Perspective Based Level of Detail Management of Topographical Data

Mastergradsoppgave i informatikk

Morten Granlund

21.12. 2004

Høgskolen i Østfold
Avdeling for
informasjonsteknologi

I believe we need a "Digital Earth". A multi-resolution, three-dimensional representation of the planet, into which we can embed vast quantities of geo-referenced data.

(Al Gore, former Vice-President of the USA)

Summary

This thesis deals with implementation and deployment of very large three-dimensional (3D) models of topographical data. Computer scientists began as early as the 1960s to research on 3D computing, and 3D visualizations has, since then, become something we almost take for granted. Today, we see 3D computer models every day, whether it is through the weather forecast, digital blue prints used for construction, digital maps, or computer games. As the Internet became prevalent through its exponential growth during the late 1990s, distributed computing emerged, and it was only natural that also 3D computing became subject to the distributed client-server paradigm. However, the vision of 3D models on the Internet being run inside an internet browser, just a mouse-click away, has not quite come true yet. By large, the user is required to install some sort of software, albeit a plugin or a full-scale runtime environment, before any 3D content can be viewed. Introductorily, different 3D technologies will be reviewed in search for one that satisfies the requirements we set in this thesis. An important paradigm that this thesis takes as its starting point, is that topographical data can easily grow to the extent where it is almost impossible to maintain as three-dimensional data. Geographical alterations, whether created by humans (roads, railroads, dikes, or buildings) or nature (volcanoes, landslips, or avalanches) lead to a continuous need for maintenance on the geographical data, albeit a Geographical Information System (GIS) or a less formal collection of data. Maintaining 2D topographical data has a relatively long tradition compared to that of 3D computing, and thus provides a multitude of existing solutions. Hence, this thesis proposes a framework that generates 3D content "on-the-fly" from a collection of 2D data. Further, this paper proposes techniques for enhancing the performance of very large 3D models. Caching, prediction of future requests, advanced garbage collection with aid from priority queues, and different measures for estimating relative value for parts of the model are the most noteworthy. All of these improvements are applied on the server side and therefore not closely tied to the 3D technology used on the client. At the end of this paper, two test cases are presented, both of them benchmarking the majority of the proposed techniques.

Acknowledgements

There are some people I would like to thank for their help and support over the last year. First of all I would like to thank my supervisor, Gunnar Misund, for guiding me well through this thesis, and for being patient with me all those times things "just did not work". Next, I would like to thank Halden Virtual Reality Centre (Institutt for energiteknikk) for being a resource on VRML and Java3D. Further, kudos goes to Knut Erik Johnsen for spending numerous hours administering and configuring servers for me, allowing me to focus on the fun stuff. At last, but not least, thanks to my fiancé for putting up with me all those times I have been absent-minded and absorbed in my work.

Prerequisite

Because this thesis deals with many special fields within computer science, it is not possible to describe everything in detail. As such, it is assumed that the reader already has knowledge of concepts such as object-oriented programming with Java, script programming, standard server/client architecture, HTTP, and basic UML. In addition, basic knowledge of data structures and algorithms (e.g. concepts such as heaps and different tree structures) will be useful when reading about the utilization of priority queues in section 5.2.2.3 In any case, a glossary of terms can be found in Appendix A, briefly describing most of the "buzzwords" used in this thesis.

The Structure of this Document

This thesis is organized as follows. Chapter 1 gives a brief introduction to promote the motivation for writing this thesis. Chapter 2 describes fundamental concepts, or building blocks, for 3D terrain visualization and technology chosen for this thesis. Modeling languages such as VRML, GeoVRML, and X3D will be explained, together with how applications for displaying 3D content work. A conceptual structure for organizing terrain in multiple levels of detail, called Quad Tree structure, is essential in this thesis, and will be described in detail. In chapter 3, a set of related projects is described in an attempt to put this thesis in perspective. Some of these projects are ongoing, and thus incomplete, whereas others are more or less abandoned. Readers who are familiar with terrain visualization, and have knowledge of both the nuances and the limitations of VRML/GeoVRML, can safely skip chapters 2 and 3, and jump directly to chapter 4. In chapter 4, we take a closer look at how we can implement an effective framework for our server-client framework for generating 3D terrain visualizations on-the-fly. First, different ways to implement Level of Detail management is described. Then we look at some workarounds (alternative methods) to overcome some of VRML's limitations, before we finally start exploring, or proposing, techniques beyond level of detail. These techniques, enhancing the performance of our framework, are calculations and builds done in the front edge of normal data flow, and are based on qualified guesswork. To improve the chances of succeeding with the guesswork, called *heuristics*, we examine both different techniques for guessing, and what information is available for the heuristics. Chapter 5 describes how the proposed techniques from chapter 4 were implemented in an attempt to benchmark them in several test cases. I recommend that the reader goes through the technique descriptions in chapter 4 before throwing him- or herself upon the tests, unless he or she is only interested in the conclusions. Each test case has its own conclusion before the thesis, as a whole, is concluded in chapter 6. Chapter 7, called "Future Work", describes the "shortcomings" of this thesis; things that I would very much have liked to add if I only had had more time. Finally, this thesis includes three appendices. The first is a glossary of terms, which should be used actively when reading this thesis. The two next are the results from a tedious study of the plugins available a year ago, when I first started this thesis. Appendices B and C are meant to be supplements to section 2.3, and should be treated accordingly.

Table of Contents

List of Figures

List of Tables

1 Introduction

This thesis deals with presentation of very large topographical models with three dimensions (3D). These are models that the user can navigate through, e.g. by using walk- or fly features, and in which the user can examine features in the model from different angles and positions. Such 3D models are often called *virtual reality* (VR). The idea of VR can be traced back to the 1950s when a former radar technician appreciated the fact that any digital data can be visualized [1]. As the technology has improved steadily since the 50s, so has also the applicability of VR. Technology that enables 3D objects to be downloaded over the Internet and being viewed in a web-browser has been freely available since the mid- 90s [2]. VRML (Virtual Reality Modeling Language) [3], for example, is an ISO-standardized script-language that evolved during the early 1990s. In 1998, Al Gore, former Vice-President of the USA, held a speech where he dissertated about his vision of a Digital Earth; a joint effort to create a community concept for geospatial information, much like the World Wide Web was originally made for online documents. A series of grand projects and initiatives started up, including the commercial industry, the academic community, and as a collaboration between the different governmental departments. In 2000, an extension to VRML, called GeoVRML [4], was released, which aimed to simplify the creation of large-scale topographical models. However, at pretty much the same time as the recession in the commercial Internet industry came, much of the fuzz around the Digital earth faded away. As a result, so did also many of the projects related to the Digital Earth vision, including several of the implementations of VRML/GeoVRML browsers and tools. The introductorily part of this thesis examines the possibilities and limitations of VRML. Much effort has been put into finding a non-commercial browser that covers the needs of this thesis (see Appendix B and Appendix C). One particularly important condition for the modeling technology used in this thesis, is that the models should be easily accessible for the users, preferably viewable through a web browser, and requiring as little setup, or installation, as possible. The rest of the thesis, proposes a client-server architecture where all the geospatial data is stored and maintained as 2D data, whereas the server converts it to 3D on-the-fly whenever it receives a request from a client. The main reason for taking this approach is the fact that there is a solid tradition for keeping maintenance on 2D geodata, and very rigid standards and systems are available for

this purpose (as opposed to the maintenance of 3D data). Besides, the rapid changes in topography, albeit human-created changes such as roads, buildings, deforestation, or canals, or changes caused by nature, such as glaciers changing size, earthquakes, forest fires, erosions, or perhaps even the weather in some visualizations, would call for revalidations of the 3D content anyway. Naturally, the introduction of on-the-fly generation of 3D content introduces an additional lag on the client-server system. Therefore, the majority of this thesis has been an effort to find techniques for optimizing the server-side performance through techniques such as introducing caches (which encourages reuse), building 3D content *before* it is needed (by guessing which parts of the model are most likely to be requested in the near future), and applying clever memory-management (which parts of the 3D content should be deleted first from the cache?). Now, as a point of departure, let us take a look at three classical challenges that arises when trying to visualize 3D topography with VRML. Then, in the next chapter, we will examine central concepts and technology used later in this thesis.

1.1 RENDERING

Rendering of 3D models, which is the projection of 3D content to the output device, is a very demanding task (given today's technology). Let us, hypothetically, imagine that we have a 3D model of the earth with such a high level of detail that a user can zoom in on his or her house. An existing 3D model of Halden in Norway [5] covers almost two square kilometers. Its size is approximately 100 MB. Given that the total area of the earth is 500,000,000 square kilometers (about 25,000,000 times bigger), we could make a mean estimate of the size of a 3D model of the entire earth: 25,000,000 * 100 MB equals 25 Petabytes (2.5 * 10^16 bytes). Immediately, we appreciate the unfeasibility of deploying extremely large 3D models in a straight-forward manner. If the user zooms out so that he or she sees the whole earth as a sphere, the computer will have to re-render gigabytes of data several times each second. Needless to say, this is an impracticable way of browsing large 3D models. Fortunately, the concept of Level of Detail (LOD) was introduced - the technique of excluding details as you move away from objects. Today, you will find implementations of LOD in 3D modeling languages such as the ISO-standardized VRML and the GeoVRML extension. I will discuss the concept of LOD more carefully later in this paper.

1.2 DIFFERENT IMPLEMENTATIONS

Virtual Reality Modeling Language (VRML) is one of the most popular languages for distributing 3D models over the Internet. However, even though VRML is an ISO standard, you will find variations in the way that the different implementations (plugins or browsers) behave. These variations lead to the fact that some models work when run in one browser, but not when run in another. This is, in my opinion, the main reason why topographical 3D modeling with VRML/GeoVRML has not been more successful. I will discuss these plugins in more detail later in this document. Also noteworthy is the X3D specification, which is meant to be the successor of VRML. It is by the time this paper is written a relatively young technology, but it is promising as it embraces XML and embeds the GeoVRML extension.

1.3 DOWNLOADING

As I mentioned earlier, rendering of 3D models is very resource-demanding. However, transmission of data can also become a bottle neck. Let us go back to the example with the fictitious 3D model of the earth. If the model is approximately one gigabyte, then how many people have enough bandwidth today to be able to download such a model from the Internet? In five years, perhaps a considerable amount of the population will, but as the bandwidth increases, so does (in all probability) the demand for more detailed models. There is a need for technology today that allows a client to download only the parts of the 3D model that the user requests. This real-time download, or download-on-demand, could also include some sort of "pre-caching" where the server, or servers, holding the model tries to expect which tiles (pieces of topographical data) that the user is going to request next. This can be estimated from recent navigation direction and navigation speed. If a user has navigated through a model in a straight line for the last 15 seconds, one can assume that he or she will continue this static movement for the next seconds too.

2 Background

2.1 THE FUNDAMENTALS OF LEVEL OF DETAIL

As I mentioned introductorily, Level of Detail eases the burden of the 3D renderer. This is done by removing details as the user moves away from an object, and vice versa: Adding details as the user approaches an object. A common approach to this is to introduce an abstract pyramid of so-called *tiles* (as seen in figure 1). Tiles are the building blocks in a 3D model and represent different parts of the model's topography. Tiles may differ in level of detail and dimension.

Let us go back to the example from 1.1, where we imagined a 3D model of the earth, with such a high level of detail that we can zoom in and distinguish two neighboring houses from each other. The root-level (level 0) in the pyramid of tiles typically include only one tile. This tile keeps a presentation of the entire model (the earth would be seen from outer space as a sphere). Necessarily, this representation needs to be very coarse-cut, or low on details. For instance, being able to distinguish England from Ireland may not be necessary at this level. Let us now assume that the user wants to zoom in on his summer-house on the French Riviera. As the user draws nearer to the model and crosses some preset distance limits, he or she will more or less transparently start experiencing tiles on different levels. A level 1 tile covers less area than a level 0 tile, but it contains more details on the relatively small topographic area it covers. This way, the level 0 tile covers the earth, a level 1 tile may cover Europe, a level 2 tile may cover France, whereas a level 6 tile may cover the French Riviera.

For such a structure to work, we need to store information about which tiles are the children of other tiles. Usually, each tile has four children (except the tiles on the most detailed level). We also need to specify information about the location and size of the area that each tile represents. We call such a hierarchy, where each tile can have four children, a *Quad Tree*. With a Quad Tree, it takes 4^n tiles to cover the entire model on level n.

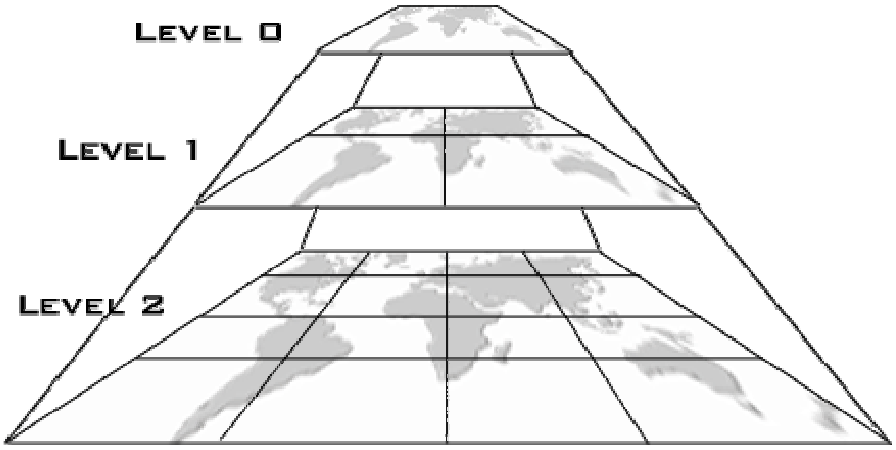


Figure 1 – The Fundamentals of Level of Detail

2.2 A QUICK GLANCE AT VRML, GEOVRML AND X3D

2.2.1 The Difference Between VRML and GeoVRML

VRML (Virtual Reality Modeling Language) is a *language*, whereas GeoVRML is an *extension* giving the developer some extra "building blocks". You can create models using VRML solely, but you cannot use GeoVRML without including VRML. It is easy to misunderstand the purpose of GeoVRML and take it for being an independent, improved version of VRML.

2.2.2 VRML Basics

Furthermore, VRML is a *script* language. That is, there is no need to compile the source code before it is executed. A VRML model consists of one or more text files with the extension '.wrl' in the file name. Figure 2 shows an example of such a text file making up a very simple VRML model. The resulting model is shown in figure 3. The syntax and semantics are, compared to OpenGL, rather easy, which makes the code easy to read and the relatively easy to master. However, it lacks some of the power and possibilities that someone familiar with OpenGL (an industry-wide standard for developing portable 3D graphics applications) might miss. Notice that node names (Viewpoint, Shape, Appearance, Material and Box) start with uppercase letters, whereas field names only have lowercase letters (position, orientation, description, appearance, etc.) A node's bounds are indicated by the balanced braces: '{' and '}'. Note also the repetition of the words 'appearance' and 'material' in lines 9 and 10. As already pointed out, 'Appearance' with a capital letter is a new node, and 'appearance' with a lowercase first letter is a field. We have simply put a new node into another node's field. This results in a hierarchic tree structure, as shown in figure 4.

```
#VRML V2.0 utf8

Viewpoint {
    position    3.0 4.0 5.0
    orientation    1.0 0.0 0.8 -0.8
    description    "Viewpoint 1"
}

Shape {
   appearance Appearance {
      material  Material {
         diffuseColor  0.1 0.2 0.9
         shininess     0.7
      }
   }

   geometry Box {
      size 2.0 2.0 2.0
   }
}
```

Figure 2 – A Simple VRML Example (Code)

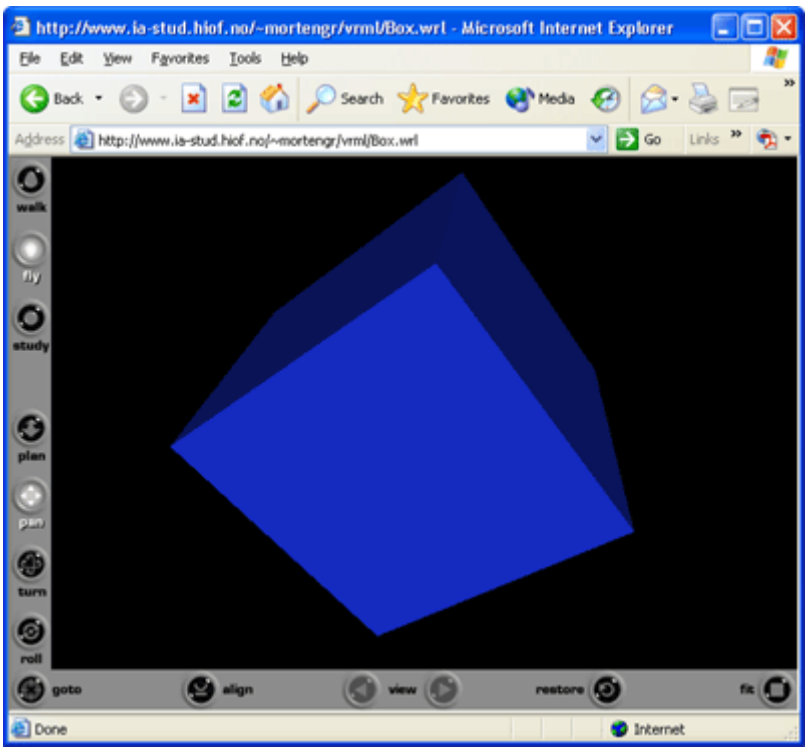


Figure 3 – A Simple VRML Example (Result)

VRML is, as mentioned above, made up of a hierarchy of nodes, constituting a graph (more specifically, an acyclic graph.) This is a common way of structuring 3D models, and it can be found in other modeling languages, whereof Java3D is the most noteworthy.

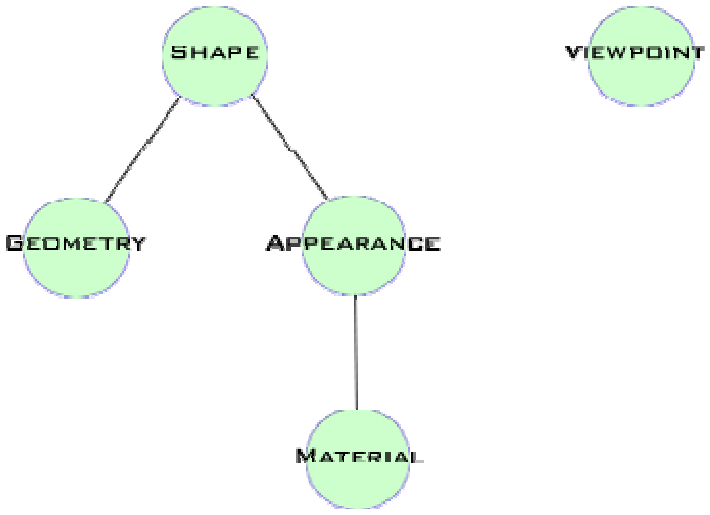


Figure 4 – The Tree Structure of a Simple VRML Example

2.2.3 GeoVRML Basics

Let us take a closer look at what the GeoVRML extension [6] offers. Perhaps the most valuable aspect of GeoVRML is the fact that it enables the use of 64 bits floating points, whereas VRML only supports 32 bits floating points. In large scale models, this improvement could turn out vital since you can improve meter-precision to millimeter-precision. Further, GeoVRML can transform "flat" topographic data into global data with the angle of inclination that the earth has (see figure 5 and 6. Both models are from the GeoVRML 1.1 examples [7]).

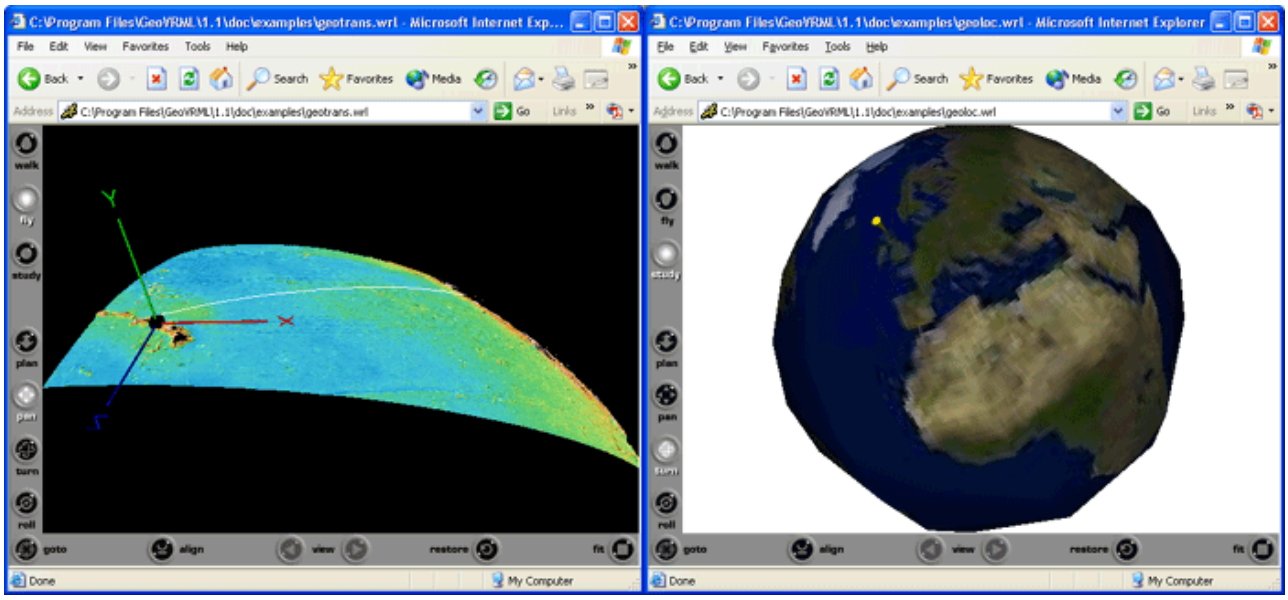


Figure 5 & Figure 6 – GeoVRML 1.1 Example A and B

Because GeoVRML consists of a set of Script nodes written in Sun Microsystem's language Java (as opposed to ECMAScript [8]), and most VRML browsers do not support Java in the Script node, the GeoVRML extension is more or less "put on hold". The extension provided by GeoVRML will be integrated in the X3D standard - VRML's successor.

2.2.4 X3D - VRML's Successor

In February 1999, the Web3D Consortium announced that they had started the process of defining a new language for describing 3D models, namely X3D. X3D is a continuation of VRML97 (the latest version of VRML), with a couple of new features, and it should (according to plans) provide backward compatibility. It also integrates the extensions provided by GeoVRML. As this paper is written, the X3D specification has been submitted for ISO (International Standards Organization) approval, and the result is expected sometimes in early 2004 [9]. When X3D takes over, many of the problems encountered when modeling with VRML are expected to vanish. Some of these problems are due to VRML browsers being differently implemented and GeoVRML being poorly supported by browsers (because support for JAVA scripting is seldom implemented).

2.3 BROWSERS/PLUGINS

A *browser,* or a *plugin,* refers, in this document, to an application that parses a document containing 3D information and draws a 3D model to the screen. Different browsers exist for showing different 3D contents. As an example, we need a VRML browser to parse and display the contents of a VRML document properly, whereas we need a X3D browser for presenting X3D contents. Note that the terms *browser* and *plugin* are in some cases interchangeable. In this paper, however, I will use the term plugin to cover the applications that enables 3D models to be viewed inside web-browsers, such as Netscape, Opera, Internet Explorer, or Mozilla. The term browser will be used as a generic term to cover all sorts of 3D interpreters, whether it is a plugin or a standalone application. Let us now take a closer look inside the browser to see how it works.

2.3.1 The Basics of the VRML Browser

As previously stated, a VRML browser is a bundle of software that creates a visual and audio presentation of a Virtual Universe. The browser's presentation is based on two factors: The VRML document describing the universe and the interaction performed by a client (the user who interacts with the Virtual Universe). Figure 7 demonstrates the basics of a plugin that shows VRML content within a web browser. The browser (represented as a set of cog wheels in the model) interprets the content of a VRML document, creates a virtual universe, and sends a visual presentation to the web browser. When the user (client) interacts with the world, e.g. navigates or turns around, the plugin will have to create and send a new presentation to the web browser, i.e. a presentation, or a snapshot, of the world seen from another position and angle. A more precise description of how the plugins work can be found on Web3D's web-site [10].

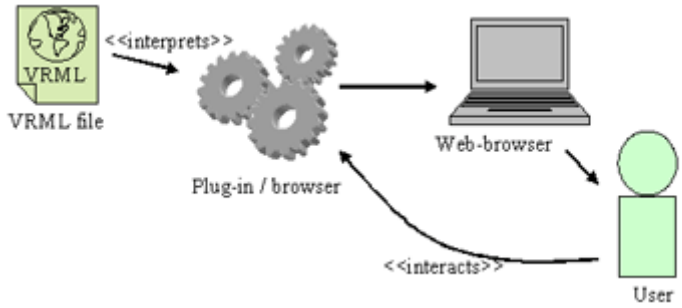


Figure 7 – An Overview of the VRML Browser, With Surroundings

For viewing standard VRML models, almost any plugin will do. However, the trouble is to find a plugin that works together with the Geo-VRML extension. I have done a test on some of the plugins that are freely available on the Internet. A summary of this test is listed in Appendix B. In short, I found that the *Cortona VRML Client* was the only plugin (of the plugins I tested), that worked properly together with GeoVRML. However, blaxxun Contact [11] (written with a lowercase 'b') was the only plugin that provided a feasible management of tile loading (see chapter 2.1 "The Fundamentals of Level of Detail" for details about tile loading).

2.3.2 VRML Browsers in Detail

So far, we have kept out all details about how the browser builds a virtual universe from plain text and what constitutes the "visual and audio presentation" that is sent to the display. If we dismantle an arbitrary implementation of a VRML browser, we will most likely find that it is divided into three components, i.e. the Parser, the Scene Graph, and the Renderer. This is shown in figure 8. Let us take a closer look at each of these three components, starting with the Parser.

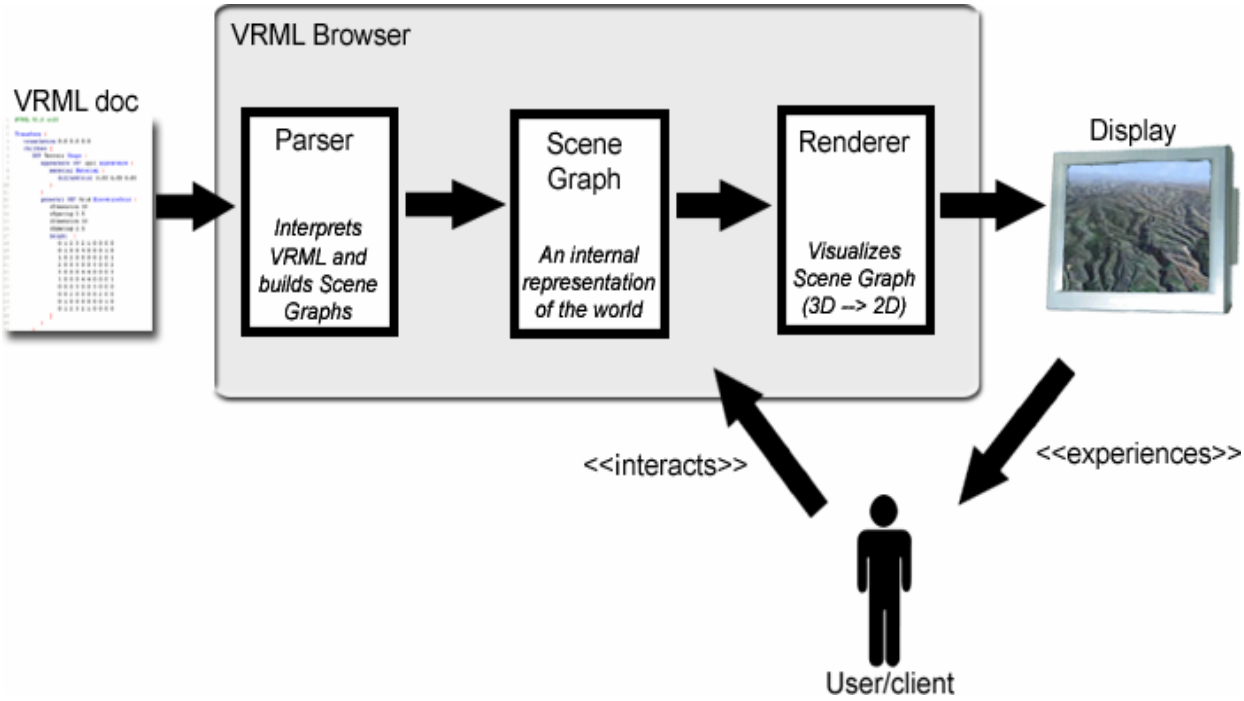


Figure 8 – The VRML Browser in Detail

2.3.2.1 The Parser (From Plain Text to Scene Graph)

The Parser is the part of the browser that converts VRML code (plain text) into a more efficient representation of 3D models. This efficient, internal representation is called *Scene Graph* and is discussed more closely in the next paragraph. Since VRML is a script language, the VRML content sent to the browser has not been validated by any compiler. The Parser must therefore check both the syntax (spelling) and the semantics (the composition of key words) of the VRML content as it tries to build the Scene Graph. Correct VRML syntax and semantics are specified in the VRML 97 Specification [12].

2.3.2.2 The Scene Graph (Internal Representation)

A Scene Graph is a hierarchical tree structure describing the virtual universe internally in the browser. This tree graph contains a set of nodes that describes the objects in the virtual world and their properties. To illustrate this, let us say that we want to create topography (a terrain model) covering 10 x 10 meters. The geometry of the topography can constitute one node in the Scene Graph, whereas the material of the surface is described in another node. According

to the VRML specification [12], this graph is both *directed* and *acyclic*. The term directed means that the direction of the parent-child relationship between the nodes is important. Node A being the parent of node B is *not* the same as node B being the parent of node A. The term acyclic means that there should be exactly one path from the root node to any other given node. Loops are not allowed and a node cannot have more than one parent (although a node can have many children). This particular type of graph is an efficient way of structuring 3D contents because, for one thing, nodes can inherit properties from parent nodes. Reuse like this is a necessity in order to create feasible VR models. In the example with the 10 x 10 meter topography, we let the node describing the surface be a child of the geometry node. Then, we let the geometry node be a child of a new node defining a translation (alteration of location). As a result, all nodes being children of the translation node is moved to the new location. Scene Graphs are used by most other 3D frameworks, e.g. Sun Microsystem's Java 3D API.

