectors* can be grouped and in such a case each *selector* has to be separated with a comma.

```
h1, h2, h3, h4, h5 {color: green}
```

Class attribute

With the *class* attribute in SVG, different styles can be defined for the same element. The element name in selector can be omitted, when the defined style is used by many elements.

```
<path class="Shop" d="M0,0 L10,0 L10,10 L0,10 Z"/>
<path class="House" d="M30,30 L50,30 L50,50 L30,50 Z"/>
<rect class="Shop" x="100" y="100" width="20" height="10"/>
<path class="School" d="M75,75 L85,75 L85,85 L75,85 Z"/>
```

The CSS for the above SVG fragment will be;

```
 path.Shop,rect.Shop {stroke:#FF00FF; stroke-width:1; fill:#00FF00}
 path.House {stroke:#FFFFFF; stroke-width:1; fill:#000000}
 path.School {stroke:#FF0000; stroke-width:1; fill:#00FFFF}
```

as mentioned early, path and rect element names can be omitted.

```
.Shop {stroke:#FF00FF; stroke-width:1; fill:#00FF00}
.House {stroke:#FFFFFF; stroke-width:1; fill:#000000}
.School {stroke:#FF0000; stroke-width:1; fill:#00FFFF}
```

Id attribute

The *id* attribute can be defined in two ways. It can be defined to match all elements with a particular *id* or to match only one element with a particular *id*.

```
<text id="intro">This paragraph will be right-aligned.</text>
```

The following first command will match all elements with id="intro". The second will match only <text> elements with "intro" *id* value.

```
#intro {font-size:110\%; font-weight:bold;
    font-family:ariel; background-color:transparent}

text#intro {font-size:110\%; font-weight:bold;
    font-family:ariel; background-color:transparent}
```

Comments in CSS

The comments in CSS are used to describe things for the use of designer or users and to ignore by the browser. A CSS comment begins with "/*" and ends with "*/", like follows

```
/* This is a comment in CSS */
```

5.7 SVG Document Structure

Since SVG is an XML application it begins with XML declaration Then the SVG Document Type Declaration which refers the DTD which is utilized to validate the accuracy of the SVG document structure.

```
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
"http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">.
```

Third line is the <svg> element that denotes the document is a SVG. All the SVG content is placed in this element and it's attributes *width* and *height* define the size of the canvas where the content will be contained. Undefining these attributes or assigning value 100% for both *width* and *height* cause the canvas size fit the browser dimensions. Assigning minus values cause errors and zero values disables rendering.

The *x* and *y* attributes on <svg> element denotes where the canvas will be placed in the browser window. Unless specified 0,0 will be the default location. For compliance with the namespaces in XML recommendation, an SVG namespace declaration must be provided so that all SVG elements are identified as belonging to the SVG namespace.

```
<svg width="200" height="200" x="0" y="0"
  xmlns="http://www.w3.org/2000/svg">
<!---The content of the SVG document is paced here -->
</svg>
```

The content can range from an empty fragment, to a simple SVG document fragment or to a complex fragments. If an SVG document is likely to be referenced as a component of another document, it will be needed to include a *viewBox* attribute on the outermost <svg> element of the referenced document.

5.8 SVG for Presenting GML Data

According to the facts mentioned in above sections, Scalable Vector Graphics is ideally suited to visualize geographic data in GML due to many reasons. Mainly it is an XML based vector format developed by W3C and has gained world wide support. Secondly, SVG presentation can be generated from GML documents through XSLT stylesheet and viewed in different viewers and browsers. And finally, the advanced and sophisticated features, graphic styling capabilities and use of script languages for interactive client features in SVG, greatly help to make the presentation attractive, more expressive, more informative and have a visual impact on the person viewing it which are important aspects in geographic data visualization.

5.9 Conclusion

SVG is a promising technology for 2D graphics in the Web. It is a perfect tool to publish geographic data in the form of maps. Using external CSS, the appearance of a SVG document can be changed easily. SVG support for script languages and ability to embed in HTML pages offer many tools required to generate interactive maps.

Chapter 6

Implementation

6.1 Introduction

In previous chapters, the XML based open standards that are used to model geographic information (creating Digital Landscape Model) and transforming them into visualizations (Creating Digital Cartographic Model) were discussed. This chapter covers the implementation phase of GML data transforming into SVG through XSLT stylesheets. First the method of implementation is discussed ,then the input GML data set and its application schema is reviewed. Thereafter creating XSLT stylesheet, external symbol/pattern library, external CSS and adding interactivity is discussed. Finally the processing of an XSLT stylesheet through an XSLT processor is described. Problems and limitations encountered during the implementation are discussed.

6.2 Methodology for Implementation

One major target in implementation is to separate sharable information. The symbol and pattern definitions, Cascading Style Sheets and Scripts are kept as external documents to enable using them by more XSLT stylesheets and SVG documents according to their need. This approach provides easy editing and alteration of documents, reduces the complexity of XSLT stylesheet and the size of SVG document as well. The GML data has to be validated to confirm the compliance with its application schema. The structure of the XSLT stylesheet is entirely based on the GML application schema and required SVG output. The output SVG document has to be compliance with SVG 1.0 specification which is supported by all available SVG viewers at the time of implementation . XMLSpy was used as XML editor for developing the XSLT stylesheet. The Figure 6.1 shows the outline of the implementation method. The following sections give details how each step was implemented.

6.3 The GML Data

The GML data used in this research has been based on the new TOP10vector data model of the Dutch Topographic Service. The TOP10vector is a digital dataset topographical information of the Netherlands territory at a scale of 1:10,000. The generation of GML data from TOP10vector has been carried out by the Technical University of Delft(TUD). The TUD has developed two GML application schemas based on GML 2.0 specifications; one with member restrictions and the other without member restrictions. But this difference is visible only in the structure of the classes and cannot be

