ARCHITECTURE AND METHODOLOGY FOR STORAGE, RETRIEVAL AND PRESENTATION OF GEO-SPATIAL INFORMATION

By

MARK E. FRASER

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2001

ACKNOWLEDGMENTS

I thank Dr. Joachim Hammer, the chairman of my thesis committee, for his role in giving me this opportunity and for his patience, expertise and dedicated assistance throughout the project. I also thank Drs. Sumi Helal and Joseph Wilson for their service on my thesis committee. I thank Ed Sims and Rick Zajac, of Vcom3D, Inc., and students Charnyote Pluempitiwiriyawej and Sushil Kumar Chordia for their technical assistance. Finally, I thank Gleim Publications, Inc., for its understanding and assistance with my educational endeavors.

TABLE OF CONTENTS

page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

ARCHITECTURE AND METHODOLOGY FOR STORAGE, RETRIEVAL AND PRESENTATION OF GEO-SPATIAL INFORMATION

By

Mark E. Fraser

December, 2001

Chairman: Dr. Joachim Hammer
Major Department: Computer and Information Science and Engineering

There are a growing amount of geo-spatial data sets available in electronic format that have been collected by various government agencies for purposes of research and exploration, commerce (e.g., maps and weather forecasting) and national defense (e.g., reconnaissance, training and simulation). The geometric portion of these data sets includes polygons, lines and points. The feature set portion contains terrain information such as forests, rivers, roads, buildings, and railroads. An important source of geo-spatial data is the Army, which distributes geo-spatial data using its SEDRIS (Synthetic Environment Data Representation and Interchange Specification) interchange format to allow the application-independent exchange of geo-spatial information.

Given the current and expected continued growth of these data sets, a method is desired for efficient, real-time, queriable access to and presentation of these geographical data sets. Such presentation should enable users to first visualize the data at a high-level

(a general overview) as well as to explore a specific area/region in greater detail (in the form of a "virtual tour"). Current methods of querying and interacting with geo-spatial data in real-time are inadequate for this type of interactive, experience-rich exploration. Prospective users of such an application are the United States military (e.g., for their battlefield simulations), and anyone else having a need to virtually visit a particular area for which geographic data sets are available.

In this thesis, we describe an architecture and algorithms to enable Web-based, 2-D and 3-D access to geo-spatial data stored in SEDRIS files. Our solution involves placing the SEDRIS transmittals into a database managed by an Oracle Relational Database Management System to support rapid access to selected portions of the data. To produce the output and make it available to the 2-D and 3-D viewers, we have developed transformations from relational and XML data into various output formats including GML, SVG and VRML. Our transformations are written in Java and Extensible Markup Language (XML) related technologies and can generate the desired output in near real-time when the desired data have been returned from the database. This thesis reports on the status of the prototype system that has been installed and tested in the Database Research and Development Center as well as on the insights and experience we have gained as the result of undertaking this project.

CHAPTER 1
INTRODUCTION

The goal of this thesis is to develop and implement an architecture which will efficiently support the real-time querying and rendering of geo-spatial data in various formats.

Problem Description

The geo-spatial data sets involved in this project consist of information about geographical terrain and man-made structures located in actual places around the globe. The data sets originate from the Synthetic Environment Data Representation and Interchange Specification, SEDRIS (www.sedris.org), a government-led association that provides the data in transmittals arranged in its own consistent format.

Because of its proprietary, non-standardized format, the data in a SEDRIS transmittal must undergo substantial transformation to be put into a format which will provide specific information to users. This thesis involves an improvement on current methods of accomplishing this transformation. In this thesis we will refer to these current methods as the pre-implementation approach and to the method we will develop as our "implemented" approach.

The pre-implementation approach relies on extracting the data from binary files using an API and manually transforming the data into a highly specialized format that only a specialized application understands. The extracted data are used in connection with an interface which simulates the subject places providing the user with the ability to

virtually experience them. This transformation process has the following drawbacks: It costs more to maintain software that uses a specialized API compared to software that uses standardized data formats for which well-established libraries and methods are available. The software is computationally intensive since, given the structure of a SEDRIS transmittal, large amounts of data must be searched in order to find specific or aggregate information. The pre-implementation approach must be run on the entire data set and cannot be run on selected manageable portions. Therefore real time access to selected portions is not possible. For the same reason, real time updating is not possible.

Summary of Research

In this thesis we will describe the development and implementation of an architecture that will allow querying of SEDRIS data with enough efficiency to allow real time creation of output based on user specified parameters regarding the scope and content of the output the user wishes to view. Specifically, the problems with the pre-implementation approach will be alleviated as follows:

1. The SEDRIS data will be maintained in a relational database (RDBMS). SQL queries will replace API access to binary files which will afford seamless, efficient access to SEDRIS data using well supported querying methods.

2. Although the approach undertaken here to access data of this type is novel, the API's and storage and retrieval methods are well-established. The resulting architecture is flexible and easily maintainable. It involves a server and a client, each of which can be run on virtually any PC-compatible machine.

Contributions Made

This thesis provides a comprehensive solution to the problem of storage and real time retrieval of geo-spatial data. Considering the desired output formats, this thesis also involves a variety of challenges which highlight the strengths and weaknesses of each of

the emerging technologies used. Specifically, a mechanism is provided to convert output from a relational database management system into an XML-based representation to facilitate additional transformations into Geographic Markup Language (GML), Scalable Vector Graphics (SVG) and Virtual Reality Modeling Language (VRML).

Current and Prospective Users

The pre-implementation application is used primarily by the United States military to make battlefield simulations. For this type of use, our implementation will allow for more realistic simulations due to the ability to explore the terrain in a more efficient manner. It is hoped that it will give these users access to more geographical terrain and in a more realistic format.

Prospective users also include anyone with an interest in exploring simulations of actual places. Although to some extent this would be possible with the pre-implementation application, our implementation will allow for a much wider audience since it will allow for a more flexible (and configurable) interface available to anyone with a web browser. Eliminating the need for specialized hardware and offering browser based access will greatly expand the potential user base.

Our implementation is being developed with the support of Vcom3D, Inc., a small Orlando-based company specializing in visualization software. Vcom3D was involved in the original conceptualization of the project and the conversion of the SEDRIS data into the relational database schema.

Thesis Organization

The remainder of this thesis describes the proposed architecture by describing the individual components and how they interact with each other. Necessary to an

understanding of this architecture is familiarity with the major technologies on which it is based. Chapter 2 introduces these technologies. Chapter 3 contains an overview of the architecture. Chapter 4 explains how the architecture was implemented. It contains a description of how the underlying technologies were applied to the problems presented. Chapter 5 evaluates the implementation, and Chapter 6 provides a summary, a conclusion and a description of possible improvements to our implemented prototype system.

CHAPTER 2
OVERVIEW OF RELATED TECHNOLOGIES

This chapter consists of a general description of the technologies used in our implementation. We will also explain how and where these technologies fit in the architecture and why they were selected over other alternatives.

Textual Processing Technologies

Given that both output from relational database queries and the output formats involved in this architecture are text-based, we focused on technologies oriented toward processing text data. Moreover, many such technologies have been emerging lately offering good prospects for tools well suited to this task.

Extensible Markup Language (XML)

Overview

XML [W3C2000] is a markup language that can be used to augment any text document with semantic information describing the nature and meaning of the data. An XML document contains content (words, tables, images, figures, etc.) as well as information (generally in the form of tags) which identifies the context and other information about the associated content.

The XML specification (the base standard is version 1.0, released in 1998) provides a universal format for adding markup to documents. The difference between an XML document and any other text document (even an HTML document) is that an XML

document must follow specified conventions in order to be considered valid or "well formed." As a result, all data to be stored in the document must be stored according to a defined document structure (often specified in a DTD, Document Type Definition, file) or XML schema, and this, in turn, allows all information in the document to be encapsulated (and queried) in a way similar in function to an object-oriented relational database management system.

XML provides a means of exchanging data across platforms and organizations. However it is strictly a means of maintaining data and does not contain presentation information. Shortly we will introduce additional technologies that we will use to include presentation information.

Why XML was used

XML provides a flexible and extensible format for relational output (output from database queries). Translating this output into XML provided an input format that subsequent processing methods understand and one that could, as the need arose, be easily restructured to make queries more efficient and simplified.

Extensible Stylesheet Language: Transformations (XSLT)

Overview

XSLT is a declarative, high-level language primarily designed for transforming one XML document into another. However it can be used to transform XML into any other text-based format. XSLT is a sub-part of the Extensible Stylesheet Language [W3C2001a], the other two parts of which are XPath (integrally related to XSLT) and XSL Formatting Objects (XSL-FO), a presentation language not used in this thesis.

Uses of XSLT

Examples of uses of XSLT involve transforming an XML document into an HTML file (the most common use), transforming an XML file into a comma delimited values file, and producing reports from business data. In most cases this process can be characterized as using XSLT to add presentation information to the data contained in an XML document.

How XSLT works

An XSLT "program" is a stylesheet that contains rules for transforming an input document into a result tree which becomes the output document. These rules specify what output should be generated for given patterns in the input. An XSLT transformation is performed by a software program called an XSLT processor, which is the XSLT equivalent of an interpreter. One source [HOL2001] lists the four top XSLT processors as Xalan [APA2001], Saxon [KAY2001], XT [CLA1999], and the Oracle XSLT processor [ORA2001]. All of these are Java-based.

An XSLT transformation is a two-stage process. First, the input document is read, validated and converted into the structure that reflects the desired structure of the output document. The first stage is usually referred to as building a "result tree." Reading and validating the document are usually done by the XSLT processor using the services of an XML parser. Additional steps taken in the first stage may involve carrying out specific instructions for selecting, grouping, sorting, and aggregating the data in the input document.

The second stage involves rearranging the result tree and adding formatting information according to rules of the stylesheet. Although XSL-FO is designed to handle this task, in practice, as here, this job is as easily handled by XSLT.

XSLT as a declarative query language

As the above description implies, implicit in XSLT is the need to extract (query) specified patterns of information from an input document. In this sense it conceptually resembles SQL. In carrying out this part of the process, XSLT uses an expression language called XPath [W3C1999]. Essentially, XPath gives XSLT access to specific nodes in an input document (based on their relative location in the document and/or their values) in a way that is analogous to how SQL gives access to records in a database meeting specified criteria. Like SQL, XSLT is declarative, meaning the user only specifies what data to get, not how it should be retrieved. It is entirely up to the XSLT processor as to how best to carry out the selection and transformation process.

Java Related Technologies

Java 2 Platform, Standard Edition

Overview

Since its introduction by Sun Microsystems, Inc., in 1995, the Java programming language has experienced an unrivaled rate of growth and acceptance [ENG1999]. It has arguably been the fastest growing software technology in history. Some of the reasons for Java's growth are its cross-platform capabilities (and corresponding usefulness in a distributed computing context), the availability and relative ease of deployment of Java applets in the context of the world-wide web, and its improvements on its implementation of the object oriented paradigm over other object oriented languages such as C++.

Why Java was used

Even though a key role in this project has been played by database and text oriented technologies offering flexibility and maintainability benefits over using a

procedural programming language, the use of such a language was nevertheless required to fill in the gaps where necessary to manage the transmission of the data among the various technologies used. There were also areas in which the sheer volume of textual and procedural processing required has in some cases rendered the textual oriented technologies inadequate.

Java was chosen over other languages due to its ability to seamlessly allow integration of smooth and efficient database access, fast server side processing, and cross portability. Java's integration of all of these factors allows for a mechanism well suited to managing various parts of a distributed system.

Java is especially well suited to working with XML (and by extension XSLT). Both XML and Java are geared toward cross-platform use. As mentioned earlier, all of the best performing XSLT processors (and XML parsers) are Java-based. In short, Java and XML work well together [FUC1999].

Java Servlet Technology

Overview

Java servlets [SUN2001a] provide a means of extending the functionality of a web server (assuming the web server itself supports them, which most popular ones do). They give the web server the ability to provide dynamic content and process HTML forms. This means that the web server can serve different pages to different users based on their characteristics and/or input.

The power of Java servlets lies in the fact that the entire Java API is available to the servlet writer. As a result, virtually any Java object can be instantiated in a servlet.

This gives the servlet the ability to interact with databases and to incorporate a site's business logic into a web application.

Alternatives to servlets

Java servlets are not the only means of providing dynamic web content. Competing technologies include Active Server Pages (ASP) [MIC2001] and Common Gateway Interface (CGI) [NCS1998].

ASP is a Microsoft solution to server-side programming, which requires Microsoft's web server, Internet Information Server, and either Windows NT or Windows 2000. It has gained popularity as a means of providing dynamic web content, but it suffers from being overly tied to Microsoft's proprietary technology.

CGI is a method for communication between the web server and a program or script. Unlike ASP, CGI can be used for application development on nearly any platform, because CGI scripts are commonly written in Perl, C, C++ or a Unix shell, which are available on all popular operating systems. CGI does not contain a method for maintaining server-level state. State can be maintained only via client-side cookies (which must be enabled by the client browser). The disadvantage of CGI is that a separate instance of the interpreter (and the script) must be executed for each request from each client. This makes CGI programs very inefficient when a large number of connections is required.

Why servlets were selected

Java servlets have been selected to manage the server side processing in our implementation for the following primary reasons:

1. The Java servlets can remain in memory once they are loaded and, unlike traditional CGI, do not need to be reloaded with each request. This makes the application much more efficient and responsive. This is especially true in this

application, since loading the servlet will involve initializing a portion of the data from the database. This initialization will need to occur only once each time the web server is restarted.

2. This application is likely to handle a large volume of data and potentially a large number of simultaneous users. The data set will often not change on an hourly or even a daily basis. It will remain static for long intervals until it is completely reloaded. It is therefore possible (and should prove advantageous) to maintain persistently in memory data that will be needed by many clients repeatedly. Among all of the alternatives, only Java servlets offered a simple way to do this.

3. To the extent that some aspects of our implemented architecture will need to be carried out in procedural program code (as opposed to stylesheets, the RDBMS, etc.), the availability of Java's object oriented methodology helps make servlets an attractive choice for management of the server side processing.

4. The XSLT transformations which form an important part of the architecture have been implemented using a Java-based processor, making these two important aspects of the architecture likely to be more compatible than they otherwise would be.