2.3.2.3 The Renderer (3D to 2D projection)

The visual representation of the virtual universe, which is sent to output devices such as monitors or VR goggles, must be 2D. This is simply because standard output devices, such as monitors or VR goggles, are limited to displaying two dimensions. The conversion from 3D to 2D is performed by the Renderer, and is by itself worth a study. Since the different ways of implementing a Renderer is outside the scope of this paper, I will only briefly describe some of its tasks. In short, the process of *projecting* (or *rendering*) the 3D virtual universe to a 2D image plate (e.g. a monitor or an HTML page) involves calculating which objects are visible from the avatar's (a representation of the user in a virtual universe) current position and orientation at all times. Besides, since all 3D objects have a back side, and any object may be partially or completely obscured by other objects, the Renderer must at all times calculate *what parts* of the visible objects can be seen. This might seem like a trivial task at first glance. However, when we include the ability to render complex objects that obscure each other interchangeably, manage objects with various degrees of transparency, and calculate the amount and angle of light reflected by each and every surface when multiple light sources are allowed, the task of projecting/rendering can hardly be called anything else than intricate.

2.3.3 Limitations of VRML Browsers

The VRML standard has not been revised for years. The VRML 2.0 specification released in August 1996 is the last *major* revision. This ought to be more than enough time for developers to make the VRML browsers resilient and rigid, and we would expect a dozen fully functional plug-and-play browsers to be accessible on the Internet today. In effect, this is not the case. Finding a browser that fits your needs may even prove impossible, forcing you to implement your own browser. Below, I will state some of the limitations that I have encountered so far.

2.3.3.1 Lack of Maintenance

At the time of writing, X3D has logically enough taken much of the focus away from the VRML browsers, so that very few of them have been maintained properly. For example, the once so popular plugin, Cosmo Player, is not compatible with the latest versions of web browsers and is therefore useless unless you get hold of an old version of Netscape Communicator.

2.3.3.2 Different Browser Implementations

The VRML specification is, in part, vaguely defined, leaving room for individual interpretations by those who implement the browsers. To quote what the VRML 97 Specification [12] says about the Box node: "[...]When viewed from the inside the results are undefined." The same applies to the Inline node: "The results are undefined if the contents of [some property defining an URL] change after it has been loaded." A particular browser may allow runtime changes of the URL property mentioned in the last example, allowing the modeler to easily create a dynamic universe. The problem is that this dynamic model is likely to result in unwanted artifacts, or odd behavior, when viewed in another browser. This problem could have been minimized if there were one leading browser which "everyone" used, but the fact is that there is a swarm of VRML browsers freely available on the Internet.

2.3.3.3 Insufficient Script Support

The VRML 97 Specification [12] states that "Browsers are required to support the ECMAScript language and may support scripting in any other language". In practice, few browsers have implemented support for other languages than ECMAScript up to now. This is sad because Java provides a much wider range of possibilities than ECMAScript. Recall from chapter 2.2 “GeoVRML Basics” that the GeoVRML extension is written in Java and is therefore useless for all browsers without support for Java. Of all the VRML plugins (allowing VRML content to be shown in a web browser) I have tested, *Cortona VRML Client* was the only plugin supporting Java (see Appendix C for more details about Java support). Unfortunately, this plugin had another limitation to it, which is infeasible cache management.

2.3.3.4 Infeasible Cache Management

The properties and capabilities of a browser often reflect the developers' intentions of use. Companies creating small or medium-sized models such as vehicles, mechanical constructions, or even kitchen solutions, would be likely to make other design choices than designers creating a complex Geographic Information System (GIS). To illustrate this, take Parallel Graphics' browser, Cortona VRML Client, that loads every part of the model at start-up. This extensive pre-caching yields very good results for small and medium-sized models. However, this approach is unfit for extremely large terrain models, where the browser will hardly ever be finished pre-caching all tiles. In Appendix B two popular VRML plugins, Cortona VRML Client and blaxxun Contact, are tested regarding cache management.

2.3.3.5 Limited Set of Communication Channels

VRML has limited means of transmitting information between a client and the server. The client can request VRML content, but the content will automatically replace the current model or parts of it. Rick Carey and Gavin Bell [13] point out that "[...] VRML does not yet define the networking and database protocols necessary for true multi-user simulations." (p. 2). This was written in 1997, and to a certain extent, it still applies. The *External Authoring Interface*

(EAI) was proposed as an extension to VRML 2.0. Unfortunately, the EAI is basically meant to be an interface for letting external objects control the VRML content, not the other way around. Furthermore, not all browsers include the EAI. In much the same way as with Java, the EAI is not compulsory for VRML browser implementations, merely encouraged. The ability to send client information such as current frame rate, navigation speed, and position and orientation of the avatar to the server, can prove valuable when trying to increase the efficiency of the tile building on the server side. Later in this paper we will discuss, in more detail, how to increase server side efficiency, and how to work around the data transmission paradigm.

2.3.4 Implementing a New VRML Browser

Even though chapter 2.3.3 describes a series of weaknesses in the existing VRML plugins, anyone may implement their own. This is becoming an even easier task than earlier, as the Xj3D project of the Web3D Consortium is being implemented - a toolkit for VRML97 and X3D written completely in Java. Using Xj3D to convert VRML geometry into a Java3D Scene Graph, developers should be able to gain full control of all the problem areas stated above. Unfortunately, at the time this paper is written, Xj3D is still under development. Stable development "snapshots" have been released, though, for testing purposes.

2.4 THE QUAD TREE STRUCTURE

Tree structures are very central in the paradigm of global terrain visualizations, regardless of how many dimensions the application displays. We touched the Quad Tree briefly in section 2.1. “The Fundamentals of Level of Detail”, and saw a conceptual model of one in figure 1. Since tree structures are going to be essential in all implementations later in this thesis, we need to give a thoroughfare on how to incorporate Quad Trees in an application for visualizing topography. Structuring terrain models in a Quad Tree structure, allows us to experience terrain models in a highly optimized manner. This is possible because the virtual environment is represented by a set of tiles that each can recursively be exchanged with four more detailed ones. Nothing prevents one part of the model from being very detailed while other parts are displayed with low level of detail. As such, it is possible to examine important parts of the model in great detail, since less important parts are kept at low level of detail. Often, but not necessarily, the parts of the environment being considered important, are those being close to the user.

In this thesis, we are going to use a character naming convention for the tiles in a Quad Tree Structure, in addition to the standard terms used for trees in mathematics. When four tiles together represent the same topographical area as a single tile, the four tiles are said to be the children tiles of the single tile. The single tile, on the other hand, is said to be the parent tile of the four others. The four children tiles get the names A, B, C, and D respectively, starting with the lower left tile (when the tile is seen from above), moving counter clockwise. Each of these four children can have their own set of children. These children will inherit their parent’s name, but will get an additional character at the end of it. The top-level tile that represents the entire model (at a very low level of detail) is called the *root tile*. This uniform division of tiles is highly predictable, and one can easily calculate the size or position of any given tile just by knowing its name and the size and position of the entire model. Let us start out by finding a general recipe for calculating the size of an arbitrary Quad Tree tile.

2.4.1 Calculating the Size of a Quad Tree Tile

The Tile Builder is able to calculate both size and position of any tile in the model by looking at the tile ID and the by knowing the size and position of the entire model (the root tile). This derives from the fact that every children tile has an ID with one more character than its parent. At the same time, the children tile is 1/4 of its parent's size (all edges of a children tile are half the length of those of its parent's). Therefore, a tile with an ID containing *n* characters, has edges that are $1/2^n$ of the length of the root tile's edges. In figure 9 we can use this formula to calculate the length of the edges of tile DDD. Three characters in the tile ID gives edge length $L_{DDD} = (1/2^3) * L_{ROOT} = (1/8) * L_{ROOT}$. We can see from the figure below that this is correct; It takes 8 lengths of DDD (L_{DDD}) to cover the length of ROOT (L_{ROOT}).

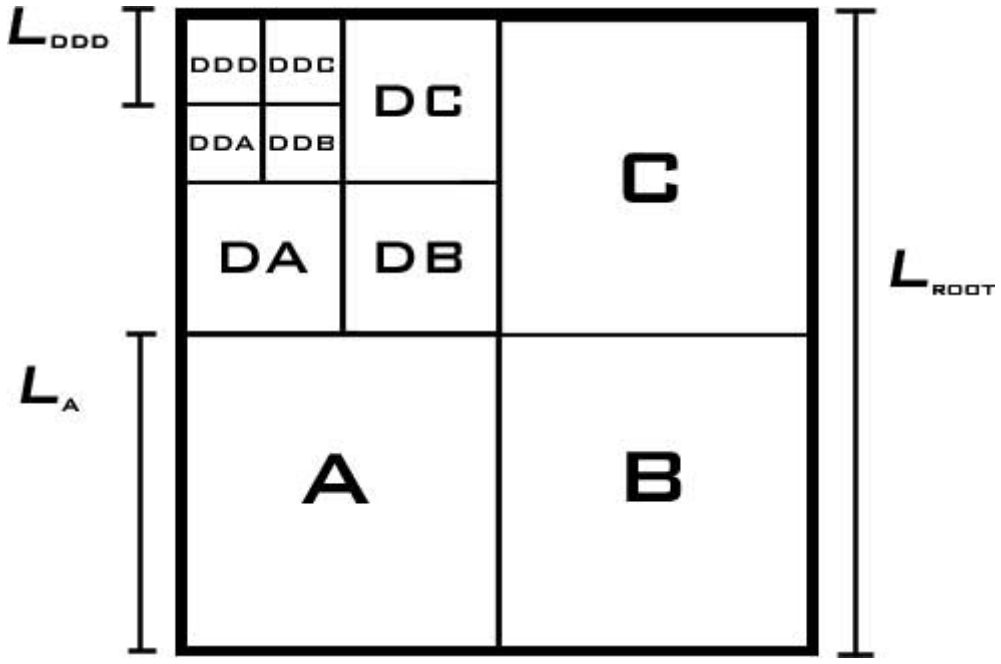


Figure 9 – Illustration of Quad Tree Tile Proportions

2.4.2 Calculating the Position of a Quad Tree Tile

In order to calculate the position of a tile, we need to know three things: The size of the root tile, the position of the root tile, and the unique name of the tile (ID). Let us take the coordinate of the root tile as a point of departure. We define origo to be the "upper left" corner of the root tile (see figure 10). Then we define $L_x(n) = L_x(0) / 2^n$ to be the length of a level n tile (a tile with name/ID made up of n characters) in the x-direction, and $L_z(n) = L_z(0) / 2^n$ to be the length of a level n tile in the z-direction.

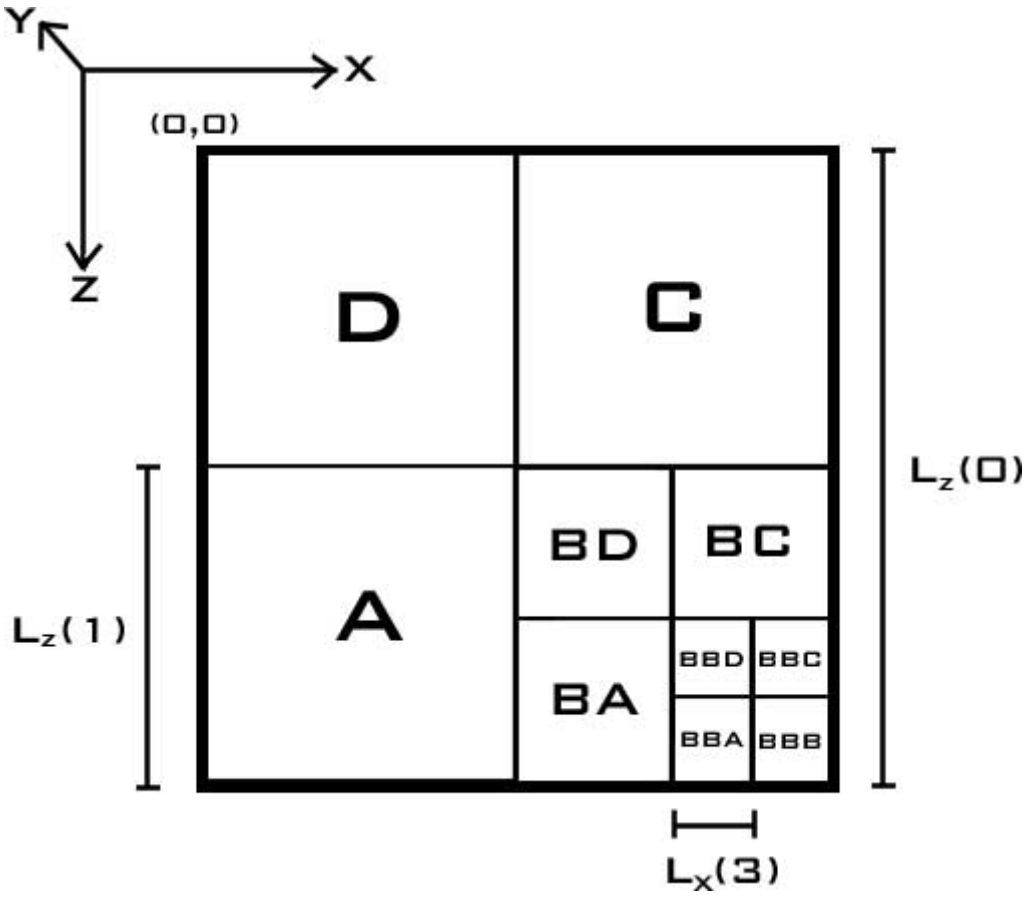


Figure 10 – Illustration of Quad Tree Tile Positions

If we now traverse the tile ID, character for character, left from right, each character specifies a relative, cumulative movement (offset) away from origo. The character's position in the ID denotes the length of the movement, and which character is used denotes the direction of the movement. This is summarized in table 1.

$Char_{pos}$	x-offset:	z-offset:
A_n	0	$L_z(0) / 2^n$
B_n	$L_x(0) / 2^n$	$L_z(0) / 2^n$
C_n	$L_x(0) / 2^n$	0
D_n	0	0

Table 1 - The Cumulative Movement Associated With Each Character in the Tile ID

2.4.3 Example - Size and Position of a Quad Tree Tile

Let us now show the calculation of size and position of a tile with an example: The root tile of a model is located (has its upper left corner) at (x=100, y=100, z=100). The size of the root tile is (x=64, z=64). Let us try to find the size and position of tile ABCD.

Since the ID of the tile has length 4, we know that the length of tile ABCD in the x-direction is $L_x(4) = L_x(0) / 2^4 = 64 / 16 = 4$. The same equation, only inserting the z-length of the root tile instead of the x-length will give the same result for the length in the z-direction ($L_z(4) = 4$). We now know that the size of tile ABCD in this model is (x=4, z=4).

In order to find the position of tile ABCD, we start out by looking at the offset in the x-axis. According to table 1, we see that an 'A' in position 1 tells us that tile A has the same x-position as the root tile (offset = 0). The 'B' in position 2 tells us that tile AB has an additional offset $L_x(0) / 2^2 = 64/4 = 16$ in the x-direction. The character 'C' in the third position gives an additional offset $L_x(0) / 2^3 = 64/8 = 8$ in the x-direction. Since the fourth character, which is a

'D', gives no offset in the x-direction, we can add the cumulative offsets in the x-direction. The tile ABCD has a total offset $\Delta x = 0 + 16 + 8 + 0 = 24$. The absolute x-position of tile ABCD is: $ABCD_x = root_x + \Delta x = 100 + 24 = 124$. We then perform the same calculation once more, now using the last column in table 1, to find the offset in the z-direction. It should give the z-offset $\Delta z = 32 + 16 + 0 + 0 = 48$. The absolute z-position of tile ABCD is then: $ABCD_z = root_z + \Delta_z = 148$. We have now found the absolute position of tile ABCD: (x=124, z=148).

2.4.4 Finding Parent and Children Tiles in a Quad Tree Model

Finding a tile's parent- or children tile is a trivial task. Every tile T has a parent T_{parent} with the same tile name (tile ID) minus the last character (needless to say, the root, having no characters in its tile name, has no parent). In addition, every tile T has potentially four children with the same tile name, where the characters 'A', 'B', 'C', and 'D' are appended.

2.4.5 Finding Neighboring Tiles in a Quad Tree Model

Finding neighboring tiles is slightly more complicated than calculating size and position. First of all, we need a definition of a tile that is positioned on the border of the model. A tile with an ID containing only the characters 'A' and 'B' (or just one of them) belongs to a tile positioned on the southern border. A tile with ID containing only the characters 'B' and 'C' (or just one of them) is positioned on the eastern border. A tile to the north only contains 'C' and/or 'D', whereas a tile to the west only contains 'A' and/or 'D' (see figure 11).

Let us now start out by making a recipe for finding the given tile T's closest neighbor to the south (positive z-direction). The first step is to find the latest occurrence of either character 'C' or 'D'. Since we just defined a tile with an ID *not* containing the characters 'C' and 'D' as a southern border tile, we know that every tile with a neighbor to the south must have a tile ID with at least one 'C' or 'D'. Then we replace this character with the character specified in the column "South" in the Quad Tree Neighbor Matrix (table 2). Next, we swap all the subsequent characters in the ID with the characters specified in the same column. Using this approach to find the tile DDA's closest neighbor to the south, we find that the second

character in the tile name is the latest occurrence of either 'C' or 'D'. We see in the column called "South" in the Quad Tree Neighbor Matrix that a 'D' should be converted to an 'A' and the subsequent characters, being a single 'A', should be converted to a 'D'. We can see from the complete level 3 Quad Tree that the tile DAD is in fact tile DDA's closest neighbor to the south.

When finding the tile to the east we search for the latest occurrence of either the character 'A' or 'D'. To find the tile to the north, we search for the latest occurrence of either the character 'A' or 'B', whereas we search for the latest occurrence of the character 'B' or 'C' when finding the tile to the west. Finding Tile T's closest neighbor to the north-east is simply solved by dividing the task into finding the tile that is east to the tile that is north of T.

A complete level 3 Quad Tree:

DDD	DDC	DCD	DCC	CDD	CDC	CCD	CCC
DDA	DDB	DCA	DCB	CDA	CDB	CCA	CCB
DAD	DAC	DBD	DBC	CAD	CAC	CBD	CBC
DAA	DAB	DBA	DBB	CAA	CAB	CBA	CBB
ADD	ADC	ACD	ACC	BDD	BDC	BCD	BCC
ADA	ADB	ACA	ACB	BDA	BDB	BCA	BCB
AAD	AAC	ABD	ABC	BAD	BAC	BBD	BBC
AAA	AAB	ABA	ABB	BAA	BAB	BBA	BBB

Figure 11 – A Complete Level 3 Quad Tree

The Quad Tree Neighbor Matrix:

Original char:	South	East	North	West
A	D	B	D	B
B	C	A	C	A
C	B	D	B	D
D	A	C	A	C

Table 2 - The Quad Tree Neighbor Matrix

2.4.6 Calculating the Number of Levels Needed in a Quad Tree

The depth of the Quad Tree is not a result of the model's size only. Two other factors play an equally important role, viz. the resolution of each tile and the desired accuracy. The formula is shown in figure 12, where the nearest positive number greater or equal to *n* is the number of levels required. Model Area is the total area of the root tile of the model. We saw in section 2.4.1. "Calculating the Size of a Quad Tree Tile" that the edges of the tiles are halved each time we move to a higher level of detail. Since all tiles, regardless of level, has a constant dimension, or resolution, we know that the precision is doubled for every new level added to the Quad Tree. Calculating the number of times we must double the precision before it suits our needs is narrowed down to the logarithmic equation below.

$$n = 1 + \log_2\left(\frac{\sqrt{\text{Model Area}}}{\sqrt{\text{Tile Dimension}} \bullet \text{Precision}}\right)$$

Figure 12 – The Three Factors Deciding the Depth of a Quad Tree

As an example, let us calculate how many levels we need to reproduce the topography of a 100 x 100 meter area with centimeter precision, using meshes consisting of 101 x 101 vertices. The square root of 100^2 is 100, and the dimension of the meshes is 100 x 100, of which the square root is 100. This gives us the following equation: $n = 1 + \log_2 (100/(100/0.01))$, which gives $n = 7.644$. Rounding this number up gives us 8, which is the number of levels we need in order to make the described model more accurate than one centimeter.

3 Related Work

In this chapter I will represent some existing projects which all approach the paradigm of modelling global 3D topography, although from different angles. The military has, historically, been an impetus in the research of large 3D models, utilizing them for combat simulations or major incident drills. In the latter years, however, due to the increased capacity of personal computers, the special field of terrain modeling has gradually become more common as desktop applications; or even as internet applications.

3.1 TERRAVISION

TerraVision [14] is one of several great projects that seem to have been abandoned, despite its great potential. It is a server-client application where the client runs as a browser (as opposed to a plugin). TerraVisionis built on top of already existing technology, such as OGC's [15] *Web Map Servers* (specification for online servers providing texture images), VRML and the GeoVRML extension. What is most interesting about TerraVision is that the developers have implemented a set of techniques to reduce the delay caused by the transmission of data. More specifically, TerraVision utilizes techniques called *prediction* and *prefetching* (in this paper this is referred to as *precaching,* and will be introduced later in this paper), which aim at predicting which parts of the model are going to be needed in the future, and start downloading them in advance. The source code can be downloaded at SourceForge.net (a repository for open source code).

3.2 PLANET EARTH / 3MAP

Planet Earth [16] is an effort to collect user-created geodata into a live map of the entire world. Planet Earth is built on top of the 3map framework [17], which provides features such as for instance real estate search and publishing possibilities. The Planet Earth map is modeled using VRML/X3D, and has, as such, quite a few things in common with the test models used later in this paper. Parts of the source code for this project are made available for download at SourceForge.net. This project is in an early state, provides highly detailed data for Sydney and Perth only, and provides insufficient documentation to decide whether they have implemented optimizing techniques, such as the one described in the latter section.

3.3 VIRTUAL GLOBE

Rune Aasgaard, working at SINTEF Applied Mathematics, is at the same time as this paper is being written, developing the application "Virtual Globe" [18]. It is a "client-server application for displaying [...] global scale terrain models" [19]. The application combines J2SE and Java3D, and uses Sun Microsystem's "Java Web Start" (which enables the application to be run from a web server). Experimental prototypes, using the *JOGL* API (Java bindings for OpenGL), is also under development. The application loads and throws out (flushes) already preprocessed data, stored on a server, as needed. An additional cache is optionally added on the client machine's hard drive. These prototypes have a great performance and seem to be highly optimized, particularly within the field of database storage (accessing and updating existing data).

3.4 DIGITAL EARTH

In 1998, Al Gore (former Vice President of the United States) held, what today has become, a famous speech [20] about his vision of a "[...] multi-resolution, three-dimensional representation of the planet, into which we can embed vast quantities of geo-referenced data". This vision, more than a project, has become known as the *Digital Earth* [21]. Further, he claimed in his speech that this task was too big for any organization alone, and could only be carried out through the "grassroots efforts of hundreds of thousands of individuals, companies, university researchers, and government organizations", much like the evolution of the World Wide Web. Concluding his speech, he said "In the months ahead, I intend to challenge experts in government, industry, academia, and non-profit organizations to help develop a strategy for realizing this vision. Working together, we can help solve many of the most pressing problems facing our society, inspiring our children to learn more about the world around them, and accelerate the growth of a multi-billion dollar industry". Unfortunately, some of the "hype" around the Digital Earth faded away, and as a result, many of the projects ebbed away. Other projects branched off to other work groups, e.g. the *Geospatial Applications & Interoperability* working group[22].

3.5 COMMERCIAL SUPPLIERS AND COMPUTER GAMES

Commercial products, especially from the computer game industry, has been a tremendous impetus for 3D visualizations and virtual environments as a whole. The gaming industry has in particular been essential in the rapid improvement of 3D hardware for desktop computers. However, the gaming industry has mainly focused on extreme optimization of relatively small environments, such as a maze, a castle, or a town. Today, flight simulators are perhaps the closest we get to a convergence between computer games and the Digital Earth vision of Al Gore (see section 3.4 "Digital Earth"). There is still a considerable gap between these two, especially considering the magnitude of data involved. No storage facility available today is even close to providing the required capacity for storing global terrain data. The special field of providing global terrain visualizations is therefore so much more than simply creating 3D data. Distributed data sources (networking), standardizing different geospatial data sources (or converting non-compatible data), generalization of data, and highly effective data storage are *some* of the fields included in a global terrain visualization application. This comes in addition to the 3D modeling (whether it is done automatically or manually, or whether it is done in advance or on-the-fly). Still, there exists numerous commercial providers of global, 3D terrain visualization applications. ESRI [23], MultiGen-Paradigm [24], and There.com [25], are three examples, all taking different approaches. ESRI provides a large package of GIS tools and technologies, including tools for 3D analysis of GIS data. MultiGen-Paradigm seems to be highly specialized in single-purpose solutions, such as battle-simulations, but also provides some 3D GIS visualization tools (some of the compatible with SRI's GIS tools). There.com, on the other hand, takes a cartoon-like approach to an online multi-user VR environment for online communication and entertainment. However, the size and shape of this environment approximates that of Earth.

4 Research and Implementation

4.1 IMPLEMENTATION OF LOD (LEVEL OF DETAIL)

One aspect of the "on-the-fly" generation of 3D models is that the client receives data only when it needs it. But when does the client need new data or new models? As for VRML (and Java 3D), nodes called *sensors* are provided. One type of sensor, called ProximitySensor in VRML, registers the movement of the user. This sensor an be used in different ways to sense when a new model should be requested from the server. I will first describe "Maximum Distance Model", which is the standard approach to Level of Detail in most 3D applications. Then I will describe the "Enter/Exit Model", which is an experimental, alternative model.

4.1.1 The "Maximum Distance Model"

The most common way to apply LOD to a model is to calculate the distance from an object with multiple levels of detail to the user's viewpoint (the avatar). Which detail level that is rendered for an object depends solely on the distance between the avatar and the object. The LOD nodes incorporated in VRML and Xj3D use this approach. Figures 13 to 15 show screenshots of a prototype built with VRML. In this prototype, three detail levels are applied to the red tile. The browser is calculating the distance between the avatar (the user's viewpoint) and all objects with multiple levels of detail each time the user moves/navigates. The advantage of applying this technique is its simplicity. The drawback is the inaccuracy related to what geospatial point that should represent the location of the object. Usually, the center of the object is chosen.

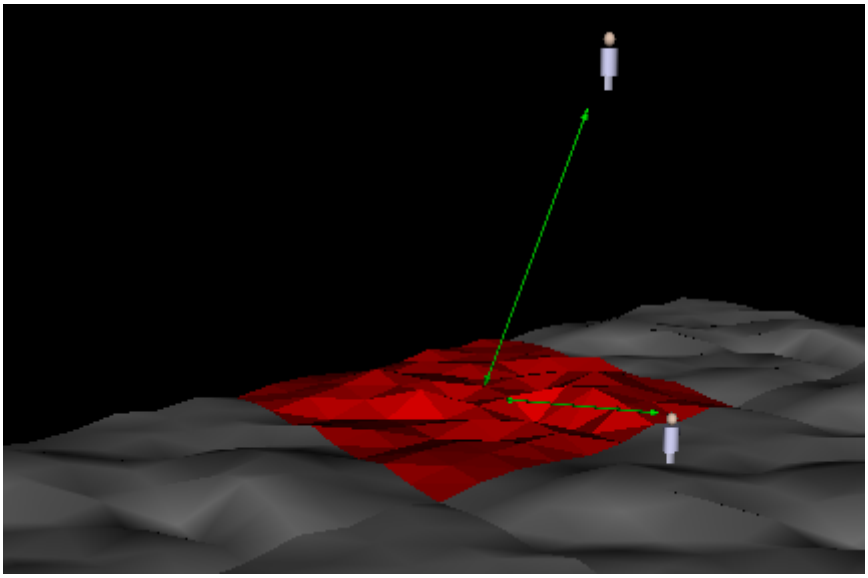


Figure 13 – Illustration of the Maximum Distance Model (A)

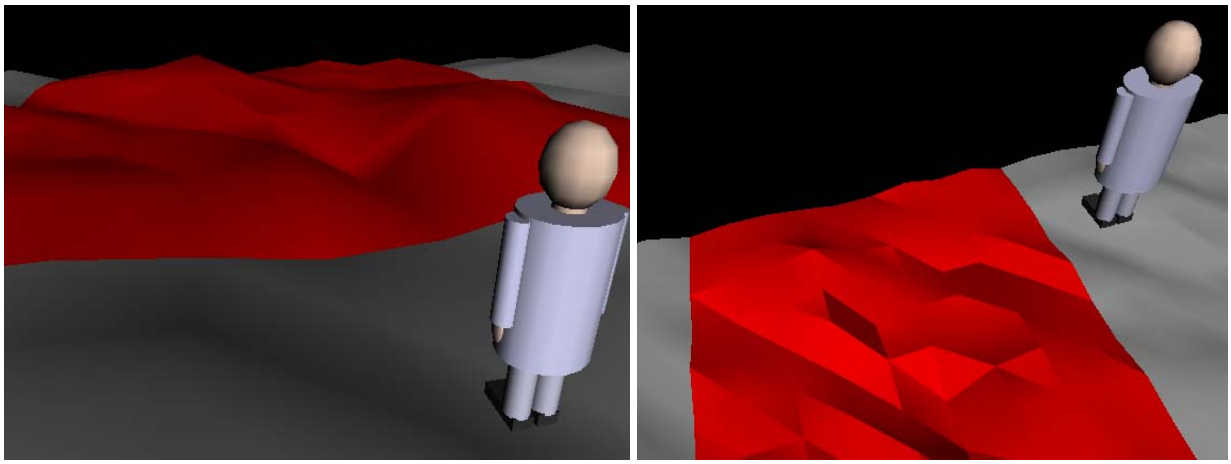


Figure 14 & Figure 15 – Two Examples of the Maximum Distance Model

4.1.2 Level of Detail Based on Projected Object Size

Another variation is to base the selection of detail level on the size of the 3D objects projected to the screen. This eliminates the inaccuracy attended with the Maximum Distance Model described above. It is, however, a more expensive (in terms of CPU resources) approach than the distance approach described above. This is due to the extra effort introduced by calculating the projected size of objects (the number of pixels it takes up on the screen).

4.1.3 Level of Detail Based on Fixed Frame Rate

A third approach is to automatically adjust the level of detail so that the system is able to maintain a constant frame rate. This approach assumes that the client has a mechanism for monitoring the frame rate on the client side, and increase, or reduce, the objects' level of detail accordingly. This approach is most suitable for applications where the 3D objects have a very nuanced set of detail levels, e.g. applications utilizing automatical polygon reduction, and where the data is quickly accessible.

4.1.4 The "Enter/Exit Model"

One way to decide when a new model is to be requested, is to make use of the ProximitySensor VRML. It senses when the avatar violates areas/boundaries. Whenever the user enters or leaves the area specified by a ProximitySensor, the sensor will generate one of two events; which one depends on whether the user enters or exits the specified area. We can then create a *ROUTE* between the ProximitySensor and a Switch-node that requests a new model to be rendered.

I have come up with this technique in order to guarantee that the client does not collect more data than is absolutely needed (which is often the case with VRML browsers' implementation of LOD). In figure 16, you will find screenshots of two prototypes made in VRML, each of them illustrating this LOD technique for which I have called the "Enter/Exit Model". I will divide this model into two sub-categories, viz."Single Sensor" and "Multiple Sensors".