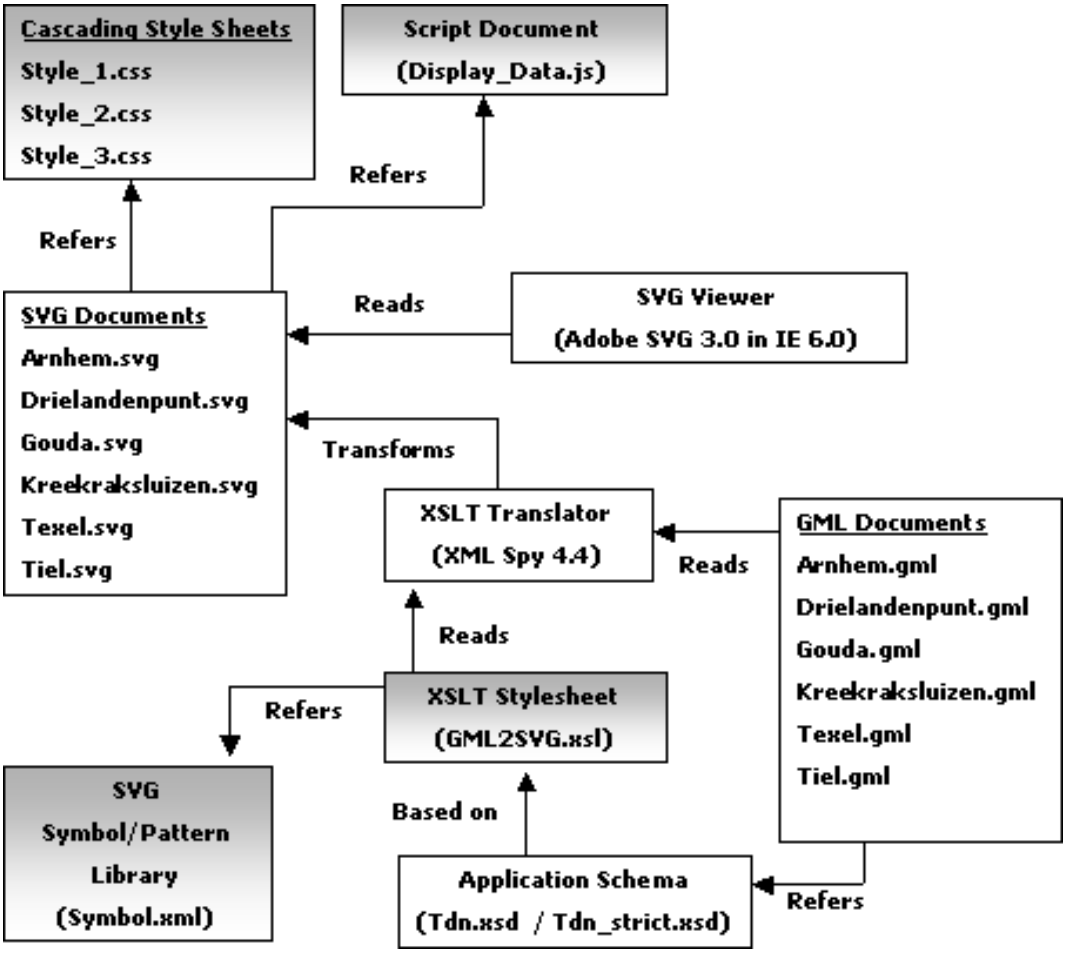

Figure 6.1: Implementation of GML data visualization in SVG.

observed in the final instances (GML data). As TUD states, in order to have explicit control over the members of a set, the GML rules require explicit *FeatureAssociation classes*, which is an complicating factor. This schema introduces restrictions to the values of the attributes. The differences in data modeling are clearly visible in UML application models shown in Appendix C and D.

According to the data model, there are 10 feature types ; road section (Weg Deel), railroad section (SpoorbaanDeel), water part (WaterDeel), building (Bebouwing), terrain (Terrein), construction element (InrichtingElement), functional area (FunctioneelGebied), administrative area (AdministratiefGebied), geographic area (GeoGrafischGebied) and conservation area (Beheersgebied). In the GML schema, their geometry property has been defined as *geometryProperty* and therefore their geometry type (class) could be any of the geometry types in GML.

The GML data set consists of six data sets for Arnhem, Drielandenpunt, Gouda, Kreekraksluizen, Texel and Tiel and each area has two GML data sets based on tdn.xsd and tdn_strict.xsd schemas. All these data sets were validated against corresponding application schema using XMLSpy version 4.0.

6.4 XSLT Stylesheet Generation

The general procedure explained in section 4.8 was followed in XSLT stylesheet generation. The rendering order in SVG leads to some complications in geographic data visualization. Therefore rendering order of feature types has to be decided at the beginning of stylesheet design. The steps in XSLT stylesheet are as follows.

1. XML declaration.

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

2. The Stylesheet declaration.

```
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:gml="http://www.opengis.net/gml"
   xmlns:tdn="http://www.gdmc.nl/tdn"
   xmlns:xlink="http://www.w3.org/1999/xlink">
```

3. Output element, that describes output method and other attributes related to output document.

```
<xsl:output method="xml" encoding="iso-8859-1"
 standalone="no" indent="yes" omit-xml-declaration="no"
 media-type="text/xml"
 doctype-public="-//W3C//DTD SVG 20010904//EN"
 doctype-system="http://www.w3.org/TR/2001/REC-SVG-20010904/
  DTD/svg-10.dtd"/>
```

4. Template that match the root element of input document contains the following template rules.

```
 <xsl:template match="/">
```

4.1 Processing instruction refers to external style sheet.

```
<xsl:processing-instruction name="xml-stylesheet">
  href="Style_2.css"  type="text/css"
</xsl:processing-instruction>
```

4.2 Determine *viewport* and *viewBox* attributes. In order to do that, the stylesheet reads `tdn:Top10Themas/gml:boundedBy/gml:Box/gml:coordinates` node in source document to get the coordinates of total area covered by Top10Themas feature collection type. With that information, stylesheet determines the following attribute values which are used in creating SVG element of the result document.

```
Width="Maximum 675 pixels or number of pixels proportionate
to Height"
Height="Maximum 675 pixels or number of pixels proportionate
to width"
viewBox="Minimum_x Minimum_y (Maximum_x-Minimum_x)
        (Maximum_Y - Minimum_y)"
```

Another variable *Scale* is calculated to use in scaling symbols.

4.3 Create SVG element that contains all graphic elements, and the connected attributes and values. In addition to the *xml:space, width, height,* and *viewBox* attributes, *id, onmousemove, onzoom, onscroll,* and *onload* attributes were added to enable event handling. The following is the corresponding stylesheet fragment.

```
<xsl:element name="svg">
 <xsl:attribute name="xml:space">preserve</xsl:attribute>
 <xsl:attribute name="width">
    <xsl:value-of select="$Width"/>
 </xsl:attribute>
 <xsl:attribute name="height">
    <xsl:value-of select="$Height"/>
 </xsl:attribute>
 <xsl:attribute name="viewBox">
  <xsl:value-of select="$Min_x"/>
  <xsl:text >  </xsl:text>
  <xsl:value-of select="-$Max_y "/>
  <xsl:text >  </xsl:text>
  <xsl:value-of select="round($XD)"/>
  <xsl:text >  </xsl:text>
  <xsl:value-of select="round($YD)"/>
 </xsl:attribute>
 <xsl:attribute name="id">svgAll</xsl:attribute>
 <xsl:attribute name="onmousemove">
    showCoords(evt)</xsl:attribute>
 <xsl:attribute name="onzoom">
    resetCoords()</xsl:attribute>
 <xsl:attribute name="onscroll">
    resetCoords()</xsl:attribute>
 <xsl:attribute name="onload">
    startMap(evt)</xsl:attribute>
```

4.4 Create Script element that refers to script type and location.

```
<xsl:element name="script">
 <xsl:attribute name="xlink:href">
    Display_Data.js</xsl:attribute>
 <xsl:attribute name="type">
    text/ecmascript</xsl:attribute>
</xsl:element>
```

4.5 Extract Symbol and Pattern definitions from the external library (Symbol.xml) document.

```
<xsl:copy-of select="document('Symbol.xml')/defs"/>
```

4.6 Determine the number of height levels to be rendered in GML source file. Then, for each height level (zero onward), stylesheet reads the input document for following features according to the same rendering order and create corresponding SVG graphic elements and attributes in result document.

```
<!--  Terrein  -->
<xsl:element name="g">
<xsl:attribute name="id">
<xsl:value-of select="concat('Terrein_',$L)"/>
```

```
</xsl:attribute>
<xsl:apply-templates select="//tdn:Terrein[tdn:hoogteniveau=$L]"/>
</xsl:element>

<!--  SpoorbaanDeel  -->

<!--  WaterDeel  -->

<!--  WegDeel  -->

<!--  Bebouwing  -->

<!--  InrichtingsElement  -->
```

Thereafter render the feature types that has no height_Level attribute according to the following order.

```
<!--  AdministratiefGebied  -->
<xsl:element name="g">
<xsl:attribute name="id">AdministratiefGebied</xsl:attribute>
<xsl:apply-templates select="//tdn:AdministratiefGebied[@fid]"/>
</xsl:element>

<!--  GeografischGebied  -->

<!--  FunctioneelGebied  -->
```

4.7 Create SVG graphic elements that renders grid lines. Even though grid interval is 500 in the prototype, it can be changed through a single variable in the XSLT stylesheet.

4.8 Create SVG text elements that renders coordinates and feature types through event handeling and insert some map cosmetics like north arrow, scale bar and data source.

4.9 Annotate the features in AdministratiefGebied, GeografischGebied, and FunctioneelGebied feature classes. Stylesheet computes the text size according to the data display scale, and text is placed at the centroide of the area concern.