Presentation Technologies

The technologies described up to this point were evaluated based on how well they fit in with and contributed to the overall architecture. Presentation technologies, however, have been selected based on the needs of the prospective users since they play a critical role in formatting the output, which affects the users more directly than the internal structure of the system.

Geography Markup Language (GML)

Overview

GML is "an XML encoding for the transport and storage of geographic information, including both the spatial and non-spatial properties of geographic features." [OGC2001] It was developed by the OpenGIS Consortium (OGC). It allows organizations to share geographic information with each other. GML is a way of

encoding spatial data, i.e. properties and geometry of objects in the world. It is not concerned with ways of displaying or visualizing that data. Thus, GML separates data from presentation in a way analogous to XML (in fact GML is, not coincidentally, a form of XML). For our purposes, GML is an encoding for two-dimensional data. Throughout the remainder of this thesis, the terms GML and "two-dimensional" may be used interchangeably.

GML is based on an abstract geographic model developed by the OGC. In this model the spatial world is made up of features. A feature is a collection of properties and geometries. Properties are made up of a name, type and value; and geometries are made up of areas, lines and points. Features can be (and usually are) made up of other features. Lake provides a thorough introduction to GML [LAK2001].

Figure 1 shows a typical example of a GML linear feature. This file, which is a complete and valid GML file, is an illustration of how data about a road could be presented in GML. Later we show how we transform this file into SVG which contains the presentation information needed to allow a user to view it and interact with it.

An explanation of notable parts of the file follows (line numbers not part of the code have been added for explanation purposes in this and various other figures throughout this thesis): Line 1 identifies the file as an XML file and line 2 indicates that it is a *feature collection* conforming to `gmlfeature.dtd`, which is a document type definition file provided through the OGC and which provides rules for the XML encoding of geometric features. Line 3 contains the root `FeatureCollection` tag, which is a set of `FeatureMember` tags, the latter of which contains the feature details. Specification of the features themselves begins following the location extent of the

feature collection which is given in lines 4 through 8. After various properties are listed and defined, the location of the road is specified, which in this case is defined by a collection of connected points.

```
1.  <?xml version="1.0" encoding="utf-8"?>

2.  <!DOCTYPE FeatureCollection SYSTEM "..\stylesheets\gmlfeature.dtd">

3.  <FeatureCollection typeName="Root">
4.     <boundedBy>
5.        <Box>
6.           <coordinates>0,0 10000,10000</coordinates>
7.        </Box>
8.     </boundedBy>

9.     <featureMember typeName="RootMember">
10.       <Feature typeName="Road">
11.          <Property typeName="Surface Material Category (Units Enumeration)"
12.             type="PDV SAC ENUM VALUE">104</Property>
13.          <Property typeName="Height Above Surface Level (Units meter)" type="PDV
14.             Int 32">0</Property>
15.          <Property typeName="Width (Units meter)" type="PDV Int 32">5</Property>
16.          <geometricProperty typeName="centerLineOf">
17.             <LineString srsName="UTM">
18.                <coordinates>2069.7,3799.7 2167.6,3883 2216,3910.8 2332.3,3935.6
19.                   2353.6,3960.4 2328.3,4087 2304.8,4287.8 2336.7,4320
20.                </coordinates>
21.             </LineString>
22.          </geometricProperty>
23.       </Feature>
24.    </featureMember>

25. </FeatureCollection>
```

Figure 1: A GML road.

Why GML was used

SEDRIS data sets are rich with two-dimensional geographical information that needs to be encoded and provided as an integral part of the description of the subject location. This information is explicitly described in terms of the same types of features and properties contemplated by the GML specification. Thus, the amount of translation necessary to extract two-dimensional information from the RDBMS into GML was minimized. Further, GML is based on XML. Therefore it carries with it all of the

benefits of XML discussed previously, including the ability to easily transform the GML data into presentation output formats. These factors made GML a logical choice for encoding the results of two-dimensional queries of SEDRIS data.

Scalable Vector Graphics (SVG)

Overview

SVG is "a language for describing two-dimensional vector and mixed vector/raster graphics in XML." [W3C2001b] SVG is the subject of a W3C recommendation (version 1.0 was published on September 4, 2001). It is primarily a presentation format. For our purposes, we can think of SVG as being to GML what HTML is to XML.

SVG came about as an answer to the limitations inherent in the two most commonly used graphics formats on the web, i.e., GIF and JPG. The limitations center around the fact that these are pixel-based formats. With these formats, resizing the picture invariably distorts (or degrades) the picture. Also, the image file must contain data fully describing each pixel in the image. SVG, on the other hand, involves a much more efficient approach of merely describing all relevant shapes and paths needed to construct the image. As a result, the image can be resized without distorting it.

In a web-based context, SVG is rendered by the browser based on data sent by the server. One advantage of this is that it distributes some of the processing load more evenly between the server and the client. A more important advantage is that it allows the client to redraw the image when zooming or panning without having to contact the server to get a different version of the image.

Figure 2 is an example of how the road described in GML in Figure 1 could appear in SVG. It constitutes a complete SVG file which when loaded into an SVG

```
1.  <?xml version="1.0" encoding="iso-8859-1" standalone="no"?>

2.  <!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20000303 Stylable//EN"
3.
4.  "http://www.w3.org/TR/2000/03/WD-SVG-20000303/DTD/svg-20000303-
5.     stylable.dtd">

6.  <svg viewBox="0 0 10000 10000">
7.     <g transform="matrix(1,0,0,-1,0,10000)">
8.        <rect x="0" y="0" width="10000" height="10000"
9.           style="fill:lightgray"/>
10.       <polyline style="fill:none;stroke:black;storke-width:30"
11.            points="2069.7,3799.7  2167.6,3883 2216,3910.8 2332.3,3935.6
12.            &#xA; 2353.6,3960.4 2328.3,4087 2304.8,4287.8 2336.7,4320
13.            &#xA;"/>
14.    </g>
15. </svg>
```

Figure 2: Road depicted in SVG.

capable browser provides a two-dimensional picture of the road. Like the GML file, it refers to a DTD and then specifies the coordinates in which it is drawn (lines 4 and 5). It also contains a line connecting the points (lines 10 through 13). The main difference is that it contains information about how to render the road. In this case, it provides a width and a color.

Why SVG was used

SVG was used due to its close proximity to XML. This fits well with the overall theme of this project, that is, building results based on transformations of textual documents produced from database queries. Compared to other, binary graphics formats, XML to SVG transformations provide a relatively simple and efficient mechanism to add presentation information to query results. We expect to benefit from the fact that it is easy (and fast) to transform XML documents into SVG. Since we are developing a web-

based application, the benefits of SVG over other web graphics formats described in the previous section are all applicable. SVG plug-ins, such as Adobe's SVG viewer [ADO2001], are readily available for all popular web browsers, simplifying deployment.

Virtual Reality Modeling Language (VRML)

Overview

VRML is a textual format that allows the creation of three-dimensional virtual worlds. Although VRML is not XML based (an XML based corollary known as X3D is under development [W3D2001]), it provides many of XML's benefits in that it is text based and follows a well defined structure. As with SVG, VRML plug-ins, such as Blaxxun's [BLA2001], are readily available for all popular web browsers, making the distribution of VRML content via the web possible. VRML was recognized as an ISO standard in 1997 [VRM1997].

VRML is distinguished by its facilities for the creation of sensory-rich, realistic worlds through its support for features such as animation, sound, and scripts. As of version 2.0 (also referred to as VRML97), even the world itself can move and interact with the visitor.

Version 2.0 specifically states that it was designed with a distributed environment such as the web in mind. Among the features included as a consequence are the ability to inline other VRML files and to hyperlink to another file or location and the use of a compact syntax. This compact syntax is accomplished via constructs such as a *Proto* (user-defined node types) [HUB2000] and the *Use* construct, which allows reuse of nodes once defined in the same file.

A VRML file is a representation of what is referred to as a "scene graph." The term arises from the fact that the scene, or world, modeled in the file is presented in terms of a root x, y, z coordinate system. This is actually a tree of coordinate systems, as many objects, or groups of objects, are positioned in terms of their own coordinate system, which is translated relative to the root coordinate system or another descendant of the root coordinate system [AME1997].

A scene graph is a collection of VRML "nodes." A node is a description of an object and its properties. There are many types of nodes defined in VRML. Nodes are grouped hierarchically in such a way as to also describe the relationships among the nodes in the scene graph.

An important feature of VRML is *event routing*. VRML nodes can be specified to generate events in response to environmental changes or user interaction. This process can be analogized to plugging in wires to input and output jacks. For example, a node that represents a door in the world can be wired to open when it is pushed. In this case, the door itself would generate the event to which it, in turn, would respond. Events normally, however, occur when an action occurring in another node takes place. For example, the door can be wired to open when a button near the door is pushed.

A VRML example

Figure 3 depicts a complete VRML file. It depicts a cone, sitting on a ball which is sitting on a box. It illustrates the use of VRML built-in shapes (*Box*, *Sphere*, and *Cone*). The physical arrangement of the objects is achieved via the *translation* field of the *Transform* node of each shape. Each translation specifies the location of the object in relation to the file's root coordinate system. Since the sphere (ball) has a translation of `0, 0, 0` it is located at the origin of the root coordinate system. The box and the cone

are located below and above the sphere, respectively, by virtue of their `-1` and `1` y-coordinate values.

```
#VRML V2.0 utf8

   Transform{
      translation 0 -1 0
      children
      [
         Box { size 1 1 1 }
      ]
   }

   Transform {
      translation 0 0 0
      children
       [
          Sphere { radius 0.5 }
       ]
   }

   Transform {
      translation 0 1 0
      children
      [
         Cone {
            bottomRadius 0.5
            height 1
            side TRUE
            bottom TRUE
         }
     ]
   }
```

Figure 3: Sample VRML file.

Alternatives to VRML

Java 3D Java 3D [SUN2001c] is a scene graph-based API provided by Sun Microsystems as an extension to the Java programming language. Its biggest advantage is also its greatest weakness. Its advantage is that, since it is a part of a well defined programming language, there are no issues as to differences in implementation (which can be seen in differences among VRML browsers). However as a programming language with a vast API it is often seen to be beyond the reach of non-programmers.

Viewpoint Viewpoint (formerly Metastream) [VIE2001] is a proprietary web content solution geared toward e-commerce sites. It supports the creation of virtual worlds by combining a variety of multimedia formats with an XML-based scene graph file. It is supported on the client side via a browser plug-in. One of its advantages is that it supports file streaming. The entire file need not be downloaded in order for the client to begin rendering or playing the content. The primary disadvantage is that it is proprietary and not free and it has not gained broad reaching acceptance.

Why we used VRML

VRML is used in this application first and foremost because of the ease with which one can create and, more importantly, modify, interactive worlds. More specific benefits are as follows:

1. The use of VRML eases deployment requirements. On the client side, only a browser plug-in is required to view and interact with VRML worlds.

2. VRML is an excellent choice for a web based application since its structure and text-based format generally lead to relatively small files. Any shape that can be rendered in VRML will most likely take up much less space than the same object rendered as a pixel-based graphic image.

3. Also because of its text based format, VRML can potentially be indexed (i.e., in a search engine). This may provide more benefits as VRML worlds become more prominent, complex and intertwined. Libraries of objects should become more widely available.

4. VRML is gaining in acceptance. This will lead to continued growth in browser support, reusable objects and the availability of worlds that can seamlessly be integrated.

CHAPTER 3
OVERVIEW OF ARCHITECTURE

Overview

The architecture described in this thesis encompasses the loading, retrieval, transformation, processing and presentation of geo-spatial data in various formats. This architecture is depicted schematically in Figure 4.

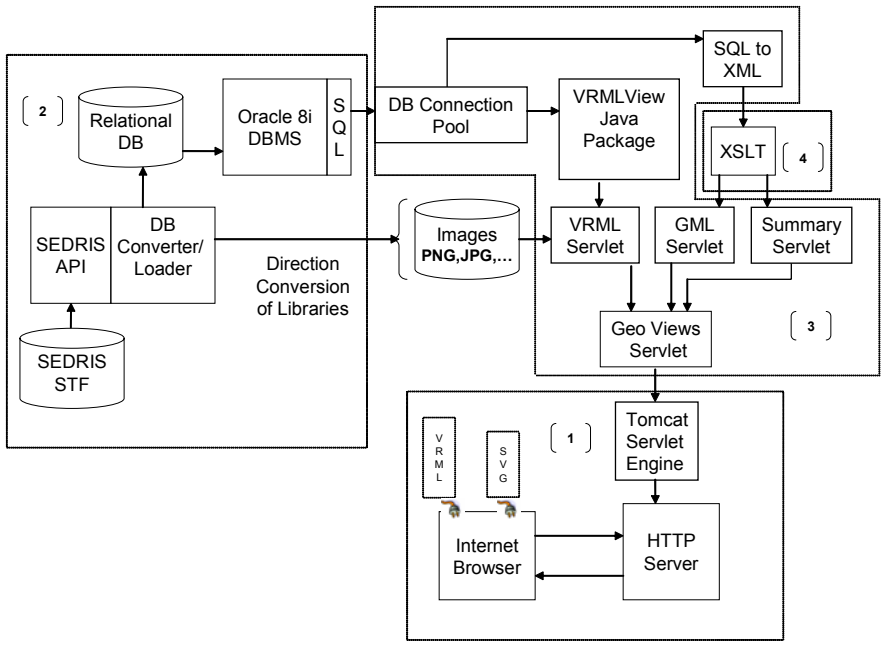

Figure 4: Geo-Spatial data storage, retrieval and presentation architecture.

Description

This architecture consists of four main elements (each is outlined and numbered accordingly in Figure 4): (1) the web server and client applications (including the three available output formats); (2) the geo-spatial data as it is stored in the RDBMS; (3) the

Java classes that manage the processes that make up the architecture; (4) and the XSLT stylesheets with which the transformation of the data into SVG and GML occurs. These elements are described in turn in this chapter.

Web Server and Client Applications (Box 1 in Figure 4)

The interface to the architecture is implemented via a web server in order to expand accessibility to anyone with a web browser and access to the internet (or, where applicable, to the private network on which it is installed). From any web browser the visitor can request output in text format. In most cases, however, a more interactive experience will be desired, and a "plug-in" will be required in order to view output in SVG format or interactive VRML. There are several freely available browser plug-ins for SVG and VRML that are supported on the two major web browsers, Internet Explorer and Netscape. Our implementation was tested using Adobe's SVG plug-in [ADO2001] and Blaxxun's VRML plug-in [BLA2001].