4.1.4.1 Enter/Exit Model With Single Sensor

The left-hand side model in figure 16 shows the user located in a single sensor (visualized by a semi-transparent box). Whenever the user (visualized by a beige shape) *exits* the sensor, an event will be generated, telling that a new model should be loaded. The advantage of this model is that it is simple, since only one sensor is needed. The drawback is that it can be hard to load different parts of the model depending on where the user is violating the sensor-boundaries. To do this with only one sensor, we need additional information about the user's coordinates and orientation. Fortunately, VRML's ProximitySensor also generates events with information on changes in position and orientation.

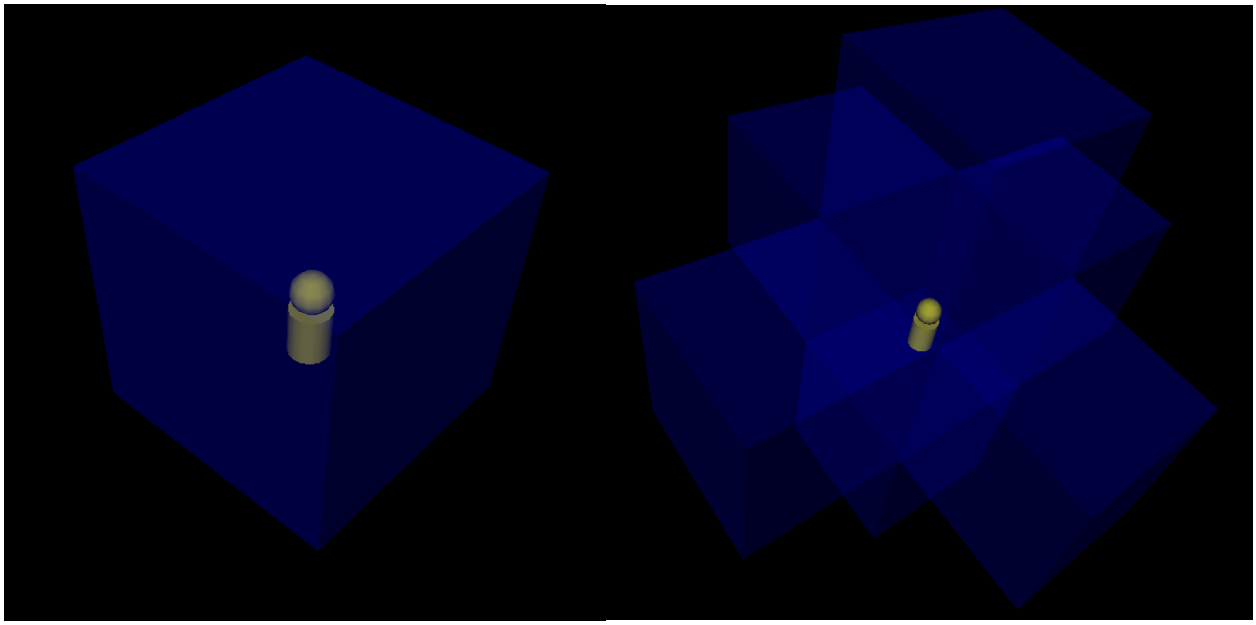


Figure 16 – Enter/Exit Model Illustrated

4.1.4.2 Enter/Exit Model With Multiple Sensors

The right-hand side model in figure 16 shows another approach. It is based on *entering* sensors instead of *exiting*. Note that the user is not *inside* a sensor, as opposed to the user in the first model. The user to the right is *surrounded* by six sensors, which will trigger an event whenever the user *enters* one of them. The advantage of this approach is that it makes it easier to implicitly tell what new model should be requested. The sensor located above the user can refer directly to a new model that shows a larger part of the 3D world with less details, whereas the sensor located to the north can refer directly to a new node with terrain representing the area further north. However, if the tiles (the area between the sensors - visualized as blue boxes in the figure) are not very small, the browser should keep track of the exact location where the user violates the sensor's boundaries in this approach as well. The reason why, is that the initial viewpoint of the avatar in the new model depends on where the avatar left the old model. If we do not base the coordinates for the starting point in a tile on the coordinates where the user left the previous tile, we might experience "jumping behavior" whenever a new tile is loaded. Another unfavorable aspect of the Multiple Sensor approach is that you might load an entire tile just to fly through one of its corners. This could be avoided by building tiles containing sensors that are not square. Unfortunately, the VRML97 specification only specifies square shaped sensors.

4.1.4.3 Enter/Exit Model With Small Sensors and Overlapping Tiles

Typically, we want the tiles to be big enough to show a decent amount of terrain, but at the same time the sensors must be small enough not to let the user come close to the edge of the tile. This results in overlapping tiles. Figure 17 shows an illustration of this. The initial tile, in which the users start, covers the gray and blue area. It also contains an invisible ProximitySensor (shown as a blue box) that senses when a new tile should be loaded. When the user navigates along the arrow and exits the sensor (the blue box), the browser will request a new tile from the server, which covers the gray and *red* area and a new ProximitySensor (shown as a red box). The idea behind this approach is that the user should only barely see terrain being added in the horizon, and, if navigating backwards, only barely see terrain being removed.

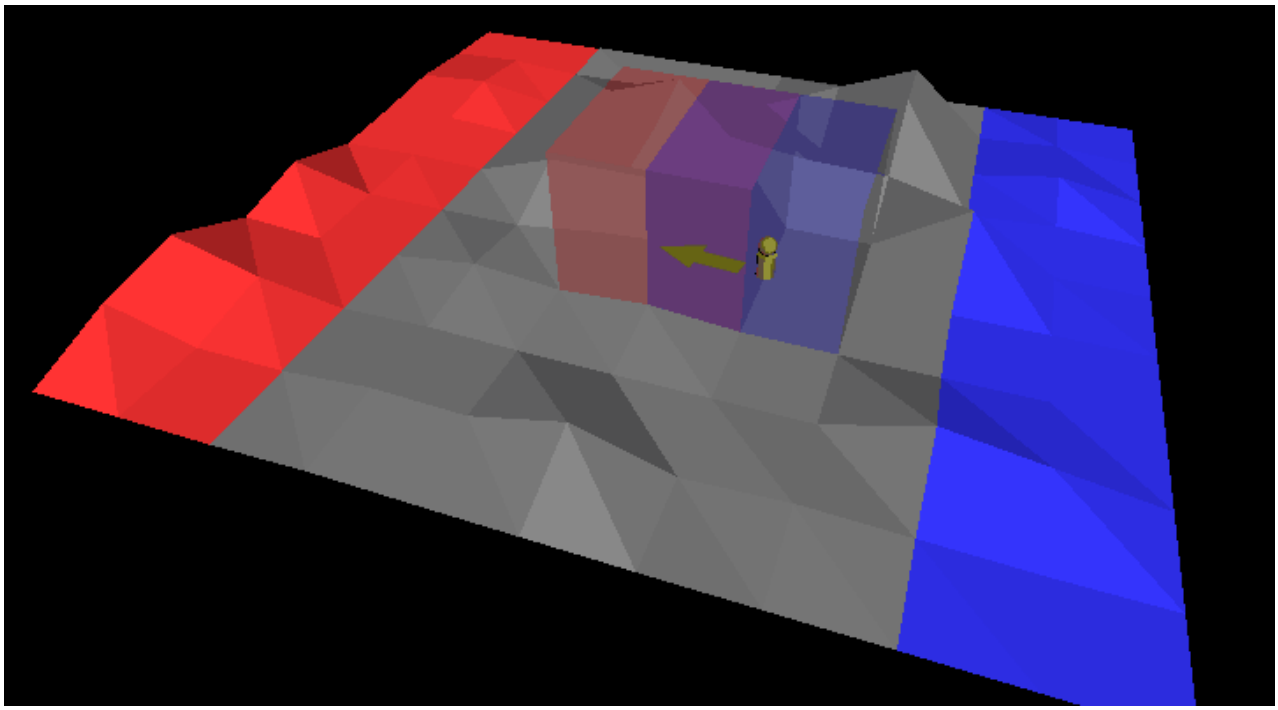


Figure 17 – Illustration of the Enter/Exit Model with Small Sensors and Overlapping Tiles

4.1.5 Perceptually Modulated LOD

A more ambitious approach is to base the LOD management on the perception of the human eye. Martin Reddy's doctoral thesis, “Perceptually Modulated Level of Detail for Virtual Environments”[26], describes a highly complex solution for removing "extraneous detail from an environment which the user cannot perceive [...] with little or no perceptual artifacts". I will not even try to describe this solution in this paper.

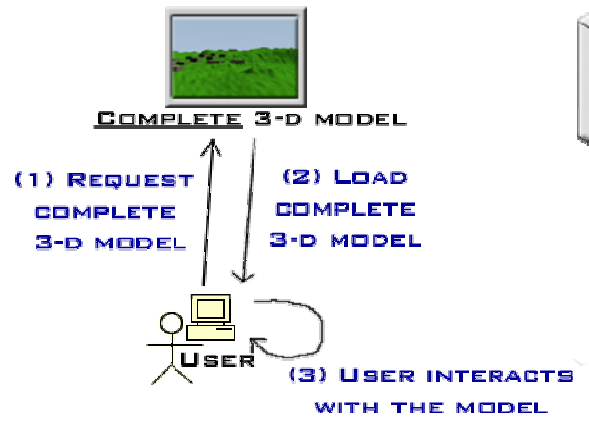
4.1.6 LOD Implementation Summary

We have seen that the VRML specification is already equipped with a distance-based LOD implementation. Compared to the size-based LOD, it is more effective and easier to use. Basing the LOD on a fixed frame rate seems to be inappropriate for client-server solutions.

The Enter/Exit Model and the Perceptually Modulated LOD seem to be unnecessarily complex for carrying out the tests in this thesis (we will mainly be concentrating on the server-side heuristics). The distance-based level of detail that comes with the VRML specification (as default) will therefore be used throughout the rest of this thesis.

4.2 ON-THE-FLY GENERATION OF 3D MODELS

The ultimate goal of my master thesis is to find a way to generate 3D models "on-the-fly". This means that some sort of *engine*, more specifically a generator, receives a request from a client who wants to browse a part of the world, e.g. the Ostfold county in Norway. The engine then fetches 2D data covering this geographical location from a database and creates one or more relatively small 3D models of Ostfold solely, which is then sent back to the client. The idea behind this seemingly awkward deployment of 3D models is three-fold. First of all, we do not want to force a client to download an infeasible file consisting of gigabytes of data before being able to interact with the model. Secondly, even the tiniest changes to the database with 2D data will not force the entire 3D model to be re-built. This would be the case in static, indivisible 3D models. The last advantage of the "on-the-fly" generation model is actually a result of the first advantage. Breaking down the model into smaller temporary models, ensures us that no one downloads a complete, free-standing model which probably will become obsolete sooner or later.



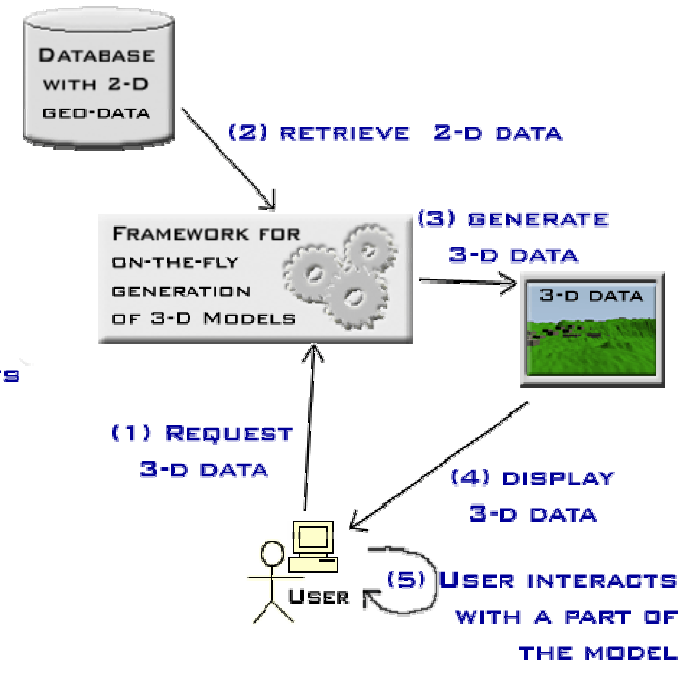


Figure 18 & Figure 19 – Conceptual Overview of a Standard and an "On-the-Fly" Implementation of a 3D Model

Figure 18 and 19 show two models. The left-hand side model shows the classic way to deploy a 3D model on the Internet. The model is a set of pre-built, static files. As opposed to this, the right-hand side model shows the system as I strive to implement it. The user (or client) requests some 3D data to browse in, and receives a relatively small model which is generated according to the user's request. This means that some sort of mechanism is required on the server side to interpret the requests from the clients, and to build the proper models. I will get back to this mechanism at a later point.

What is important to note, is that both of the techniques described above will provide the same interface for the client. Server side choice of implementation is transparent to the client. The delay caused by a complicated model being built "on-the-fly" could just as well have been caused by a slow, or overloaded, network connection.

When implementing an engine, such as the one described above, we can use different approaches. A series of approaches will be addressed later in this paper, i.e. in chapter 4.4.8 "Forecast - Looking Into the Crystal Ball".

4.3 MAKING THE CLIENT WAIT FOR DATA

One thing we have to keep in mind when implementing an "on-the-fly" model, such as the one described in paragraph 4.1, is to make sure that the client accepts delayed model deliveries/transmissions. This delay is typically caused by network delay and the time it takes for the server to build the model. If we could, in some way, make sure that the user is unable to visit parts of the model that the server has not had time to prepare yet, we would not have to worry about delayed transmissions of models. However, since the navigation can be implemented differently for each browser, we can easily end up in situations where an incomplete or non-existing tile is activated. This would result in an undesired effect if the client does not have the ability to *sleep* and try to activate it later. To appreciate the need for a sleep-capability, let us consider the built-in feature in some VRML browsers called "jump". It lets the user click on a location in the model and set it as the new viewpoint very quickly.

4.3.1 Making the Client Wait Using VRML and Scripts

The solution described in 4.1.1 is based on redirecting the user to tiles that have not yet been built. This approach requires a VRML browser that do not load such referenced models (also called *inlined tiles*) before they are needed. As of today, blaxxun Contact 5.1 is the only VRML browser I have found that do *not* load all the inlined tiles recursively (can be an exhaustive process) at start-up.

A primitive, but simple, way to make the client wait is to make use of the fact that the Inline node in VRML is able to take a whole list of URLs. The Inline node tries to load the content of the first URL. If that one fails it will try the second, and if that one fails it continues down the list until it finally manages to load the content of a file. If the first URL in an Inline node points to a tile that is under construction, and the second URL points to the file that the Inline node is located in (a reference to itself), the effect would in theory be a loop - a node that keeps on referring to itself until the tile with the highest priority URL (the one being constructed) is complete. Unfortunately, creating a simple VRML model to test out this approach seems to result in a finite number of "loops" (whereas we want an *infinite* number of loops until the first URL points to a valid VRML-file). blaxxun Contact's browser gave up

and neglected the Inline-node after it had referred to itself 18 times, indicating that the browser has some sort of built-in "loop-detection".

4.3.2 Limitations - Scripts Cannot Create Events.

One of the limitations of embedding scripts in your VRML world is that you cannot *create* an event without the Script-node receiving an event. Only sensor-nodes and exposed fields can create events "from nothing". This means that the only way to evaluate (execute) scripts is to pass an event to the script-node. The events we can use in order to start scripts must be generated by one of the VRML sensors or by changes made in an "exposedField". Since a LOD node does not emit any events when it switches children node to be rendered, only when the whole set of nodes is added, removed, or replaced, we might have to build our own LOD-mechanism from scratch using the Switch node. The Switch node, as opposed to the LOD-node, sends events whenever the browser switches children node to render.

4.3.3 Dynamic Servers - Overriding the HTTP Response From the Server.

Due to the limitations of VRML and ECMAScripts, it is natural to check whether CGI-scripts or servlets, being server side technology, can force the client to wait by customizing the response from the server. We are now assuming that the 3D browser will wait for an HTTP response (either a "404 File Not Found" or a "200 OK") before proceeding. As described briefly in 4.1 the server needs some sort of mechanism to interpret the request received from the client and build a model, which is then returned to the client.

Another great advantage of introducing Dynamic Servers is that it enables two-way communication over the HTTP protocol. For instance, it can add the coordinates for the user to the HTTP request to the server. I will return to this in section 4.4.7.2 "Client-Side Data (Fine-Grained Data)".

4.4 HEURISTICS FOR ON-THE-FLY GENERATION OF TILES

4.4.1 About Heuristics

The term *heuristics* descends from the Greek adjective *'heuriskein*', which means "to discover". Using heuristics involves performing qualified guesswork and can be contrasted with using algorithms (which involves using a predefined set of rules or a formula). Another definition describes heuristics as gaining knowledge or some desired result by following a "rule-of-thumb" [27].

4.4.2 Motivation for introducing Heuristics

The main purpose for introducing heuristics on the server side is to minimize the *Request Delay* (RD), i.e. the time it takes for the server to respond to an HTTP request. The response takes form as an *HTTP* response and should include the requested 3D content (which in our case will have to be built from 2D data). Building 3D data from 2D data is a time-consuming process and is therefore a strong candidate for becoming the bottle neck on the server side. The introduction of a cache encourages reuse instead of re-creation of redundant data, and is a common strategy to increase efficiency. However, in an attempt to improve the efficiency even further, we also introduce heuristics, which aims at improving the percentage of tiles that will be reused in the cache.

4.4.3 How to Measure Heuristic Efficiency

In order to compare different implementations of heuristics and try to decide which one is the best, we need to define what characterizes good heuristics. Although the heuristics that we try to make efficient is located on the server, we can monitor and measure several parameters both on the server side and the client side. From the client's point of view, the primary goal is that the client experiences "smoothness" when navigating through the model. That is, motion

should appear to be continuous, not discrete. This smoothness is tightly connected with the measure called *frame rate*, which I shortly will describe in more detail. Periodical occurrences where the model seems to "freeze", possibly caused by large amounts of data suddenly being downloaded, are a nuisance and can also be a measure of inefficient heuristics. Let us take a closer look at some parameters that can be monitored, both client and server side.

4.4.3.1 Frame Rate at the Client

What we perceive as motion on the screen, albeit a monitor or a television, is an illusion created by updating the screen frequently with relatively small changes to the visual content. If the updates happen frequently and the changes to the visual content are small, the human eye will perceive the movement (animation) as smooth and realistic. The measure of this "smoothness" is called *frame rate* and as an example, professional movies are supposed to have a frame rate of 24 frames per second.

Every VRML browser supporting ECMA-script (JavaScript) in the "script node" should support the object, `Browser`, with a set of static methods. One of these methods is `getCurrentFrameRate()`, which returns, as the name implies, a number indicating what frame rate the model is keeping. By creating a pre-defined navigation path through the model (often called a *Tour*), we can repeat the exactly same navigation after exchanging the heuristics part with a new one. The deviation in the mean frame rate can then only be explained by the difference in the heuristics' efficiency.

4.4.3.2 Number of Requested Tiles not Being Built yet.

On the server side, we should keep track of how many times the requested tiles can be fetched from the cache (which means that the tiles have already been built). Similarly, we also should keep track of how many tiles the client have requested in total. The client will receive tiles more quickly when they have been pre-built. Therefore, the percentage of the requested tiles that can be fetched directly from the cache can be an indication of how effective our heuristics implementation is, at least concerning cache handling. In order to get a high percentage of requested tiles that are pre-built, we need the heuristics to estimate in a sophisticated manner

which tiles should be pre-built and saved in the cache. However, since all caches have limited capacity, we also need to free cache space by estimating which tiles are least likely to be requested in "near future" and therefore can be deleted.

4.4.4 Heuristics in Detail

The heuristics we use in this document consists of two agents, namely the *Pre-builder* and the *Garbage Collector* (see figure 20). The Pre-builder is responsible for predicting which tiles are most likely to be requested and see to that these tiles are built and put into the tile cache (given that these tiles are not already built and already resides in the tile cache). Because the tile cache has a limited capacity, as with all storage facilities, there must be another mechanism that expunges tiles that are unlikely to be reused from the cache. This is the responsibility of the Garbage Collector. Strikingly, the two agents have one thing in common: They both have to calculate the tiles' *Likelihood Of Utilization* (LOU), viz. how likely it is that tiles will be requested. Before presenting different techniques to calculate the LOU of tiles, let us investigate the two agents of heuristics in detail.

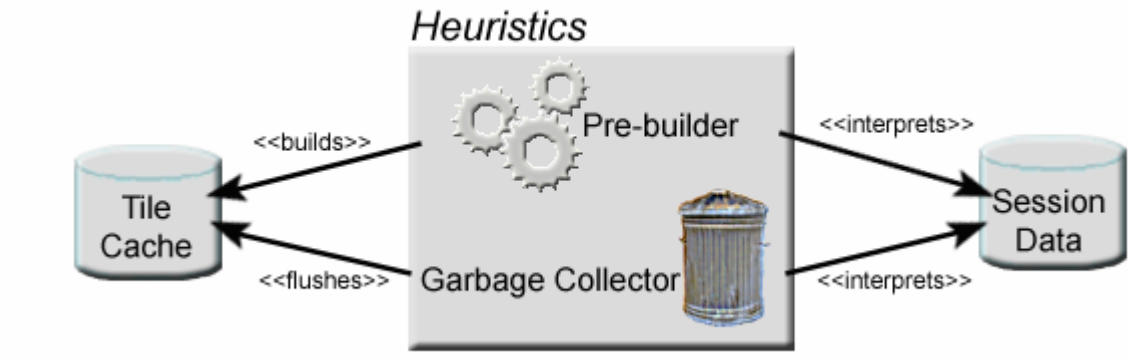


Figure 19 – The Heuristics' Two Agents (The Pre-builder and the Garbage Collector)

4.4.4.1 Pre-building vs. Flushing

As already mentioned, both agents of the heuristics calculate the LOU of tiles. However, the angle of attack is different for the two agents. Whereas the Pre-builder strives for finding tiles that do not exist yet with high LOU based on Session Data only, the Garbage Collector is trying to find one or several tiles with low LOU from the Tile Cache (which is a restricted set of tiles). This allows the Garbage Collector to compare a finite set of tiles, as opposed to the Pre-builder, which has to calculate which potential tile, when built, will have high LOU.

4.4.5 LOU (Likelihood Of Utilization) - different techniques

Estimating the likelihood for a tile to be requested can be done in several ways. In this section we will take a look at several techniques for estimating the LOU, especially those that will be implemented in the "on-the-fly" framework (described later in this paper). Each technique may be used in combination with the others. This is done by basing the LOU on either a *ranking system* (where some techniques are only used when other techniques with high precedence are not able to rank the tiles satisfactorily) or a *system of weighting,* where each technique is given a weight and the LOU is calculated from the total of all techniques.

4.4.5.1 Number of Times Utilized

Perhaps the most common way to calculate LOU is to use a technique that is used by all kinds of caches, viz. to find out how many times the contents have been used (or requested). A tile that has been requested many times must somehow be an important tile; perhaps a tile that is loaded each time the user jumps to a certain viewpoint. There is a greater chance that a tile that has been requested 20 times should be requested again than a tile that has been requested only one time. This is of course if we ignore *when* the tiles were requested, which leads us to the next criterion - Time.

4.4.5.2 Time

The timestamp for when a tile was last requested, or even built, can also predict the LOU of tiles. A tile that has been requested a long time ago might be less likely to be reused than a tile that has been requested recently.

4.4.5.3 Distance

Looking at the distance between a tile and the avatar is, in our case, perhaps a more rigid way to estimate the LOU of a tile. This technique rests on navigation that is as continuous as possible. If a tile is far away, then it is not very likely that it will be requested for some time because of the time it takes to get there. Tiles being close to the avatar is very likely to be requested next. However, when we introduce several levels of detail for each tile it gets more complicated. Tiles with low level are loaded at a longer distance than tiles of high level. This can cause low-level tiles positioned relatively far away from the avatar being requested before high-level tiles positioned relatively near the avatar. Hence, we introduce the term *Detail Boundary* (*DB*) as being a virtual sphere surrounding each tile. Breaking the surface of this sphere results in the detail level of the tile to change. Entering a sphere yields a higher level of detail and vice versa. Figure 21 shows a selection of the model's topography (a gray elevation grid) with a transparent, gray sphere representing the detail boundary of the root tile. Breaking the surface of this sphere (moving inwards) will result in tiles 'A', 'B', 'C', and 'D' replacing the root tile, revealing a higher level of detail for the client. The four blue spheres show the detail boundaries for the tiles 'A', 'B', 'C', and 'D', which will replace these tiles with even more detailed tiles. As a result, any given tile is visible only when the avatar is inside its DB and outside the DBs of its children. As an example, assume that a client navigates through the gray sphere in figure 21 (which is the DB of the root tile). The root tile will then become invisible as tile 'A', 'B', 'C' and, 'D' replace it. Tile 'A' will again be exchanged by 'AA', 'AB', 'AC', and 'AD' as soon as the client enters the DB (blue sphere) of tile 'A'.

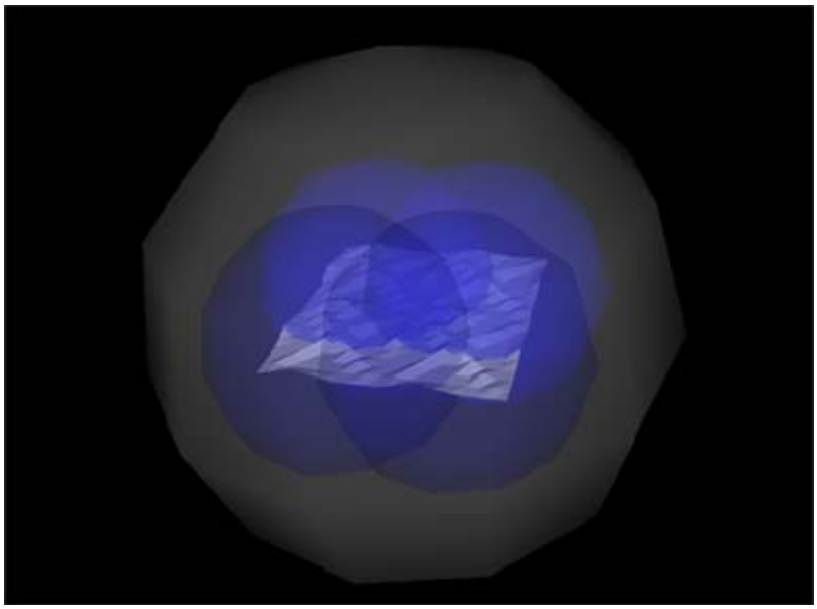


Figure 20 – The Detail Boundaries (DB) of a Tile

So the distance we are really interested in is the shortest distance between the avatar and the detail boundary of a tile, since this would tell us how far the client will have to navigate before a specific tile is needed. The shortest distance between a sphere and the avatar is at all times the normal of the sphere which runs through the avatar. This distance will be referred to as *Detail Boundary Distance (DBD*) and is shown in figure 22. If the DBD of a tile is short, then the tile's LOU is high and vice versa.

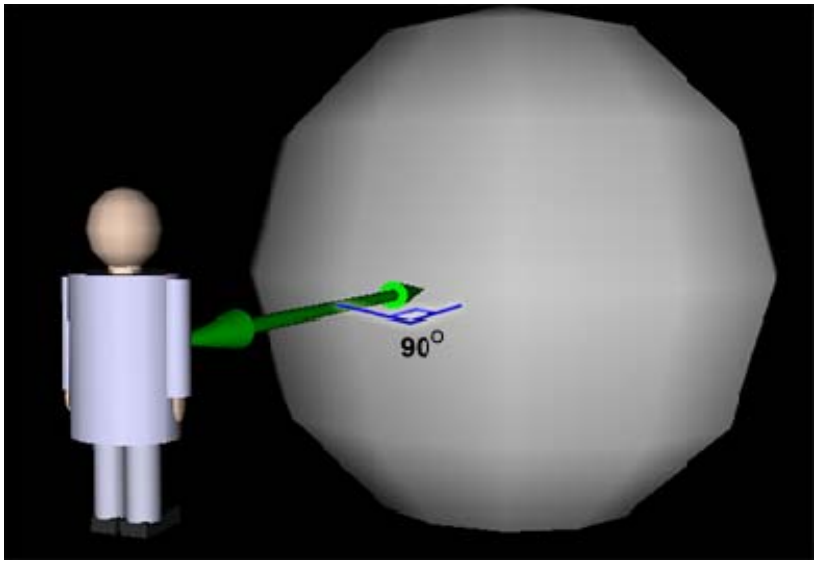


Figure 21 – The Detail Boundary Distance (DBD)

4.4.5.4 Trends in the Navigation

Another technique is to analyze the navigation of the client and try to predict future navigation. This may prove impossible for long-term prediction, but if the client has zoomed in on the model by navigating in a straight line for the last five seconds, then it is likely that this static navigation will proceed for the next few seconds as well. One specific case in which this technique may prove effective is when the client has been jumping from viewpoint to viewpoint in rapid succession, perhaps even chronologically. It would be natural to assume that the client will continue jumping in this manner, which means that tiles needed for the next viewpoints will have a high LOU.

4.4.6 Heuristics Input

In order to implement heuristics that can perform qualified guesses, for example about which tiles are going to be needed in the near future, we need some data, or input, to base these guesses on. We can divide data utilized by the heuristics into several categories, starting out by introducing the terms *History Based Session Data* (HB) and *Present State Session Data* (PS).

4.4.6.1 History Based vs. Present State Session Data

As already mentioned, the two main categories of heuristics input are History Based (HB) and Present State Session Data (PS). The difference lies in the extent to which the server keeps track of the past. PS Session Data is a *snapshot* of the current state either in the server, in the client or in both. An example of utilizing PS data would be if the Pre-builder bases all its activities on only the *latest* data collected from the client, e.g. orientation of the avatar. In contrast, HB is, as the name implies, *a set of* snapshots resulting in a *history* of data. The snapshots may be taken at regular intervals or each time the client requests new tiles. As a matter of fact, HB is a set of PS snapshots from different moments in time. This is illustrated in figure 23. Because the HB approach results in much more information collected, the heuristics is able to perform more sophisticated predictions as it may recognize trends, repetitions, and patterns in the way the client navigates through the virtual universe. If we modify our previous example so that the Pre-builder utilizes HB, it could pick up tendencies telling that a particular client repeatedly jumps back to the initial viewpoint before zooming in on the model again. The server can then choose to keep the initial low-detailed tiles in the cache even though the client navigates away from the initial viewpoint. One of the drawbacks of relying on HB is that the implementation can get quite complex without *necessarily* improving the average LOU (Likelihood of Utilization) drastically.