5. Close SVG element.

```
</xsl:element>
```

6. Close Template that match source root element.

```
</xsl:template>
```

7. Templates that apply to render features in step 4.6 as well as other supporting templates and named templates have to be defined individually outside the main template. For example Terrain feature type rendering procedure is explained below. The <xsl:apply-template> elements in step 4.6 apply corresponding templates developed similar to the following one.

```
<xsl:template match="tdn:Terrein">
 <xsl:variable name="ID" select="tdn:top10_id"/>
 <xsl:variable name="TdnType" select="tdn:landgebruik"/>
 <xsl:variable name="Type">
```

```
 <xsl:call-template name="StyleMake">
 <xsl:with-param name="Class" select="normalize-space($TdnType)"/>
 <xsl:with-param name="Pre" select="'Ter-'"/>
 </xsl:call-template>
 </xsl:variable>

 <xsl:variable name="Area_out">
 <xsl:variable name="clist"
   select="..//gml:Polygon/gml:outerBoundaryIs//gml:coordinates"/>
 <xsl:variable name="nlist1" select="normalize-space($clist)"/>
 <xsl:call-template name="Area_Cal">
 <xsl:with-param name="nlist" select="$nlist1"/>
 </xsl:call-template>
 </xsl:variable>

 <xsl:for-each
  select="..//gml:Polygon/gml:innerBoundaryIs//gml:coordinates ">
 <xsl:variable name="Area_in">
  <xsl:call-template name="Area_Cal">
   <xsl:with-param name="nlist" select="normalize-space(.)"/>
  </xsl:call-template>
 </xsl:variable>
 <xsl:value-of select="$Area_out - $Area_in"/>
 </xsl:for-each>

 <xsl:element name="g">
  <xsl:attribute name="id"><xsl:value-of select="$ID"/>
  </xsl:attribute>
  <xsl:attribute name="class"><xsl:value-of select="$Type"/>
  </xsl:attribute>
  <xsl:attribute name="onmouseover">
   showType('<xsl:value-of select="$TdnType"/>')
   ;changeColorOver(evt)
  </xsl:attribute>
  <xsl:attribute name="onmouseout">
   emptyType();changeColorOut(evt)
  </xsl:attribute>
  <xsl:attribute name="onclick">
   idClick('Top10 ID :<xsl:value-of select="$ID"/>Extent:
   <xsl:value-of select="round($Area_out)"/> sq.m.')
  </xsl:attribute>
  <xsl:apply-templates select="..//gml:Polygon"/>
 </xsl:element>
</xsl:template>
```

This template uses many other templates for different purposes. First it reads the content of <tdn:top10_id> and <tdn:landgebruik> elements which are two child elements of the <tdn:Terrein>. The content of <tdn:top10_id> element is assigned as the *id* attribute in corresponding SVG <g> element and the other is used to create the relevant *class* attribute of the <g> element.

To distinguish different sub types in terrain feature class, <tdn:landgebruik> (land use) property has been used and StyleMake template creates unique class name for the features in terrain type but having different land use types. It adds

”Ter-” prefix for the class name of each terrain type feature and rest of the name is taken form the content of <tdn:landgebruik> element.

It also calculates the area of each terrain feature that will be included as a extra information to be displayed in SVG document. And finally the template creates SVG <g> element in the result document that contains *id*, *class*, *onmouseover*, *onmouseout*, *onclick* attributes and <path> element that renders the feature graphically in SVG. The *onmouseover* , *onmouseout* and *onclick* attributes add events to the SVG document which are explained in section 6.7.

Even though GML application schema defines geometryProperty as the geometry property of all features, in reality one feature type may hardly be represented by all geometry classes. For instance; Terrain features are represented by polygons and not by line features or points. Roads may be represented by polygons and lines but not with points. According to the suitable geometry classes, the <path> element will be created by the template.

In case of transforming GML features in InrichtingsElement class, in to SVG, most of them are represented by point symbols or lines. Then SVG <use> element with *id*, *translate* and *scale* attributes is used to render appropriate symbol at the correct location at the suitable scale.

8. After including all templates, finally the XSLT stylesheet element has to be closed.

```
</xsl:stylesheet>
```

6.5 Symbol and Pattern Library

This is an XML document that contains all symbol and pattern definitions. All these symbols have been created, in such a way, that their original size will fit 1:10000 display scale, and their insertion point is at the origin.
Symbols are defined using svg <g> element and *id* attribute that is used to identify groups.

Symbols are used to represent point data and patterns are used to fill area symbols. In the TDN GML data only, *tdn:InrichtingsElement* feature type can be represented by symbols. But this feature class contains about 45 different sub types given by the content of <tdn:typeInrichtingsElement> element. Out of these 45 types, only seven types that mainly appear in provided data sets were symbolized. The symbol names (*id* attribute) have been assigned from the allowed names of <tdn:typeInrichtingsElement> element which is used in tern by the stylesheet while creating svg <use> element.
The appearance of the symbols defined here are not the same as the symbols found in Dutch Topographic maps. The symbol having id=”Wegwijzer” has been defined as an animated one to show the possibilities.
Symbols and Patterns also can be created with pre-defined geometries. The following fragment of the symbol library clearly shows the possibilite.

```
<?xml version="1.0" encoding="UTF-8"?>
<defs xmlns:xlink="http://www.w3.org/1999/xlink">

<g id="Hoogspanningsmast">
 <path style="stroke:cyan;fill:blue;stroke-width:0.25"
 d="M-2.00 2.00 L2.00 2.00 L2.00 -2.00 L-2.00 -2.00
 L-2.00 2.00 Z"/>
</g>
```

```
<g id="Wegwijzer" stroke="#ff6600" fill="red"
  stroke-width="0.25">
<path d="M-2.00 2.00 L2.00 2.00 L2.00 -2.00 L-2.00
 -2.00 L-2.00 2.00 Z" />
<animate attributeName="fill" values="red;blue"
 dur="1s" repeatCount="indefinite"/>
</g>

<g id="LoofbosGeometry">
<path d="M0,-1.6L-0.2,-0.8a0.6 0.6 0 1 0 -0.8
  0.86a0.55 0.55 0 0 0 0.45 0.89a0.56 0.56 0 0 0
  1.1 -0.0a0.55 0.55 0 0 0 0.45 -0.89a0.6 0.6 0
  1 0 -0.8 -0.86L0,-1.6z"/>
</g>

<g id="Loofbos" transform="matrix(1 0 0 -1 0 0),
  scale(2)" style="stroke:rgb(107,142,35)
  ;fill:rgb(34,139,34);stroke-width:0.087">
<use xlink:href="#LoofbosGeometry" x="0" y="1.6"/>

</g>

<pattern id="LoofbosPattern" patternUnits="userSpaceOnUse"
  x="0" y="0" width="30" height="30">
<rect x="0" y="0" width="30" height="30" fill="#CCFFCC"/>
<use x="4" y="20" xlink:href="#Loofbos" />
<use x="5" y="10" xlink:href="#Loofbos" />
<use x="7" y="29" xlink:href="#Loofbos" />
<use x="14" y="18" xlink:href="#Loofbos" />
<use x="15" y="8" xlink:href="#Loofbos" />
<use x="18" y="29" xlink:href="#Loofbos" />
<use x="25" y="10" xlink:href="#Loofbos" />
<use x="26" y="28" xlink:href="#Loofbos" />
<use x="25" y="20" xlink:href="#Loofbos" />
</pattern>


</defs>
```

The graphical output of LoofbosPattern is shown in Figure 6.2.
In the same way, any kind of symbol or pattern can be defined.

6.6 Cascading Style Sheet Creation

As elaborated in chapter 5 of this thesis, stylesheet declaration

```
<?xml-stylesheet href="style.css" type="text/css"?>
```

refers to the Cascading Style Sheet that contains style information. The style applicable to each element in the SVG document is specified by its *class* attribute. In the Cascading Style Sheet, the *class* attribute is used as the selector. Following CSS fragment defines the styles apply to Terrain feature types.