With each query the user must specify one of three available view types. These are summarized below.

Summary view

The summary view is a statistical overview of the data. It allows a user to obtain various counts as to how much of a given geo-spatial element is present in a specified area of the geography covered by the selected database. For example, an "instance count" tells the user, by tile, how many model instances (e.g., buildings and other man-made structures) are present in the selected tiles. The user can also specify a count based on the terrain's description (i.e., how many forests are in a particular area). The purpose of the summary view is to identify areas with a high concentration of particular elements

(information which can then be used in selecting what portions to view with the other view types).

If the user elects to receive the output in SVG format, the user will see a colored grid, with each cell representing one tile and the color indicating the range in which the count falls. An example of this output is presented in Chapter 4 (Figure 20). The user can alternatively elect to view the XML results of the query (from which internally the program creates the SVG output).

2-D view

The "2-D view" provides two-dimensional attributes about geometric features within a selected geometric area (in this thesis the 2-D view is also referred to as the "GML view"). The user can elect to view in GML format information about features delineated by areas (e.g., forests and bodies of water), lines (e.g., rivers, roads, and railroads) and/or points (e.g., trees and buildings). The user can elect to view this information in raw GML or SVG. In the latter case the output represents a two-dimensional map giving the user a schematic diagram of the features in the selected range. An example GML map is shown in Chapter 4 (Figure 24).

3-D view

The 3-D view is a VRML world built from selected tiles in the database (the 3-D view is also referred to in this thesis as the "VRML view"). Similarly to the other two view types, the user can elect to receive the output as either raw text or as a VRML stream which she can experience in a VRML plug-in. The user can elect to view a particular range of tiles or the area within a bounding box created from two coordinates. In the latter case, however, the specified range is converted to a set of tiles in order to

allow the application to efficiently deliver the information (since the VRML for each tile is cached as a unit) .

Geo-spatial Data Element (Box 2 in Figure 4)

The data element encompasses the conversion and storage of the geo-spatial data. From the standpoint of the architecture, data arrive in a specialized format called SEDRIS transmittal format (STF) [SED1999]. As the name implies, the format is specific to SEDRIS. It is a binary format accompanied by an elaborate API and a suite of utilities to assist in extracting data from the transmittal [SED2001]. Once the initial conversion has taken place, the information is stored in a SEDRIS independent format. It follows that this architecture could support data from non-SEDRIS sources with the requirement only that a corollary to the SEDRIS API be available to take the place of the DB converter/loader.

In the loading process, the geo-spatial elements in each transmittal are extracted and placed into an Oracle 8i relational database. The database schema into which the data set is imported is described in detail in Chapter 4. The software designed to handle this extraction was developed and is maintained by Vcom3D, Inc.

Each SEDRIS transmittal consists of a manageable chunk of data, usually consisting of a city or some distinct geographic unit. A transmittal covering an area of a few hundred kilometers is common. It is anticipated that user queries and interaction will be confined to data within a transmittal. Therefore each transmittal is stored in a distinct database instance (using the same underlying schema). The user can choose the database (i.e., which geo-spatial data source) to which he wishes to connect. The design is nevertheless scalable because, since each data source is stored using the same schema, it

will allow us to merge data sources (i.e. combine smaller geo-spatial areas into larger ones) should that become desirable in the future.

The geo-spatial data set in a SEDRIS transmittal is divided up conceptually into relatively equal sized, square grids referred to as "tiles." A typical tile size is one square kilometer, but the sizes vary. The tile arrangement is retained in the database and it is used in this architecture in order to divide queries into manageable chunks.

Java Element (Box 3 in Figure 4)

The Java element consists of Java classes that (1) manage and transmit queries to the database and (2) produce the web pages that gather input from and provide results to the user. These tasks are described more fully in the following sections.

Database queries and connection pool

The Java classes contain parameterized queries which contain the SQL statements needed to formulate the three view types. For two of the views, the Java classes convert the results into XML in order to prepare these results for transformation via XSLT into the proper output format. In the case of the VRML view, the results are placed directly into Java classes specifically designed for processing the database query results and creating the output. XSLT is not used in the transformation to VRML for reasons that are discussed in Chapter 4 where the mechanics of the VRML production are explained.

Each time a query is processed, a database connection is required. Java Database Connectivity, which forms a part of the Java developer's kit, and is supported as a part of Oracle's call interface, is used to establish and maintain the database connection. It is anticipated that multiple queries from multiple clients will need to be simultaneously supported. Because of the overhead required to establish a database connection, the

server maintains a “pool” of database connections. Each time a client needs to process a query, it need only check out a connection from the pool, eliminating in most cases the need to re-establish and wait for a database connection.

Servlets

Java servlets form perhaps the most pervasive part of the architecture since they are involved in the entire client interaction from start to finish. The role of each of the four servlets is described in the following paragraphs.

Geo views servlet. The Geo Views servlet is essentially an application that manages the client’s interaction with the other servlets. It directs traffic among the servlets. The Geo Views servlet is the first servlet invoked when a new user visits the site which houses the application. At this initial step the servlet consults the database connection pool for the list of available databases. It then creates and returns an HTML page allowing the user to select one of these databases.

Once the user has selected a database, the user must select one of the three view types, Summary View, 2-D View or 3-D View. Because each of the three views is managed by a distinct servlet, the Geo Views servlet has the task of providing an HTMLl page on which the user selects a view type and then when the user submits this request the Geo Views servlet responds by redirecting to the appropriate servlet.

Summary view servlet. Requests that the Geo views servlet receives for summary views are redirected to the summary view servlet. The summary view servlet asks the user to specify one of three possible count types and an output type (i.e. XML or SVG). The user can also specify a description. In both cases, the servlet pre-fetches the extent of available data so that the user is aware of what is available to select from. The servlet obtains this information from the database (i.e. by getting the extent of the

available tiles and the list of potential descriptions) but it only has to do this once each time the web server is restarted since the servlet is a persistent process that maintains this information in memory.

Once the user has specified count type, dimensions, description, and output type, the servlet (through the use of specialized Java classes that is described more fully in the next chapter) queries the database and runs an XSLT transformation to produce the output which is sent to the browser.

GML View Servlet. The GML view servlet produces the 2-D output in a similar way as the summary view servlet. Here, instead of a tile range, the user specifies two x/y coordinate pairs and the servlet uses these parameters, along with XSLT stylesheets to produce the desired output. The user has the option of obtaining a GML file or to view the SVG in his browser.

VRML View Servlet. The VRML view servlet operates similarly to the previous two with the exception that all of the output is produced using the underlying Java classes (also more fully described in the next chapter) and not using XSLT. The user can specify an area (by tile range or x/y coordinates) and to view the output as a plain text VRML file or interactively using a VRML plug-in.

As mentioned earlier, the VRML servlet creates the VRML content on a tile by tile basis, and the servlet architecture allows the VRML data for each tile to be retained in memory. This leads to very fast response times for a given tile or range of tiles for all users after the first user has requested a given tile or range.

XSLT Stylesheet Element(Box 4 in Figure 4)

XSLT is the work horse of the summary view and 2-D views. The Java classes instantiated in the summary view and GML view servlets send SQL queries the database and convert the results into XML. These classes do not encapsulate the information necessary to produce output in any of the desired formats. XSLT plays this role.

The XML results produced by the Java classes can be thought of as the XML equivalent of an SQL result set. In the case of the summary view, for example, this might be a list of information about the polygons in the specified row and column range (with one XML element per polygon). In this case the stylesheet processes the XML results by counting the number of polygons and producing a much smaller XML file that contains just the requested counts. If the user has requested SVG formatted output (for either the summary view or 2-D view), another stylesheet is used to process the latter XML file in order to provide the colored grid described earlier.

CHAPTER 4
IMPLEMENTATION

In previous chapters we described in general terms the technologies used and explained why we used them. We have also given an overview of the architecture in an attempt to explain how these technologies were used together. In this chapter we describe the technical details of the implementation for each of the major technologies implemented in the architecture. We start with a brief introduction to geo-spatial data. We will then highlight the database schema, followed by the Java and then the XSLT implementations.

Database Implementation

Introduction to Geo-spatial Data

An understanding of the main concepts of geo-spatial data will facilitate an understanding of our database schema. Geo-spatial data contains information about the *location*, *shape* and *attributes* of real objects [ORD2001].

Location and shape

Each shape is formed by a single point or an ordered collection of points in a coordinate system (which could be specified as latitude and longitude or distance from a specified origin). Since each point is defined by its location in the coordinate system, the location of a feature is also determined by reference to the points that make up its shape.

In this sense the shape and the location of a feature are defined together. There are three principal types of shapes.

Points. Points are objects that are defined with reference to one *x*, *y* coordinate. Since each distinct point object is not defined by a unique shape, they are usually rendered with reference to a prototypical shape. Examples of objects that are commonly defined in this way are trees and buildings (the latter can also be defined as an area).

Lines. Linear features are defined by connecting a group of points in a specified order. No point appears more than once in the specified order. Common examples of linear features are rivers and roads.

Areas. Area features are defined by connecting a series of points in a specified order to form a linear ring (closed boundary). Examples of areas are forests, lakes and grassland.

Attributes

Attribute data (also referred to as properties) are descriptive information about features. Each attribute has a type name which indicates what it defines, a data type which describes how it is measured, and a value. For example, a forest might have a density attribute with a type name of "Density Measure (% of Tree / Canopy Cover) (Units percent)", a data type of integer, and a numeric value indicating the percentage of tree cover.

Schema

The database schema is depicted in Figure 5. The schema has been designed with the dual goals of encapsulating the geo-spatial information from the SEDRIS transmittals and to allow efficient retrieval of that information. In order to describe the schema in

context, in the following sections we will explain which elements of the schema are used in representative queries generated for each of the view types.

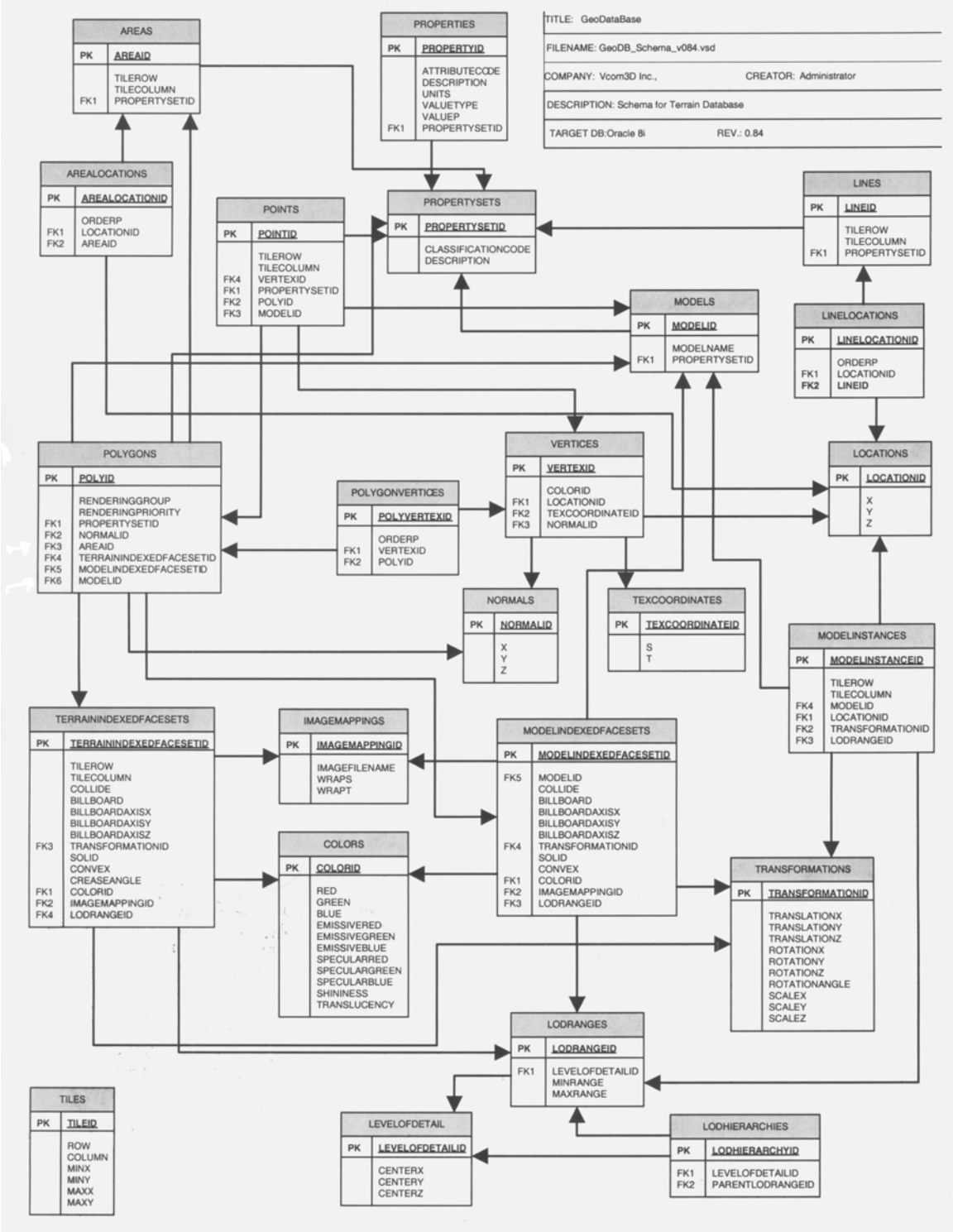

Figure 5: Database schema.

Summary view

The representative query which we will examine is one used to obtain the terrain count. The terrain count is designed to give the user an idea of where within a given area exist high concentrations of terrain polygons. Terrain polygons are distinguished from model polygons in that terrain polygons, even though they may be made up of features that are reused in other places, are each considered a separate entity with a unique location. Model polygons, on the other hand, exist within a model that, in turn, may be reused in different locations. This generally is the distinction that is made in the schema between terrain and models. Terrain features are individually locatable and are not repeated, while models are.