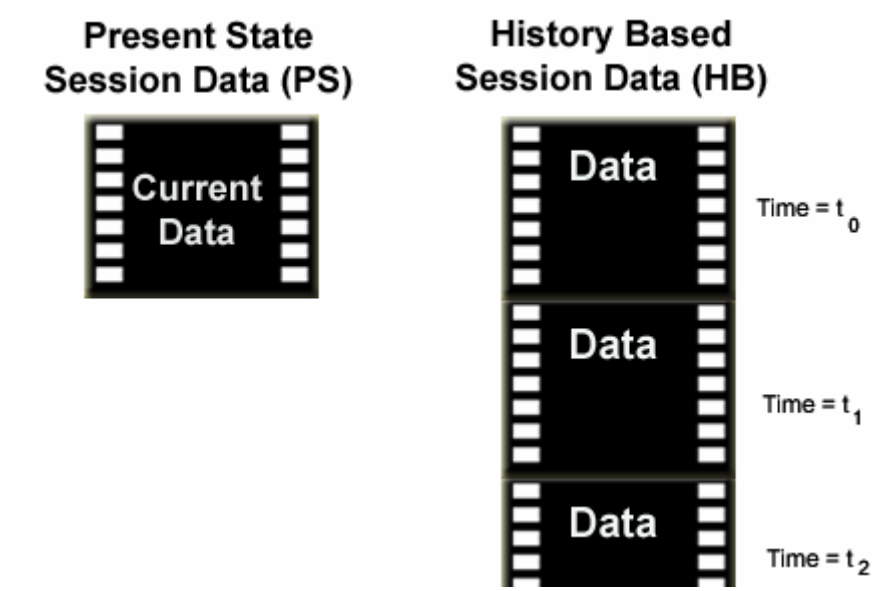


Figure 22 – History Based vs. Present State Session Data

4.4.6.2 Collecting History Based Session Data from Discontinuous Navigation

To prevent extremely large models from getting cumbersome to navigate through, it is common to introduce a set of viewpoints. The client can at any time choose whether to navigate continuously or to relocate by means of jumping to an arbitrary viewpoint. Allowing the client to perform such jumps is potentially a problem when trying to interpret History Based Session Data; Calculating the client's general direction and speed from the HB would give obscure, if not meaningless, results (because the client might just have jumped thousands of kilometers within the last second).

On the other hand, the discontinuity in the navigation may also be exploited. It can serve as a means for grouping bundles of tiles together. This allows the heuristics to delete and rebuild multiple tiles at the same time. Before explaining this with an example, we introduce the term *History Sequence* (HS), or *Sequence* for short, as being "continuous History Based Session Data, starting with a viewpoint (e.g. at start-up) and ending with a jump". The jump marking the end of a sequence can either be a jump back to a previously visited viewpoint or to a new one. In much the same way as we defined the LOU for tiles, we can also define the LOU for a History Sequence. The LOU of an HS is a unit of measure for the probability of the client revisiting the viewpoint associated with this HS in the immediate future.

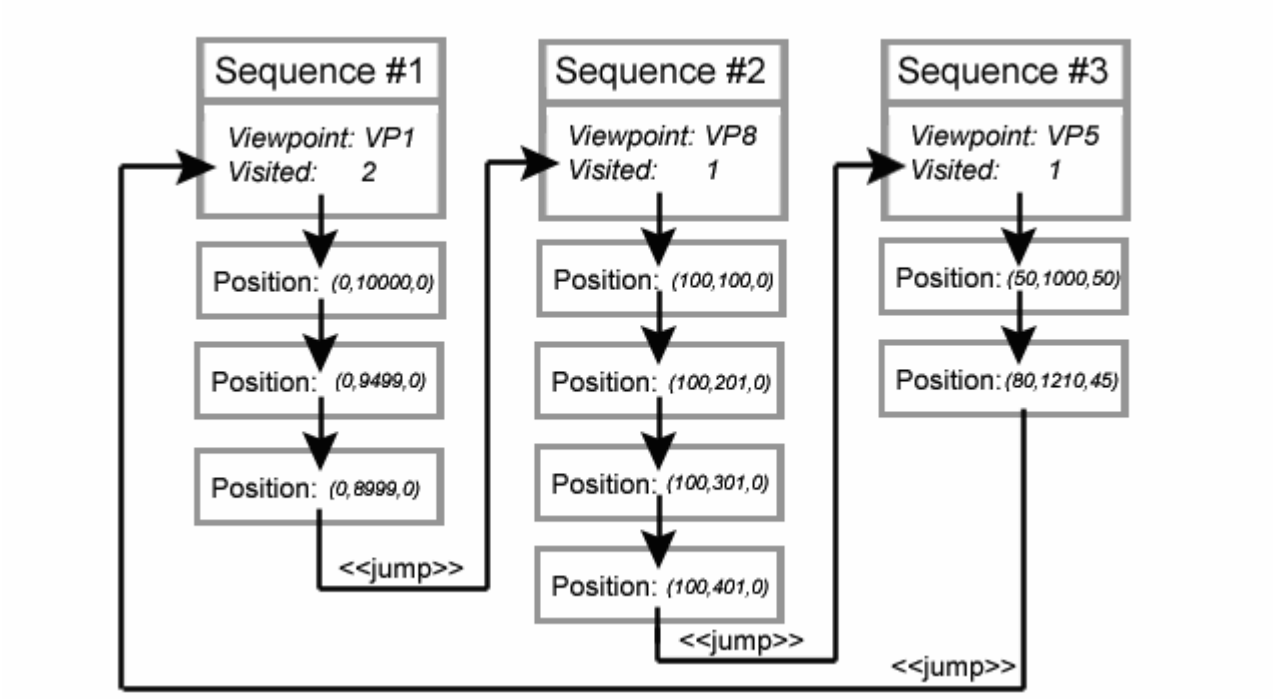


Figure 23 – Introducing History Sequences (HS) in History Based (HB) Session Data

Let us now elaborate on the dividing of HB into History Sequences with an example. Consider a fictitious client, navigating through a 3D model by means of both continuous navigation and jumping to arbitrary viewpoints. For the sake of simplicity, the only data given to the server is the exact position of the avatar. If the client in our example starts out at the initial viewpoint VP1, positioned at such a high altitude that only the root tile is loaded. The client, wondering what his home town looks like, starts diving straight towards the model. Figure 24 is an attempt at visualizing the data stored on the server after our client has navigated through the model. As soon as the client visits a viewpoint, a new *Sequence* object is created in the history. Each Sequence refers to a viewpoint and a list of snapshots. In our example, the avatar's position is the only data stored. In addition to the snapshot list, each Sequence keeps track of the number of times the viewpoint has been *visited* (we say that the client *visits* a viewpoint each time he or she jumps to it). In the figure, we can see that the client is diving towards the model as three snapshots are registered with decreasing y-value. The client may by now have realized that it takes too long to navigate this way and chooses to jump to a viewpoint very close to his home town. This ends the first Sequence and a new Sequence is created, referring to viewpoint *VP8*. Since this is the first time the client has visited VP8, the *visited* field is set to 1. The client realizes that he has zoomed in too much

and can therefore not find out where he is. He tries to regain control by navigating upwards. This results in several snapshots in the second Sequence. Finally, he finds a viewpoint that takes him directly to his home town, and he performs one more jump, this time to viewpoint VP5. A new Sequence is created, but the Tile Cache is by now running out of free space. Here, the Garbage Collection part of the heuristics can choose to flush tiles in one of two ways: either one tile at the time or several tiles based on an entire Sequence. If the client now returns to the first viewpoint, the Pre-builder faces much the same choice as the Garbage Collector did. It can choose to re-build tiles individually or it can reconstruct either parts of, or the entire set of, tiles that were requested the last time this History Sequence was visited. By doing this, the Pre-builder assumes that the navigation performed in the immediate future will not deviate much from that of the last time this viewpoint was visited.

Before going any further, it is important to note that the History Sequences are stored in the Session Data part of the heuristics, and do not contain any actual tiles. Being part of the Session Data, History Sequences are meant to be input to the heuristics agents maintaining the actual tiles in the Tile Cache (as shown in figure 20 previously in this paper). Similarly, a snapshot is not the same as a tile, but may in some implementations contain references to, or tile ID of, one or more tiles.

4.4.6.3 Tile Level vs. Sequence Level Operations

In our previous example, where we had introduced History Sequences, we saw that the Garbage Collector faced two choices when flushing tiles from the Tile Cache. It could either flush one tile at the time (a *Tile Level Operation*) or flush all tiles connected to a History Sequence (a *Sequence Level Operation*). An advanced implementation would try to find out which of the two choices seems most convenient. If the heuristics needs to free lots of space in the Tile Cache, then a Sequence Level Flush would seem appropriate. However, there may be special situations where a Sequence Level Flush is not suitable, e.g. if all the History Sequences have high LOU. It may be more effective to delete a few tiles individually, since a History Sequence may have a totally different LOU the next time the Garbage Collector is flushing. As we saw at the end of the previous example, the Pre-builder part of the heuristics can also operate on either Tile Level (pre-building each tile individually) or Sequence Level (pre-building all tiles needed last time the client jumped to the current viewpoint).

4.4.7 Server-Side Data vs. Client-Side Data

Both SBD and HBD can be further subdivided into *Server-Side Data* and *Client-Side Data*. This gives us four different types of data (as shown in figure 25 with its four leaf nodes).

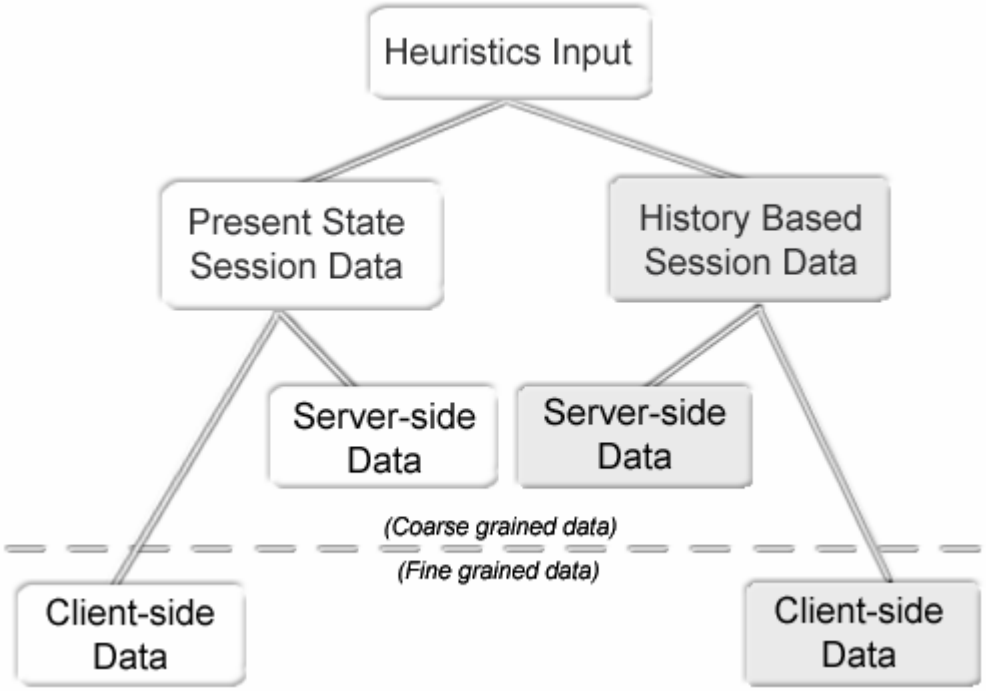


Figure 24 – The Four Different Types of Heuristics Input

4.4.7.1 Server-Side Data (Coarse-Grained Data)

When relying on Server-SideData solely, the server perceives the client as being a "black box"; The server does not know what happens internally on the client side (such as orientation of the avatar, the navigation speed or frame rate) but can extract estimates from the small amount of information it receives. The server receives HTTP requests and is responsible for delivering the proper 3D content back to the client as fast as possible. The only information the server retrieves from the client is the ID of the tiles requested. Since heuristics based on Server-Side Data can only calculate trends and navigation based on which tiles are loaded when, we sometimes refer to Server-Side Data as *Coarse-grained Data.* Note that even

though the tile ID of the requested tile is, strictly speaking, data sent from the client, tile ID is the absolute minimum of information that can be received from the client side. The server is virtually self-sufficient and independent of advanced parameters from the client. There are, however, much information to be collected from Server-Side Data. Let us assume that a client browses a 3D model and starts out at an initial viewpoint so high above the ground that the root tile with lowest level of detail is requested. The server can from this information predict that the next tile being requested is either the 'A', 'B', 'C', or 'D' tile since these are in immediate proximity to the root tile.

As mentioned in section 4.4.5.3 "Distance", we can make a rough estimate as to the position of the avatar just be looking at what tiles are requested. When a client requests tile T, we know that the avatar must be outside the DB (Detail Boundary) of T (if not, four more detailed tiles would have been requested) and at the same time inside the DB of tile T's parent (if not, a tile with less details would be requested). Note that in order to distinguish between the DB of a tile and the DB of the tile's parent, we sometimes refer to a tile's DB as the *Inner Boundary (*IB*)* and the parent tile's DB as *Outer Boundary* (OB) (see figure 26). As a rude estimate of the avatar's position, we define the term *mean center*: The mean center of Detail Boundary B is the average point between the Inner Boundary and the Outer Boundary.

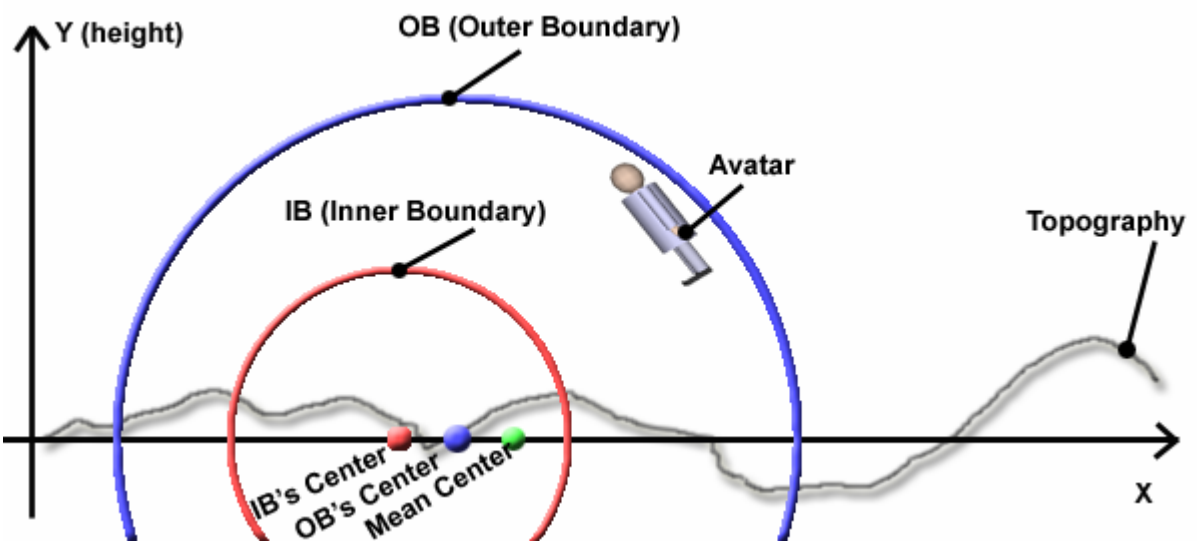


Figure 25 – Introducing Inner Boundary, Outer Boundary, and Mean Center

The mean center can be thought of as the weight center of the space between the two boundaries. It depends on two factors. The first is the dislocation of the IB within the OB, which can be described as the vector between the center of the IB and the center of the OB. If the IB's center is displaced far from the OB's center, then the average position of the avatar is somewhere in the opposite direction. If the position of the IB's Center is identical to the position of the OB's Center, then the Mean Center is also located in this position. The second factor is the difference in size between the IB and the OB. An IB with radius = 0 does not displace the average avatar position from the OB's Center whereas a huge IB (relative to OB) will displace the average avatar position a great deal from the OB's center. An important thing to note is that the Mean Center will always have height y=0 when we represent the boundaries as spheres. This is due to the fact that the distance from a tile, whether it is detailed or coarse, is measured from the center of a tile and the center is always at "sea-level". The center of Detail Boundaries will therefore always be positioned at y=0 as the vector between two points with y=0 will also have y=0. Fortunately, we can benefit from the fact that the avatar is never, or at least seldom, positioned below sea-level. This results in hemisphere shaped detail boundaries, as shown in figure 27 and 28. Calculating the weight center will now yield a 3D coordinate with a meaningful y-value that is greater than zero.

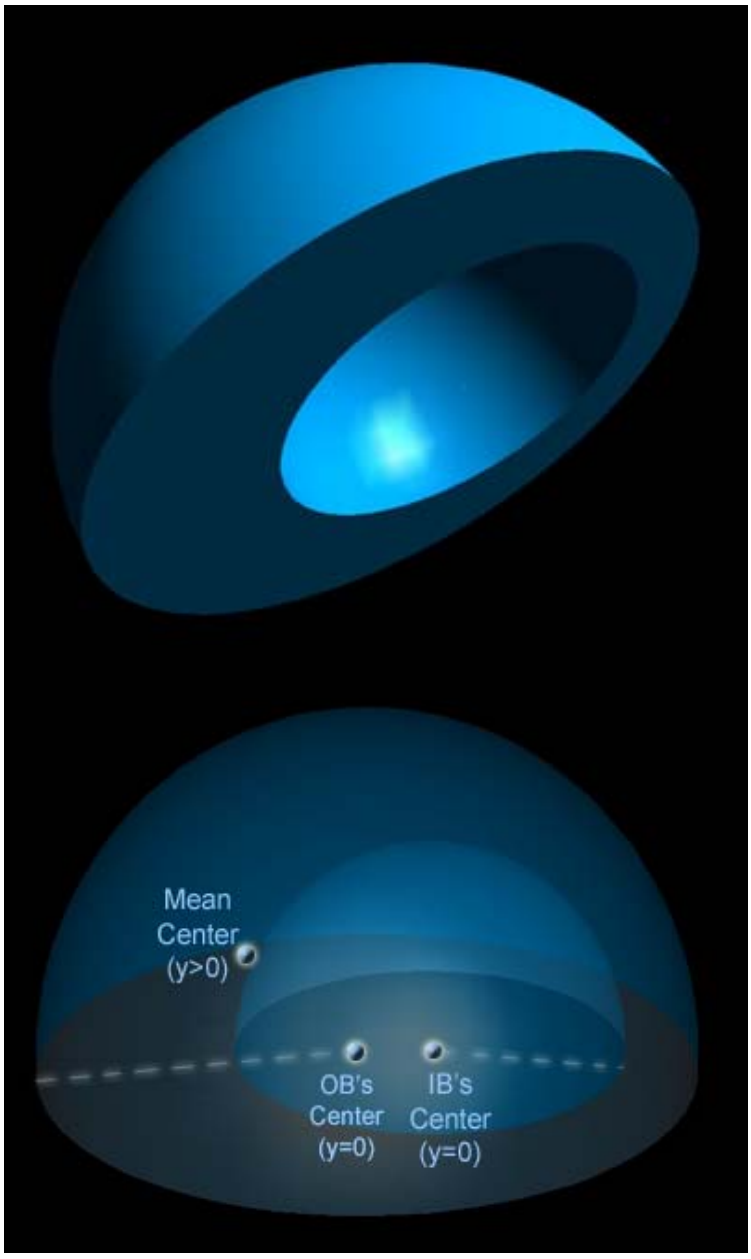


Figure 26 & Figure 28 – Estimating the Y-value of the Mean Center By Using Detail Boundaries Shaped as Hemispheres (A at the top, B at the bottom)

It is also possible to estimate the orientation of the avatar from Server-Side SessionData, as long as the data is history based (HB). The session data contains a list of snapshots containing tile ID and timestamp for each requested tile. The client will request tiles that are small, detailed, and have relatively long tile IDs for the terrain that is close to the avatar, and big, coarse tiles with relatively short tile IDs for terrain that is far away from the avatar. The orientation of the avatar in the xz plane can be calculated by finding the general direction going from the location of the high-level tiles being requested to the location of the low-level tiles requested. Figure 29 shows that the orientation of the avatar in the xz plane is directly connected with the level of detail in the requested tiles (From highly detailed tiles towards tiles with few details).

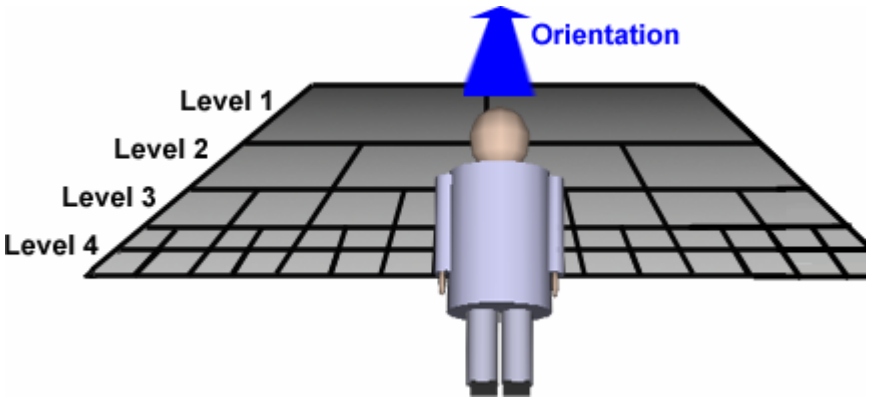


Figure 27 – Estimating the Orientation of the Avatar Based on Coarse-Grained (Server-Side) Data

4.4.7.2 Client-Side Data (Fine-Grained Data)

Client-Side Data is information collected and passed on to the server by the client. How this data is passed on to the server depends on the technology used. If the client side application is a VRML browser and the server's interface is a servlet, then the Client-SideData can be added as parameters to the URL in the HTTP requests. Since Client-SideData can be very accurate (we can for example calculate navigation speed, and the avatar's orientation every half second) we often refer to Client-Side Data as *Fine-Grained Data*.

One way of sending Client-SideData to the server is to add parameters to the URL in the HTTP requests sent to the server. If the server, being a servlet, has the URL "http://url/servlet" and we want to send the following Client-Side Data to the server: value1, value2 and value3, then the resulting HTTP request would be: "http://url/servlet?param1=value1&param2=value2&param3=value3". A considerable problem with this approach is that in VRML, changing the URL in the LOD nodes after the tile has been loaded results in "undefined behavior". This means that it is up to those who implement the VRML browser to decide what happens. It might give the desired result when viewed in one particular browser, whereas it might give an erroneous result within another browser. Another drawback with changing the URLs in the LOD-node is that it is arbitrary *when* these requests are sent to the server. At worst, if the user is navigating in such a way that no tiles or objects change level of detail, the server will never receive any data from the client.

Another approach is to exchange the current virtual universe with a new copy at given intervals. In VRML this can be done by using ECMAScript (also called JavaScript). The *Browser* object defines the method *loadURL,* taking an URL as a parameter that specifies the location of the new virtual universe. By adding Client-SideData as parameters to this url, including the current orientation and position of the avatar, the server can both build an exact copy of the universe and retrieve Client-SideData. Since the server receives the orientation and position of the Avatar, it is capable of setting the avatar back in the same position and with the same orientation as it was in the original universe.

A third approach, and perhaps a more rude one, is to rely on the browser's ability to ignore invalid links or links to non-existing VRML content. This can be done by letting the servlet reply with an HTTP error message instead of an HTTP response, e.g. "404 - Not Found", each time the client sends Client-SideData. The servlet stores the Client-SideData found in the parameters but ignores the actual request. When the client receives the "404 - Not Found" error message, it has no choice but to continue browsing the world as if the request was never sent. This means that we can invoke the *loadURL* method to send Client-SideData to the server without swapping the current universe with a new one. However, as I have already stated, this is a rude and indelicate implementation since different VRML browsers might deal with non-existing VRML contents differently.

The last method I will mention here is the use of the *External Authoring Interface* (EAI). EAI was proposed as an addition to the VRML 2.0 specification. is an interface that "allows you to

control the contents of a VRML browser window embedded in a web page from a Java (tm) applet on the same page" [28]. It allows so-called "embedded objects" on a web page to communicate with each other. An embedded object is simply an object, such as a plugin or an applet, that resides in an HTML document [29]. According to the *blaxxun Developer Information pages* [30], the EAI enables the applet to interact with the VRML world in four ways: "Accessing the functionality of the browser interface (i.e. to create new geometry)[,] sending events to eventIns of nodes inside the scene. (i.e. to change positions or colors of objects) [,] reading the last value sent from eventOuts of nodes inside the scene. (i.e. to find the last position of an object) [and] getting notified when events are sent from eventOuts of nodes inside the scene. (i.e. when an object is being clicked)". The third and fourth way of communicating should be suitable for passing Client-SideData to the server.

4.4.8 Forecast - Looking Into the Crystal Ball

Predicting which tiles are going to be requested in the near future is far from trivial. First of all, it is essential that the client navigates in a more or less predictable manner; We can for examples assume that the client performs continuous navigation such as "flying" or "walking" (as opposed to discrete navigation which is solely based on "jumping" from place to place), and that he or she tends to navigate in an approximately linear way. The latter assumption is often the case with VRML models due to the nature of the navigation controls in VRML browsers. Unlike the navigation in VR models such as flight simulators, the navigation possibilities in VRML browsers are a bit more awkward with a view to doing multiple forms of navigation simultaneously. As an example, changing the position of the avatar is often limited to straight forward, straight back, and, by pressing some sort of shortcut key, side stepping (straight left or right), up, and down. Changing the orientation can be done simultaneously as changing the avatar's position, but changing the avatar's position and changing the orientation both along the xz plane and the yz plane at the same time (analogous to jerking the flight stick back and to the left) requires, in the *blaxxun Contact browser*, the following combination: Pushing the mouse button while moving the mouse and scrolling the mouse wheel simultaneously. Fortunately, the somewhat gawky navigation in most VRML browsers can in some cases make the client's navigation more predictable and therefore increase the Heuristics' chances of predicting which tiles needs to be pre-built. In the following sections, several approaches for Heuristics prediction is proposed.

4.4.8.1 The Neighbor Tile Pre-Processing Model

The first and most simple approach, I have called the *Neighbor Tile Pre-Processing Model*. It is a "blind" approach, which means that it does not make use of neither the avatar's orientation nor position. It simply assumes that the client is navigating continuously and will therefore request one or more of the neighbor tiles (tiles that are adjacent to the tiles already requested). Since the Heuristics cannot decide *which* of the adjacent tiles are going to be requested next, it will have to build either all of them or randomly select some. To extend this model even further, the Heuristics can also preprocess the neighbors of the neighbors of the requested tiles. The definition of how near two tiles need to be before they are defined as neighbors, which decides how many tiles that should be pre-built for each tile requested, should be determined by factors such as: the size of the model, the LOD range used (the limits deciding when to apply more details also decides, indirectly, how many tiles are visible at the same time), and the performance of the hardware on which the Heuristics is deployed). Figure 30 shows Heuristics that pre-builds (prepares) only the closest neighbors of the already requested tile. This includes the parent tile, the four children tiles and the tiles to the north and east of the requested tile. (There are no tiles to the west and to the south of the requested tile.)

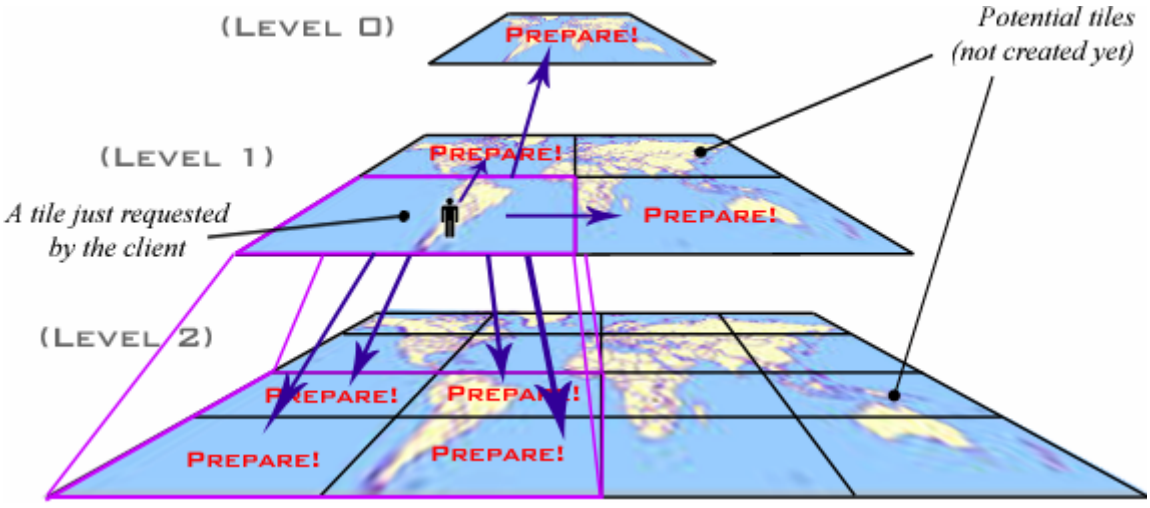


Figure 28 – An Illustration of the Neighbor Tile Pre-Processing Model

4.4.8.2 Extrapolation - Predicting the Future Mathematically

Extrapolation is the act of "[...] estimating a value by projecting or extending known values" [31]. Theoretically, it should be possible to derive advantage from this mathematical paradigm and forecast the position of the avatar in near future based on information collected throughout the session. However, it is important to note that "[...] extrapolation is not very reliable and the results so obtained are to be viewed with some lack of confidence. In order for extrapolation to be at all reliable, the original data must be very consistent" [32]. In this thesis, I will try to use extrapolation, possibly in combination with other techniques, to reduce the lag in 3D-models which are generated "on-the-fly" and thereby enhancing the temporal fidelity of the models. However, it should be mentioned that navigation in 3D models deployed on the Internet is hard to predict compared to e.g. flight simulations, and therefore decreasing the consistency of the data. As an example, it is not possible to fly with a very high speed, then stop dead and start a movement in the opposite direction in most flight simulators. This is navigation behavior expected by most clients in a topographical 3D model, as is side-stepping and jumping between different viewpoints. I will later in this thesis present results from different test cases where extrapolation is implemented.

4.4.8.3 Extrapolation Based on Different Types of Data

Perhaps needless to say, extrapolation requires History Based Data (HBD), as described in section 4.4.6 "Heuristics Input", as input for its projections, or rather calculations. Hopefully, the client is sophisticated enough to provide the server with Client-Side Data (or fine-grained data) such as exact avatar positions. If Client-Side Data is not available for the server, avatar position must be estimated, e.g. through calculating the mean center such as described in section 4.4.7.1 "Server-Side Data (Coarse-Grained Data)". Using extrapolation to estimate future avatar positions based on data that is estimated in the first place, leads to an accumulative inaccuracy in the calculations. It may be the case that this inaccuracy causes the gain of extrapolation not to be worth the effort. Nevertheless, in a bold (or perhaps futile) attempt, extrapolation based on Server-Side Data is tested in the first test case presented later in this thesis, if nothing else, to exclude the combination.