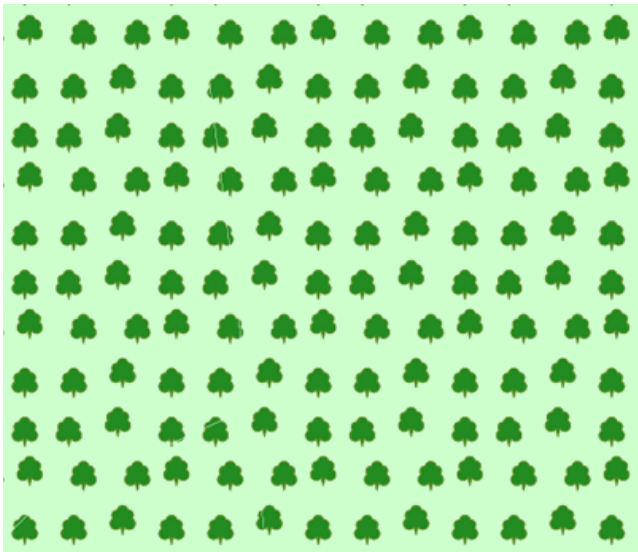

Figure 6.2: Graphical output of LoofbosPattern Pattern in SVG.

```
/*   Terrein :landgebruik  */
.Ter-Loofbos{stroke:#ccffcc;stroke-width:0.1;fill:url(#TreePattern)}
.Ter-Naaldbos{stroke:#ccffcc;stroke-width:0.1;fill:#ccffbb}
.Ter-Gemengdbos{stroke:#ccffcc;stroke-width:0.1;fill:#ccff33}
.Ter-Griend{stroke:#ccffcc;stroke-width:0.05;fill:#ccff44}
.Ter-Populieren{stroke:#ccffcc;stroke-width:0.05;fill:#ccff55}
.Ter-Heide{stroke:#ccffcc;stroke-width:0.05;fill:#ccff66}
.Ter-Zand{stroke:#ccffcc;stroke-width:0.05;fill:#ccff77}
.Ter-Akkerland{stroke:#ccffcc;stroke-width:0.05;fill:#ccff88}
.Ter-Grasland{stroke:#ccffcc;stroke-width:0.05;fill:url(#GrassPattern)}
.Ter-Boomgaard{stroke:#ccffcc;stroke-width:0.05;fill:#ccff15}
.Ter-Fruitkwekerij{stroke:#ccffcc;stroke-width:0.05;fill:#ccff26}
.Ter-Boomkwekerij{stroke:#ccffcc;stroke-width:0.05;fill:#ccff37}
.Ter-Dodenakker{stroke:#ccffcc;stroke-width:0.05;fill:#ccff48}
.Ter-Plaveisel{stroke:#ccffcc;stroke-width:0.05;fill:#ccff59}
.Ter-Overig{stroke:#ccffcc;stroke-width:0.05;fill:#ccff70}
```

Some of the above features have been filled with previously defined patterns. The same method can be extended to style all other feature types as well. A SVG document can be styled differently by applying different CSS stylesheets. In order to achieve that, the stylesheet name in the stylesheet declaration in the SVG document has to be changed and load the new SVG document to viewer. For example Arnhem.svg document was visualized using three Cascading Style Sheets Style_1.css, Style_2.css and Style_3.css and the corresponding visualizations are shown in Appendix E.

6.7 Adding Interactivity

Interactivity can be added to SVG elements through scripts, and in order to trigger the execution of scripts, event-handlers are used. Some of the most important SVG event-handlers are; *onmouseover*; executed when the pointer moves above an object, *onmouseout*; executed when the pointer leaves an object, *onclick*; executed when the

user clicks on an object, *onload*; triggered after a SVG document was loaded, *onzoom*; triggered after the user changes zoom factor, *onscroll*; triggered after the user scrolls within a SVG graphics and *onresize*; triggered when a document-view was resized.

During the transformation, the above XSLT stylesheet creates event attributes on <svg> document element as well as in svg <path> elements. These event attributes define what happens when the specified event is triggered. Action to be followed by the event is defined as a method in the script file. The location path (Display_Data.js) and script type is specified by <script> element as follows.

```
<script xlink:href="Display_Data.js" type="text/ecmascript" />
```

For example; the stylesheet outputs `onmousemove="showCoords(evt)"`, `onzoom="resetCoords()"`, `onscroll="resetCoords()"` and `onload="startMap (evt)"` event attributes and their values on the <svg> element. The event attribute *onmousemove* triggers the showCoords(evt) function whenever the mouse pointer is moved on the <svg> element content. Likewise all other attributes also triggers their respective functions according to the event-handler. These functions are defined in Display_Data.js ecmascript file.

The other occasion is the use of event attributes on svg graphic elements. For example, a SVG <path> element that represent a Terrain feature has the following event attributes and values.

```
onmouseover="showType('Overig');changeColorOver(evt)"
onmouseout="emptyType();changeColorOut(evt)"
onclick="idClick('Top10 ID : 5100298  Extent :445 sq.m.')"
```

In this case when mouse is moved over the specific graphic element, it will trigger both showType() and changeColorOver() functions. Following is a fragment of the ecmascript file.

```
var svgObjType;
var svgDoc;
function startMap(evt) {
    // Retrieve the SVG document object

    svgDoc = evt.getTarget().getOwnerDocument();

    //get reference to text elements

    svgObjType = svgDoc.getElementById("varType");
    svgObjType = svgObjType.getFirstChild();
}
```

The viewer reads the startMap(evt) method after loading SVG document because of the *onload* event attribute in <svg> element. In order to change SVG elements per script one need to first get a reference to the SVG document and elements. The two global variables *svgDoc* and *svgObjType* are used as references to the SVG document and a text object that needed to change. This text object has the ID ”varType” and is referenced by the method *getElementById()*. In order to change the text-string itself, the method *getFirstChild()* have to be used.

```
function showType(type) {
    //change text-Value
    svgObjType.setData(type);
}
```

This function changes the text-value itself using setData() method, triggered by the mouse-over event. The function changeColorOver extract the class attribute value from the target through getAttribute() method and substitute it with another value highlightFill which is a fill style defined in CSS.

```
function changeColorOver(evt) {
    var data=evt.getTarget();
    fillStyle=data.getAttribute("class");
    data.setAttribute("class","highlightFill");
}
```

The functions emptyType() and changeColorOut() triggered by the onmouseout event, reset the text string at ”varTyep” and class attribute on target element by setAttribute() method.

```
function changeColorOut(evt) {
    var data=evt.getTarget();
    data.setAttribute("class",fillStyle);
}

function emptyType() {
    //empty text-String
    svgObjType.setData("");
}

function idClick(text) {
    //show an alert message
    alert(text);
}
```

The idClick function is triggered by a mouse-click event and sends a JavaScript alert message with the message-text given by the functions parameter.

In addition to these functions, the implementation phase has one more function to determine the mouse pointer coordinates and display them at the same location on the screen over zooming and scrolling.

Another aspect in implementation is embedding SVG in HTML. Through this layer on/off facility has been added. Even though the same interactivity can be achieved through SVG and ECMAScript, that could be more complicated than this.

6.8 XSLT Processing and Viewing SVG

Even though many XSLT processors are available at free sources, XMLSpy was used because of its editing facilities and convenience in processing. In XMLSpy, all GML source files and the XSLT stylesheet can be included in a project and specify the output files location and file name extension. By simply executing XSLT transformation, all files in that project can be transformed at once. The project setup is shown in Figure 6.3.

Whatever the technique used to create SVG, still a SVG viewer is needed to render the SVG images onscreen. Among many SVG viewers, Adobe and Batik SVG viewers are the most widely used SVG viewers that comes closer to complete implementation of SVG 1.0 Recommendation. In this research, Adobe SVG viewer 3.0 plug-in in Internet Explora 6.0 has been used.