Figure 6 illustrates a terrain count query to count the number of forests in the area encompassing the tiles ranging from row 1, column 1 and row 20, column 20 (this query

```
SELECT  IFS.TileRow, IFS.TileColumn,  count(P.PolyID) AS TERRAINCOUNT

FROM    Polygons P, TerrainIndexedFaceSets IFS, PropertySets PS

WHERE          P.TerrainIndexedFaceSetID = IFS.TerrainIndexedFaceSetID AND

  IFS.TileRow     >= 1                                    AND
  IFS.TileRow     <= 1                                    AND
  IFS.TileColumn  >= 20                                   AND
  IFS.TileColumn  <= 20                                   AND
  P.PropertySetID  = PS.PropertySetID                     AND
  PS.Description   = 'Forest'

GROUP BY       IFS.TileRow, IFS.TileColumn

ORDER BY       IFS.TileRow, IFS.TileColumn
```

Figure 6: Terrain count query.

covers about a four hundred square kilometer area in Bellevue, Washington). In order to understand this query it is necessary to be familiar with *indexed face sets*. An indexed face set is a collection of polygons whose edges are connected (similar to a jigsaw

puzzle) to form an overall shape. It allows for rendering efficiency in that when the internal polygons are drawn it is not necessary to redraw common coordinates.

The query in Figure 6 counts all of the terrain polygons that are in any indexed face set that is in the selected tile row and column range. This is possible since all terrain polygons form a part of an indexed face set. It is necessary to query the `TerrainIndexedFaceSets` table since it is only via this table that it can be determined which row and column the polygons within the indexed face sets lie.

The distinction between terrain and models referred to above is reflected in the schema in part with reference to the fact that the `TerrainIndexedFaceSets` table contains a `TileRow` and `TileColumn` value and the `ModelIndexedFaceSets` table does not. Instances of the latter could appear anywhere, and the actual location must be obtained with reference to the `ModelInstances` table.

The other table referenced in the query is the `PropertySets` table. Property sets, as the name implies, are collections of properties. Properties are attributes such as density, height above surface level, etc., depending on the type of PropertySet. Property sets are a way that SEDRIS uses to group together objects of similar attributes. This allows polygons that share similar attributes to use the same property set, where the alternative would be to provide a distinct list of properties and values for every polygon. One would expect this to be convenient since many polygons could make up the same forest, and one would expect these polygons to have similar attributes. This is borne out by the fact that there are approximately 13,964 polygons in the Bellevue database and only 296 distinct property sets.

2-D view

The 2-D view involves GML features that are structured in sections divided by areas (e.g. forests), lines (e.g. roads) and points (e.g. buildings). The user has the option of selecting one or any combination of these three feature types to be included. As a consequence of this, it makes sense from an efficiency and simplicity standpoint to divide the queries into separate queries for each type. This also made it easier to structure the XML output from the database (converted from the SQL output) so as to allow the XSLT transformation process to efficiently access the feature types that it needs.

For the 2-D view we will examine two representative queries since they are both used together to, in this case, generate the "area" features for the entire extent of the Bellevue database. Figure 7 shows the query needed to get the area features.

```
SELECT  A.AreaID, PS.PropertySetID AS PSID, L.X || ',' || L.Y AS APOINT,
        PS.Description AS PSDesc

FROM    Areas A,              PropertySets PS,      Locations L,
        AreaLocations AL,     AreaBoundingBox ABB

WHERE   A.PropertySetID  = PS.PropertySetID          AND
        A.AreaID         = AL.AreaID                 AND
        AL.LocationID    = L.LocationID              AND
        A.AreaID         = ABB.AreaID                AND
        NOT     ( (ABB.MaxX <= 0)             OR
                  (ABB.MaxY <= 0)             OR
                  (ABB.MinX >= 10000)         OR
                  (ABB.MinY >= 10000) )

ORDER BY       A.AreaID, AL.OrderP
```

Figure 7: Areas query.

Each area is outlined by a series of points that connect to form an enclosure (a *linear ring*). The points that make up the ring for each area are stored in the `AreaLocations` table. The `AreaLocations` table contains the points, the x, y coordinate values for which are obtained from the `Locations` table, as well as the

order in which the points are connected to form the ring (via the `OrderP` field in the `AreaLocations` table). In the areas query, the locations are concatenated in order to simplify the processing since they are shown as *x*, *y* pairs in the GML output (this makes for one distinct XML element to extract from the query results where otherwise there would be two). The lines query is similarly derived using the using the `Lines`, `LineLocation` and `Locations` tables.

There is one "table" referenced in the query that is not in the schema depicted in Figure 5. This is the AreaBoundingBox *view*, which is simply a *materialized view* (stored summaries of queries containing precomputed results [ORA1999]) which provides the minimum and maximum coordinate extents of each area. There is also a LineBoundingBox view. These views were created in order to identify which areas and which lines fall within a given range. Even though it would be possible with one or more queries to filter areas and lines based on the locations of each of their points, these views make such filtering much more efficient.

The query in Figure 7 is not enough to make the GML output. The properties that make up the property sets are needed. These are obtained in a separate query since the property values are identified with property sets and not areas or locations. The query to obtain the properties is depicted in Figure 8. This query gets the data needed to list in the GML information about all of the properties of the property sets. The property sets are given a "Feature" tag in the GML, and each of the properties is a "Property" element within the associated feature.

```
SELECT          DISTINCT P.PropertyID) AS PID,PS.PropertySetID AS PSID,
                P.Description AS PDesc,       P.Units,
                P.ValueP,                     P.ValueType AS Type

FROM            Areas,  PropertySets PS,      Properties P,  AreaBoundingBox

WHERE           Areas.PropertySetID = PS.PropertySetID       AND
                PS.PropertySetID    = P.PropertySetID        AND
                Areas.AreaID        = AreaBoundingBox.AreaID        AND
                NOT     ((AreaBoundingBox.MaxX <= 0)         OR
                         (AreaBoundingBox.MaxY <= 0)         OR
                         (AreaBoundingBox.MinX >= 10000)     OR
                         (AreaBoundingBox.MinY >= 10000))
```

Figure 8: Area properties query.

3-D view

The queries involved to generate the 3-D output are much more numerous and complex, which is a reflection of the fact that the output itself is more voluminous and complex. Conceptually, however, the process is similar in that for each of the views the queries are mainly concerned with obtaining all instances of a given entity along with various attributes associated with the entity.

The summary view was concerned with polygons and the 2-D view was concerned with areas, lines and points. The 3-D view is centered around indexed face sets and levels of detail. For each of these two elements a distinct query is generated in the process of generating the data for the VRML view. Additionally, one query each is generated for the indexed face set's coordinates and texture coordinates. Next we will examine these queries.

Indexed face set query. In creating the VRML from our schema, each indexed face set forms a part of a distinct VRML *shape* node. A VRML shape node describes the geometry and appearance of an object in a VRML world. The indexed face

set, including its coordinates and attributes, appears as the geometry. A typical query to generate the terrain indexed face sets in a tile range is depicted in Figure 9.

```
SELECT  IFS.TileColumn,               IFS.TileRow,
        IFS.TerrainIndexedFaceSetID,  LODR.LevelOfDetailID,
        IFS.LODRangeID,               IFS.Collide,
        IFS.Billboard,                IFS.BillboardAxisX,
        IFS.BillboardAxisY,           IFS.BillboardAxisZ,
        IFS.TransformationID,         IFS.Solid,
        IFS.Convex,                   IFS.CreaseAngle,
        C.ColorID,                    IMG.ImageFileName,
        IMG.WrapS,                    IMG.WrapT,
        C.Red,                        C.Green,
        C.Blue,                       C.EmissiveRed,
        C.EmissiveGreen,              C.EmissiveBlue,
        C.SpecularRed,                C.SpecularGreen,
        C.SpecularBlue,               C.Shininess,
        C.Translucency,               T.TranslationX,
        T.TranslationY,               T.TranslationZ,
        T.RotationX,                  T.RotationY,
        T.RotationZ,                  T.RotationAngle,
        T.ScaleX,                     T.ScaleY,T.ScaleZ

FROM    TerrainIndexedFaceSets IFS,   ImageMappings IMG,
        Colors C,                     Transformations T,
        LODRanges LODR

WHERE   IFS.TileColumn         >= 1                          AND
        IFS.TileRow            >= 1                          AND
        IFS.TileColumn         <= 1                          AND
        IFS.TileRow            <= 1                          AND
        IFS.ImageMappingID      = IMG.ImageMappingID(+)      AND
        IFS.ColorID             = C.ColorID(+)               AND
        IFS.TransformationID    = T.TransformationID(+)      AND
        IFS.LODRangeID          = LODR.LODRangeID(+)

ORDER BY       IFS.TileColumn,              IFS.TileRow,   IFS.TerrainIndexedFaceSetID,
LODR.LevelOfDetailID,  LODR.MinRange, LODR.MaxRange
```

Figure 9: Indexed face sets query.

Notable elements of this query are:

Levels of detail. A *level of detail* (LOD) is a VRML construct that helps a VRML browser to make decisions about how objects should be rendered, taking into account the distance from the viewer. It promotes efficiency and realism since in the real world one cannot see in as much detail, if at all, objects that are far away. Thus, using LOD's, only nearby objects need be drawn in full detail.

Levels of detail in VRML are depicted via `LOD` nodes. An `LOD` node is a node that contains one or more "versions" of one or more objects. An `LOD` node contains three fields. The *center* field contains an *x*, *y*, *z* coordinate in the current coordinate system. The *level* field contains one or more versions of a set of objects. The *range* fields contain a set of ranges equal to the number of versions in the level field. When the participant moves around in the virtual world depicted in the VRML file, and the distance between the participant's position and the `LOD` node's center point changes from one range to another, the version of the object drawn changes accordingly.

Information about levels of detail are stored in our schema in the `LevelOfDetail`, `LODRanges`, and `LODHierarchies` tables. The `LevelOfDetail` table contains only the center coordinate triple for each `LOD` node. The `LODRanges` table stores one or more ranges for each `LOD` node (one for each level in the LOD). An `LOD` node can be specified to appear within a specific range of another `LOD` node. These relationships are specified in the `LODHierarchies` table, which pairs `LOD` nodes with their parent `LOD` ranges.

Transformations. An indexed face set may reference a record in the `Transformations` table. If it does, a VRML *transform* node is created to contain the indexed face set. A transform node establishes a local coordinate system relative to the global coordinate system or the coordinate system of the node that contains the transform node. All coordinates within the transform node are relative to the local coordinate system.

Image mappings. The appearance of a shape is largely influenced by its texture. The `ImageMappings` table contains a list of file names which are used to

construct a reference to a file to which the VRML servlet has access and which appears in the appearance node of the corresponding indexed face set(s).

LOD query. The indexed face sets query contains enough information to determine into which LOD range an indexed face set must be placed. But it does not contain the details needed to construct the LOD's themselves. This information is obtained in the LOD query, a representative example of which is depicted in Figure 10.

```
SELECT  Distinct(IFS.LODRangeID),     IFS.TileColumn,        IFS.TileRow,
        LODR.LevelOfDetailID,         LODR.MinRange,         LODR.MaxRange,
        LOD.CenterX,   LOD.CenterY,   LOD.CenterZ,
        LODH.ParentLODRangeID

FROM    TerrainIndexedFaceSets IFS,   LODRanges LODR,
        LevelOfDetail LOD,            LODHierarchies LODH

WHERE   IFS.TileColumn         >= 1                          AND
        IFS.TileRow            >= 1                          AND
        IFS.TileColumn         <= 1                          AND
        IFS.TileRow            <= 1                          AND
        IFS.LODRangeID          =     LODR.LODRangeID(+)     AND
        LODR.LevelOfDetailID    =     LOD.LevelOfDetailID    AND
        LODR.LevelOfDetailID    =     LODH.LevelOfDetailID(+)

ORDER BY       IFS.TileRow, IFS.TileColumn, LODR.LevelOfDetailID, LODR.MinRange,
               LODR.MaxRange
```

Figure 10: Levels of detail query.

The LOD query combines the task of querying the attributes of the both the LOD's and the LOD ranges. The query returns a list of `LODRange` records, and each row contains the attributes of the LOD in which the ranges resides. There is some redundancy in the sense that, since there is a one-to-many relationship between the LOD's and the ranges, the attributes of the LOD's are returned multiple times (i.e. a number of times equal to the number of ranges in the LOD). However, doing so is more efficient than adding the overhead of an additional query since the number of ranges within each LOD is relatively small.

Coordinates query. The coordinates query obtains all of the coordinates used by all of the indexed face sets acquired in the indexed face sets query. In the VRML output, these coordinates will be listed explicitly in the first indexed face set node in the VRML for each tile. Subsequent indexed face sets will incorporate the coordinate list by reference (via the USE construct). The coordinates query is shown in Figure 11.

```
SELECT  IFS.TileColumn,               IFS.TileRow,
        IFS.TerrainIndexedFaceSetID,  P.PolyID,
        PV.OrderP,                    L.X AS VRMLX,
        L.Z AS VRMLY,                 (L.Y * -1) AS VRMLZ

FROM    TerrainIndexedFaceSets IFS,
        Polygons P,
        PolygonVertices PV,
        Vertices V,
        Locations L

WHERE   IFS.TileColumn                >= 1                            AND
        IFS.TileRow                         >= 1                            AND
        IFS.TileColumn                <= 1                            AND
        IFS.TileRow                         <= 1                            AND
        IFS.TerrainIndexedFaceSetID    = P.TerrainIndexedFaceSetID  AND
        P.PolyID                             = PV.PolyID                    AND
        PV.VertexID                          = V.VertexID                   AND
        V.LocationID                   = L.LocationID

ORDER BY       IFS.TileColumn, IFS.TileRow, VRMLX, VRMLY, VRMLZ
```

Figure 11: 3-D view coordinates query.

As previously explained, indexed face sets are made up of polygons. The coordinates query illustrates the way that polygons and their coordinates are stored. In order to render a polygon, we need to know some attributes of the vertices, including their locations and the order in which they are connected. As the query illustrates, vertices are associated with polygons via the `PolygonVertices` table. Locations can serve as vertices for more than one polygon (i.e. where polygons are connected). Therefore vertices are stored separately in the `Vertices` table.