4.4.8.4 An Implementation of Extrapolation

In this section, one possible way of implementing extrapolation based on Server-Side Data will be described. Of the latest tiles requested by a client, those with the highest level of detail is selected, and a Mean Center is calculated for each by extracting an Inner and an Outer Boundary and then estimating the weight center between the boundaries (as described in section 4.4.7.1). Since each Detail Boundary is far from accurate by itself, we group tile requests that occurred approximately contemporarily before we finally calculate an *Average Avatar Location* (AAL) for each group. Figure 31 shows the detail boundaries generated from eight tile requests. The tile requests are, as already mentioned, sorted into a set of groups based on the timestamp of the requests. In this example, they are divided into two groups: recent requests (colored red) and relatively old requests (colored blue). For each Detail Boundary, the Mean Center is calculated, marked by human-like symbols. Next, the Average Avatar Locations are calculated, being an average position from all boundaries within a group (symbolized by a black cross), one for each group. These AALs constitute the known values for which the projection of future values is based on. Naturally, if the server has History-based, Client-Side Data, containing a list of exact avatar positions, this would enhance the accuracy considerably. Since we in our example only have two request groups, and therefore only two avatar locations, we are only capable of extrapolating the future point as an extension of the linear movement performed between AAL#1 and AAL#2.

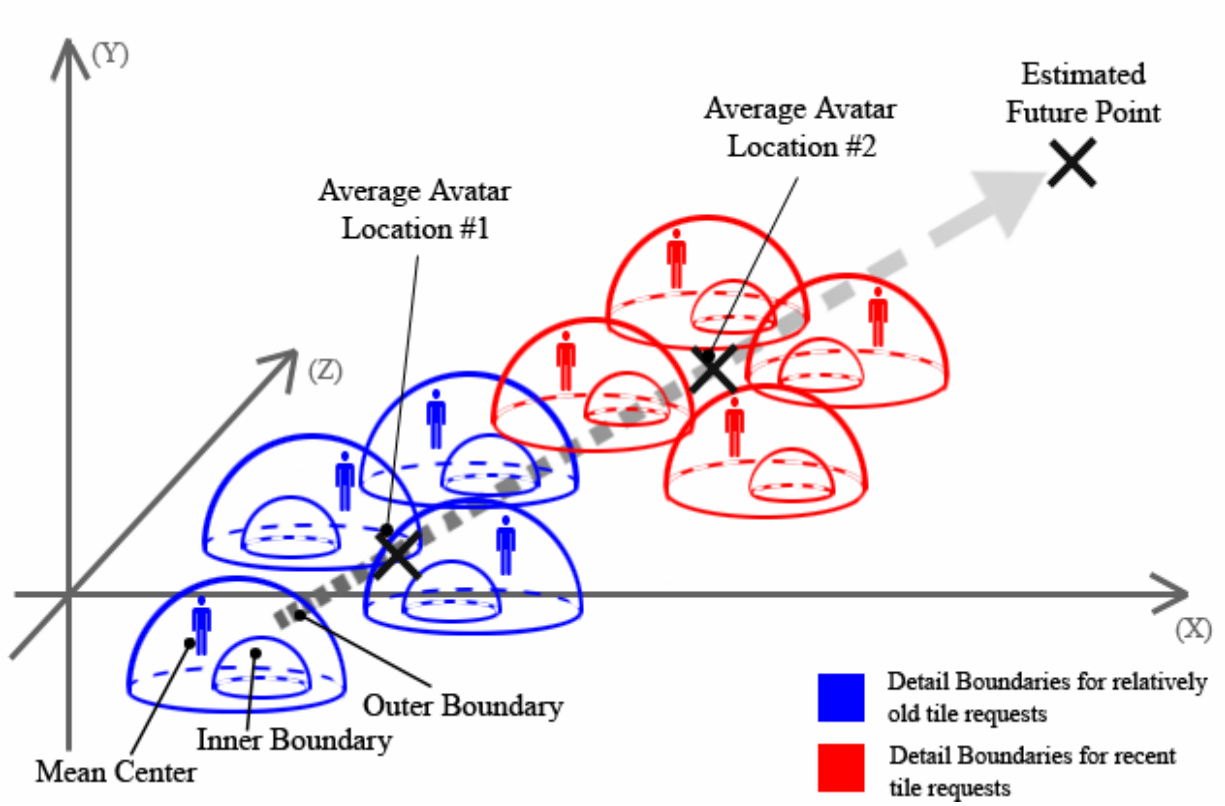


Figure 29 – Extrapolating Future Points From History Based Data

4.5 A FIRST DRAFT OF AN "ON-THE-FLY" GENERATOR

In order to measure the efficiency of different approaches, it is best to dismantle the generator into many interchangeable modules. This allows us to implement a variety of variants of the framework just by replacing a single module. This way we can measure the effect of the different implementations of the modules without needing to re-implement the rest of the generator. Figure 32 shows a draft of how a generator can be divided into modules.

In the lower right corner, we find the client, who sends requests to the HTTP Interface, which can be a CGI script or a servlet. The HTTP request is then passed on to the Dispatcher, whose job is threefold; First of all, it is responsible for storing the parameters that follow each HTTP request. This data becomes valuable when the heuristics makes qualified guesses about which tiles are going to be requested in the near future. The second responsibility of the Dispatcher is to retrieve the tile name (or tile ID) from the request and fetch the tile from the Tile Manager. The Tile Manager then checks if the requested tile is currently located in the Tile Cache. If not, it issues an order to the Tile Builder, telling it to build the tile with the given tile ID. The Tile Manager then returns the tile to the Dispatcher, and the Dispatcher passes it on to the HTTP Interface (e.g. a servlet) which then returns it to the Client (with an appropriate MIME-type so that the returning file is interpreted correctly by the client, e.g. "model/vrml" for VRML models).

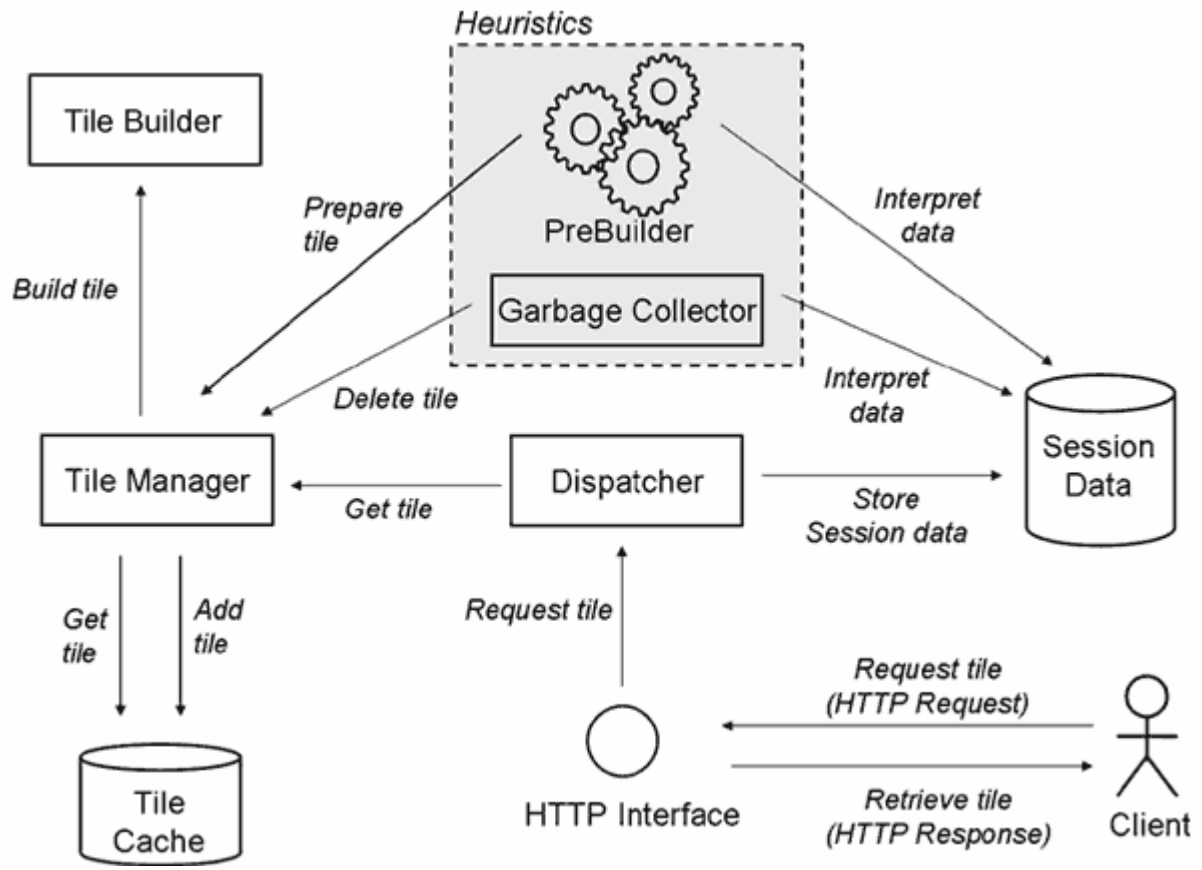


Figure 30 – A Conceptual Model of the Framework for "On-the-Fly" Generation of 3D Tiles

The heuristics part of the framework (placed in the gray box) is an attempt to enhance performance by building the tiles before they are requested. To allow this, we need a mechanism that adds tiles that are likely to be used again in the near future to the tile cache. This is the "PreBuilder" module. However, since all forms of storage are limited, we also need a mechanism that can delete the least valuable tiles from the cache. This is the "Garbage Collector" module. Both modules consult the session data (a collection of request parameters) stored by the Dispatcher in order to rank the value of the tiles. In addition, the Garbage Collector is able to send a request to the Tile Manager for the least frequently used, or oldest, tiles.

5 Results

5.1 TEST CASE 1 - A COMPLETE, SMALL-SIZED MODEL

5.1.1 The Simulations

The test case consists of 32 isolated simulations using a 3D model of Fredriksten Fortress in Halden. This model is a slight modification of Herman Kolås' VRML model of Halden [5], which consists of 341 tiles dispersed over 5 levels of detail. Figure 33 shows a snapshot of the model. This is, strictly speaking, a small model compared to the models this thesis is meant to enhance, but, nevertheless, it is a model that is easily accessible and easy to use. (It has been built using the Rez toolkit created by Chris Thorne, and therefore, it can be deployed on a servlet after running it through the *Rezerver* converter tool [33]). Hopefully, this test case should give us an idea of the efficiency of the basic precaching heuristics. It should be further stated that the model is accommodated and deployed on an implementation of the framework described in section 4.5 "A First Draft of an "On-the-Fly" Framework" (see also figure 32) so that each tile must be requested from a servlet where the tile's ID is given as an HTTP parameter. Various information is collected from the requests, such as timestamp and ID of the requests, and this makes up a set of data on which the heuristics can base its qualified guesswork. Furthermore, since we need to test the heuristics on a model that builds the tiles on an "on-demand" basis, or rather an "on-the-fly" basis, (which suffers from an arbitrary build delay,) and the Halden model is static (it suffers only from delay caused by I/O and transmission delay), we need to add an extra delay on each tile request. It is not possible to predict the exact length of the build delay, since it is dependent upon a multitude of factors, e.g. the complexity and amount of 2D data for which the tiles are generated, or the current workload on the server. Therefore, the simulations are repeated eight times, each time with a different build delay, ranging from 32 to 4196 milliseconds. Each tile request is handled in its own thread on the server side. In order to make sure that all simulations are identical, a predefined tour, taking 120 seconds, is programmed, controlling exactly how fast and where to navigate in the model. The path of this tour is recorded while navigating through the model using the mouse controls in the blaxxun Contact 3D (version 5.104 Direct3D) browser. Neither the jump function nor the sidestepping function is used. The tour is made extensive enough for the tile cache containing recently requested tiles to be congested. This way we

make sure that both the Garbage Collector performs some flushing and that the Tile Cache is not filled with all the tiles in the model. (If that were possible, the model would act as if it were static from the moment all the tiles in the model were resident in the tile cache.) The capacity of the tile cache is 128 tiles, and whenever the tile cache is congested, the 25 percent of the tiles that are oldest is purged. This garbage collecting heuristics applies for all the simulations in this test case. Since the predefined tour should, ideally, cause the client to request 235 tiles, the garbage collector should be activated three times, deleting 32 tiles (25%) each time. Finally, the set of eight simulations is repeated four times, each time using the same model and the same tour (navigation), but using different heuristics setup for precaching. The next section describes these setups in more detail. The test case results in 32 isolated simulations, as mentioned introductorily in this section, allowing for a decent comparative study of the four different heuristics.

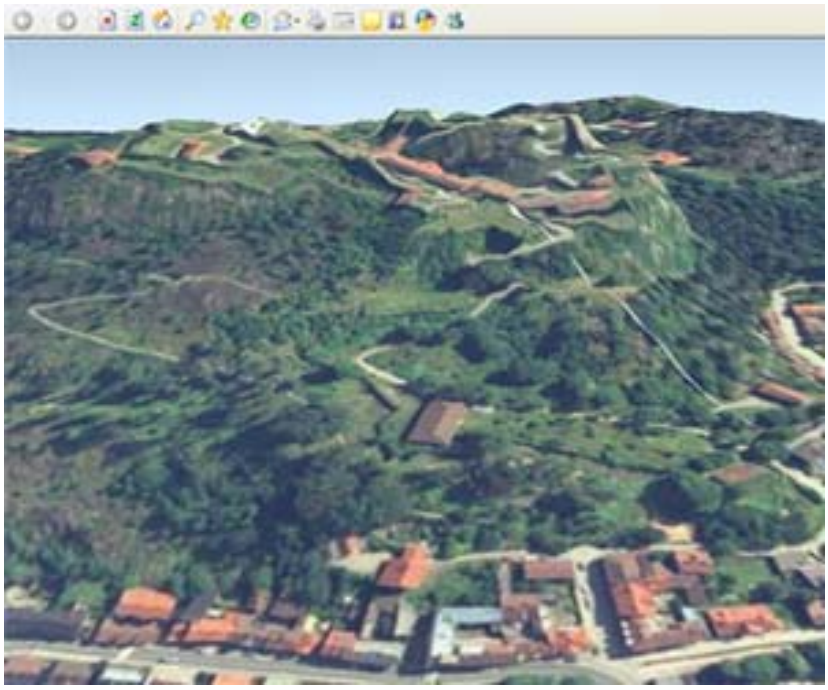


Figure 31 – Snapshot of Herman Kolas' 3D model of Fredriksten Fortress in Halden

5.1.2 Precaching Setup

As mentioned above, the set of eight simulations is repeated four times; each with a different setup for the precaching part of the heuristics. These setups are referred to as *None*, *Random*, *Extrapolation*, and, finally, *Neighbor,* and will be described in further detail in this section.

5.1.2.1 None

The setup called None has, as the name implicates, no form of heuristics in which precaching of tiles is performed; the tiles are built only when the client requests them. Running simulations with a bare minimum of improvements gives us a basis to compare the other heuristics setups with.

5.1.2.2 Random

The Random heuristics is a special setup that is impracticable for very large models or for server implementations with a very large tile cache. This setup will function as a very small improvement with which other setups can be compared. It precaches tiles arbitrarily so that the tile cache is filled with a random set of tiles before the simulations start. The reason for perform these simulation, despite the fact that they are not feasible for very large models, is to check the reliability of the Extrapolation setup (which we will describe next). The thought behind this setup is to test whether the Extrapolation precaching setup is more efficient than the precaching of tiles that have been selected arbitrarily. If this is not the case, then the implementation of Extrapolation may not be worth all the extra calculations. Either that, or a new and perhaps more sophisticated heuristics setup, should be explored, such as a combination of Extrapolation, Neighbor, and Random (a randomly chosen number of the extrapolated tile's neighbors might be precached, alternatively in addition to a randomly chose number of the requested tile's neighbors).

5.1.2.3 Extrapolation

The Extrapolation setup is an implementation of what is described in section 4.4.8.4 "An Implementation of Extrapolation". It includes calculations of AALs from two groups of detail boundaries which are based on coarse-grained Server-Side Data. Using only two groups of boundaries results in linear extrapolation. This, combined with the fact that the "known" values on which the projection of future point is based are in fact estimated (uncertain), leads to the fact that this setup is only applicable for estimating values in *near* future. For more details, see figure 31. The extrapolation precaching is triggered each time the client requests a tile.

5.1.2.4 Neighbor

The Neighbor setup is implemented as described in section 4.4.8.1 "The Neighbor Tile Pre-Processing Model". It defines the term neighbor to include only the tiles that are in immediate proximity of the requested tiles, and is therefore not a very complex or costly heuristicssetup regarding server performance. As is the case with the Extrapolation setup, the Neighbor precaching setup is triggered each time the client requests a tile.

5.1.3 The Output of the Simulations

The output of each of the 32 isolated simulations can be divided into four measures, viz. frame rate, response time, tile loss, and, finally, precache ratio. Each of them will be explained in detail below.

5.1.3.1 Frame Rate

The frame rate measure presented in this paper's tests stands out from the frame rate measure describing most 3D applications, or even 3D games, and thus, needs some explanation. The frame rate is retrieved from the VRML browser using the `getCurrentFrameRate`

method. The frames per second returned by the browser tells us *how fast the browser renders the part of the 3D world that is both currently retrieved by the client and visible from the avatar's viewpoint*. If some parts of the model have not yet been received by the client, possibly due to high build delay and transmission delay, the browser will ignore these parts of the model, and it is therefore able to deliver a higher frame rate. Hence, the frame rate may serve as an indication of lag in the tile request process, where a high frame rate indicates build delay (and/or transmission delay). Finally, it should be mentioned that the frame rates returned from the browser may surpass the frequency limit of the input device, i.e. the monitor. It is therefore not likely that the client will be able to distinguish between 500 and 600 frames per second with the bare eye.

5.1.3.2 Response Time

Response time is by large self-explanatory. This is the average time it takes for the server to give a reply to a tile request, or more precisely, the average time it takes from the HTTP request is retrieved by the servlet to the servlet is able to send the HTTP response back to the client. The reason why we do not measure the response time on the client side is that the transmission delay between the client and the server is irrelevant since it does not depend on any of the changes done between the simulations. Naturally, we strive to keep the average response time as low as possible, since lag reduces the temporal fidelity significantly, and thereby also reduces the experience of immersion and presence. The heuristics setup which results in the lowest average response time is preferred.

5.1.3.3 Tile Loss

Tile Loss is what occurs when the browser requests a tile, but is not able to retrieve it from the server and render it until its use has expired. The client will experience a hole in the model which is not filled with topography before either the hole is out of sight or the hole is replaced by a tile with a different level of detail. This differs from *lag*, which is when the client can see the hole being filled with the delayed tile before its use expires. Tile loss depends on three factors, viz. the build delay, the transmission delay between the client and the server, and the amount of time the tile, or the hole where the tile should be, is in the sight of the the avatar.

High degree of tile loss reduces the temporal fidelity of the model significantly, and, with that, also the sense of presence. Even though tile loss is closely connected to response time, tile loss is by far worse with a view to the fidelity of the model. Hence, the heuristics setup which gives the lowest degree of tile loss is preferred.

5.1.3.4 Precache Ratio

Precache Ratio is the degree to which the requested tiles can be directly accessed from the server side tile cache, and thereby avoiding the build delay caused be the tedious task of building tiles. Improving the precache ratio will decrease the response time directly, and therefore, the heuristics setup which gives the highest precache ratio is preferred.

5.1.4 The Results of the Simulations

In this chapter, the results of Test Case 1, described in the section prior to this, will be presented, starting with the measuring of frame rates. Conclusions and possible improvements will be presented in the next section.

5.1.4.1 Frame Rates

Table 3 and figure 34 show the frame rates for the different heuristics setups simulated. The first column in table 3 is the delay simulated by the on-the-fly generation of each tile measured in milliseconds. Note that high build delay will severely diminish the temporal fidelity of any 3D model. Build delays higher than 1,000 ms should therefore be avoided at all costs. Recall from section 5.1.3.1 "Frame Rate" that the frame rates measured by the browser are the frequency of images delivered *by the browser*. Therefore, low frame rates suggest good heuristics because a larger percentage of the topography has been successfully retrieved and this increases the amount of graphics to render. Figure 34 shows relatively small differences in frame rate for build delays below 256 ms, giving us little to go by when deciding which setup is the best. However, when the building of 3D tiles takes more than 256 ms, we can see that the Neighbor setup gives the client more topography to render. When it

comes to the difference between Random and Extrapolation, it is disappointingly, although not surprisingly, small. Note also the seemingly small benefit from all kinds of heuristics for build delays less than 64 ms; at 64 ms, the lowest frame rate is actually measured for the None setup which contains no precaching heuristics at all. Finally, it must be mentioned that he measuring of frame rates delivered by the browser is inaccurate and should only be used as a supplement to the three other measures, either verifying or disapproving them.

Build Delay	None	Random	Extrapolation	Neighbor
32	269	263	255	278
64	238	271	259	263
128	269	253	253	269
256	287	289	286	262
512	342	323	340	293
1024	429	402	404	325
2048	494	457	449	408
4196	529	559	552	508

Table 3. The Average Frames per Second Delivered by the Browser Throughout the Simulations

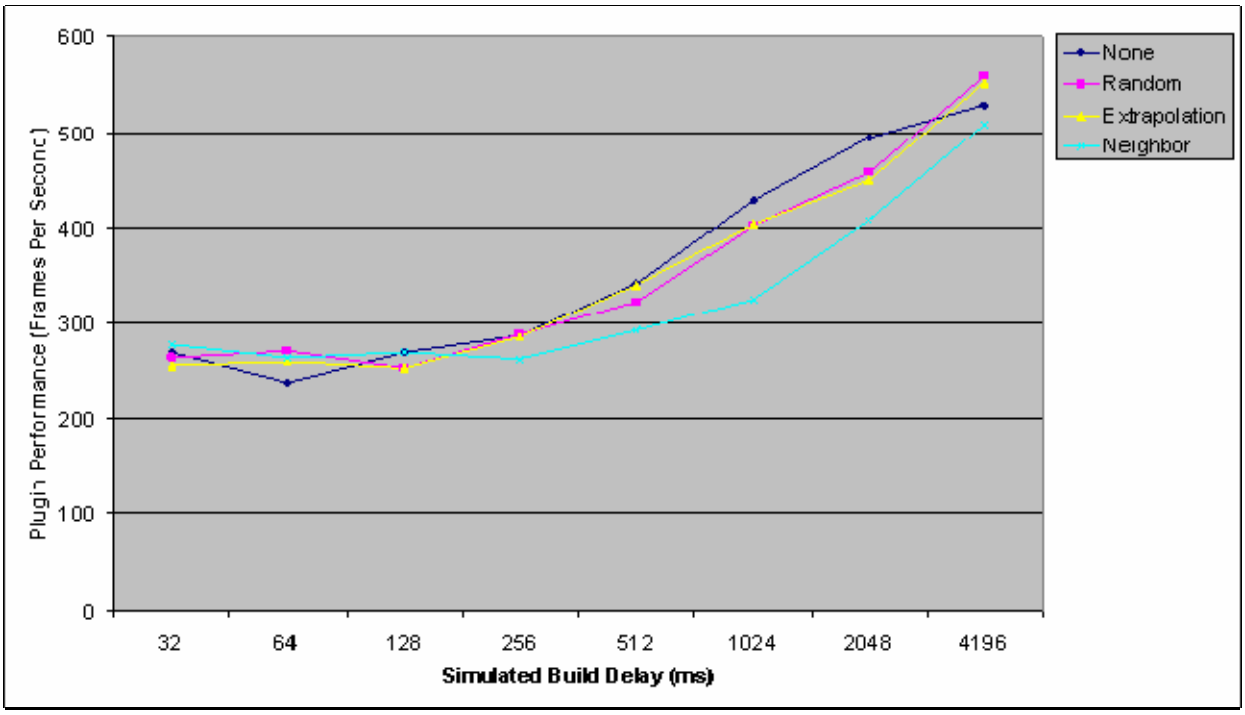


Figure 32 – The Average Frames per Second Delivered by the Browser Throughout the Simulations

5.1.4.2 Response Time

Response Time is the time elapsed from the server receives a request until it is able to respond, as described in 5.1.3.2 "Response Time". Keep in mind that this is measured twice by using the `System.currentTimeMillis()` method in the Java API to get two timestamps and then looking at the difference. According to Jack Shirazi [34], this method takes half a millisecond to perform, which gives us a potential inaccuracy of 2% when measuring spaces of time around 50ms (which is the case here). Nevertheless, the results from the Response Time measure are evident (see table 4 and figure 35). When the build delay is 128 ms, using the Neighbor setup leads to a response time that is only 40% of the build delay maintained by the second best setup, i.e. Random. The differences between the None, Random, and Extrapolation setups are, at this stage when the build delay is simulated to be only 128 ms, not prevalent. Not until the build delay increases to 1,024 ms can we see a perceptible difference between the two other precaching setups (Random and Extrapolation) and the None setup. The Random and Extrapolation setups are then accomplishing response times that are 83% and 87%, respectively, of what the None setup accomplishes. The Neighbor setup is at that time the only setup that produces respectable response times, approximately 38% of what the None setup is capable of. These measures verifies what we saw hints of in the previous section; the Neighbor setup seems to yield the best performance, whereas the extra effort of introducing extrapolation seems to give poor results compared to precaching tiles arbitrarily (even when the tiles are arbitrarily cached only once, before the tour starts).

Build Delay	None	Random	Extrapolation	Neighbor
32	90	78	94	53
64	166	137	149	61
128	283	243	262	98
256	538	463	482	160
512	1038	938	943	382
1024	2050	1697	1775	783
2048	4116	3105	3140	1749
4196	8009	5775	6034	4516

Table 4 – The Response Time Delivered by the Server Side Throughout the Simulations

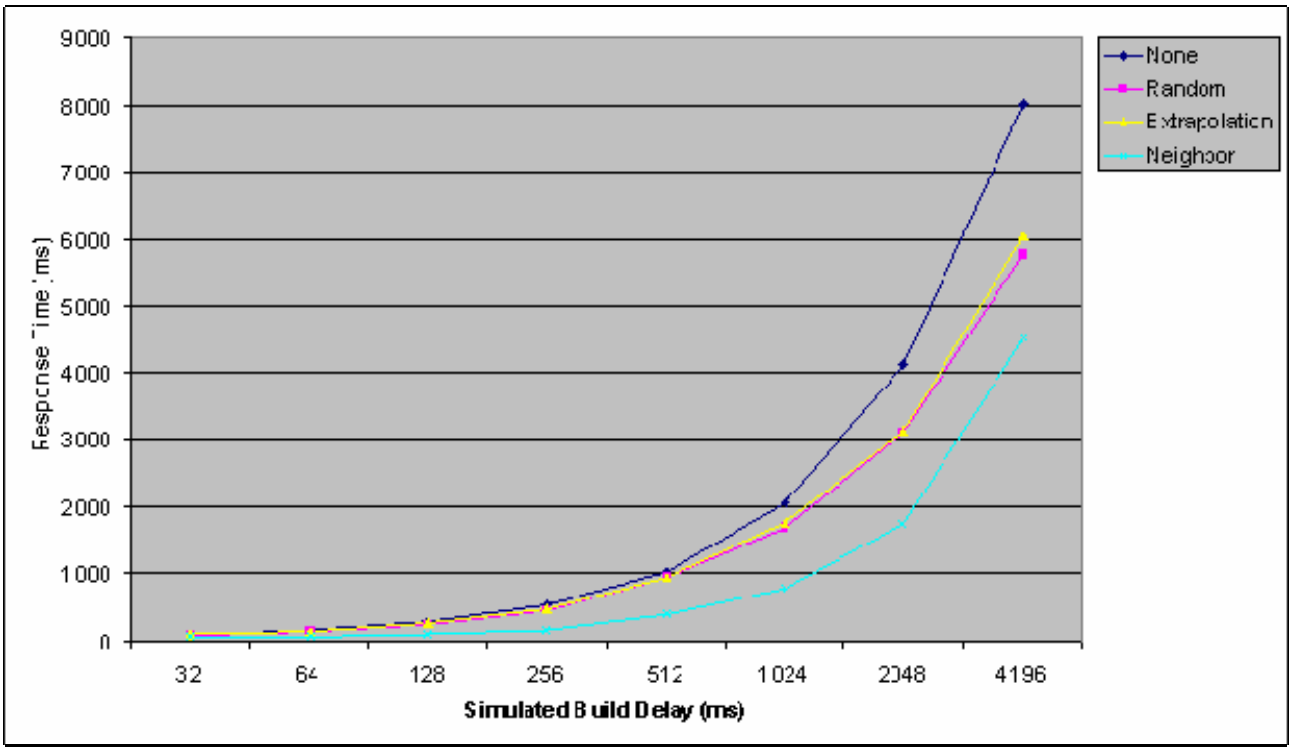


Figure 33 – The Response Time Delivered by the Server Side Throughout the Simulations

5.1.4.3 Tile Loss

Table 5 and figure 36 show the tile loss experienced by the client while running the different simulations in the test case. As described in section 5.1.3.3 "Tile Loss", this is the extent to which tiles become obsolete before the client is able to render them to the monitor (or any other output device). One thing that should be added is that tiles may be visible from the avatar's viewpoint for only a fraction of a second, for instance when it is visible only in one of the monitor's corners, and it may therefore not be possible to obtain 0% tile loss. Once again, the Extrapolation setup yields disappointing results, whereas the Neighbor setup turns out to be proficient. Neither the Extrapolation nor the Random setup reduced the tile loss significantly; the best result being the Random setup reducing the loss from 12.8% to 8.5% at 256 ms build delay (compared to the None setup). The Neighbor setup, however, kept the tile loss below 2% all the way up to the same build delay. For more heavy build delays, just above one second, the Neighbor setup has still been able to halve the tile loss percentage (21.3 % compared to None's 48.5%). For more extreme build delays, exceeding two seconds, the gain of using the Neighbor setup is less even though it is still prevalent. At a build delay of 2,048 ms the Neighbor and None setup yield 44.7% and 71.1% respectively.