The SVG output was embedded in HTML to make the presentation more interactive and manageable. SVG outputs were embedded to a HTML file (Display.html) using `<EMBED>` element and SRC attribute as follows.

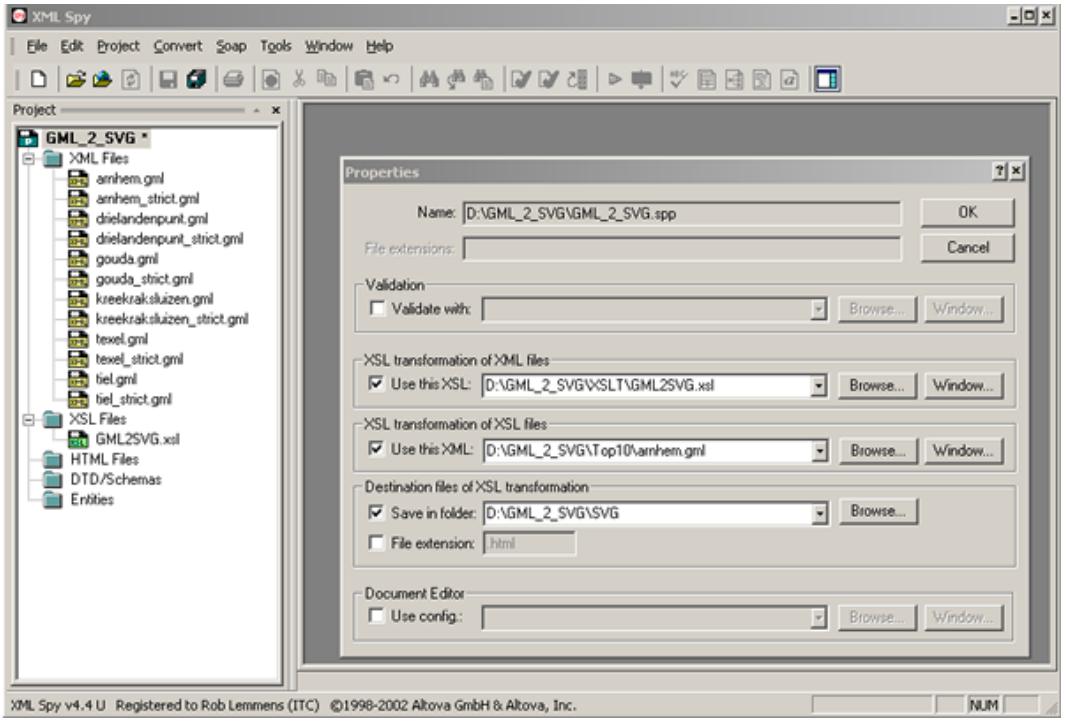

Figure 6.3: XSLT stylesheet processing in XMLSpy.

```
<EMBED WIDTH="675" HEIGHT="675" SRC="Arnhem.svg" NAME="gml_svg_map"
type="image/svg-xml" align="left"/>
```

Through this, Layer on/off facility was added and the SVG image was restricted to 675x675 pixels on the screen. A SVG document (Texel.svg), embedded in a HTML page is shown in Figure 6.4.

The final outcome was checked against the Topographic Maps.

6.9 Problems in Implementation

One of the main problem in implementation was determining rendering order of the features. As explained above this was succeeded using the height level property in GML data. Another requirement was keeping symbol and Pattern definitions in an external file. It could not achieved due to the limitations in <use> element, because it cannot refer definitions outside the svg document. In visualizing, features like roads, it is necessary to show their outer boundary for highlighting continuation and priority. For this, a solution could not be found. Even though the use of external Cascading Style Sheet offers many advantages it leads to delay of the rendering as well.

Another problem was linking original GML data with the corresponding SVG graphic element for extracting more information. One way of doing is including all the gml data as metadata in the svg file. This was not a practicable solution. If the file was generated on server side (e.g gml fetched by a WFS), then it is possible to generate the svg paths from the gml geometry, while preserving the attribute data. But this not applicable for standalone systems. Another way is to embed the svg and the gml

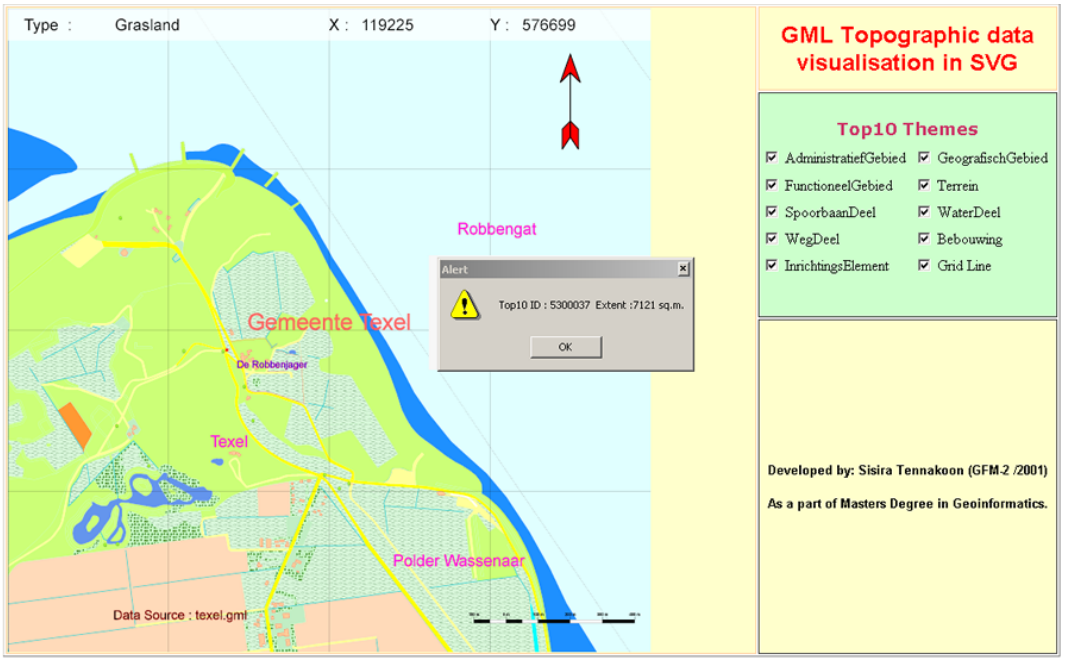

Figure 6.4: A SVG document (Texel.svg) embedded in HTML.

(using xml tag) in a html page to read both DOMS. Once both documents are loaded, the required elements have to be accessed through DOM methods. This is a complicated process and could not be achieved due to the time constrain.

Another problem is loading time, because of the size of SVG document. This can be reduced considerably by compressing SVG document into gzip format.

6.10 Conclusion

GML data can be transformed into SVG through XSLT and viewed in Web browsers. In SVG, Symbols and Patterns can be created using pre defined graphics and the SVG document must contain all symbols and patterns referenced by <use> element. The Cascading Style Sheets, Script languages and HTML can be used to enhance the effect of the SVG visualization. Some limitations in SVG specification, and viewers lead to difficulties in geographic data visualization.

Chapter 7

Conclusion and Recommendations

According to the research objectives, this research is aimed at visualizing geographic data in GML using Extensible Stylesheet Language Transformation. All research questions have been answered in 3rd, 4th and 5th chapters. The proposed method for GML data visualization has been implemented in chapter 6 through developing an XSLT stylesheet, SVG pattern library and CSS.

7.1 Conclusions

GML is an open standerd based on XML technology to describe geographic data for transportation and store in Internet environment. Even though it is still in developing mode, the world wide adoption and response to it indicates the success and need for such a standerd. The limitations in GML 2.0 such as describing topology, 3D geometries, arcs and surfaces are expected to be resolved in GML 3.0. The adoption of GML to describe geographic data is a solution for the problems in data sharing among heterogenous users.