The location of each vertex is stored in the `Locations` table. Translation of the *y* and *z* values, as shown in the SELECT clause, is necessary because of differences between the SEDRIS and VRML coordinate systems. This could have been done when loading the data, but we chose not to do so since this would have required conversions when generating the other views.

Texture coordinates query. The remaining set of information that we need in order to render the VRML is texture coordinate information. Texture coordinates are a way to specify portions of the texture image file (referred to in the section above on the ImageMappings table) to apply to faces in the indexed face set. In effect it amounts to, for each polygon in the indexed face set, cutting out a piece of the image and placing that portion over the polygon, matching the coordinates of the resulting image with the coordinates of the polygon (the latter is three dimensional and the former is two-dimensional, so the texture is "wrapped" around the polygon). The query to obtain texture coordinate information is depicted in Figure 12.

```
SELECT  IFS.TileColumn,        IFS.TileRow,    IFS.TerrainIndexedFaceSetID,
        P.PolyID,              PV.OrderP,      TC.S,
        TC.T

FROM    TerrainIndexedFaceSets IFS,    Polygons P,
        PolygonVertices PV,            Vertices V,
        TexCoordinates TC

WHERE   IFS.TileColumn                 >= 1                            AND
        IFS.TileRow                    >= 1                            AND
        IFS.TileColumn                 <= 1                            AND
        IFS.TileRow                    <= 1                            AND
        IFS.TerrainIndexedFaceSetID     = P.TerrainIndexedFaceSetID    AND
        P.PolyID                                = PV.PolyID                    AND
        PV.VertexID                     = V.VertexID                   AND
        V.TexCoordinateID               = TC.TexCoordinateID

ORDER BY       IFS.TileColumn, IFS.TileRow, TC.S, TC.T
```

Figure 12: 3-D texture coordinates query.

Java Implementation

Summary View and GML View

As mentioned in Chapter 3, the Java classes used in making the summary view and the GML view do little more than make use of the JDBC classes, and Oracle's call interface, and an additional set of classes added in this project which convert the query results to XML. The latter is done in order to perform the XSLT transformations to the desired output format (since XSLT generally requires an XML input document). Most of the rest of the processing for these views occurs in the XSLT transformation, the processing details of which are presented in the final section of this chapter.

VRML View

The bulk of the VRML processing is done in a class called `VRMLViewGen`, one instance of which per database is present in the `VRMLViewServlet`. The VRML view is built on a tile by tile basis. The smallest unit that one can create at a time is a tile. This is done in part because it is more efficient than allowing the VRML to be constructed in smaller units, since the time it takes to access the database and construct the VRML for a full tile is not proportionately greater than it takes to do so for a small subset of a tile. Also, it allows the data to be saved in well defined and limited increments (as opposed to areas bounded by random points, allowing for potentially millions of distinct areas), which makes saving the information for later reuse much easier. It is possible, however, build a range of tiles at a time (as the queries listed previously in this chapter illustrate). The Java processing architecture will be detailed in the sections below by summarizing the major classes used in creating the VRML output.

CyberVRMLV97 (CV97) package. The CV97 package [CYB2000] is a Java library that encapsulates the process of programmatically reading and writing VRML files. Each type of VRML node is encapsulated in the class hierarchy, with member variables corresponding with each of the available fields that make up the nodes. A VRML scene graph is easily created via nesting of nodes which the library accomplishes via linked lists. The classes were modified for this project primarily to make the output of default values for fields optional.

VRMLViewGen class. The `VRMLViewGen` class is instantiated once per database and persistently resides in the VRML servlet. When the servlet needs to create the tiles within a given range, it calls the `makeView` method in the `VRMLViewGen` class and passes in the tile range. The data about each tile are encapsulated in one instance per tile of the `Tile` class, which is discussed shortly. If the `makeView` method is called and a set of tiles that have *all* already been created is requested, no queries to the database need be made, and the VRML creation is accordingly greatly sped up. If a range of tiles is requested for which one or more tiles has not been requested and created, the entire range is created (since it is more efficient to query for all tiles at once, instead of breaking the request up into chunks and just getting the chunks that have not yet been created).

As the queries are processed, instances of CV97 node objects (i.e., `IFSNode`, `GroupNode`, etc.) are created. These nodes are built, sorted hierarchically, and added to one overall group node that represents the outermost group node for each tile. This work is done in the `Tile` class, which is discussed next. In the `VRMLViewGen` class, there is an array of `Tile` instances, one member for each potential tile in the database. When a

tile range is requested, these tiles are built or retrieved, and an overall `Group` node is created into which the top group of each tile is placed (and by extension all of the data in the tile). This group node becomes the outermost VRML node in a new scene graph that is sent to the browser.

Tile class. The `Tile` class is where the bulk of the VRML data are processed and stored. You will note from the VRML queries displayed earlier in this chapter that the queries all return a tile row and tile column. This is because there are distinct tile objects for each tile, and when the result set is processed the data for each row returned will potentially be added to a different member of the tiles array.

The `Tile` class is most easily explained with reference to how the query results are processed. Therefore we will revisit the representative queries shown earlier in this chapter. Also, we will introduce a few other classes that are contained in the Java implementation.

LOD query (Figure 10). Potentially all VRML data reside within a certain range of a level of detail (it is also possible that a given database may not contain LOD information, in which case the query would return nothing and all indexed face sets would be located within one group node per tile). The `LOD` nodes act as parent nodes to the shape nodes into which indexed face sets are placed. Therefore we run the LOD query first, since it gives us a chance to build the `LOD` nodes first. If we ran the indexed face set query first, it would result in more complicated code, since initially when the shape nodes are created we would have no `LOD` node in which to put them.

Each row returned in `VRMLViewGen.makeTiles` is sent to the applicable `Tile` object's `addLODRange` method (explained below). Since the query is structured

to distinctly get the LOD ranges for each tile in the range, this method will only be called once per LOD range. However we also create the `LOD` nodes at the same time (because the LOD's and the ranges are obtained in the same query). Since there are more ranges than LOD's, we will not need to create a new `LOD` node each time we add an LOD range.

The `Tile` class uses hash maps extensively because in processing the queries it repeatedly becomes necessary to look up nodes by their ids. For example, when an LOD range is created, we need to put it into its parent `LOD` node. If other ranges have already been created, the `LOD` node will have already been created, so we need a way to access the `LOD` node to associate the new range with it. Thus, when we add an LOD range, we need to either create or obtain the `LOD` node into which we will place the new range. The code in Figure 13 accomplishes this. The `LOD` node is retrieved from the hash map, based on its ID, and if a null object is returned, it indicates that the LOD has yet to be created, so it is created in the block beginning on Line 7. On Line 10, the new `LOD` node is placed in the hash map, so that the next time a range is encountered belonging to this LOD, the node will be found in the hash map.

Once the `LOD` node is obtained, we can create the new `Group` node, representing the range that we are adding (there is no such thing as a "Range" node, so we use a `GroupNode` object to represent each range), which ultimately we will add to the LOD. The query results contain the minimum and maximum range values. A slight diversion on how ranges work is necessary here.

An `LOD` node in VRML contains a number of ranges one greater than the number of range values. For example, if we are given range values of 2000 and 4000 for a given LOD, we will need to create three ranges. This is because it is given that there is always

```
/** Get an LOD node (from the hash map) using its LevelOfDetailID.  It will be
*  created if it does not already exist
*/
1.  private LODNode getLODNode(Integer lodID, Integer parentLODRange)
2.  {
3.    // LODNode = CV97 LODNode
4.    // lodMap  = Java HashMap
5.    LODNode lodNode = (LODNode)lodMap.get(lodID);
6.    //create it if necessary and put it in the hashmap
7.    if (lodNode == null)
8.    {
9.      lodNode = new LODNode();
10.     lodMap.put(lodID, lodNode);     //each LOD is hashed by id
11.     lodNode.setName("LODID" + lodID.intValue());  //give node meaningful name
12.
13.
14.     //each LOD will have at least one empty group
15.     GroupNode emptyGroup = new GroupNode();
16.
17.      emptyGroup.setName("RANGE_" + "0");

18.      //add child node sets the hierarchy in CV97, internally the relationship is
19.      //represented as a linked list
20.      lodNode.addChildNode(emptyGroup);

     // this LOD node may be nested within an LOD range of another LOD.  This
     // is specified via the LODHierarchies table, which is queried in the LOD
     // query.  Each CV97 node instance has an Object placeholder for data, which
     // we use to store the parent LOD range id...later we will add the LOD
     // to its parent node..we cannot do it now because we do not know that it
     // has been created yet..we only have the ID at this point
21.     lodNode.setData(parentLODRange);
  }

22.    return lodNode;
}
```

Figure 13: `Tile` class `getLODNode` method.

a range representing a distance of zero to the first given value, and another from the highest value to infinity. Thus, in the example, the ranges would be from 0-2000, 2000-4000, and 4000-infinity. As a result of the above if an LOD is processed from the database for which there is no range with a minimum of zero, one must be added (likewise one must be added if no range exists with a maximum of infinity, infinity being represented in the database by a sentinel value).

It follows from the above that every LOD has at least one implied range from 0 to infinity (which doesn't really mean anything to the browser since the distance from the viewpoint would always be within that range). Therefore as soon as we create an LOD

we create an LOD range to represent that level. When we process the results of the LOD query and we create the first new range for the LOD, whether we use the original empty range or not is dependent on the minimum value of the new range. If it is zero, then we use the original range (because the implied minimum of the original one is zero). If the specified maximum of the new range (included in the query results) is infinity, we know we will not need any more ranges for this LOD. Otherwise, the specified maximum of the new range will then become the minimum of a new, empty (temporarily, at least) range which will have a maximum of infinity.

If the minimum of the new range is not zero, the original one will remain empty permanently. We know this because the query sorts the ranges by their minimum values, so there cannot be a range with a minimum of zero if the first one has a minimum larger than zero.

Once a `Group` node has been created for the new range, the node is added to its `LOD` node. LOD ranges are also kept in a hash map by id for retrieval when needed to add their child nodes (primarily indexed face sets and their related nodes).

At this point we have reviewed the processing in the Java classes of the LOD query results. Next we will review the processing of the indexed face sets query.

Indexed face sets query (Figure 9). The indexed face sets (IFS) query works similar to the LOD query in that each row of results potentially deals with a different tile in the Tiles array. In this case, for each new row we add a new IFS to the tile. This process is only more complicated than that for adding LOD's in the sense that there are more attributes to be concerned with when it comes to creating indexed face sets.

When adding an IFS, potentially several new nodes may need to be created. These may include `Shape`, `Geometry`, `Appearance`, `Collision` (a node which detects and informs the browser that the viewer has run into something in the VRML scene), `Billboard` (a grouping node which causes all objects contained within it to rotate as the viewer moves around so that they are always facing the viewer), and `Transformation` nodes. Several of these are optional (i.e. the data may or may not contain them), but the ones that are included must be nested according to a prescribed hierarchy. Establishing this hierarchy in the code was done by creating each node as necessary from outermost to innermost, and then adding each node to a locally created vector. Only when all nodes for the IFS have been created are the parent child groupings created by going through the vector in order and establishing a parent-child relationship for each adjacent node in the list.

The following outlines the steps taken to add an IFS to a tile:

1. If the IFS is contained in an `LOD` node (which we will know if the query results contain a non-zero and non-null `LevelOfDetailID`), we retrieve the `LOD` node from the LOD hash map.

2. If the IFS is contained in `Transformation`, `Billboard`, and/or `Collide` nodes, we create these in turn and add them to the hierarchy list.

3. We create the `Shape` node (all `IFS` nodes are inside `Shape` nodes) and add it to the hierarchy list.

4. We create a new appearance node which will be added to the `Shape` node.

5. We create and set the attributes of a new `Color` node if there is a `ColorID` associated with the IFS.

6. If the results reveal an image mapping, we create an image texture node and set its url equal to the file name included in the results.

7. We create the IFS node and add it to a hash map that maps `IFS` nodes by their ID.

8. We make `Coordinate` and `TextureCoordinate` nodes for the IFS and add them to the `IFS` node (if we are adding the first IFS in the tile, we make new ones, otherwise we apply the `USE` construct which allows us to incorporate by reference the ones in the first `IFS` node).

9. We set various attributes of the IFS, then add the `Material` and `Appearance` nodes to it.

10. We set the hierarchy of the new nodes based on the values in the hierarchy list.

11. If there is a non-zero value for the ID of the LOD range that should contain the new IFS, we obtain this node from the hash map containing the ranges and add the result of the above steps to the range. If there is no containing range, then we add the result to the group node that represents the tile itself.

As a result of processing the LOD and IFS queries, the structure of the VRML is set. What remains is to add the coordinates and texture coordinates. Although these represent a large part of the volume of a typical VRML file, adding these values is not much more complex than processing the previous two queries.

Coordinates query (Figure 11). There are two main tasks involved with processing the coordinates for the indexed face sets in a tile. As stated previously, all coordinates for all of the sets in a tile are listed in a Coordinate node in the first IFS in the tile. So the first task is to get all of the coordinates for the whole tile and put them in a Coordinate node with the first IFS in the tile. The polygons that make up each IFS are specified by listing their coordinates in a `coordIndex` node inside each IFS. This is the second task, and it is done by printing an index of the coordinate into the coordinate list in the first IFS instead of by printing the actual coordinate.

In order to assist in sorting and comparing coordinates and texture coordinates, a class called `IFSLocation`, which implements the Java `Comparable` interface, has

been created. Sub-classes of `IFSLocation` for coordinates and texture coordinates have been created to encapsulated these two types. In addition to the coordinate point values, the `IFSLocation` class also stores the `IndexedFaceSetID`, `PolyID` and `OrderP` fields associated with each location.

The coordinate values for a given tile are stored in a Java vector as instances of the `Coordinate` class. Filling up this vector is simply a matter of cycling through the results of the coordinates query and, once for each record, adding a coordinate triple to the coordinates vector in the appropriate tile.

After all coordinates have been added (and all queries have been processed), another method is called to setup the coordinate nodes and to create the polygons that go into the indexed face sets. This involves iterating through all of the `Coordinate` objects and performing the following steps:

1. Using the IFS ID member of the `Coordinate` object, obtain the `IFS Node` from the IFS hash map.

2. Add to the `Coordinate` object a reference to the `IFS` Node obtained in the previous step.

3. For each distinct (with respect to *x*, *y*, and *z* values) coordinate, add a point to the first `IFS` node in the tile. This creates the master coordinate list which all polygon coordinates will reference.