Build Delay	None	Random	Extrapolation	Neighbor
32	0,4 %	0,4 %	1,3 %	0,4 %
64	3,0 %	1,7 %	2,6 %	0,0 %
128	5,5 %	4,3 %	4,3 %	0,9 %
256	12,8 %	8,5 %	8,9 %	1,7 %
512	25,1 %	23,0 %	19,1 %	10,6 %
1024	48,5 %	43,0 %	43,0 %	21,3 %
2048	71,1 %	62,6 %	63,4 %	44,7 %
4196	83,4 %	78,7 %	77,9 %	70,2 %
Average Tile Loss	**31,2 %**	**27,8 %**	**27,6 %**	**18,7 %**

Table 5. The Tile Loss Suffered by the Client Throughout the Simulations

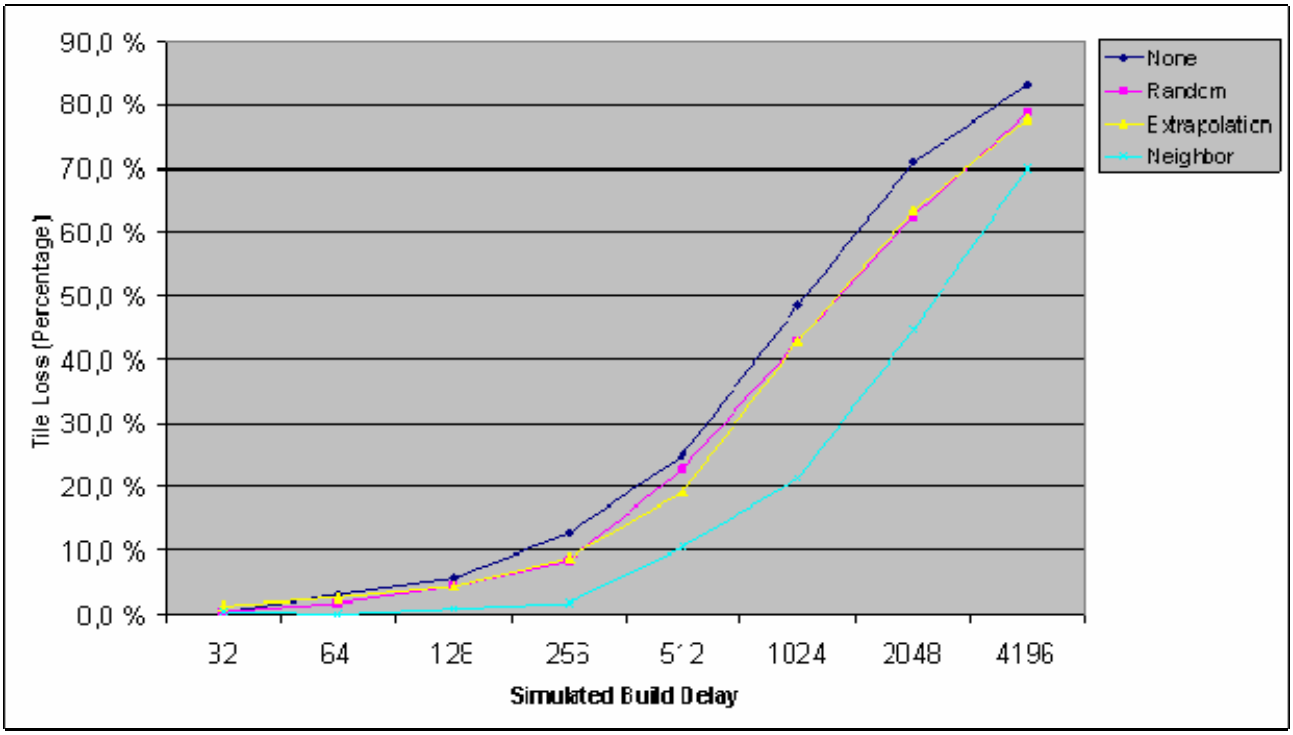


Figure 34 – The Tile Loss Suffered by the Client Throughout the Simulations

5.1.4.4 Precache Ratio

While it is the response time that matters when it comes to good heuristics, it is the the precache ratio that best measures the precaching part isolated. Table 6 and figure 37 show the resulting ratios in table form and as graphs. Once again, we see insignificantly small improvements from the Random setup to the Extrapolation setup, telling us that the Extrapolation does not succeed more than the simple Random setup. The Neighbor setup, on the other hand, is, at its best, 85% more successful than both the Random and the Extrapolation setups. Using the Neighbor setup and 256 ms build delay, which should be a realistic build delay, 74% of the tiles were already built when requested. As a comparison, the Random and Extrapolation setups resulted in 13% and 10% respectively. For build delays bigger than 256 ms, we see that the percentual gain of the Neighbor setup is taking a dive. It is, nevertheless, far much better than the other setups. Additionally, for "on-the-fly" generation, 512 ms is a long time, and all values collected with greater build delays should basically be used only to examine the trends in more detail.

Build Delay	None	Random	Extrapolation	Neighbor
32	0 %	9 %	10 %	61 %
64	0 %	10 %	10 %	70 %
128	0 %	12 %	9 %	70 %
256	0 %	13 %	10 %	74 %
512	0 %	9 %	11 %	66 %
1024	0 %	17 %	14 %	63 %
2048	0 %	23 %	23 %	58 %
4196	0 %	27 %	24 %	44 %
Average	**0 %**	**15 %**	**14 %**	**63 %**

Table 6. The Ratio of Tiles Successfully Prebuilt Throughout the Simulations

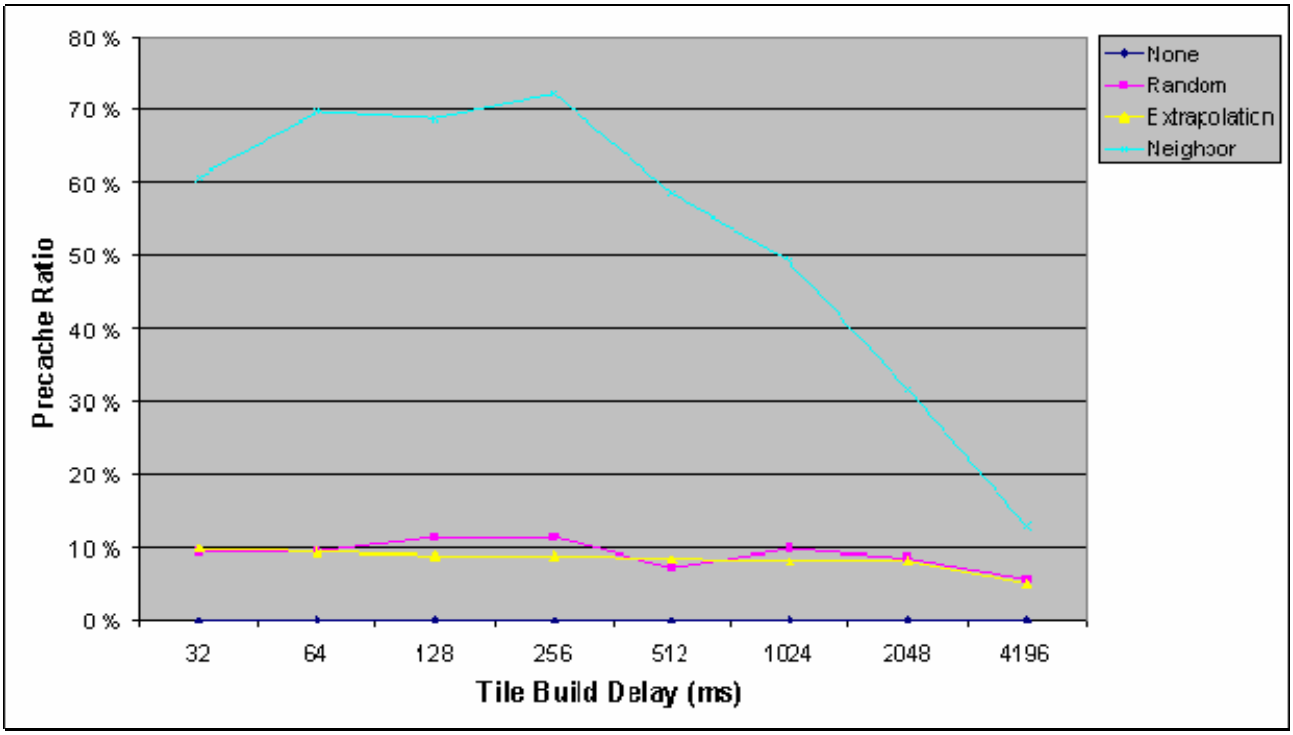


Figure 35 – The Ratio of Tiles Successfully Prebuilt Throughout the Simulations

5.1.4.5 Test Case 1 Summary

The results from all of the four measures (frame rates, response time, tile loss, and precache ratio) are unisonous: the Neighbor setup is very efficient, but the Extrapolation setup, on the other hand, is far from good enough (at least as it is now, and when not combined with other types of heuristics). It is important to note that the Extrapolation setup has plenty of room for improvement, both when it comes to retrieving fine-grained data from the client instead of estimating AALs (see figure 31) and when it comes to using more sophisticated formulas for projecting the Estimated Future Point. In addition, "tweaking and tuning", e.g. extrapolating points closer to the present time, can also increase the performance. When it comes to the Random setup, it gave better results than expected. Unfortunately, this setup is not feasible for very large models, the way it is implemented in this test case (as the number of potential tiles in the model becomes thousands of times bigger than the tile cache).However, it could have been interesting to combine this technique with others. One possible precaching heuristics may be to precache all tiles that are immediate neighbors of the requested tiles, and, additionally precache a random selection of a set of tiles which would be too costly to precache as a whole. Ideally, this approach could also be applied to the extrapolated tiles so that fields of tiles are precached rather than isolated tiles.

5.2 TEST CASE 2 - A VERY LARGE SIMULATED MODEL

5.2.1 Simulating the Model's Data Set

In the following test case, we aim at measuring the performance of heuristics in a model with the magnitude of the Earth with a 10-meter precision. The surface of the Earth is estimated to 510,000,000 km^2 [35], which is equivalent to a quadratic model with edges that are 22,583,180 meters. If we structure our quadratic model after the Quad Tree principle (as shown in figure 30), with tiles composed of meshes with 101 x 101 vertices, we need 15.46 ≈ 16 levels (which means a total of 1,431,655,765 tiles) to meet the 10-meter precision requirement (see the calculation in figure 38). Since we do not have such a 2D data set, on which the on-the-fly conversion to 3D data should be based, we will have to simulate the 2D data set. In this thesis, "simulating a model" implies that no attempts are made to create a high fidelity reproduction of topography. In practice, this means that whereas the first test case projects an existing environment, and thereby gives the user the better sense of immersion and presence of the two, this test case focuses on performance and scalability only, and thereby ignores the shape and appearance of the topography, as long as its complexity, i.e. the amount of data being transmitted between client and server, is realistic.

$$n = 1 + \log_2\left(\frac{22.58 \cdot 10^6 \text{m}}{(101-1) \cdot 10\text{m}}\right) = 15.46$$

Figure 36 – Number of Tiles Needed to Create a Quad Tree Model of the Earth with a 10-Meter Precision

The simulated model can have a data set that consists of either a small set of predefined building blocks, such as meshes and textures, that we use repetitively, or implement a tile builder that generates tiles on the fly with pseudo-random altitude (y-value) for each vertex in the tiles' meshes. The tile builder used here, in Test Case 2, builds tiles with pseudo-random altitude for each vertex in the meshes (resulting in random terrain), but uses a predefined set of textures to be applied to the tiles. In order to be able to monitor and keep track of the level of each visible tile, the textures of a level *n* tile is an image of the number *n* (see figure 39). In addition, applying textures makes the amount of data to be transmitted from the server to the

client more realistic. As we shall see shortly, the generation of tiles using pseudo-random height-values for the meshes, gives a rather unrealistic, short build delay. Therefore, the simulations described in this chapter are therefore divided in two. The first set of simulations is run with a tile builder that generates tiles relatively fast. This set is described in chapter 5.2.6 "The Results of the Simulations (build delay <50 ms)". Then, in chapter 5.2.7 "The Results of the Simulations (build delay > 2,000 ms)", we have, as the title implies, added an extra delay to make sure the server needs at least 2,000 ms to generate a tile. A comparison, and a summary of the results from these two parts of Test Case 2, can be read at the end of this chapter, more specifically in chapter 5.2.8 "Test Case 2 Conclusion". But first, let us take a look at what the two parts of Test Case 2 have in common, and what makes this test case stand out from Test Case 1.

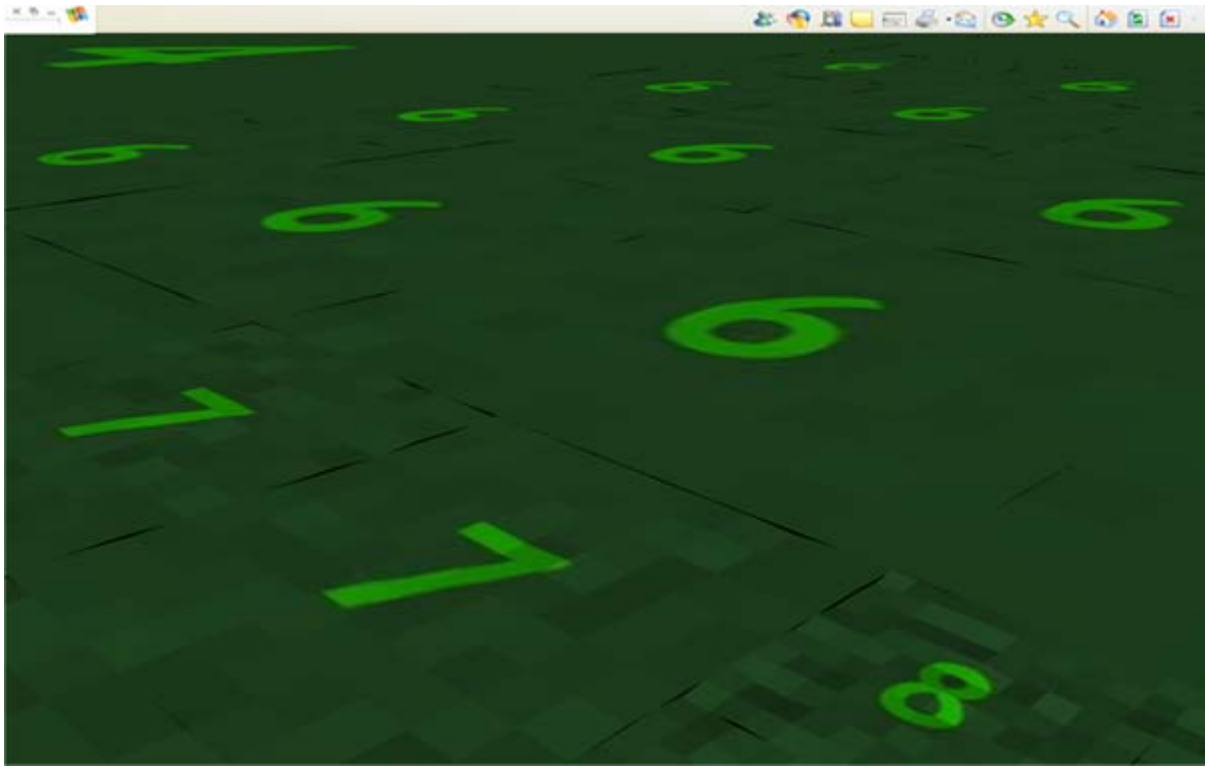


Figure 37 – Snapshot of the Simulated Model of the Earth

5.2.2 Amendments (Improvements from Test Case 1)

Test Case 2 is an improvement of Test Case 1 in more than one area, of which there is, in particular, two differences that stand out. The first difference is that the server now receives fine-grained information about the avatar position every 1.5 seconds. (In Test Case 1 the avatar position had to be calculated by the server from History Based Server-side data.) We will come back to this later. The second improvement is that the server flushes tiles from the cache in a much more sophisticated manner. Instead of consequently flushing the oldest tile in the cache, several components called measures are introduced to calculate the priority of tiles based on different criteria. In the upcoming sections, the implementation of Test Case 2, including the two improvements just mentioned, will be described in more detail.

5.2.2.1 Reporting Accurate Avatar Positions (Client-Side & History Based Session Data)

The client, running the model in a VRML browser, retrieves the exact location of the avatar continually, and makes a report which is sent periodically to the server. The interval implemented in Test Case 2 is 1.5 seconds. The reports are client-to-server data modules implemented as the third approach described in section 4.4.7.2 "Client-Side Data (Fine-Grained Data); the data is wrapped by a fake HTTP request to the server asking for a new model, upon which the server interprets the wrapped data before consequently returning a "404 File Not Found" HTTP response. The client continues to show the current model as if nothing happened. The reason for choosing this seemingly indelicate way of sending reports is described in more detail in the section referred to. Despite this rude implementation, enabling the heuristics to base its guesswork upon exact avatar positions is a considerable improvement from the previous test case. This means that the server does not have to operate with the spatial volume between two detail boundaries when describing the avatar position. Nevertheless, we still need to calculate detail boundaries for estimating the distance from the avatar to the nearest point where a tile becomes visible. We will return to this in section 5.2.3.3 "Combined". Hopefully, the fine-grained session data will improve the accuracy of the Extrapolation precaching.

5.2.2.2 The Introduction of Measures and Tile Priority in Flushing

In section 4.4.2 "Motivation for introducing Heuristics" we explained the necessity for "improving the percentage of tiles that will be reused in the cache.". To accomplish this, we need a means to compare two tiles and tell which one of them is the more valuable. There is no easy way to find the absolute answer to this, since there are many factors deciding how likely it is that a tile will be requested in the near future. These factors will be called measures in this paper and four of them have been implemented and used in Test Case 2, viz. Distance, Reuse, Idle Time, and Creation Time. Most people would agree on the assertion that distance is closely connected to a tile's probability of being requested. The same goes for the number of times a tile has been requested in the past; it is natural to believe that a tile that has been requested five times during the last minute is more likely to be requested again than a tile that has not been requested at all. One obvious reason for this may be that some tiles are requested when the client performs a jump to a viewpoint. Another reason could be something as trivial as the fact that some tiles contain an interesting landmark. The third measure is the Idle Time of a tile, i.e. the time since a tile was last requested. After all, it may be irrelevant that a tile has been requested three times if this was five minutes ago. Finally, the fourth and last measure implemented, is the time since a tile was created and put in the tile cache. This last measure is essential in preventing recently prebuilt tiles from being flushed before they have had the chance to become "valuable".

5.2.2.3 Implementing a Priority Queue in the Tile Cache

Now that we have defined a set of measures, we are ready to prioritize the tiles in the tile cache so that we can, at any time, remove the "worst" tile, i.e. the tile with the lowest LOU (Likelihood Of Utilization, described in section 4.4.4 "Heuristics in Detail"). Implementing a heap-structured priority queue (complete binary tree in which the keys along any leaf-to-root path are ascending) is a very effective technique to keep the best, or the worst, tile ready for direct access at all times. In fact, it runs in logarithmic time, which is acceptable, as it performs 2 lg n comparisons at most when adding or removing a tile (where n is the number of tiles in the queue). However, there is a small problem with our implementation of the priority queue: measures based on criterions that change dynamically have been allowed,

causing the value of the tiles to change independently of each other. Take the Distance measure as an example: two tiles, A and B, are added to the priority queue at a moment in time when the avatar is positioned closer to A than to B. This leads to a higher evaluated priority for A than for B, and tile B will be flushed from the tile cache before tile A. Now, it is not unlikely that during the next seconds, the client navigates closer to tile B, ideally making it switch place with tile A in the priority queue. Unfortunately, we do not have any means of telling which tiles have been affected by the last navigation, and which tiles have not. Therefore, we need to revalidate the priority queue with given intervals. Naturally, there is a trade-off between performance and the correctness of the priority queue at all times; revalidating the queue frequently gives high integrity of the priority queue's content, but is costly, and vice versa. Despite the fact that revalidating a heap-structured priority queue is done in linear time, we will trade good integrity for performance in this test case, so that we can add as many tiles to the cache as possible. The priority queue is therefore revalidated every second time that the server receives client-side data. Since the client sends reports about the avatar position every 1.5 seconds, we revalidate the queue every third second. Finally, it is important to note that, since we still need to perform direct access on the Tile Cache, we store the actual tiles in a hash table, or a hash map, just like in Test Case 1. The priority queue is only storing references/pointers to the tiles in the priority queue.

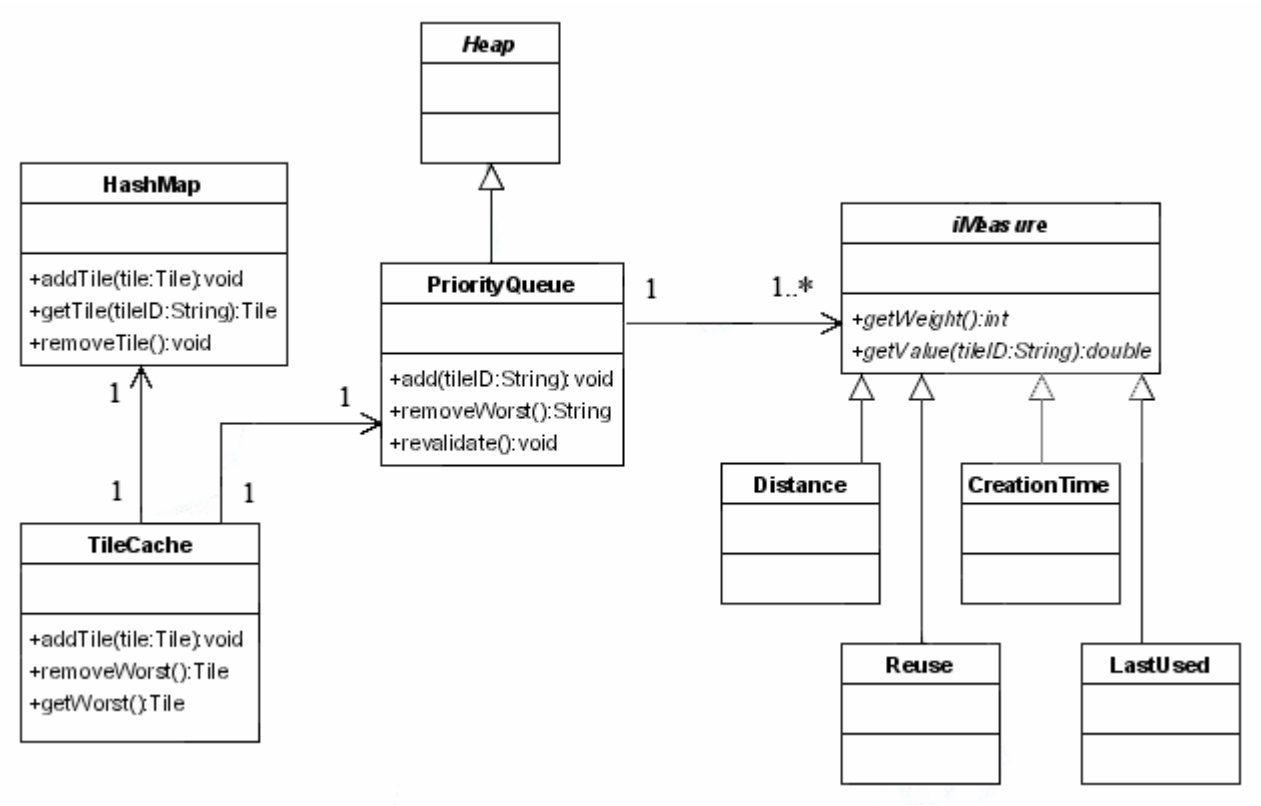


Figure 38 – A Conceptual Model of the Tile Cache With Priority Queue (UML Style)

Figure 40 shows a conceptual class model of the tile cache with the most important methods. Note that the HashMap's addTile-method takes a Tile object as an argument (the actual tile), whereas the addTile-method in the PriorityQueue class takes a String object (a reference, or a pointer) as an argument. Note also that the PriorityQueue can use one *or more* Measure objects to calculate the value of a tile. When more than one measure is used, the different measures can be weighted differently so that, for example, distance counts more than the time of creation. In section 4.4.5 "LOU - different techniques" we called this a system of weighting, as opposed to the ranking system. This can be used to tweak, or tune, the heuristics to improve the performance. The trial-and-error approach to optimize the weight distribution is required since it is hard, if not impossible, to calculate the optimal weight distribution.

5.2.3 Precaching Setup

As with Test Case 1, we have implemented several precaching setups in this test case, though some of them are altered slightly. The Random setup, precaching tiles arbitrarily, would be meaningless to implement here in the same manner as in Test Case 1, since we in this case talk about billions of tiles to choose from. However, we have included the Random setup in our attempt to create a combined precaching setup, utilizing all of the three setups (except None) from the other test case. The three setups included here are thus: None, Neighbor and Combined. Note that precaching is triggered by tile requests from the client only, and not by server initiative to precache tiles, since this would cause an infinite loop where the number of tiles to be built grows exponentially. Finally, precached tiles are only inserted into the tile cache if the tile's LOU is higher than that of the worst tile currently residing in the tile cache. This is not the case for tiles built as a result of a tile request from the client. The tile will then be inserted into the tile cache regardless of its LOU.

5.2.3.1 None

The None setup is more a point of departure than a setup; a setup to which the other two setups can be compared. The None setup utilizes no tile cache on which precaching and flushing is performed. As a result, all tiles will be built on request.

5.2.3.2 Neighbor

The Neighbor setup is implemented in the same way as in the previous test case. It is described in closer detail in section 4.4.8.1 "The Neighbor Tile Pre-Processing Model".

5.2.3.3 Combined

The thought behind the Combined setup is to increase the odds of actually visiting the tiles precached by extrapolation. This is done by precaching a cluster of tiles instead of

extrapolating one tile at the time. First, the Future Avatar Position (FAP) has to be estimated through extrapolation. The immediate problem is that the prebuilder do not automatically know which tiles are visible from this avatar position (the FAP). The prebuilder solves this by calculating the *nearest visible tile*. The nearest tile is obviously a tile that is "straight beneath" the FAP. More specifically, a vector is projected from the FAP through the xz-plane so that the vector is parallel with the y-axis. Figure 41 is an attempt to illustrate this. The bold, broken arrow is the extrapolation, which leads us to a Future Avatar Position (shown as a humanoid avatar in the figure). The thin, broken arrow pointing downwards is the vector projected from the avatar, through the xz-plane, and runs through the nearest tile. The problem is that there are potentially several tiles which envelopes the intersection of the vector and the xz-plane, and all are "nearest tiles" with different level of detail. However, there should only be one nearest tile that is *visible* at any given time, and this is found by looking at the detail boundaries of these "candidate" tiles. The definition of the nearest visible tile is hence: the most detailed of the candidate tiles (tiles enclosing the intersection between the xz-plane and the vector projected from the FAP, normal to the xz-plane) whose detail boundaries encloses the FAP is the nearest visible tile. As mentioned in the beginning of this paragraph, the idea was to precache a cluster of tiles. This is done by precaching all the nearest visible tile's neighbors by applying the technique used in the Neighbor setup implemented in Test Case 1. We can adjust the circumference of the cluster by extending the definition of neighbor tiles to include more tiles, or we can reduce the amount of tiles being precached by randomly selecting only a sub-set of the neighbors. In figure 41, the prebuilder has prebuilt only the nearest visible tile (in the middle) and three randomly selected neighbor tiles.

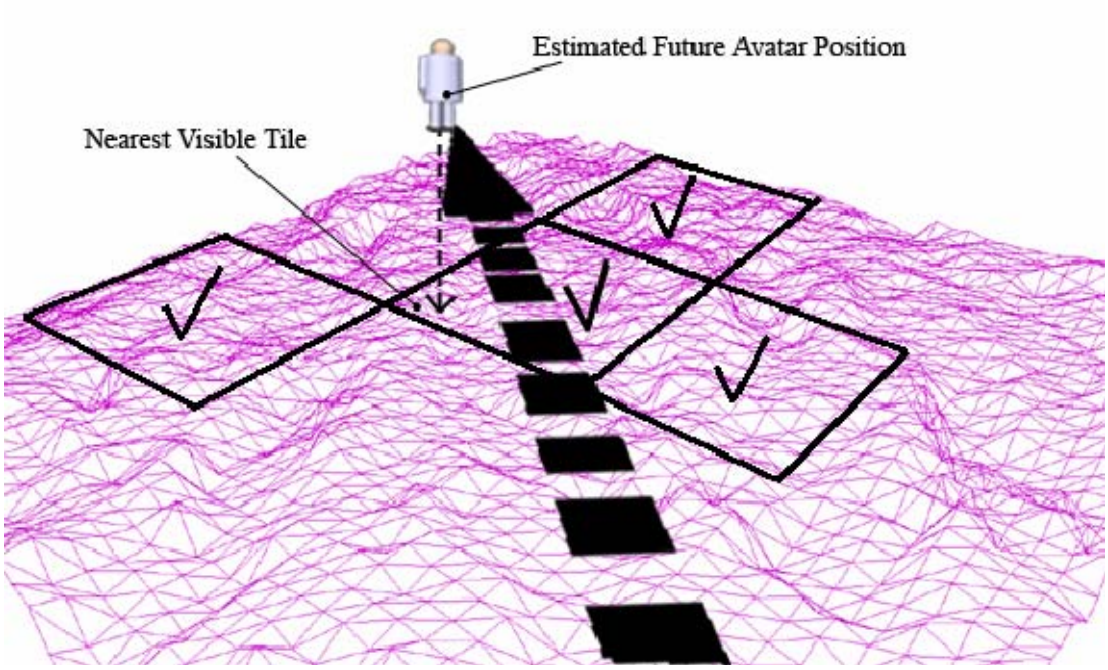


Figure 39 – The Combined Setup for Flushing – Using Both Extrapolation, Neighbor, and Random

5.2.4 Flushing Setup

Flushing is an almost equally important part of the heuristics as precaching when it comes to ensuring the highest possible LOU for the tiles residing in the tile cache. In Test Case 2 the heuristics will flush the worst tile (the tile with lowest LOU) from the tile cache each time a new tile is inserted if, and only if, the tile cache has reached its capacity. Four measures will be used interdependently to calculate the LOU, viz. Distance, Reuse, Idle Time, and Creation Time, whereof the system of weighting will be used to optimize the individual significance of the measures.

5.2.5 The Fly Tour

A fly tour, taking 300 seconds (five minutes), is activated on startup. This is meant to simulate a hypothetical case where the client wants to zoom in on familiar places, and perhaps perform a low-altitude travel between two locations. The tour starts out at an overview viewpoint where only the root tile is requested. The fly tour zooms in on imagined places by diving towards the model, along the y-axis, arbitrarily adjusting the position by *panning* (angular

displacement of a view along any axis of direction in a three-dimensional world [36]). Several times, when the user has zoomed in on the model, the user performs high-detailed "sweeps" (low altitude navigation along the xz-plane). It is natural to believe that the client might get lost when navigating in low altitude. To simulate this, the fly tour returns to the overview viewpoint two times before zooming in on other imagined places in the model and performs some sweeps.

5.2.6 The Results of the Simulations (build delay< 50 ms)

We will now show the results from Test Case 2. 18 simulations were run, using the different setups described in the preceding sections. In these simulations, the generation of tiles is a non-demanding process (since it is based on a random algorithm rather than data that needs parsing). The generation of a single tile is measured to take less than 50 milliseconds. This time will increase when the server gets a considerable amount of tile requests from a client. Table 7 shows, in detail, the differences between the 18 simulations with build delay less than 50 ms.