XML data can be presented in many ways. To present XML based geographic data in a web browser, they have to be transformed into a graphical format (not text) that web browsers can interpret. The Scalable Vector Graphics is an XML based vector graphic format that web browsers can interpret either with or without the support of a viewer plugin. Extensible Stylesheet Language Transformation is an W3C recommendation to transform one XML document to another. Since GML being an XML data format, XSLT can transform GML into SVG. These transformation rules are specified in XSLT stylesheets. Through the stylesheets many additional information can be generated from base data in GML document and added them to the output document. For example; extent of the area features, length of linear features, placing names of streets and places etc. Apart from that, scaling the graphic symbols according to display scale also can be done.

Technically, software can be developed to transform GML data into any graphic format, but the conversion of GML to SVG through XSLT has advantages. It makes use of XML, the feature base-technology for the Internet, supported by the world leading software developers. One can relatively easy generate different representations because XML accommodates the separation of data content from data presentation. Furthermore, if GML data use a predefined application schema, one can represent any

data set using the same stylesheet.

The use of external Cascading Style Sheets provide a powerful tool to change the SVG display via the reference to the stylesheet. The scripting languages add interactivity to the SVG display which is an important aspect in a digital environment. They also provide methods to manipule the SVG document through the Document Object Model. This facility can be used to change SVG elements dynamically. Through embedding SVG documents in HTML, the presentation can be controlled more easily.

Even though open standard browsers support SVG natively they are not supporting SVG specifications fully. At the moment Adobe SVG viewer 3.0 is the most convenient and sophisticated SVG plugin available that supports widely used web browsers. But it also does not support SVG 1.0 specification fully. The fast implementations of SVG specifications in browsers and viewers will make the use of SVG in geographic data visualization more common.

7.2 Recommendations

This thesis is focused on visualizing GML data in a standalone system. In reality GML is mostly used in network environment. Therefore another important aspect to be considered is, how GML data can be styled at the server side. Even though basically the same XSLT stylesheet can be used, that process involves client side interference too.

At the moment no SVG viewer or browser supports SVG specification fully or to the same level. Different viewers and browsers have implemented SVG specifications at different levels and therefore the same results cannot be expected from all of them.

An attachment to Chapter 7, Section 7.1 (Conclusions)

In visualization it is required to change the display order of the feature layers for many reasons. Normally in other graphic packages this can be achieved dynamically through the interface. But in SVG, to change the display order of features, their order in the document itself has to be changed. It cannot be performed dynamically through the viewer interface at the moment. It is a useful feature that should be supported by SVG viewers in the future.

As mentioned in the section 6.9, point symbols and fill patterns used in the SVG document have to be defined in the SVG document itself. The *href* attribute in <use> element which refers to the symbols, and the *fill* attribute which refers to fill patterns cannot locate symbols or patterns residing outside the SVG document. Therefore all the time they have to be defined in the document.

In geographic data visualization, sometimes it's required to show the outer boundary of features to highlight their continuation or priority (for instance road under/over passes). This kind of problems in visualization cannot be solved easily in SVG. The topology of the feature should be taken into consideration.

Another important aspect in data visualization is ability to interact with the display to get information of the features. This needs to have a link with the source data. The process of visualizing GML data in SVG involves data reformatting using XSLT. Once the SVG document is created, no link remains with the original source data. Therefore to query the GML content through the SVG visualization, features in SVG have to be linked to GML features through DOM methods and script languages. This can be done because each feature has it's own unique identifier.

Bibliography

[1] A. Cuthbert, R. Lake, et al. - Open GIS Consortium. Geography Markup Language (GML)v 1.0, OpenGIS implementation specification. http://www.opengis.net/gml/GML.html, May 2000. Accessed on 2002-09-16.

[2] B. Bos - World Wide Web Consortium. Cascading Style Sheets home page. http://www.w3.org/Style/CSS/, October 2002. Accessed on 2002-10-30.

[3] M. Birbeck, J. Diamond, et al. *Pofessional XML 2nd Edition*. Wrox Press Ltd., Arden House, 1102 Warwick Road, Acocks Green,Birmingham, B27 6BH, UK2001, second edition edition, 2001.

[4] C. Lilley. Graphics Activity Statement. http://www.w3.org/Graphics/Activity, September 2002. Accessed on 2002-10-04.

[5] K. Cagle. *SVG Programming: The Graphical Web*. Springer-Verlag GmbH & Co. KG, Trergartenstr. 17, 69112 Heidelberg, Germany., 2002.

[6] D. C. Fallside - World Wide Web Consortium. XML Schema Part 0: Primer, W3C Recommendation. http://www.w3.org/TR/xmlschema-0/, May 2001. Accessed on 2002-10-03.

[7] Rolf A. de By, R. A. Knippers, et al. *Principles of Geographic Information Systems*. The International Institute for Aerospace Survey and Earth Sciences (ITC), Hengelosestraat 99, P.O.Box 6, 7500 AA Enschede, The Nethrelands, second edition edition, 2001.

[8] E.R. Harold. The XML Bible, 2nd Edition. http://www.ibiblio.org/xml/books/bible2/index.html, November 2001. Accessed on 2002-09-26.

[9] P. Flynn. The <XML> FAQ v. 1.6. World Wide Web Consortium, July 2000.

[10] C. F. Goldfarb and P. Prescod. *The XML Handbook*. Prentice-Hall, Inc., Upper Saddle River, NJ 07458, 1998.

[11] D. Goodman. *Java Script Bible*. Hungry Minds, Inc., 909 Third Avenue, New York, NY 10022, gold edition edition, 2001.

[12] S. W. Houlding. XML—An opportunity for <meaningful> data standards in the geosciences. *Computers & Geosciences*, 27:839–849, 2001.

[13] J. Bowler, J. Ferraiolo, et al. - World Wide Web Consortium. Scalable Vector Graphics (SVG) 1.0 Specification, W3C Recommendation. http://www.w3.org/TR/SVG/, September 2001. Accessed on 2002-10-30.

[14] P. A. Mansfield and D. W. Fuller. Graphical stylesheets – using xslt to generate svg. http://www.idealliance.org/papers/xml2001/papers/html/05-05-02.html, 2001. Accessed on 2002-08-22.

[15] M. Meneghello. XML (eXtensible Markup Language)—The New Language of Data Exchange. *Cartography*, 30(1):51–57, June 2001.

[16] A. Neumann and A.M. Winter, editors. *Time for SVG—Towards High Quality Interactive Web-Maps*. International Cartographic Association, 2001.

[17] R. Lake - Galdos Systems Inc. Making Maps for the Web with GML. http://www.geojava.com/company/glados/article/, December 2001. Accessed on 2002-07-29.

[18] A. H. Robinson and J. L. Morrison. *Elements of Cartography*. John Wiley & sons Inc., USA, sixth edition edition, 1995.

[19] S. Adler, A. Berglund, et al. - World Wide Web Consortium. Extensible Stylesheet Language (XSL) Version 1.0, W3C Recommendation. http://www.w3.org/TR/xsl/, October 2001. Accessed on 2002-08-20.

[20] S. Cox, A. Cuthbert, et al. - Open GIS Consortium. Geography Markup Language (GML) 2.0, OpenGIS implementation specification. http://www.opengis.net/gml/01-029/GML2.html, February 2001. Accessed on 2002-08-20.

[21] S. Sol. XML For Web Developers. http://wdvl.internet.com/Authoring/Lang ua-ges/XML/Tutorials/Intro/toc.html, May 1999. Accessed on 2002-10-03.

[22] SVG resources at Adode.com. Frequently asked questions. http://www.adobe.com/svg/, 2002. Accessed on 2002-08-25.

[23] T. Bray, J. Paoli, et al. - World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation. http://www.w3.org/TR/2000/REC-xml-20001006, October 2000. Accessed on 2002-08-15.