4. As the master list is created, an index number is used to keep track of how many coordinates have been added. Each coordinate is given an index number into this list so that it will be available as each polygon using the coordinate is created.

The code that accomplishes these steps is shown in Figure 14 . This code, and the code in the next section that establishes the coordinate index lists, highlights the reason that it was ultimately chosen to create the VRML in Java without using XSLT. With large numbers of coordinates (easily tens of thousands), XSLT did not provide a

```
private void setupCoordinates()
{
  int idx = -1;
  boolean throughList = false;

  //coords are ordered in the query by IFSID, X, Y, and Z

  //get an enumeration of all of the coordinate objects in the tile
  Enumeration e = coords.elements();

  Coordinate priorLocation = null;


  while (e.hasMoreElements())
  {
    Coordinate nextLocation = (Coordinate)e.nextElement();

    //get the coordinate's IFS node
    IndexedFaceSetNode ifsNode =
           (IndexedFaceSetNode)ifsMap.get(new Integer(nextLocation.getIFSId()));

    if (ifsNode == null)      //all coords should have an IFS id
     {//fatal error handling omitted}

     //set an internal reference in the coord to its IFS node
     nextLocation.setIFSNode(ifsNode);
     //since coordinates can appear in multiple IFS's, they will also appear
     //multiple times in the array, but we only want to list them once in the
     //coordinate list
     if (!nextLocation.equals(priorLocation))
     {
        idx++;

        //when we created the IFS nodes, we saved a reference to the first one
        //so we can add coordinates to it as in the following
        firstIFSNode.getCoordinateNodes().addPoint(nextLocation.getX(),
             nextLocation.getY(), nextLocation.getZ());
        priorLocation = nextLocation;
      }
      //set the index for this coordinate in the coordinate list
      //we do this for all coordinates, so now once we print out the coordinate
      //as a polygon coordinate, it will be easy to get the index
      nextLocation.setIndex(idx);
  }
}
```

Figure 14: VRML coordinate setup.

convenient and, more importantly, efficient way that we could find that would allow processing and sorting these numbers in an acceptable amount of time. This code works well because of Java's ability to encapsulate references to objects (in this case nodes and coordinates). XSLT's tree based model could not process in acceptable time the moderate amount of data we used for evaluation purposes. For example, processing an

averaged sized set of tiles using XSLT was found to take over ten minutes even without processing the texture coordinates.

What is left in processing the coordinates is the creation of the geometry (that is, the polygons) of the IFS's. This involves the steps described below(shown in Figure 15).

```
private void setupCoordIndices()
{
  Collections.sort(coords);  //this sorts by polygon id
  //get all coordinates in the tile
  Enumeration e = coords.elements();

  int currentPoly = -1;  //keep track of the current polygon
  IndexedFaceSetNode currentIFSNode = null;
  while (e.hasMoreElements())
  {
    Coordinate nextLocation = (Coordinate)e.nextElement();
    int thisPoly = nextLocation.getPolyID();

      // if we have a new IFSnode, we need to 'close' the last
      // polygon on the last IFS
    if (nextLocation.getIFSNode() != currentIFSNode)
    {
      if (currentIFSNode != null)  //unless this is the first IFS node
        currentIFSNode.addCoordIndex(-1);
      currentIFSNode = nextLocation.getIFSNode();
      currentPoly = -1;
    }
    if (currentPoly != thisPoly)
    {
      if (currentPoly != -1)
        currentIFSNode.addCoordIndex(-1);
      currentPoly = thisPoly;
    }
    currentIFSNode.addCoordIndex(nextLocation.getIndex());
  }
  if (currentIFSNode != null)
    currentIFSNode.addCoordIndex(-1);
}
```

Figure 15: Coordinate index setup.

1. We sort the coordinate list by polygon id. Within each tile, the coordinates have, up to this point, been sorted by location. This was, in part, to make it easy to avoid duplicating coordinates in the master coordinate lists. Now, though, within each IFS node, we need to list the coordinate indices for each polygon, one polygon at a time (and we need to place a "-1" value at the end of the coordinate index list for each polygon to delineate the end of the list for each polygon). Naturally, retrieving the coordinates from the list by polygon ID each time we create a new polygon would not be acceptable, since it would mean the list would need to be searched once for each polygon, resulting in near exponential time to create the polygons.

2. Iterate through the newly sorted list and keep track of the polygon id value of each member of the list.

3. Add the index of each coordinate to the IFS node with which the coordinate's polygon is associated.

4. When the next coordinate in the list has a new different polygon id than the previous, a new polygon list is started, so a "-1" is added to the coordinate index list of the associated IFS.

Texture coordinates query (Figure 12). Texture coordinates and their index setup works very similar to coordinates. This is not surprising since in the database there is one texture coordinate for each polygon coordinate. In fact, the steps and the Java code are almost identical to the steps and code shown above for the coordinates, so they will not be listed separately.

XSLT Implementation

In this section we provide the details of the XSLT implementation which, for purposes of this thesis, covers the production of the SVG and GML output from the results of the database queries. The relative simplicity of the stylesheets presented here is arguably due to both the elegance of XSLT and the relative simplicity of the two view types for which it is being used compared to the VRML production process.

Summary View Stylesheets

Since the summary view queries return only a total count value, one per tile, the output is relatively simple and, as a result, so are the stylesheets. As stated previously, transmission of the SQL query and conversion of the results to XML is performed by the Java classes. An example of the XML generated from this process is contained in Figure

16. The XML results consist of one count value per tile. To transform these results, one or both of two stylesheets is used, depending on the format requested.

```
<?xml version='1.0'?>
<!--Text of query:

"SELECT IFS.TileRow, IFS.TileColumn, count(P.PolyID) AS TERRAINCOUNT

 FROM   Polygons P, TerrainIndexedFaceSets IFS

 WHERE P.TerrainIndexedFaceSetID = IFS.TerrainIndexedFaceSetID      AND
       IFS.TileRow              >= 1                                AND
       IFS.TileRow              <= 2                                AND
       IFS.TileColumn           >= 1                                AND
       IFS.TileColumn           <= 2

 GROUP BY IFS.TileRow, IFS.TileColumn

 ORDER BY IFS.TileRow, IFS.TileColumn"-->

<!—header information omitted -->

<ROWSET>
  <ROW>
        <TILEROW>1</TILEROW>
        <TILECOLUMN>1</TILECOLUMN>
        <TERRAINCOUNT>13</TERRAINCOUNT>
  </ROW>
  <ROW>
        <TILEROW>1</TILEROW>
        <TILECOLUMN>2</TILECOLUMN>
        <TERRAINCOUNT>32</TERRAINCOUNT>
  </ROW>
  <ROW>
        <TILEROW>2</TILEROW>
        <TILECOLUMN>1</TILECOLUMN>
        <TERRAINCOUNT>32</TERRAINCOUNT>
  </ROW>
  <ROW>
        <TILEROW>2</TILEROW>
        <TILECOLUMN>2</TILECOLUMN>
        <TERRAINCOUNT>29</TERRAINCOUNT>
  </ROW>
</ROWSET>
```

Figure 16: Summary view query results after conversion to XML.

XML results. The intermediate XML results are always computed (whether or not SVG output is desired). The XSLT code that creates the XML result is simply prints out the count element for each row. The tag for the count element varies with the count

type. The XSLT that performs this work, is shown in Figure 17. Finally, the XML transformation results (using the Saxon XSLT processor) are shown in Figure 18.

```
<!—grid_type is the count type passed to the stylesheet as a parameter -->
<xsl:for-each select="//ROW">
   <gridelement row_index="{./TILEROW - 1}" column_index="{./TILECOLUMN - 1}">
      <xsl:choose>
         <xsl:when test="$grid_type='terrain_count'">
            <xsl:value-of select="./TERRAINCOUNT"/>
         </xsl:when>
         <xsl:when test="$grid_type='polygon_count'">
           <xsl:value-of select="./POLYGONCOUNT"/>
         </xsl:when>
         <xsl:otherwise>
           <xsl:value-of select="./INSTANCECOUNT"/>
         </xsl:otherwise>
      </xsl:choose>
   </gridelement>
</xsl:for-each>
```

Figure 17: XSLT code to generate summary view XML output.

```
   <gridelement row_index="0" column_index="0">13</gridelement>
   <gridelement row_index="0" column_index="1">32</gridelement>
   <gridelement row_index="1" column_index="0">32</gridelement>
   <gridelement row_index="1" column_index="1">29</gridelement>
```

Figure 18: Summary view XML output.

In most cases SVG output will be desired, so the user can determine where in the database high concentrations of polygons are located. The SVG stylesheet creates the SVG output from the XML results of the first transformation and not directly from the query results. This makes the stylesheets more manageable (changes in the queries necessitate only that one stylesheet be changed). Moreover, there is no measurable loss of efficiency since producing the summary view requires negligible processing time.

The SVG transformation is initiated directly from the results of the initial transformation (i.e. within the same call to the Saxon XSLT processor). This occurs via a

directive to that effect inside the first stylesheet. The notable portion of the SVG stylesheet is shown in Figure 19.

```
<xsl:for-each select="gridelement">
  <rect> <!-- one rectangle per tile -->
         <!-- set the x and y positions of the tile's cell -->
    <xsl:attribute name="x">
      <xsl:value-of select="@column_index * $column_width"/>
    </xsl:attribute>
    <xsl:attribute name="y">
      <xsl:value-of select="($row_count - @row_index - 1) * $row_height"/>
    </xsl:attribute>

         <!-- all cells are the same width and height -->
    <xsl:attribute name="width">
      <xsl:value-of select="$column_width"/>
    </xsl:attribute>
    <xsl:attribute name="height">
      <xsl:value-of select="$row_height"/>
    </xsl:attribute>

 <!-- calculate the count value for the current tile relative to the rest of
          the tiles...this is how the color of the cell is determined -->
    <xsl:variable name="ratio">
      <xsl:value-of select="(. - $min) div  ($max - $min)"/>
    </xsl:variable>

     <!-- color the cell based on the ratio -->
    <xsl:choose>
      <xsl:when test="$ratio &lt; .25">
      <xsl:attribute name="style">fill:rgb(0, <xsl:value-of select=
                      "(.5 + $ratio * 2) * 255"/>, 0)
      </xsl:attribute>
      </xsl:when>
      <xsl:when test="$ratio &lt; .5">
        <xsl:variable name="square_root" select="math:sqrt(($ratio - .25) * 4)"
                         xmlns:math="http://Java.sun.com/Java.lang.Math"/>
       <xsl:attribute name="style">fill:rgb(<xsl:value-of select="$square_root *
                                    255"/>, 255, 0)</xsl:attribute>
      </xsl:when>
      <xsl:when test="$ratio &lt; .75">
         <xsl:attribute name="style">fill:rgb(255, <xsl:value-of select="
                       (3 - $ratio div .25) * 255"/>, 0)
         </xsl:attribute>
      </xsl:when>
      <xsl:otherwise>
         <xsl:attribute name="style">fill:rgb(
           <xsl:value-of select="(2.5 - 2 * $ratio) * 255"/>, 0, 0)
         </xsl:attribute>
      </xsl:otherwise>
    </xsl:choose>
  </rect>
</xsl:for-each>
```

Figure 19: XSLT code to create SVG summary view grid cells.

The SVG stylesheet creates a two-dimensional table of squares, one square representing each tile in the range. Each tile is colored with a color that is redder the higher the count value for the tile. The coloring scheme was provided by Vcom3D. Notice that it calls in two cases for the computation of an intermediate square root value. This could have been computed with a function inside the XSLT stylesheet, but this would have resulting in adding another method to the stylesheet (or calling another stylesheet with such a method) that is several lines long, so it makes sense here to use Java's static function to compute the square root. This is an illustration of the convenience of using an XSLT processor that runs inside the Java virtual machine. Virtually any static Java method can be taken advantage of from within the stylesheet.

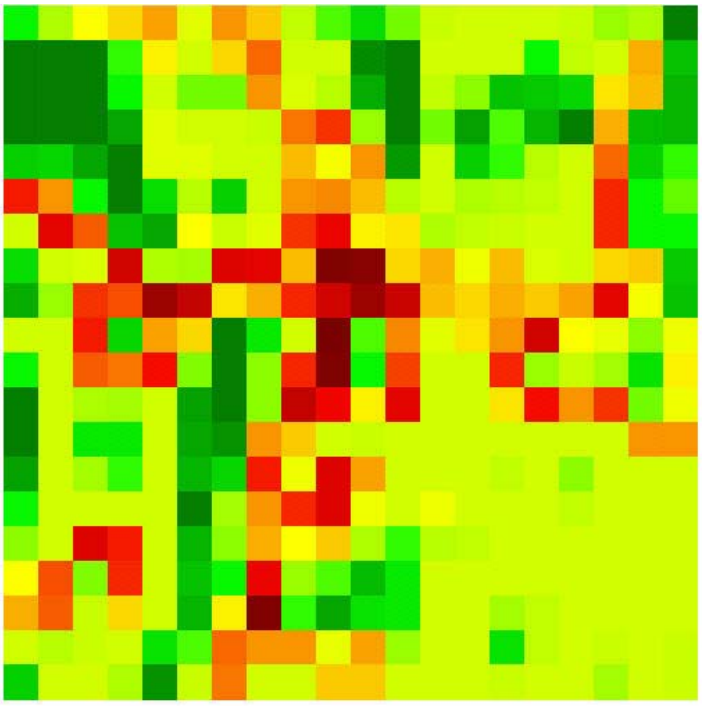

Figure 20: Summary view in SVG.

The SVG output (as rendered in the Adobe SVG viewer) that results from a transformation of a terrain count of all tiles in the Bellevue database is shown in Figure 20. This example is typical of how the data for a city might look, with the highest concentration of objects existing in the center.

2-D View Stylesheets

The overall process involved with creating GML or SVG for the 2-D view is similar to the summary view. There is a set of queries that gets sent to the database, the results are converted into XML, then the results are transformed into GML and, if the user requests SVG output, into SVG using XSLT stylesheets.