Setup	Description
1	No heuristics applied. All tiles are built "on demand".
2	Tile cache is included with room for 2,000 tiles. The cache is using a priority queue (with same capacity as the cache). Advanced flushing is performed using four measures: Distance, Reuse, IdleTime and CreationTime. All four measures are weighted equally by the priority queue when calculating the tile value. No prebuilding is performed, however.
3	The same as setup 2, except that the tile cache is now capable of holding 15,000 tiles.
4	Tile cache with room for 15,000 tiles. The cache using a priority queue with the same four measures as in setup 3, namely Distance, Reuse, IdleTime and CreationTime. All four measure are weighted equally (0.25). Prebuilding and precaching of tiles is performed using the Neighbor approach described in section 4.4.8.1. "The Neighbor Tile Pre-Processing Model".
5	This setup is the same as setup four, except that the weight distribution on the four measures have been tweaked. In stead of giving all the four measures equally weight, the Distance measure is now predominant. The weight distribution is as following: W={Distance(0.6), Reuse(0.13), IdleTime(0.13), CreationTime(0.13)}. The Distance measure is weighted more than the others.
6	This setup is the same as the one described above, except for a slightly different weight distribution: W={Distance(0.13), Reuse(0.13), IdleTime(0.6), CreationTime(0.13)}. The IdleTime measure is weighted more than the others.
7	This setup is the same as the one described above, except for a slightly different weight distribution: W={Distance(0.13), Reuse(0.6), IdleTime(0.13), CreationTime(0.13)}. The Reuse

	measure is weighted more than the others.
8	This setup is the same as the one described above, except for a slightly different weight distribution: W={Distance(0.13), Reuse(0.13), IdleTime(0.13), CreationTime(0.6)}. The CreationTime is weighted more than the others.
9	This setup is the same as setup 4 except that it has a much smaller tile cache. The capacity has been limited to 500. The weight distribution is even between the four measures: W={Distance(0.25), Reuse(0.25), IdleTime(0.25), CreationTime(0.25)}.
10	This setup is the same as the one described above, except for a slightly different weight distribution: W={Distance(0.6), Reuse(0.13), IdleTime(0.13), CreationTime(0.13)}. The Distance measure is weighted more than the others.
11	This setup is the same as the one described above, except for a slightly different weight distribution: W={Distance(0.13), Reuse(0.13), IdleTime(0.6), CreationTime(0.13)}. The IdleTime measure is weighted more than the others.
12	This setup is the same as the one described above, except for a slightly different weight distribution: W={Distance(0.13), Reuse(0.6), IdleTime(0.13), CreationTime(0.13)}. The Reuse measure is weighted more than the others.
13	This setup is the same as the one described above, except for a slightly different weight distribution: W={Distance(0.13), Reuse(0.13), IdleTime(0.13), CreationTime(0.6)}. The CreationTime measure is weighted more than the others.
14	This setup is the same as setup 9, except that it revalidates the priority queue every 3rd second instead of every 1.5 second.
15	This setup is the same as the one described above, except that it revalidates the priority queue every 4.5 second.
16	This setup is the same as the one described above, except that it revalidates the priority queue every 9th second.
17	This setup utilizes Neighbor tile prebuilding to insert tiles into a cache with capacity of 500 tiles. However, it provides no advanced flushing mechanism with priority queue and measures. The tile first inserted in the cache is the tile first tile expunged.
18	This last setup includes the same heuristics as in setup 16, except that the Distance measures is removed, leaving only the three other measures to calculate the value of the tiles in the tile cache.

Table 7. Description of the Different Simulation Setups Used in Test Case 2 (Build Delay < 50ms)

5.2.6.1 The Precache Ratio

The precache ratio measured in this test case denotes the percentage of the requested tiles that could be retrieved directly from the tile cache. Since responding to these requests is a matter of performing direct access on memory, they can be responded to much quicker than those requests where the tiles must be converted/built from 2D data by the tile builder. Therefore, as in Test Case 1, a high precache ratio indicates a good heuristics setup. Figure 42 shows the precache ratio for the 18 setups tested in this test case. The immediate impression is that, except from the three first setups, there are only small variations between the setups. The highest precache ratio obtained was with setup 8, which managed to anticipate 75.3% of the

requested tiles. Ignoring the three first setups, not being able to precache any tiles, the lowest precache ratio comes from using setup 12 which managed to precache 62.9% of the requested tiles. From the results, we can see a slightly higher average ratio from using a tile cache with tile capacity of 15,000 (setup 4 to setup 8 with 73.5% average ratio), than for setups with tile capacity of 500 (setup 9 to 13 with 65.6% average ratio). Although this seems rather logical, it is a small surprise that the difference between the setups was not greater. When it comes to the interval between each revalidation of the priority queue, we can see from setup 9, 14, 15 and 16, which uses the intervals 1.5 seconds, 3 seconds, 4.5 seconds and 9 seconds respectively, that very short intervals is not an important factor for keeping the most valuable tiles in the tile cache. The precache ratio is only slightly decreasing: 1.5 second resulted in 65,4%, 3 seconds in 65.9%, 4.5 seconds in 65,5%, and, finally, 9 seconds resulted in 64.0% precache ratio.

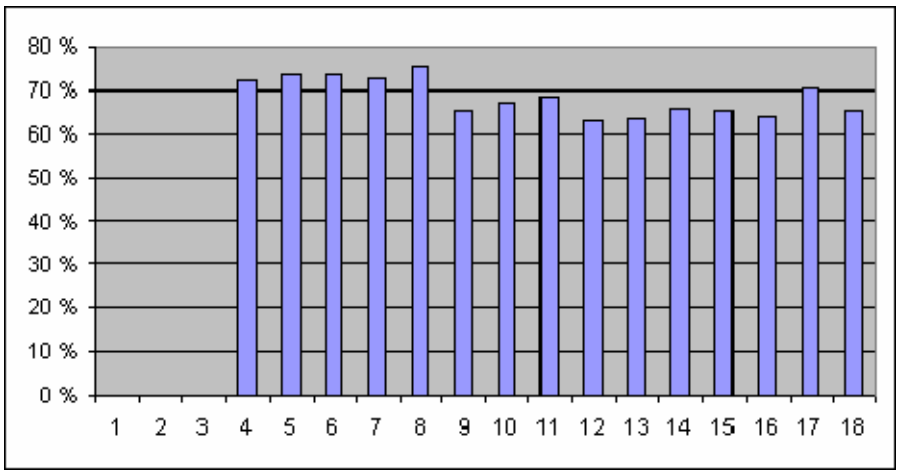


Figure 40 – Successful Precache Ratio with 50 ms Delay

5.2.6.2 The Average Response Time

The average response time is, as in Test Case 1, the average time the server takes to reply to a HTTP request, independent of whether the tile was built on demand, or was accessed directly from the tile cache. The response time is the key factor when determining the efficiency of the "on-the-fly" framework. Low response time means shorter time before the tile is visible on the client-side. What we see here, is that setup 1 (visualized by the pillar to the far left in figure 43) is by far the most efficient heuristics, which in fact is no heuristics at all; no cache, no prebuilding, and no flushing (garbage collection). The requested tiles are, on average, being sent back to the client in 14 milliseconds. The next three best setups are setup 2, 3, and 17 which include only a bare minimum of heuristics. Through these setups, the server obtained an average response time of 16, 17, and 17 milliseconds respectively. With setup 4 to 9, the server obtained on average a response time of 18 milliseconds, whereas setup 9 to 13 lead to a response time of 23 milliseconds. What these results indicate, is in fact that for response times below 20 milliseconds, the extra effort of computing and maintaining the heuristics costs more than it gives. We saw the same tendency in the previous test case. There, the gain of applying heuristics became prevalent only when the build delay exceeded 60-70 milliseconds. The results given by measuring the average response time gives very useful results when it comes to the priority queue revalidation interval. Setup 9, 14, 15, and 16 is, as in the last section, the same heuristics setup except that the interval between each time the priority queue is updated, is 1.5 seconds, 3 seconds, 4.5 seconds and 9 seconds respectively. The average response times performed by the server using these setups, were: 25 ms, 23 ms, 21 ms, and 23 ms. From these simulation, 4.5 second seems to be the optimal interval for revalidating the priority queue, revalidating the priority queue seldom enough to not load the server too much, and frequently enough to keep the priority of the tiles up to date.

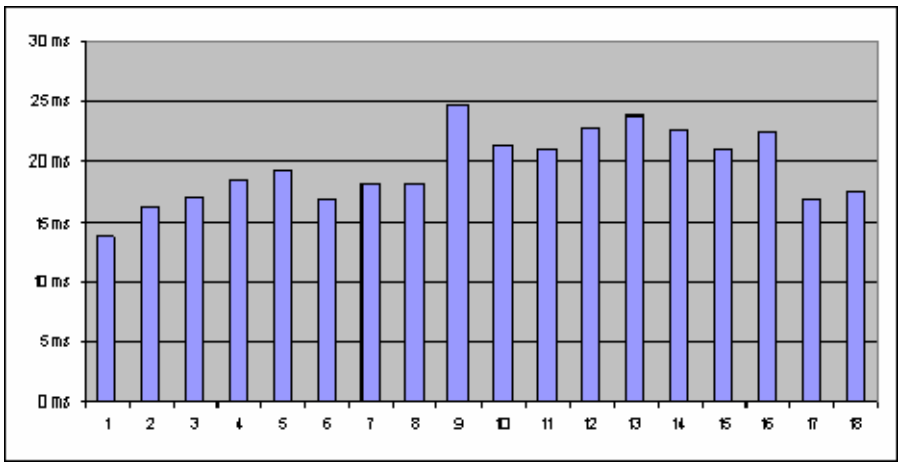


Figure 41 – Average Response Time with 50 ms Delay

5.2.6.3 Browser Frame Rate

Figure 44 shows the frame rates measured by the client-side browser. Recall from section 5.1.3.1 "Frame Rate" and Test Case 1 that the browser frame rate is basically the opposite of the frame rate measured by most 3D application; low frame rate indicates good response time, because the renderer has more terrain to model. In case of high response time, the browser experiences tile loss and has less terrain to model. Hence, the browser frame rate increases. We can see from the figure that there are very small differences between the 18 setups, maybe with an exception of the three first. This backs up the results we got from measuring the avg. response time (figure 43). Simple heuristics, or no heuristics at all, seems to work very well for tile building taking less time than 20 milliseconds. When it comes to the effect of altering the interval between each time the priority queue is revalidated, the frame rate results coincides with the results retrieved from measuring the average response time (although less prevalent now). The revalidation intervals: 1.5, 3.0, 4.5, and 9.0 seconds gave the results: 138.8, 137.5, 136.5 and 141.9 frames/s respectively. These results match the corresponding response times: 25 ms, 23 ms, 21 ms, and 23 ms. The results from measuring the browser frame rate seems to support the results in the preceding sections.

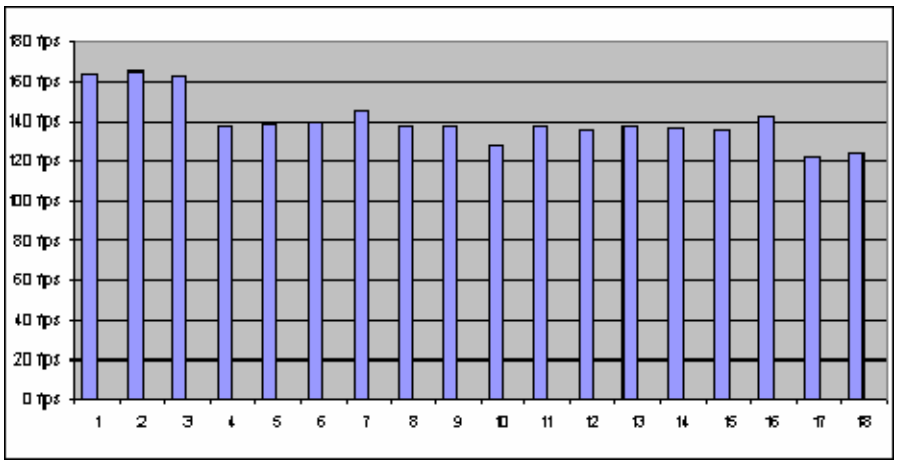


Figure 42 – Browser Frame Rate with 50 ms Delay

5.2.6.4 Test Case 2 Summary for Build Delay < 50 ms

The immediate impression one get from looking at the results, is that, for systems with extremely fast terrain builders, the introduction of advanced heuristics seems to hamper the server more than it relieves its work load. Figure 43, which shows the average response time, clearly shows an increase in response time for the simulations where lots of heuristics have been applied. This set of simulations clearly shows that the proposed framework does not yield a desirable result on systems where the terrain is built extremely fast (in this case less than 50 milliseconds), and is, in that respect, an important result. It remains to test the heuristics on a system where the delay caused by building tiles is more prevalent. This is done in the next section, where the build delay is increased by 2,000 milliseconds.

5.2.7 The Results of the Simulations (build delay> 2,000 ms)

In order to be able to compare the following results with the previous set of simulations, we use the same 18 heuristics setups as described in section 5.2.6. and Table 7. The only difference is that we have added 2,000 milliseconds to the time it takes to build a tile. The results from these simulations are described in the next sections.

5.2.7.1 The Precache Ratio

The three first bars in figure 45 show simulations with no precache function, and, as a result, the precache ratio is zero. If we examine the impact of tile cache capacity, we will find that the correlation between cache capacity and the ability to reuse previously requested tiles is present, although prudent. In figure 45 we see that simulations 4 to 8, where the server had a tile cache with a capacity of 15,000 tiles, the server found, on average, 65% of the requested tiles already prebuilt in the tile cache. As a comparison, simulations 9 to13, where the cache capacity was only 500 tiles, resulted in a precache ratio of 58%. The results are surprising, but important, because they show that increasing the size of the tile cache does not drastically improve the performance, as one might have expected. We saw the same result for the simulations with low build delay (see section 5.2.6.1 "The Precache Ratio"). Further, we see that there are relatively small variations between the different setups, with the exception of setup 17 (which had no priority queue with measures). Taking a look back at simulation 17 in figure 42, we see that excluding the concept of priority slightly *improved* the precache ratio, whereas now, with high build delay, excluding the concept of priority drastically reduces the precache ratio. Comparing the average of setups 9 to 13 with setup 17, we find that the precache ratio increases from 31% to 58% when introducing a priority queue and a set of measures. As for which measure seems most valuable, it is surprising to see that tweaking the different measures does not yield greater variations. Finally, when it comes to intervals between each time the priority queue is validated, we can see a clear connection between the validation frequency (how often the priority queue is rebuilt) and the percentage of tiles that can be retrieved directly from the tile cache. A 3.0 seconds validation interval (simulation 14) gave a precache ratio of 59.8%. When increasing this interval to 4.5 seconds (simulation 15), the ratio dropped to 55.0%, and, at last, the ratio dropped to 52.7% when using a 9.0 seconds

build delay. This connection was less visible when the build delay was less than 50 ms. Still, it appears to be a limit as to how low the validation interval can be before the extra workload on the server pays off. Simulation 9 is the same setup as simulations 14, 15, and 16, except that the validation intervals are lowered all the way to 1.5 seconds. As we can see by it to simulation 14, the precache ratio has been reduced from 59.8% to 56.7%. It seems that a 3.0 seconds interval is decent for build delays just above two seconds.

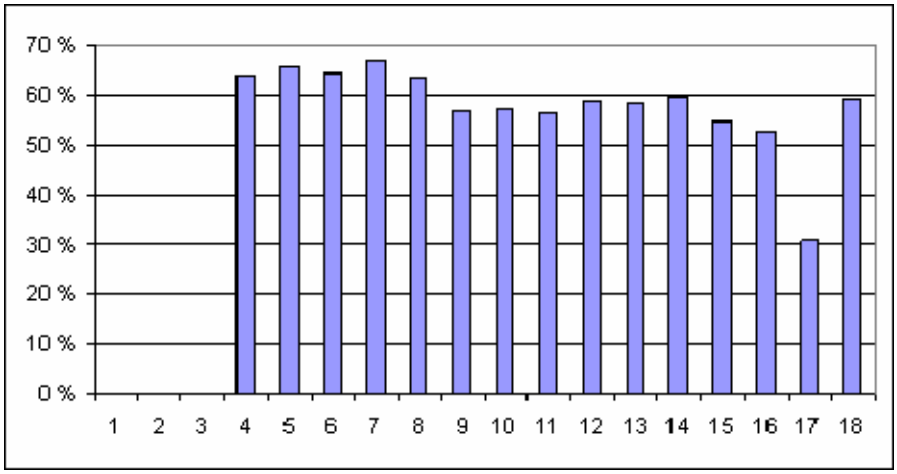


Figure 43 – Successful Precache Ratio with 2,000 ms Delay

5.2.7.2 The Average Response Time

Whereas the precache ratio described in the previous section serves more as an indication of efficiency, the average response time is the actual performance, perceptible by the client. The measured average response times from the simulations (see the bar graph in figure 46) show that the heuristics proposed, or described, in this paper can improve the perceived performance of a global scale terrain model drastically. The three first bars in the figure represent three separate simulations without advanced heuristics. The resulting avg. response times were 2,097, 2,110, and 2,127 milliseconds, respectively. The highest measured performance, coming from setup 7 with advanced heuristics, gave an average response time of 676 milliseconds. This is a reduction by a factor of three. When it comes to the significance of cache size, the latest results confirm what we saw traces of when studying the precache ratio: there is a connection between cache size and performance, although it is not a prevalent one. This is shown in figure 46. Setups 4 to 8, utilizing a cache with capacity of 15,000, gave an average response time of 729 milliseconds. When lowering the cache capacity to 500 (setups

9 to 13), the average response time became 864 milliseconds. Seemingly, the results from measuring the response time seem to match the results we got from measuring the precache ratio, also when it comes to tweaking the weight of the four different measures, and when excluding the priority queue. The differences between setups 4 to 8, where the only difference is the weighting of each measure, are minimal. The same applies for the differences between setups 9 to 13. Setup 17 shows our attempt to exclude the priority queue altogether. Except for the priority queue, this setup is identical to setup 9. From the graphs we see that excluding the priority queue decreases the performance relatively much; the average response times for the two setups are 878 milliseconds for setup 9, and 1,402 milliseconds for setup 17. This is a 63% increase which confirms our previous results. Finally, as regards the validation interval, we see that, in conformity with the previous results, that the best measured validation interval is three seconds. In figure 46, we can see that setup 14 (3.0 seconds interval) gives a lower average response time than both setup 9 (1.5 seconds interval) and setup 15 (4.5 seconds interval). This shows that there is a trade-off between frequent validations of the priority queue and avoiding weighing down the server with unnecessary calculations.

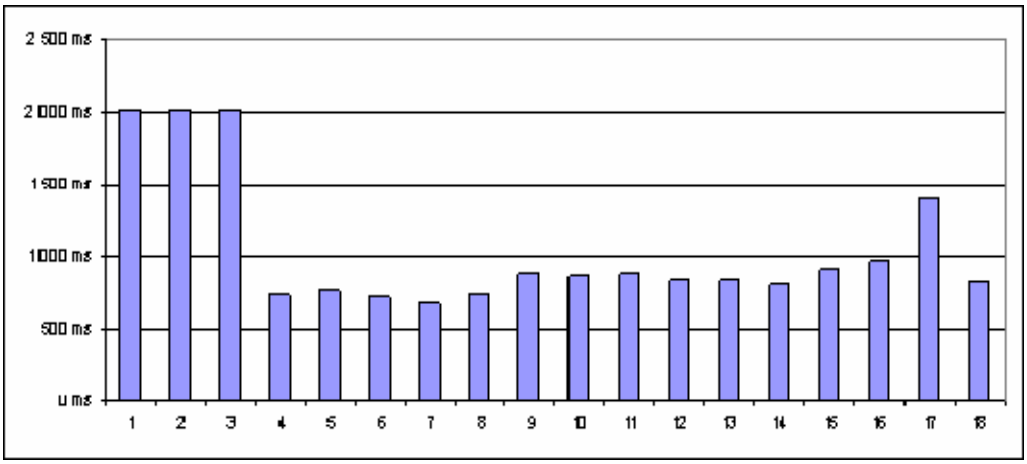


Figure 44 – Average Response Time with 2,000 ms Delay

5.2.7.3 Browser Frame Rate

The measured browser frame rate confirms the results described in the two preceding sections (see section 5.1.4.1 "Frame Rates" for a description of the 'browser frame rate' measure). Applying no heuristics gave high frame rates (the three first bars in figure 47). This indicates that a relatively small part of the 3D content has been retrieved from the server, which the

browser can quickly render and display. The setups with a tile cache taking 15,000 tiles (bars 4 to 8) were on average rendered 254 times per second, while the setups with a tile cache taking only 500 tiles (bar 9 to 13) were on average rendered 264 times per second. This matches the prudent increase in performance already measured when increasing the capacity of the tile cache. Further, the browser frame rate results also seem to match previous results regarding different weighing of measures, and the exclusion of the priority queue. The internal differences between setups 4 to 8 and setups 9 to 13 are minimal, whereas setup 17 gives a perceptible increase in the browser's frame rate (indicating sluggish performance).

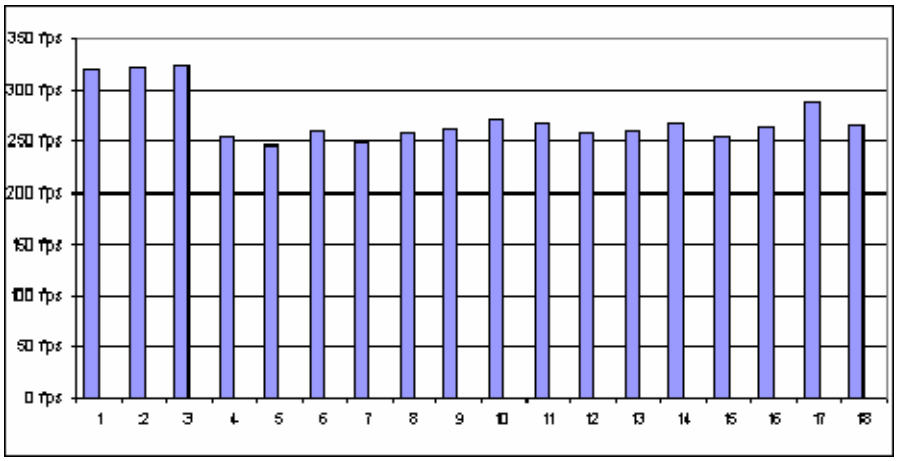


Figure 45 – Browser Frame Rate with 2,000 ms Delay

5.2.7.4 Test Case 2 Summary for Build Delay> 2,000 ms

Whereas the heuristics slowed down an already fast system in 5.2.6 (where the build delay was less than 50 ms), we witness a significant improvement when utilizing the described set of heuristics for systems with build delays somewhere around two seconds. All three sets of results were uniformly describing this enhancement, where, at best, the build delay was reduced by a factor of three. Further, the optimal validation interval (how frequent the priority of the tiles in the cache is updated) seemed to be 3.0 seconds with the current build delay (approx. 2,000 ms), whereas 4.5 seconds gave the best results for smaller build delays (less than 50 ms). This can probably be explained by the observation that extra calculations caused by the heuristics seem to be futile when the server is already very fast, while the extra calculations become more valuable as the build delay increases. The tweaking of the four different measures gave small or no variations at all. This may be explained by the fact that

the workload caused by the four measures are the same regardless of how much weightthe priority queue put on each of them. So whereas one measure, e.g. the distance measure, may be better in terms of predicting a higher percentage of tiles that will be requested, it may be more resource demanding. Calculating the distance between the avatar and hundreds, if not thousands, of tiles obviously requires more resources in terms of CPU cycles than keeping count of how many times each tile has been requested since it was last built. We also saw the value of incorporating a priority queue in the tile cache to encourage flushing of the tiles least likely to be reused. The simulations show an increase by 63% in response time when turning off the priority queue. Finally we established a prudent connection between cache capacity and performance.

5.2.8 Test Case 2 Conclusion (All Build Delays)

Test Case 2, as a whole, shows some interesting results, not only because it proves the increasing value of good heuristics as the build delay escalates. It also shows areas of use where these additions *reduces* the server performance. As a guideline, it seems that the longer it takes the server to build a tile, the more it will benefit from heuristics and all the calculations that come along. However, this does not seem to apply for the tile cache capacity. In fact, running the simulations with a high capacity tile cache had a bigger *percentual* impact on the server for small build delays (less than 50 milliseconds) than was the case with 2,000 milliseconds build delay. More specifically, the reduction of the cache size (from 15,000 to 500) increased the average response times with 28% for 50 ms and 19% for 2,000 ms. Nevertheless, both test cases showed a real improvement when increasing the cache capacity. Finally, we also saw the need for, at least the potential enhancement in the performance caused by, a "quasi-intelligent" mechanism that flushes the cache based on a priority estimated by a set of different measures.

6 Conclusions

6.1 EXISTING TECHNOLOGY AND PROJECTS

In this paper we have taken a closer look at some of the available technologies for visualizing global 3D terrain models. VRML, being an easy-to-read 3D script language, was chosen relatively early due to the steep learning curve associated with its syntax and semantics. In an attempt to find a free, yet feasible, VRML plugin, which also supports the GeoVRML extension described in section 2.2.1, we conducted a rather extensive compatibility test where eight different plugins where installed along with GeoVRML. This test is described in Appendix C. The test shows that only one plugin was compatible with the extension, namely the Cortona VRML Client. Unfortunately, it lacked a feasible memory management, which ultimately forced me to abandon the GeoVRML extension and use the blaxxun Contact. VRML seems to suffer from a somewhat lenient specification, which has left room for individual interpretations by those who implement browsers/plugins. Rune Aasgaard's ongoing project, Virtual Globe (described in section 3.3), shows that other technologies, such as Java3D and JOGL, may be equally suitable (if not more) for global 3D models. This model also shows Sun Microsystem's elegant Java Web Start technology, which has simplified the installation process of Java applications. Hence, one of the main arguments for running the 3D models inside the web browser, as is the case with VRML plugins, disappears. Fortunately, the technology used for browsing the 3D content on the client side is transparent for the heuristics on the server side. And in order to change the format of the 3D data generated, e.g. from VRML to X3D, one only has to make a new implementation of the Tile Builder interface and set a new MIME type for the HTTP response.

6.2 THE IMPORTANCE OF HEURISTICS

Although we have demonstrated the potential power of heuristics in global terrain models, existing documentation and guidelines seem to be relatively scarce. We have seen that there already exists 3D models with heuristics incorporated, e.g. the TerraVision server-client application (see section 3.1). However, using these heuristics requires that you dig into the code without necessarily having good documentation available. Other models run under different premises than what we are looking for. The JCanyon demo [37] is a good example, where the navigation is limited to that of an airplane, and where the data set is small enough to download in one big mouthful before the application starts. The test cases demonstrate the difficulty of basing the heuristics on prediction of future navigation, especially when the user has a free hand when it comes to how he or she can navigate (as opposed to the propulsion-based navigation in flight simulators, the user can, in VRML, do advanced navigation such as sidestep, pan, teleport (jump), and stationary rotation. Still, we have seen that heuristics based on prebuilding adjacent and children tiles can be very effective. Further, the test cases also show the importance of managing cached tiles properly. Introducing a priority queue and different measures to estimate the value of all tiles in the cache increased the "quality" of the tiles residing in the cache at all times. At best, the heuristics improved the performance of a global scale model by a factor of three. Equally important, we also saw that the extra calculations introduced by applying heuristics can curb the server rather than improve it. In particular, this seems to be the case when the server is able to build 3D content and deliver it extremely fast.

6.3 THE FUTURE - WHAT CAN WE EXPECT?

Al Gore's vision of the open community-driven Digital Earth (see section 3.4.) came early in 1998, which was in the middle of the *Internet Boom* [38] (often referred to as the *Internet Bubble*). As the name indicates, the bubble burst in late 2000 and through 2001. At this time, all publications on the Digital Earth's web-site stopped dead. Some of the work was moved to the newly formed Geospatial Applications and Interoperability (GAI) working group of the Federal Geographic Data Committee (FGDC) [39]. As such, the Digital Earth has lost some of its trenchancy as a unifying vision of a new WWW for geospatial information about the Earth. However, as Gordon E. Moore observed already in 1965, the growth in computer processing power is getting improved exponentially (something like a doubling every 18th months). At the same time, we are at all times collecting and storing a vast amount of information, lots of it being georeferenced (associated with a global position). It would be a very strange thing if this vision, if not a similar one, should open a passage for a totally different way of acquiring electronic information.

7 Future Work

When a project reaches a certain extent, there will always be things one wants to improve, but cannot, for reasons such as availability (software may be commerical and hardware too expensive) or thesis deadline. Other potential improvements came to my attention at a stage in my thesis where it was too late to go back. The improvements may consist of adding extra features, restructuring the underlying design/architectural structure, replacing the underlying technology with something new and better, or simply spending more time tuning and tweaking minor configurations. In this chapter, I will mention a few improvements that I would very much like to carry out if I get the chance, or things I would like to do different if I were to start all over again.

7.1 IMPLEMENTING A VRML/X3D BROWSER IN JAVA3D

The limitations in the VRML specification and in the browsers existing today prevented some ideas to be tested out, and forced much of the workload over to the server-side (which can easily become a bottle neck in a realistic multi-user environment). Given the fact that desktop computers are more powerful today than the servers were yesterday, it would be interesting to move as much workload over to the client as possible. The on-the-fly conversion from 2D to 3D could for example be moved from the server to the client side. To overcome the VRML limitations, one could make use of Web3D's toolkit Xj3D (still under development as this thesis is being written) to import VRML/X3D into a specialized browser implemented in Java/Java3D. This way, one will no longer be limited to the VRML specification, but rather do all the work with Java3D and Java. As such, one will have a much more free hand as to what can be implemented (most importantly how, or when, tiles should be loaded).

7.2 IMPLEMENTING AND TESTING ALL PROPOSED ENHANCEMENTS

Not all of the enhancements proposed in this thesis were tested. The most interesting feature not tested is probably the introduction of History Sequences (HS) in History Based Session Data. See section 4.4.6.2 "Collecting History Based Session Data from Discontinuous Navigation" for more details. In short, it divides navigation into manageable "chunks", called sequences, by assuming frequent use of the jump function. The session data, or the cache, can then operate on sequence level instead of tile level. Another feature to implement is to exploit the orientation of the avatar, which is currently being reported to the server with given intervals. As it is in the included test cases, only the position of the avatar, not the orientation is utilized by the heuristics.

7.3 MORE TEST CASES

Additional tests should have been run on the framework. For one thing, the four different measures should have been applied one by one, instead of all being used in each test and only tweak the weighting. In Test Case 2, I tried to find the optimal weighting between the measures, but we did not take into consideration that some measures were more CPU-intensive than others. As a result, the tweaking gave only minor variations in the performance. Maybe is a combination of only two of the measures the optimal solution.

7.4 OPTIMIZATION

The implementation of the framework has not been examined minutely, so, in all probability, there should be several opportunities to optimize the code in order to enhance the server's performance. In addition, it would have been interesting to make a formal test as to how much the framework requires from the server on which it runs (create a minimum system requirements list). Another possible optimization, would be to restrict the user's navigation possibilities. This would result in much more predictable navigation, which again would increase the heuristics' ability to prebuild the correct tiles. The navigation restrictions can be

such things as refusing the user's attempt to jump to locations, reducing the maximum navigation speed (giving the heuristics better time to perform its calculations and builds), or even refusing sidestepping and reversing and thereby creating a flight simulator-like navigation.

Appendix A - Glossary of Terms

Avatar : A representation of the user in a virtual universe. The name has its origin from the ancient Hindu language, Sanskrit, and means "visible deity". In most cases, the avatar is not a visible object in the virtual environment, rather than a conceptual denotation.