[24] G. Taladoire. Geospatial data intergration and visualisation using open standard. http://www.ec-gis.org/Workshops/7ec-gis/papers/pdf/taladoire.pdf, June 2001. Accessed on 2002-08-22.

[25] M. E. Vries, T. P. M. Tijssen, et al. The GML prototype of the new TOP10vector object model. Technical report, Technical University of Delft, The Netherlands, GIS Technology section, Department of Geodesy, Faculty of Civil Engineering and Geosciences, December 2001.

[26] A. Watt, C. Lilley, et al. *SVG unleashed*. SAMS, 201 West 103rd Street, Indianapolis, Indiana 46290, 2002.

Appendix A

UML model of GML Feature Schema

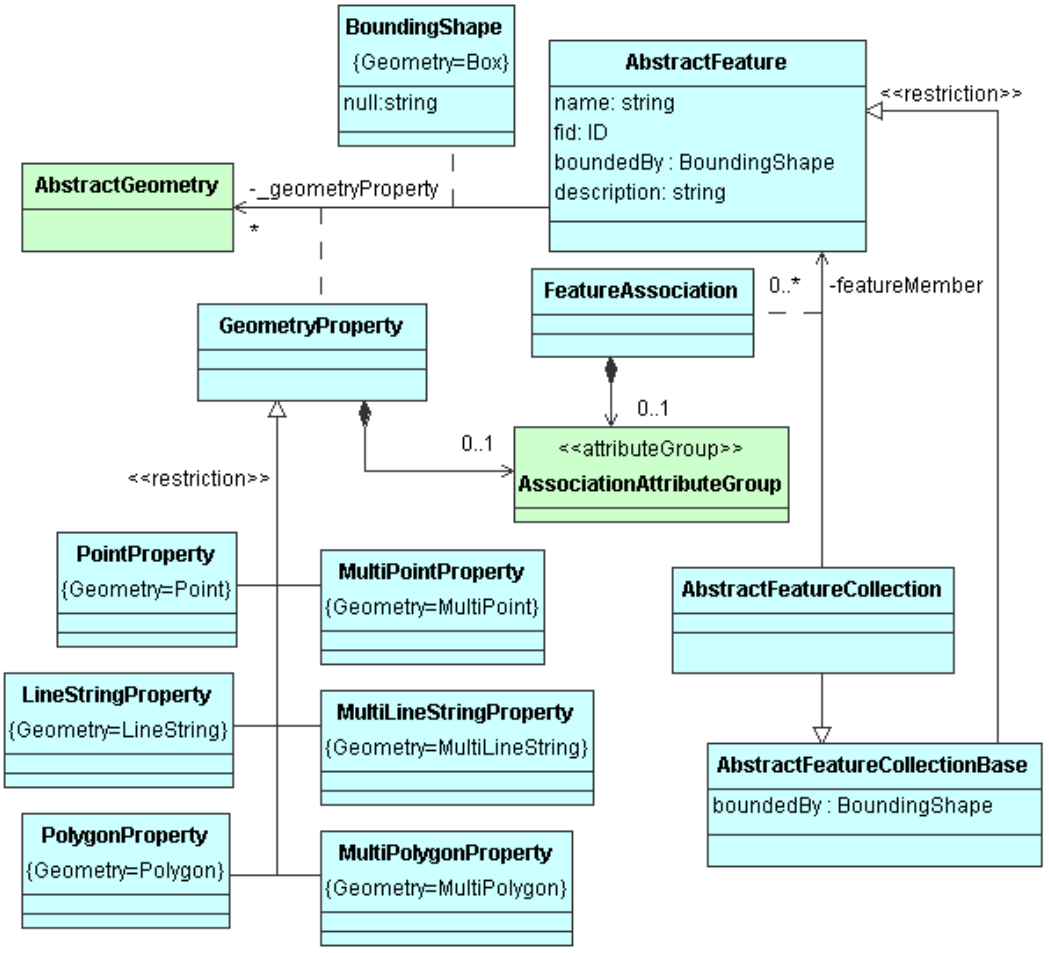

Figure A.1: UML model of GML Feature Schema. (Source: GML 2.0 Specification)

Appendix B

UML model of GML Geometry Schema

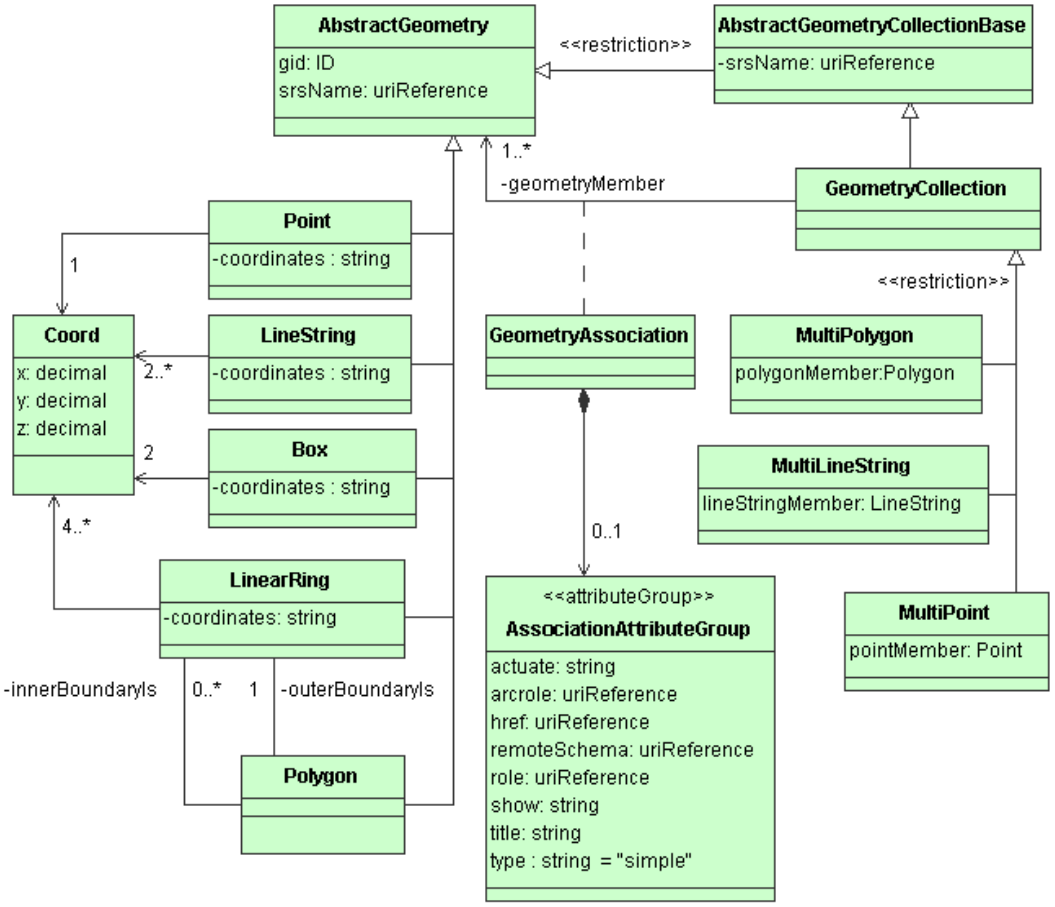

Figure B.1: UML model of GML Geometry Schema. (Source: GML 2.0 Specification)