Intermediate XML results. Recall that the user has the option of requesting GML based information on any combination of areas, lines and points in the database. The queries for these three are structured similarly. In each case there is a separate query that gets the area, line and point information for the selected tile range, and then a distinct query that gets information about the properties in those features. Here to illustrate the process we will show some examples from an areas query.

Raw XML output. Recall from earlier in this chapter that the query to obtain the areas returns one row for each point in the area. Thus the entire XML output for any area in most cases will be hundreds of elements or more. A representative query and XML results is shown in Figure 21.

```
<?xml version='1.0'?>

<TWODQUERY>
   <!--Text of query:

"SELECT    A.AreaID, PS.PropertySetID AS PSID, L.X || ',' || L.Y AS APOINT,
           PS.Description AS PSDesc

FROM       Areas A, PropertySets PS, Locations L, AreaLocations AL,
           AreaBoundingBox ABB

WHERE      A.PropertySetID  = PS.PropertySetID      AND
           A.AreaID         = AL.AreaID             AND
           AL.LocationID    = L.LocationID          AND
           A.AreaID         = ABB.AreaID            AND
           NOT(   (ABB.MaxX <= 6000)         OR
                  (ABB.MaxY <= 6000)         OR
                  (ABB.MinX >= 7000)         OR
                  (ABB.MinY >= 7000) )
ORDER BY A.AreaID, AL.OrderP"-->

<AREAS>
   <AREA>
      <AREAID>13</AREAID>
      <PSID>42</PSID>             <!-- Property Set ID -->
      <APOINT>3750,4000</APOINT>  <!-- Location coordinate pair -->
      <PSDESC>Geographic Information Area</PSDESC><!--Property Set description -->
   </AREA>
   <AREA>
      <AREAID>13</AREAID>
      <PSID>42</PSID>
      <APOINT>3625,4000</APOINT>
      <PSDESC>Geographic Information Area</PSDESC>
   </AREA>
   <AREA>
      <AREAID>13</AREAID>
      <PSID>42</PSID>
      <APOINT>3625,3750</APOINT>
      <PSDESC>Geographic Information Area</PSDESC>
   </AREA>
</AREAS>
```

Figure 21: 2-D Areas query and results after conversion to XML.

The XML in Figure 21 provides enough information to create the GML area features and to identify which points belong with each area. The only data still needed to complete the areas are the properties that make up each of the property set identified with the area. Figure 22 shows the query and XML results containing this data. The figure

```
<!--Text of query:

"SELECT DISTINCT(P.PropertyID) AS PID, PS.PropertySetID AS PSID,
        P.Description AS PDesc, P.Units, P.ValueP, P.ValueType AS Type
FROM    Areas, PropertySets PS, Properties P, AreaBoundingBox
WHERE  Areas.PropertySetID = PS.PropertySetID       AND
       PS.PropertySetID    = P.PropertySetID        AND
       Areas.AreaID        = AreaBoundingBox.AreaID AND
       NOT( (AreaBoundingBox.MaxX <= 6000)   OR
            (AreaBoundingBox.MaxY <= 6000)   OR
            (AreaBoundingBox.MinX >= 7000)   OR
            (AreaBoundingBox.MinY >= 7000) )"     -->

<APROPS>
  <APROP>
    <PID>125</PID>     <!-- Property ID     -->
    <PSID>42</PSID>    <!-- Property Set ID -->
                                          <!-- Property Set Description -->
      <PDESC>Surface Trafficability Group; SIMNET Mobility Model</PDESC>
      <UNITS>Units Enumeration</UNITS>      <!-- Units of measure -->
      <VALUEP>3</VALUEP>                    <!-- Property value -->
      <TYPE>PDV EDCS AC ENUM VALUE</TYPE>   <!-- Data type -->
   </APROP>
</APROPS>
```

Figure 22: 2-D Properties query and results converted to XML.

shows all of the results for this property query example (this property set only has one property associated with it). The query in effect gets all of the properties for all of the property sets identified with all of the areas within our range (which in this example is the box created using two dimensional coordinate `6000, 6000` in the lower left and coordinate `7000, 7000` in the upper right.

GML stylesheet. The XSLT stylesheet that produces the GML output for the above example is shown in the Appendix. In this example one can see the simplicity of the XSLT stylesheet processing model. The stylesheet simply processes each unique area, adds some GML tags, then adds the properties and boundary ring and outputs the resulting GML. Most users, however, will choose to view an SVG rendering of the results, so one more stylesheet is needed.

SVG stylesheet. As was the case with the summary view, the SVG stylesheet takes as input not the query results but the GML output. The SVG stylesheet creates a "map" from the GML by defining a color for each feature type, drawing each feature on the map (using the boundaries of the areas, the points on the lines, and the individual points) and filling in or drawing the features using the specified color. The area in the example above, then, would be filled in as shown in Figure 23.

```
<--For each element containing a Polygon element a SVG polygon element is
    created with the specified @color.  Do not confuse the "Polygon" tag used
     here with the polygons in our database schema.  Here by "Polygon" we
     refer to the linear ring that makes up the entire perimeter of one area-->

<xsl:for-each select = "./geometricProperty/Polygon">
  <xsl:element name = "polygon">
    <xsl:attribute name = "style" >fill:
      <xsl:value-of select="$color"/>
    </xsl:attribute>
    <xsl:attribute name = "points" >
      <xsl:value-of select="./outerBoundaryIs/LinearRing/coordinates"/>
    </xsl:attribute>
  </xsl:element>
</xsl:for-each>
```

Figure 23: Creation of SVG area in XSLT.

This shows the creation in the SVG output of an SVG "polygon" element, which in this case is a filled in ring. The lines and points are created similarly. Each line feature results in the creation of SVG "polyLine", which is a series of dots connected with a line of a given width. The line is filled in with the specified color from the SVG stylesheet. The point features are created by making a small square filled in with the specified color. Figure 24 shows how these features look together. It shows the full SVG map for the entire Bellevue database (all features requested for all tiles) .

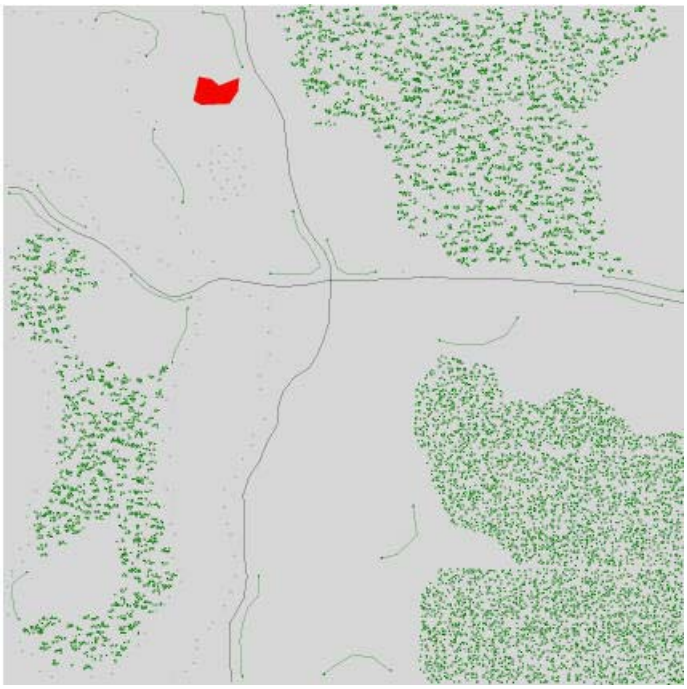

Figure 24: Bellevue SVG output as viewed in Adobe SVG plug-in.

CHAPTER 5
EVALUATION

In this thesis we have endeavored to describe, develop and implement an architecture which will efficiently support the querying and rendering of geo-spatial data in real time and in various formats. Now that we have described our implementation, we will evaluate the results. We will do so in relation to the specific goals mentioned in Chapter 1. Our desire in this evaluation is to assess how well these goals were met.

Database Schema

The problem that we sought to solve by creating the database schema and storing the geo-spatial data in an Oracle database was the fact that there was no way to query a SEDRIS transmittal in such as a way as to gain acceptable performance. Given the structure of a SEDRIS transmittal, any of the representative queries that we need to run to generate the required output will take literally hours, regardless of platform and hardware, if we do so by using the SEDRIS file structure and API. Using our database schema, we can assess the schema and queries by running each view type for the entire amount of data in the Bellevue database and timing the response. This means running a summary view for all twenty tiles (for this evaluation we will select terrain count and will not apply a description filter), a GML view for the entire extent of the database and requesting all of areas, lines and points, and, finally, running a VRML view for all tiles.

All of the evaluations in this chapter were run on a Windows 2000 system with 256 MB of RAM running Oracle 8i, locally, with an AMD Duron 750 Mhz processor.

To run this evaluation we will isolate the time taken to transmit the queries to the database and receive the results. The measured running times are shown in Table 1.

Table 1: Raw query times for Bellevue database

View Type	Query Time
Summary View	170 ms
GML View	741 ms
VRML View	51,343 ms

The query times for the first two views are negligible. The time for the VRML view is much larger, which is due to the fact that tens of thousands of points needed to be retrieved (about 98% of the query time was used by the coordinates and texture coordinates). Considering this and the time in relation to current methods, the query time is well within acceptable limits and represents a vast improvement over current methods.

Overall Processing Efficiency

Getting the data is only one part of the view generation process. Our main goal was to produce output for the whole or parts of the database in an efficient manner. As previously stated, under current methods it takes hours to generate the data and, as such, the ability to access portions of it in real time was absent. Another problem was the fact that specialized hardware and software was required. Therefore if we can produce the output using our proposed architecture even in several minutes this might be seen as an acceptable improvement.

To evaluate the overall efficiency we will run requests for each of the views. We will run the evaluation using parameters which we believe will represent a typical user

request. For this evaluation we will use the a database referred to as "CCTT", which represents a large, dense, area in Germany covering about 64 square kilometers. The information in this database is about ten times more voluminous than the Bellevue database.

In the case of the summary view and the GML view, we will request as much as can be requested. For the summary view, this means all tiles, all property set descriptions, and the terrain count. For the GML view, this means areas, lines, and points for the entire extent of the database (also including all descriptions). For the VRML view we will process one tile. The results are shown in Table 2.

Table 2: Total times for selected queries in the CCTT database

View Type	**Total Time**
Summary View	3375 ms
GML View	63,992 ms
VRML View	78,625 ms

The results here once again show significant improvement over current methods. In each case large chunks of data were queried, and the "worst" case, a request for a VRML file of over one megabyte in size, was a time of well under a minute and a half.

Also note that, as part of the implemented architecture, these improved times represent only the *first* request for the data (that is, the first request after the web server is restarted). Subsequent requests, even if from a different user, are returned virtually immediately due to the use of a caching servlet which stores the VRML for each tile.

Accuracy and Data Integrity

It would have served little benefit to make a swift new way of querying geo-spatial data if the results were erroneous. Thus, an overriding goal was to produce correct results and to maintain the integrity of the data. Vcom3D provided us with sample output produced using the current means. We also provided them with the output from this application in order for them to evaluate the results for correctness. In doing so we have relied on Vcom3D's familiarity with the underlying information and how it should be represented under each view type.

The result of these evaluations was positive. The sample files and the output from our application are formatted identically. Further, any discrepancies which Vcom3D noted from their evaluation were easily rectified and they reported no problems that would call into question the reliability of the approach.

Finally, it is apparent that the implemented architecture is much more structurally reliable than the current methods. The use of an RDBMS introduces a way to enforce integrity that is not present in a specialized binary file structure such as a SEDRIS transmittal. This will reduce the likelihood of undetected errors occurring in loading the data and it will minimize the chance of undetected corruption occurring once it has gone live.

Flexibility and Maintainability

Another goal was to build an application that is flexible enough to be adaptable to changing standards. Our architecture is based on several standards-based technologies. It presents data in formats that will change with these standards. We want to be able to easily modify the architecture to support such modifications. In order to evaluate this

aspect we will consider a few likely developments and assess how we would need to modify the application.

Presentation modifications. Modifications in the output format could be dictated by changes in standards or the needs and desires of users. Suppose for instance that one wanted to provide a more meaningful presentation of the summary view by adding a legend showing the range of colors that would appear for various ranges of count values. Such an addition would be relatively easy under the current architecture due to the fact that the presentation is controlled by the XSLT transformation. One could modify the stylesheet to provide this information and it would not be necessary to recompile or redeploy anything.

Possibility of changing standards. Similarly, suppose the GML standard changed so that the structure and tag names of the GML output needed to be modified. Once again, since the output is controlled by the XSLT transformation, such modifications can be made simply by editing a text file with no need to recompile any part of the application. This operational efficiency is more of a benefit in cases such as these where the expertise of those hosting and maintaining the application is expected to be in the area of how the information needs to be presented rather than in application programming languages.

VRML flexibility and maintainability. With respect to the VRML output generation, we cannot claim to have achieved the same level of presentation flexibility (compared to the other two views). This is because the processing required to produce the VRML was not suited to XSLT and therefore the VRML generation task was restricted to the use of Java classes. However, we have replaced the use of a

complicated, specialized API with an object-oriented approach using a VRML library and some new Java classes adapted to the needs of this application. These factors should lead to a much more maintainable application with respect to the VRML generation as well.

CHAPTER 6
CONCLUSIONS

Summary

In this thesis we have described an architecture that we have developed and implemented for the purpose of storing, retrieving and presenting geo-spatial information. We began by describing the technologies that play an important role in this architecture. These technologies include a relational database management system, XML and related technologies, and the Java programming language. We indicated the reasons why we chose to use these technologies over other alternatives.

We then described the architecture and how these technologies were combined to achieve the desired goals. We examined four key elements of the architecture and specified how they fit into the architecture. Then, in Chapter 4, we gave the details of how each of these elements was implemented. We showed how XSLT was used to generate the summary view and the 2-D view and showed example output. We then explained how the VRML output was achieved using a set of Java classes that were developed for this purpose along with some existing Java libraries. At each step along the way, we gave the reasons for the design and implementation decisions that were made.

At the outset we stated that among our goals was to develop a system for accessing geo-spatial information in real time, i.e. more efficiently than under prior methods. We also wanted an architecture that would be maintainable and flexible enough

to allow it to easily be adapted to changing standards and user needs. In the previous chapter we presented test running time results and discussed ways in which our implementation met the needs of maintainability and flexibility. We thereby concluded that we generally succeeded in improving on the efficiency and flexibility of current methods.

Future Work

The architecture we have described provides a solid foundation for a user-friendly, virtual reality experience. Although the results presented in the previous chapter reflect a tremendous improvement over current methods, with some of the output taking in excess of a minute to produce, there is still room for improvement.

In processing the database queries, the summary and GML views store the returned results internally as a Java string prior to converting to XML (which is also stored internally as a string prior to being sent to the XSLT transformation). The use of Java piped input and output streams to transmit these results could result in some improvement. Even more of a potential related improvement would be to use XSLT functionality that would allow the queries to be executed and processed within the XSLT stylesheets. This should result in faster processing and better maintainability (since it would further simplify the Java classes and bring more of the processing into the stylesheets). Oracle's XSQL could be used to accomplish this. This functionality, however, is not part of the XSLT standard. Therefore, a potential drawback is increasing dependence on proprietary or other non-standards-based technology.

Another desired improvement is in the user interface. The current servlet implementation uses distinct servlets for each view type. It does not allow one to easily

switch from one presentation type to another. A more integrated HTML frames-based interface is being designed for this purpose. This will require little changes to the internal processing architecture. That the interface can be redesigned without greatly modifying the implementation is an indication that the architecture is adaptable and extensible.

APPENDIX
GML STYLESHEET

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
                xmlns:saxon="http://icl.com/saxon">

<xsl:output doctype-system="stylesheets\gmlfeature.dtd" indent="yes"/>

<xsl:param name="xMin"/>
<xsl:param name="yMin"/>
<xsl:param name="xMax"/>
<xsl:param name="yMax"/>


<xsl:template match="/">

    <FeatureCollection typeName="Root">
        <boundedBy>
            <Box>
                <coordinates>
                    <xsl:value-of select="concat($xMin, ',', $yMin, ' ', $xMax,
                                                 ',', $yMax)"/>
                </coordinates>
            </Box>
        </boundedBy>
        <xsl:apply-templates select="/TWODQUERY/AREAS"/>
        <xsl:apply-templates select="/TWODQUERY/LINES"/>
        <xsl:apply-templates select="/TWODQUERY/POINTS"/>
    </FeatureCollection>
</xsl:template>

<xsl:template match="/TWODQUERY/AREAS">

<xsl:for-each select="./AREA/AREAID[not(.=preceding::AREA/AREAID)]">

    <xsl:variable name="currentID">
          <xsl:value-of select="."/>
    </xsl:variable><xsl:text>
</xsl:text>

<xsl:comment>AreaID: <xsl:value-of select="$currentID"/></xsl:comment>

    <featureMember typeName="RootMember">

        <xsl:variable name="desc">
            <xsl:value-of select="../PSDESC"/>
        </xsl:variable>
        <xsl:variable name="propertySetID">
           <xsl:value-of select="../PSID"/>
        </xsl:variable>

        <Feature typeName="{$desc}">

            <xsl:for-each select="/TWODQUERY/APROPS/APROP[PSID=$propertySetID]">
                <xsl:call-template name="propertyPrint"/>
                </xsl:for-each>

            <geometricProperty typeName="extentsOf">
        <Polygon srsName="UTM">
            <outerBoundaryIs>
```

```
                <LinearRing>
                    <coordinates><xsl:text>
                                </xsl:text>
                      <xsl:for-each
                          select="/TWODQUERY/AREAS/AREA[AREAID=$currentID]/APOINT
                             [not(.=preceding::AREA[AREAID=$currentID]/APOINT)]">
                            <xsl:value-of select="../APOINT"/><xsl:text>
                                                          </xsl:text>
                               <xsl:if test="(position() mod 4)=0">
                                   <xsl:text>
                                   </xsl:text>
                               </xsl:if>
                        </xsl:for-each>
                        <xsl:value-of
select="/TWODQUERY/AREAS/AREA[AREAID=$currentID]/APOINT[1]"/><xsl:text>
                    </xsl:text>
                    </coordinates>
                </LinearRing>
            </outerBoundaryIs>
        </Polygon>
         </geometricProperty>
     </Feature>

 </featureMember>

</xsl:for-each>
</xsl:template>


<xsl:template match="/TWODQUERY/LINES">

<xsl:for-each select="./LINE/LINEID[not(.=preceding::LINE/LINEID)]">

    <xsl:variable name="currentID">
          <xsl:value-of select="."/>
    </xsl:variable>

<xsl:text>
</xsl:text>
<xsl:comment>LineID: <xsl:value-of select="$currentID"/></xsl:comment>

<featureMember typeName="RootMember">

     <xsl:variable name="desc">
          <xsl:value-of select="../PSDESC"/>
     </xsl:variable>
     <xsl:variable name="propertySetID">
        <xsl:value-of select="../PSID"/>
     </xsl:variable>

    <Feature typeName="{$desc}">
      <xsl:for-each select="/TWODQUERY/LPROPS/LPROP[PSID=$propertySetID]">
          <xsl:call-template name="propertyPrint"/>
      </xsl:for-each>

      <geometricProperty typeName="centerLineOf">
         <LineString srsName="UTM">
             <coordinates>
                <xsl:for-each
select="/TWODQUERY/LINES/LINE[LINEID=$currentID]/LPOINT[not(.=preceding::LINE[LINEID=$cur
rentID]/LPOINT)]">
                    <xsl:value-of select="../LPOINT"/><xsl:text> </xsl:text>
                       <xsl:if test="(position() mod 4)=0"><xsl:text>
                           </xsl:text>
                       </xsl:if>

                </xsl:for-each>
             </coordinates>
         </LineString>
      </geometricProperty>
    </Feature>
```

```
</featureMember>

</xsl:for-each>

</xsl:template>

<xsl:template match="/TWODQUERY/POINTS">

<xsl:for-each select="./POINT">
    <xsl:variable name="currentID">
          <xsl:value-of select="POINTID"/>
    </xsl:variable>

<xsl:text>
</xsl:text>
<xsl:comment>PointID: <xsl:value-of select="$currentID"/></xsl:comment>

<featureMember typeName="RootMember">
     <xsl:variable name="desc">
          <xsl:value-of select="PSDESC"/>
     </xsl:variable>
    <Feature typeName="{$desc}">
         <name>unavailable</name>
         <xsl:call-template name="propertyPrint"/>

        <geometricProperty typeName="locationOf">
            <Point srsName="UTM">
                <coordinates>
                   <xsl:value-of select="PPOINT"/>
                </coordinates>
            </Point>
        </geometricProperty>
    </Feature>

</featureMember>
</xsl:for-each>
</xsl:template>

<xsl:template name="propertyPrint">
               <xsl:variable name="units">
                  <xsl:value-of select="UNITS"/>
               </xsl:variable>
               <xsl:if test="$units != 'null'">
                  <xsl:variable name="pdesc">
                    <xsl:value-of select="concat(PDESC, ' (', $units, ')')"/>
                  </xsl:variable>
                  <xsl:variable name="type">
                    <xsl:value-of select="TYPE"/>
                  </xsl:variable>

                    <Property typeName="{$pdesc}" type="{$type}"><xsl:value-of
select="VALUEP"/></Property>
               </xsl:if>
  </xsl:template>

</xsl:stylesheet>
```

LIST OF REFERENCES

[ADO2001] Adobe Systems Incorporated, *SVG Zone*, Web Document, http://www.adobe.com/svg/viewer/install/main.html, 2001, Accessed October 19, 2001

[AME1997] Ames, A.; Nadeau D.R.; and Moreland, J.L., *VRML 2.0 Sourcebook*, Second Edition, pp. 63-66, Wiley Computer Publishing, New York, NY, 1997

[APA2001] The Apache XML Project, *Xalan-Java version 2.2.D11*, Web Document, http://xml.apache.org/xalan-j/index.html, 2001, Accessed October 19, 2001

[BLA2001] Blaxxun Interactive AG, *Virtual Worlds Platform 5, Product Specification*, Web Document, http://www.blaxxun.com/pdfs/blaxxunplatform_productspecification.pdf, 2001, Accessed October 19, 2001

[CLA1999] Clark, J., *XT Version 19991105*, Web Document, http://www.jclark.com/xml/xt.html, 1999, Accessed October 19, 2001

[CYB2000] Konno, Sotoshi, *CyberVRML97 for Java*, Web Document, http://ns.cybergarage.org/vrml/cv97/cv97java/index.html, 2000, Accessed October 19, 2001

[ENG1999] English, J., *The Story of the Java Platform*, Web Document, http://java.sun.com/nav/whatis/storyofjava.html, 1999, Accessed October 19, 2001

[FUC1999] Fuchs, M., *Why XML is Meant For Java, Exploring the XML/Java Connection*, Web Techniques, 06/1999, CMP Media LLC, 1999

[HOL2001] Holzner, S., *Inside XSLT*, New Riders Publishing, Indianapolis, IN, 2001

[HUB2000] Huber, B., *Adding Dimensions to GIS with VRML*, Web Document, http://www.directionsmag.com/features.asp?FeatureID=36, 2000, Accessed October 19, 2001

[KAY2001] Kay, M., *Saxon, The XSLT Processor*, Web Document, http://saxon.sourceforge.net, 2001, Accessed October 19, 2001

[LAK2001] Lake, R., *Introduction to GML, Geography Markup Language*, Web Document, http://onyx.brtrc.com/webmapping/GMLIntroduction.html, 2001, Accessed October 19, 2001

[MIC2001] Microsoft Developer's Network, *Introduction to Active Server Pages*, Web Document, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/iisref/html/psdk/asp/iiwaabt.asp, 2001, Accessed October 19, 2001

[NCS1998] National Center for Supercomputing Applications, *Common Gateway Interface*, Web Document, http://hoohoo.ncsa.uiuc.edu/cgi/intro.html, 1998, Accessed October 19, 2001

[OGC2000] OpenGIS Consortium, *Geography Markup Language (GML) v. 1.0, OGC Document Number: 00-029, 12 May 2000*, Web Document, http://opengis.net/gml/GML.html, 2000, Accessed October 19, 2001

[OGC2001] OpenGIS Consortium, *Geography Markup Language (GML) 2.0, Open GIS Implementation Specification, 20 February 2001*, Web Document, http://opengis.net/gml/01-029/GML2.html, 2001, Accessed October 19, 2001

[ORA1999] Oracle Corporation, *Oracle 8i Tuning, Release 8.1.5*, Web Document, http://technet.oracle.com/doc/server.815/a67775/ch11_dwf.htm#5907, 1999, Accessed October 19, 2001

[ORA2001] Oracle Technology Network, *Oracle XML Developer's Kit for Java*, Web Document, http://technet.oracle.com/tech/xml/xdk_java/content.html, 2001, Accessed October 19, 2001

[ORD2001] Ordnance Survey, *The GIS files*, Web Document, http://www.ordnancesurvey.co.uk/gis-files/stage2/introduction.htm, 2001, Accessed October 19, 2001

[SED1999] SEDRIS, *SEDRIS Transmittal Format Description*, Web Document, http://www.sedris.org/stf_desc.htm, 1999, Accessed October 19, 2001

[SED2001] SEDRIS, SEDRIS Products and Documentation, Web Document , http://www.sedris.org/pro1trpl.htm, 2001, Accessed October 19, 2001

[SUN2001a] Sun Microsystems, Inc., *The Java Tutorial, Overview of Servlets*, Web Document, http://java.sun.com/docs/books/tutorial/servlets/overview, 2001, Accessed October 19, 2001

[SUN2001b] Sun Microsystems, Inc., JDBC Data Access API, Web Document, http://java.sun.com/products/jdbc, 2001, Accessed October 19, 2001

[SUN2001c] Sun Microsystems, Inc., *Java 3D™ API*, Web Document, http://java.sun.com/products/java-media/3D, 2001, Accessed October 19, 2001

[VIE2001] Viewpoint Corporation, *Viewpoint Technical Documentation*, Web Document, http://developer.viewpoint.com/developerzone/5-222.html, 2001, Accessed October 19, 2001

[VRM1997] The VRML Consortium, Incorporated, *VRML97, The Virtual Reality Modeling Language, International Standard ISO/IEC 14772-1:1997*, Web Document, http://www.vrml.org/Specifications/VRML97/index.html, 1997, Accessed October 19, 2001

[W3C1999] World Wide Web Consortium (W3C), *XML Path Language (XPath) Version 1.0, W3C Recommendation 16 November 1999*, http://www.w3.org/TR/xpath, 1999, Accessed October 19, 2001

[W3C2000] W3C, *Extensible Markup Lanuage (XML) 1.0 (Second Edition), W3C Recommendation 6 October 2000*, Web Document, http://www.w3.org/TR/REC-xml, 2000, Accessed October 19, 2001

[W3C2001a] W3C, *Extensible Stylesheet Language (XSL), Version 1.0, W3C Proposed Recommendation, 28 August 2001*, Web Document, http://www.w3.org/TR/xsl, 2001, Accessed October 19, 2001

[W3C2001b] W3C, *Scalable Vector Graphics (SVG) 1.0 Specification, W3C Recommendation 04 September 2001*, Web Document, http://www.w3.org/TR/2001/REC-SVG-20010904, 2001, Accessed October 19, 2001

[W3D2001] Web 3D Consortium, *Extensible 3D (X3D) Graphics Working Group*, Web Document, http://www.web3d.org/x3d.html, 2001, Accessed October 19, 2001

BIOGRAPHICAL SKETCH

Mark E. Fraser was born in Detroit, Michigan, in 1960. He received a Bachelor of Science degree with a major in accounting from Florida State University in 1981 and a Juris Doctorate from Duke University School of Law in 1988. He is licensed in Florida as a Certified Public Accountant and is a member of the Florida Bar.

He began his studies in the Department of Computer and Information Sciences at the University of Florida in 1999. Since 1998, he has worked as a software developer with Gleim Publications, Inc., in Gainesville, Florida.

His research and professional interests include database management systems and related object oriented software applications. He conducted his master's thesis research at the Database Systems Research and Development Center at the Unversity of Florida. His main research focus has been the design and implementation of an architecture and methodology for storage, retrieval and presentation of geo-spatial information.