Appendix C

UML model of GML data without member restriction

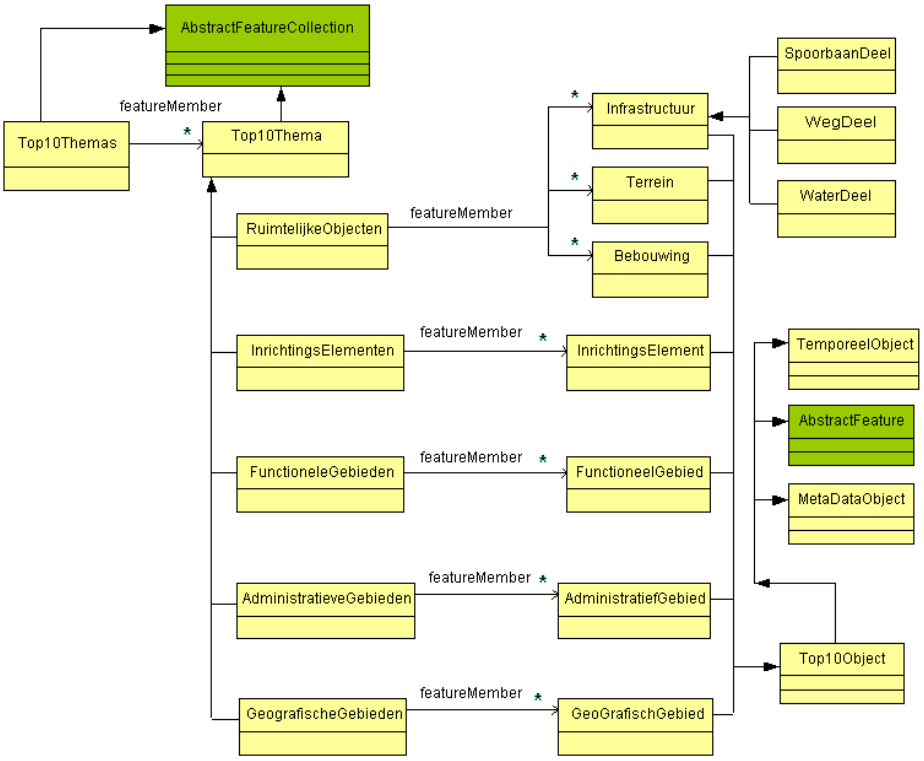

Figure C.1: UML application model of GML data without member restriction. (Source: GISt Report No.9 Technical University Delft)

Appendix D

UML model of GML data with member restriction

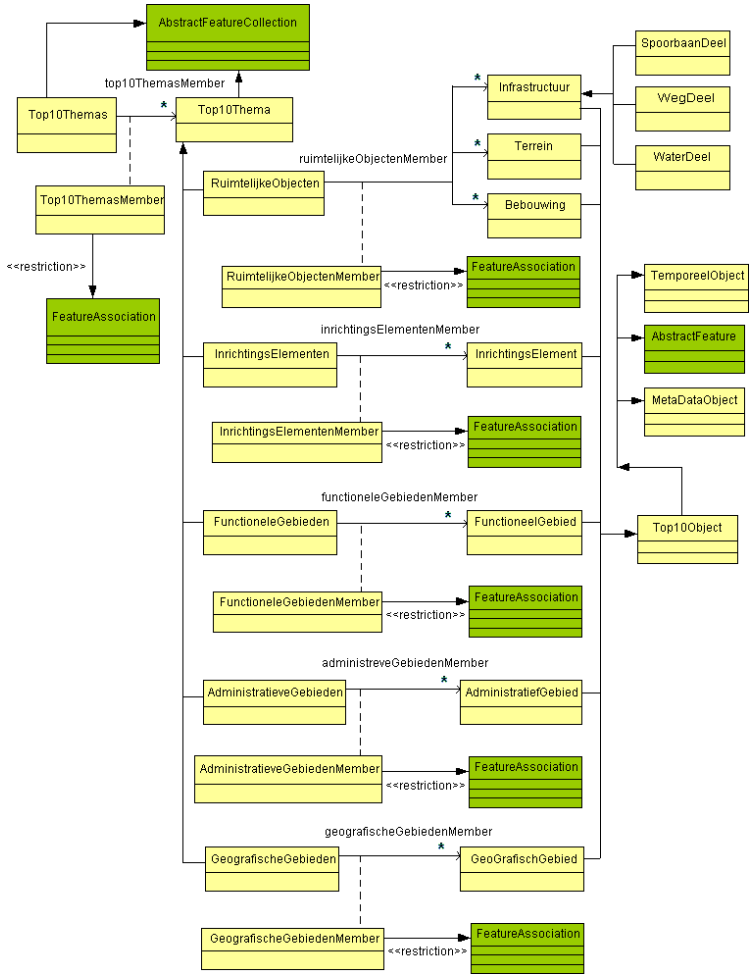

Figure D.1: UML application model of GML data with member restriction. (Source: GISt Report No.9 Technical University Delft)

Appendix E

Changing SVG visualization using CSS

Figure E.1: Visualization of Arnhem.svg using Style_1.css CSS.

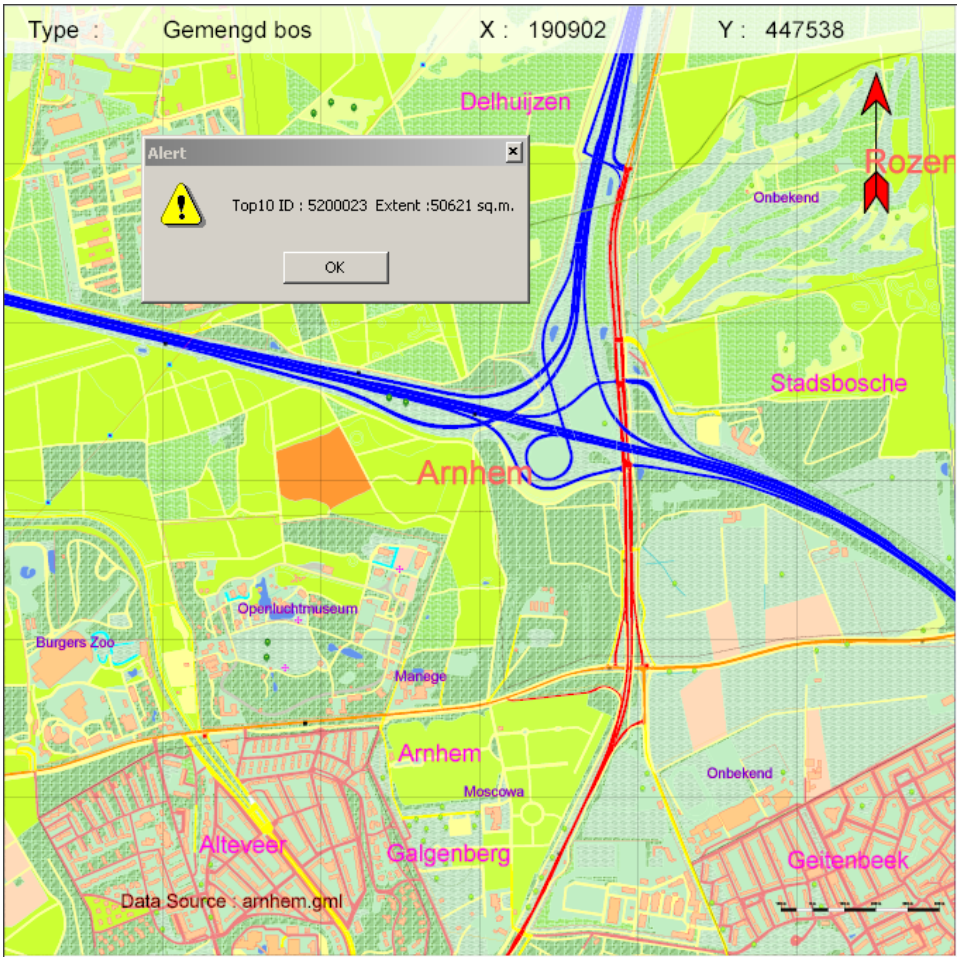

Figure E.2: Visualization of Arnhem.svg using Style_2.css CSS.

Figure E.3: Visualization of Arnhem.svg using Style_3.css CSS.