Black Box : A piece of software that can be used without knowledge of its inner workings.

Browser : An application that lets the user browse/interact with 3D models. It is often called a 'plugin' when it is an extension of a web-browser, enabling 3D models to be viewed in your web-browser.

CGI : Common Gateway Interface. A rather simple way of enabling two-way interaction in the otherwise static HTML documents. (See also: Servlet)

Client : A user or a piece of software that employs a set of services provided by a server (see Server) in a distributed client-server architecture. In this thesis, it often refers to a person (the user of the 3D models).

Client-Side data : Data collected from the client side, e.g. a VRML browser, used by the heuristics on the server side to make qualified guesses at which tiles are to be pre-generated. This data must be passed on to the server in some way. (see also: *Server Side Data)*

Daemon : A daemon is, according to the "Microsoft Windows 2000 Server Documentation" [40], a: "[...] networking program that performs a housekeeping or maintenance utility function without being called by the user. A daemon sits in the background and is activated only when needed [...]"

Detail Boundary : The conceptual boundary for a tile, denoting when to swap between two levels of detail. Whenever the avatar crosses this boundary, a new representation of the tile will be visible.

External Authoring Interface : A plugin interface for the VRML browser that allows embedded objects on a web page to control the VRML content. Not all VRML browsers implement this interface.

Evaluate : ECMAScripts' functions are said to be evaluated instead of executed.

Flushing : The act of expunging tiles from the tile cache. All caches have a finite capacity, and flushing is therefore required before adding new content to a packed cache. Flushing in a quasi-intelligent manner is an important part of an effective heuristics to prevent recently built

tiles to be flushed immediately.

Inlined tile/model/file : A *reference* to a file which contains part of a VRML world.

Geodata : (See Geospatial data).

Geospatial data : Webster's New Millennium™ Dictionary of English [41] gives the following definition: "pertaining to the geographic location and characteristics of natural or constructed features and boundaries on, above, or below the earth's surface; esp. referring to data that is geographic and spatial in nature".

GIS : Short for Geographic Information System. www.whatis.com gives the following explanation: "A GIS [...] enables you to envision the geographic aspects of a body of data. Basically, it lets you query or analyze a database and receive the results in the form of some kind of map."

HTTP : Hypertext Transfer Protocol. A *stateless* TCP/IP based protocol used for communicating over the Internet. By default, HTTP uses port 80 on the server.

JOGL : Acronym for Java bindings for OpenGL. It is an API that provides access to the 3D accelerator hardware's OpenGL interface through Java. Or with Sun Microsystem's [42] own words: "[JOGL] provide[s] hardware-supported 3D graphics to applications written in Java.". JOGL is a "lover level" API than Java3D.

Jump : Sometimes used to describe the navigation feature in some VRML browsers/plugins that lets you instantly move close to an object by clicking on it.

Likelihood Of Utilization : A unit of measurement denoting how likely it is that a given tile is being requested in the immediate future.

LOU : (See Likelihood Of Utilization)

Mean Center : Mean center is a rude estimate as to where the avatar is located in a virtual environment, based on Detail Boundaries (see Detail Boundaries). See section 4.4.7.1. "Server-Side Data (Coarse-Grained Data)" for details.

Measure : WordNet [43] provides the following definition: "[...] a basis for comparison; a reference point against which other things can be evaluated[...]".

Mesh : In the word's widest sense, it is a series of polygons grouped to form a surface. In this paper, however, the term is more specifically referring to topography represented by a 2D matrix of height values. In VRML and X3D, such a construction is called an ElevationGrid.

MIME : Multipurpose Internet Mail Extension. A description that states which type a document is, e.g. "model/vrml" for VRML documents.

Node : Nodes are the building blocks in a tree graph. Usually we divide nodes into two

categories: branch nodes, which may have children nodes, and leaf nodes, which cannot have children nodes. In 3D modeling languages such as VRML and X3D a world is described by building blocks called nodes in a tree graph (often called scene graph). It can to a certain degree be compared to an object in object-oriented programming.

OGC : (See Open Geospatial Consortium)

Open Geospatial Consortium : A non-profit organization that works with governments, private industry and academia in order to create open and extensible programming interfaces for geographic information systems (GIS).

Plugin : An application that runs within an internet browser. In this paper, often used for the application allowing VRML content to be viewed within an internet browser. (See Browser)

Prebuilder : An interface in the proposed on-the-fly framework, specifying how to implement the module that build 3D content (tiles) in in advance. The term can in some cases also refer to the implementation of this interface.

Pre-caching : The process of anticipating parts of the 3D model that are likely to be requested in near future, and then start building these in advance before storing them in a cache. A similar approach used for fetching already generated 3D content over the web is in some publications referred to as "prediction and prefetching" (which is the case in the TerraVision system by SRI).

ROUTE : A mechanism that allows a VRML *node* to pass an event on to another VRML node, thus creating a chain of events. Example: A ProximitySensor node sensing when the user exits an area can be ROUTED to a Script node which alters a URL field in a Switch node. The Script node can then be ROUTED to a Switch node, making sure that a new terrain-model is rendered when the user exits the specified area.

Scene Graph : The internal representation of the objects in a 3D browser or runtime environment. It is an acyclic and directed graph. Scene Graphs are used in VRML, X3D and Java3D.

Sensor : A mechanism that generates an event based on a condition. The ProximitySensor in VRML is a good example. It generates events when the user enters or exits a given area.

Server : The passive part in a client-server architecture (see Client) which offers a set of services. By passive, we mean that it sits idle and wait for clients to employ its services.

Server-Side Data : Data stored on the server side, used by the heuristics to make qualified guesses about which tiles are to be pre-generated (as opposed to *Client-Side Data*).

Servlet : A collection of code written in JAVA that carries out a server function. Or put more

simply: A server side technology which enables two-way interactivity between server and client. (See also: CGI)

Session : According to Callaway's definition [44] a session is "a persistent network connection between two hosts [...] that facilitates the exchange of information." Note that we can also create sessions through *virtual connections* between hosts, i.e. an HTTP session.

Stateful protocol : A protocol that stores information about prior activity. FTP is an example of a stateful protocol. (see also *Stateless protocol*)

Stateless protocol : A protocol that does not store information about prior activity. HTTP is an example of a stateless protocol because it cannot distinguish one client's request from another client's request. (See also *Stateless protocol*)

Tile : The manageable pieces into which the topography is divided in a 3D terrain visualization. In a Quad Tree structured model, each tile is divided into four more detailed tiles recursively when applying higher level of detail.

User : A person or software that navigates through a 3D model. Can be juxtaposed with the term *client* when applied to an HTTP context.

Viewpoint : The parameters that define the position, orientation, and the field of view of the avatar. Often used to specify a rendezvous, or a static position and orientation, that the user can jump/teleport to.

Virtual Reality : The concept of having an interactive world (or universe) in which a user can nagivate among and interact with objects and topographical spaces.

Virtual Universe : (also called Virtual World, World or Universe) An graphical, interactive 3D representation through which a user may navigate and explore from different positions and angles.

VR : (see: Virtual Reality)

VRML : (Pronounced: *vermal*) Acronym for Virtual Reality Modeling Language. According to Rikk Carey et. al. [13] VRML is: "a 3D file interchange format" and that "[...]serves as a simple, multiplatform language for publishing 3D Web pages."

Web3D Consortium : An open consortium providing "a forum for the creation of open standards for Web3D specifications, and to accelerate the worldwide demand for products based on these standards through the sponsorship of market and user education programs". [45]

World : (See Virtual Universe)

X3D : The successor of VRML. It is based on XML and incorporates the extensions

previously provided by GeoVRML.

Xj3D : A toolkit for VRML and X3D. The content is written solely in Java and contains both a stand-alone 3D browser and a loader which can function as a VRML/X3D-interface for other applications.

XML: eXtensible Markup Language. An unlimited and self-defining language for describing data (in contrast with HTML which also describes appearance such as format).

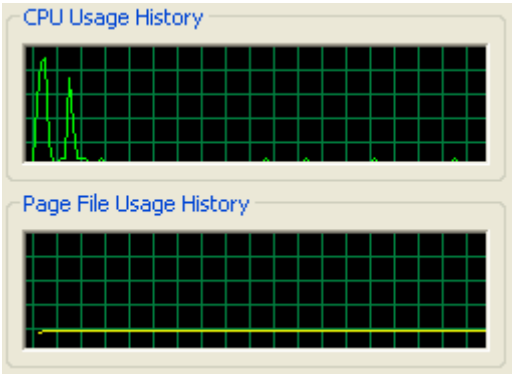
Appendix B - Tile Loading and Cache Management in Plugins

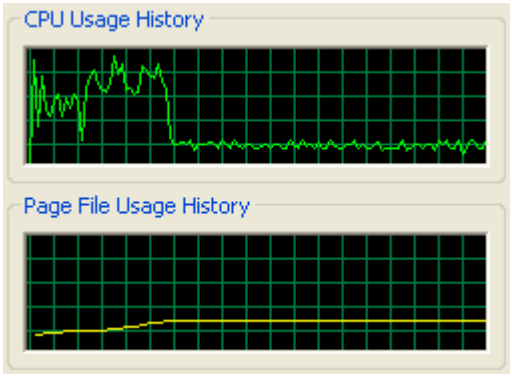
B.1 THE TEST MODEL

The model used in the tests described in this appendix is based on Herman Kolas' detailed model of Halden [5]. It is run through Chris Thorne's multi-resolution terrain management tool - *Rez* [46]. The test model is slightly modified so that every tile is located on a server and is accessed by sending a HTTP request to a servlet. All references in the VRML model are exchanged with a request to the servlet containing several parameters. Based on the parameters, the servlet finds out which VRML file that has been requested and sends a HTTP response with MIME type set to model/vrml. This results in a fairly decent simulation of a server that is generating 3D tiles "on-the-fly".

B.2 TESTING WHEN VRML PLUGINS LOAD TILES (SIMPLE TEST)

The following test monitors the CPU- and Memory usage the first minute after a very large VRML model of Halden is loaded. As soon as the model starts loading, an overview viewpoint is selected and no navigation is performed throughout the test. You can compare the two figures below to see the difference between viewing the model with blaxxun Contact and with Cortona VRML Client.





blaxxun Contact **Cortona VRML Client**

Figure 46 & Figure 49 – Monitoring blaxxun's and Cortona's CPU Usage the First Minute after Opening a Large VRML Model

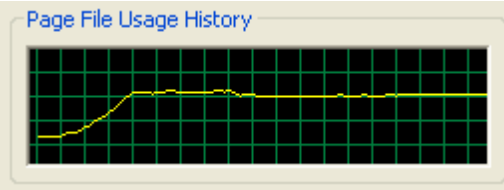
The graphs to the left represent the CPU usage and Physical Memory usage (respectively) for viewing the Halden model with blaxxun Contact, while the graphs to the right represent CPU usage and Physical Memory usage (respectively) for Cortona VRML Client.

Note that whereas blaxxun Contact seems to do no more processing after approximately 9 seconds (the north-west graph), Cortona VRML Client (the north-east graph) continues its processing all the way until the browsing of the model ends (every vertical line in the diagrams corresponds to three seconds). The question is: What is Cortona's VRML plugin working with that occupies over 20% of the CPU capacity? (There is a horizontal line for every 20% of total capacity.) Every horizontal line in the diagrams corresponds to 20% of the total capacity. In the uppermost graphs, the physical memory is stabilizing itself around 360 Megabytes, so if it is caching tiles, we have to assume that it stores this cached data on the hard disk.

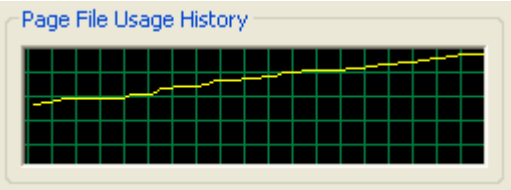
When browsing the Halden model with blaxxun Contacts, we can clearly see that the plugin stops processing the model as soon as all of the visible tiles have been rendered (which is approximately after 6 seconds). This is a clear indication of blaxxun not performing some sort of pre-caching activity (loading tiles before they need to be rendered).

B.3 CACHE MANAGEMENT

In the next test, I try to shed light on how the plugins restrain the amount of data that is cached (stored locally, or in physical memory, for faster reuse).



blaxxun Contact



Cortona VRML Client

Figure 47 & Figure 48 – Monitoring blaxxun's and Cortona's Physical Memory Usage when Browsing a Large VRML Model

With blaxxun Contact (the graph to the left), I have monitored the usage of physical memory as we navigate through the model. The memory usage increases until it reaches around 70% of the physical memory capacity (740 out of 1,024 Megabytes), and then it stabilizes. Once during the test, I performed a *JUMP* (a navigation feature supported by some plugins and browsers, which lets you navigate very quickly to a location by pointing the cursor on it). You can find this "jump" in the graph to the left. It is where the memory usage declines before it increases again. This decline in memory usage implies that the the plugin gets rid of already cached data before it can load and cache new data, and this is of course a desired result.

Strangely enough, this is not the case when browsing the Halden model with Cortona VRML Client. As we can see from the graph to the right, Cortona VRML Client seems to consume memory beyond the computer's physical memory (which was 1 GB when performing these tests). When all the physical memory is occupied, "swapping" of cached data to the hard drive becomes inevitable, and the machine resources become too busy for viewing 3D models efficiently.

Appendix C - Test of GeoVRML Compatibility Among Available VRML Plugins

C.1 OVERVIEW

Below, you will find an overview of the plugins I tested with a view to establish whether they supported the GeoVRML extension.

NAME:	URL:
blaxxun Contact	http://developer.blaxxun.com/
BS Contact	http://www.bitmanagement.de/products/fs_bs_contact_vrml.en.html
Cortona VRML Client	http://www.parallelgraphics.com/products/
Cosmo Player	http://www.ca.com/cosmo/home.htm
Flux	http://www.mediamachines.com/
Octagon Free Player	http://www.octaga.com/
OpenWorlds	http://www.openworlds.com/resources.html
Vcom 3D Venues	http://www.vcom3d.com/Viewer.htm

Table 8 – Overview of VRML Plugins Tested With a View to Establish GeoVRML Compatibility

C.2 TEST SPECIFICATION

In the following paragraph you will find out how the test was carried out:

- Every test was initiated by uninstalling all browsers/plugins previously installed on the system (including the GeoVRML extension).
- Then the plugins were, if it was possible, downloaded and installed manually (some plugins had to be installed directly).

- During installation/setup, OpenGL was always chosen as the way to render (as opposed to DirectX and other choices).

- After the installation was complete, GeoVRML 1.1 Run-Time was downloaded and installed manually (see how to).

- After GeoVRML was installed, the machine was restarted to make sure that the appropriate classpaths had been successfully updated.

- Then I tried to run the following examples with Internet Explorer:

1) GeoElevation example: http://www.geovrml.org/1.1/doc/examples/exagearth.wrl

2) Animated GeoViewpoint: http://www.geovrml.org/1.1/doc/examples/viewanim.wrl

3) On demand inlining: http://www.geovrml.org/1.1/doc/examples/geoinline.wrl

4) Georeferencing VRML objects: http://www.geovrml.org/1.1/doc/examples/geoloc.wrl

5) Using a GeoOrigin. http://www.geovrml.org/1.1/doc/examples/geoorigin.wrl

C.3 TEST ENVIRONMENT

The tests were performed on the following system:

OS:	Microsoft Windows XP Professional
Web-browser:	Microsoft Internet Explorer 6.0
Processor:	Intel Pentium 4, 2.8GHz
Physical memory:	1024 MB
Monitor adapter:	Radeon 9800 PRO, 128 MB RAM
VRML Extension:	GeoVRML 1.1 Run-Time (re-installed after each plugin)
Miscellaneous:	Wherever possible, I chose that the plugin should render using OpenGL.

Table 9 – The System Specifications for the System on which the Test Was Run

C.4 TEST RESULTS

As table 10 shows, Cortona was the only plugin, of the eight tested, that supported the GeoVRML extension. However, the only plugin that had a feasible management of tile loading was blaxxun Contact. The test performed in Appendix B shows that Cortona tries to download the whole model (all tiles) on startup without purging the cache when filled up. Sadly, this lead to the dismissal of the GeoVRML extension. The tests throughout this thesis is therefore run using the blaxxun Contact 5.104 plugin (unless otherwise stated) without utilizing GeoVRML.

Name:	Worked properly with GeoVRML:	Description:
blaxxun Contact 5.104	NO	The only plugin in this test that does not load tiles until they are needed. This feature is absolutely necessary when implementing very large Level of Detail-based models. However, it does not support Java in the script node*) and therefore the geospatial extension called GeoVRML is ruled out.
BS Contact	NO	Does not support GeoVRML, and the GeoVRML examples did not work.
Cortona VRML Client 4.2	**YES**	Supports Java in the script node and therefore also GeoVRML. Unfortunately, this browser loads all the tiles in the model on startup and is therefore unfit for browsing very large Level of Detail-based models.
Cosmo Player	NO	A very popular VRML browser. Cosmo stopped the development of this browser late in the 1990s, and the compatibility with Internet Explorer is poor. I did not get any of the examples up and running.
Flux	NO	Did not work with GeoVRML.
Octagon	NO	Did not work with GeoVRML.

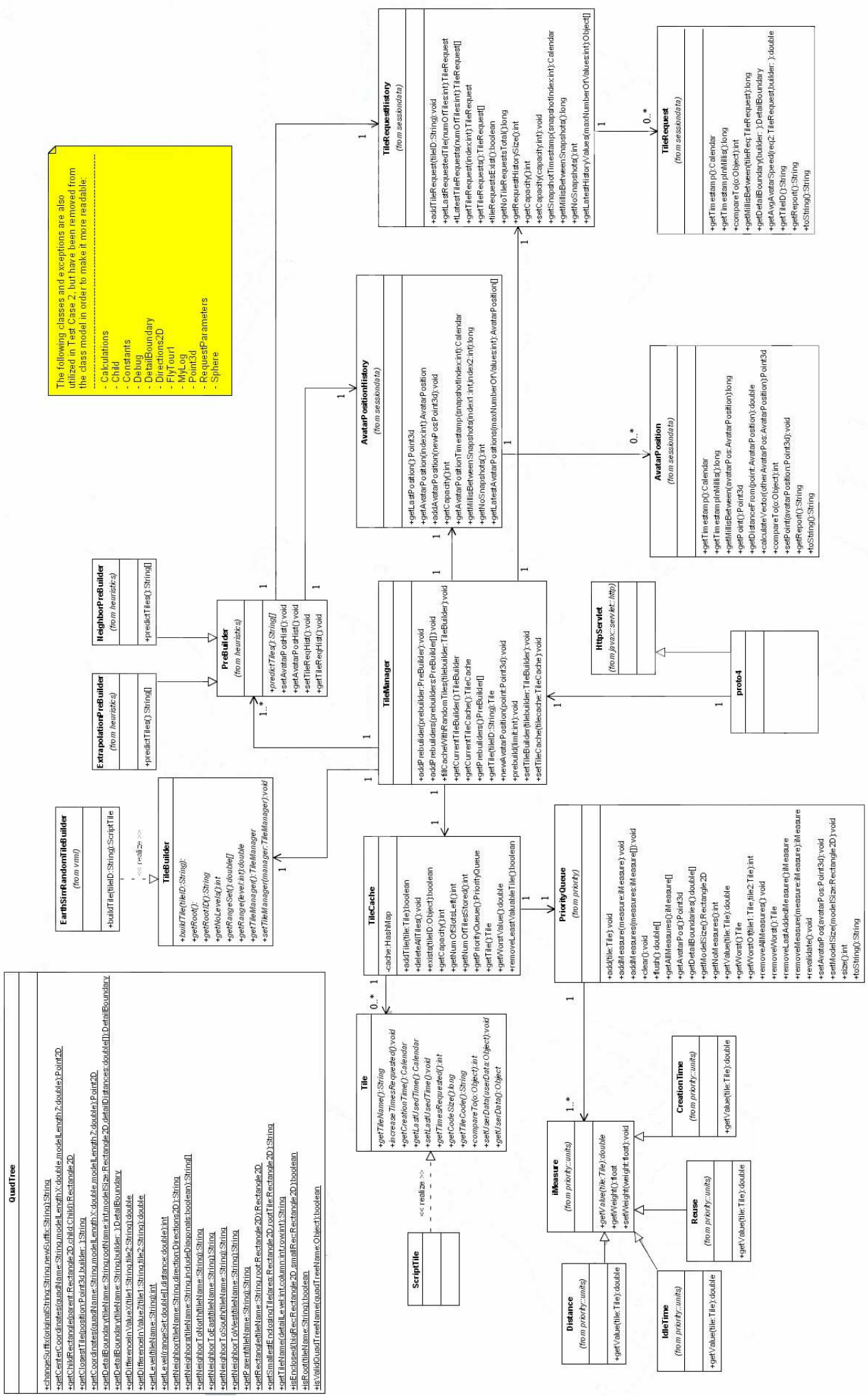
Free Player		
OpenWorlds	NO	Did not work with GeoVRML and was troublesome to uninstall.
Vcom 3D Venues	NO	Did not work with GeoVRML.

**) Java is embedded in VRML models by including a reference to the class files in a node called Script. Not all browsers/plugins support Java in this node.*

Table 10 – The Results of the GeoVRML Compatibility Test

Appendix D - Class Diagram for Test Case 2

The icon beneath is a link to a conceptual class diagram for Test Case 2, using UML notation. The most significant classes have been included, whereas the helper classes excluded from the model have been listed in the upper right corner. The diagram will not be described in detail here. It is included in this thesis in hope that it can be a supplement to help see the concepts described in this thesis in the big picture. The QuadTree class (positioned in the upper left corner) contains much of the logic and calculations regarding the quad tree structure, and is, as such, a very central class in this implementation.


QuadTree
+changeSuffix(originalString:String,newSuffix:String):String
+getCenterCoordinates(quadName:String,modelLength:X:double,modelLength:Z:double):Point2D
+getChildRectangle(parent:Rectangle2D,child:Child):Rectangle2D
+getClosestTile(position:Point3d,builder:):String
+getCoordinates(quadName:String,modelLength:X:double,modelLength:Z:double):Point2D
+getDetailBoundary(tileName:String,rootName:int,modelSize:Rectangle2D,detailDistances:double[]):DetailBoundary
+getDetailBoundary(tileName:String,builder:):DetailBoundary
+getDifferenceIn Value:X(tile1:String,tile2:String):double
+getDifferenceIn Value:Z(tile1:String,tile2:String):double
+getLevel(tileName:String):int
+getLevel(rangeSet:double[],distance:double):int
+getNeighbor(tileName:String,direction:Directions2D):String
+getNeighborsAll(tileName:String,includeDiagonals:boolean):String[]
+getNeighborToNorth(tileName:String):String
+getNeighborToEast(tileName:String):String
+getNeighborToSouth(tileName:String):String
+getNeighborToWest(tileName:String):String
+getParent(tileName:String):String
+getRectangle(tileName:String,root:Rectangle2D):Rectangle2D
+getSmallestEnclosingTile(area:Rectangle2D,rootTile:Rectangle2D):String
+getTileName(detailLevel:int,column:int,row:int):String
+isEnclosed(bigRec:Rectangle2D,smallRec:Rectangle2D):boolean
+isRoot(tileName:String):boolean
+isValidQuadTree(name:quadTreeName:Object):boolean
EarthSimRandomTileBuilder
(from vmi)
+buildTile(tileID:String):ScriptTile
<< realize >>
TileBuilder
+buildTile(tileID:String):
+getRoot():
+getRootID():String
+getNoLevels():int
+getRangeSet():double[]
+getRange(level:int):double
+getTileManager():TileManager
+setTileManager(manager:TileManager):void
ExtrapolationPreBuilder
(from heuristics)
+predictTiles():String[]
INeighborPreBuilder
(from heuristics)
+predictTiles():String[]
PreBuilder
(from heuristics)
+predictTiles():String[]
+setAvatarPosHist():void
+getAvatarPosHist():void
+setTileReqHist():void
+getTileReqHist():void
The following classes and exceptions are also utilized in Test Case 2, but have been removed from the class model in order to make it more readable:
- Calculations
- Child
- Constants
- Debug
- DetailBoundary
- Directions2D
- FlyTour1
- MyLog
- Point3d
- RequestParameters
- Sphere
TileRequestHistory
(from sessiondata)
+addTileRequest(tileID:String):void
+getLastRequestedTile(numOfTilesint):TileRequest
+LatestTileRequests(numOfTilesint):TileRequest[]
+getTileRequest(index:int):TileRequest
+getTileRequests():TileRequest[]
+tileRequestsExist():boolean
+getNoTileRequestsTotal():long
+getRequestHistorySize():int
+setCapacity(capacity):void
+getSnapshotTimestamp(snapshotIndex:int):Calendar
+getMillisBetweenSnapshots():long
+getNoSnapshots():int
+getLatestHistoryValues(maxNumberOfValues:int):Object[]
TileRequest
(from sessiondata)
+getTimestamp():Calendar
+getTimestampInMillis():long
+compareTo(o:Object):int
+getMillisBetween(tileReqTileRequest):long
+getDetailBoundary(builder:):DetailBoundary
+getAvgAvatarSpeed(req2:TileRequest,builder:):double
+getTileID():String
+getReport():String
+toString():String
AvatarPositionHistory
(from sessiondata)
+getLastPosition():Point3d
+getAvatarPosition(index:int):AvatarPosition
+addAvatarPosition(newPos:Point3d):void
+getCapacity():int
+getAvatarPositionTimestamp(snapshotIndex:int):Calendar
+getMillisBetweenSnapshots(index1:int,index2:int):long
+getNoSnapshots():int
+getLatestAvatarPositions(maxNumberOfValues:int):AvatarPosition[]
AvatarPosition
(from sessiondata)
+getTimestamp():Calendar
+getTimestampInMillis():long
+getMillisBetween(avatarPos:AvatarPosition):long
+getPoint():Point3d
+getDistanceFrom(point:AvatarPosition):double
+calculateVerdor(otherAvatarPos:AvatarPosition):Point3d
+compareTo(o:Object):int
+setPoint(avatarPosition:Point3d):void
+getReport():String
+toString():String
TileManager
+addPrebuilder(prebuilder:PreBuilder):void
+addPrebuilders(prebuilders:PreBuilder[]):void
+fillCacheWithRandomTiles(tilebuilder:TileBuilder):void
+getCurrentTileBuilder():TileBuilder
+getCurrentTileCache():TileCache
+getPrebuilders():PreBuilder[]
+getTile(tileID:String):Tile
+newAvatarPosition(point:Point3d):void
+preload(limit:int):void
+setTileBuilder(tilebuilder:TileBuilder):void
+setTileCache(tilecache:TileCache):void
HttpServlet
(from javax::servlet::http)
proto4
TileCache
-cache:HashMap
+addTile(tile:Tile):boolean
+deleteAllTiles():void
+existsTile(ID:Object):boolean
+getCapacity():int
+getNumOfSlotsLeft():int
+getNumOfTilesStored():int
+getPriorityQueue():PriorityQueue
+getTile():Tile
+getWorstValue():double
+removeLeastValuableTile():boolean
PriorityQueue
(from priority)
+add(tile:Tile):void
+addMeasure(measure:IMeasure):void
+addMeasures(measures:IMeasure[]):void
+clear():void
+flush():double[]
+getAllMeasures():IMeasure[]
+getAvatarPos():Point3d
+getDetailBoundaries():double[]
+getModelSize():Rectangle2D
+getNoMeasures():int
+getValue(tile:Tile):double
+getWorst():Tile
+getWorstOfTile(t1:Tile,tile2:Tile):int
+removeAllMeasures():void
+removeWorst():Tile
+removeLastAddedMeasure():IMeasure
+removeMeasure(measure:IMeasure):IMeasure
+revalidate():void
+setAvatarPos(avatarPos:Point3d):void
+setModelSize(modelSize:Rectangle2D):void
+size():int
+toString():String
Tile
-getTileName():String
+increaseTimesRequested():void
+getCreationTime():Calendar
+getLastUsedTime():Calendar
+setLastUsedTime():void
+getTimesRequested():int
+getCodeSize():long
+getTileCode():String
+compareTo(o:Object):int
+setUserData(userData:Object):void
+getUserData():Object
ScriptTile
<< realize >>
IMeasure
(from priority::units)
+getValue(tile:Tile):double
+getWeight():float
+setWeight(weight:float):void
CreationTime
(from priority::units)
+getValue(tile:Tile):double
Distance
(from priority::units)
+getValue(tile:Tile):double
IdleTime
(from priority::units)
+getValue(tile:Tile):double
Reuse
(from priority::units)
+getValue(tile:Tile):double

Bibliography

[1] "Virtual Reality - History". National Center for Supercomputing Applications. 24 Oct 1995. Retrieved: 14 Feb 2004. <http://archive.ncsa.uiuc.edu/Cyberia/VETopLevels/VR.History.html>

[2] "History of the VRML Specification". Retrieved: 14 Feb 2004. <http://www.web3d.org/about/historyofvrml.html>

[3] VRML97's official web-site: <http://www.web3d.org/technicalinfo/specifications/vrml97/index.htm>

[4] GeoVRML's official web-site: <http://www.web3d.org/technicalinfo/specifications/vrml97/index.htm>

[5] OneMap. VRML model of Halden. <http://www.onemap.org/resources/sentrum/WorldSingleEG/world/DisplayTerrain.wrl>

[6] GeoVRML Nodes. Reddy, Martin. <http://www.geovrml.org/1.1/doc/nodes.html>

[7] GeoVRML 1.1 Examples. <http://www.geovrml.org/1.1/doc/examples.html>

[8] Flanagan, David. "JavaScript, The definitive Guide". 3rd ed. O'Reilly & Associates. Sebastopol: 1998.

[9] Daly, Leonard. "Upgrading Basic VRML97 to X3D". 2004. Retrieved: 16 Feb 2004. <http://3dgraphics.about.com/library/weekly/aa033103a.htm>

[10] Web3D. "Plugins - Presentation and interaction". Retrieved: 03 Dec 2003. <http://www.web3d.org/technicalinfo/specifications/vrml97/part1/concepts.html#4.2.7>

[11] "blaxxun | Products | blaxxun Contact 5.1". Retrieved: 20 Dec 2004. "<http://www.blaxxun.com/en/products/contact/>

[12] Web3D.org. VRML97 Specification. <http://www.web3d.org/technicalinfo/specifications/specifications.htm>

[13] Carey, Rick et. Gavin Bell. "The Annotated VRML 2.0 Reference Manual". Addison Wesley. 1997.

[14] TerraVision. <http://www.ai.sri.com/TerraVision/>

[15] Open Geospatial Consortium (OGC). <http://www.opengeospatial.org/>

[16] "Planet Earth". Retrieved: 13 Dec 2004. <http://www.planet-earth.org>

[17] 3map. Retrieved: 4 Dec 2004. <http://www.ping.com.au/3map/>

[18] SINTEF Virtual Globe. Rune Aasgard. <http://globe.sintef.no/>

[19] Rune Aasgard. "Virtual Globe". Retrieved: 13 Dec 2004. <http://globe.sintef.no/>

[20] "The Digital Earth: Understanding our planet in the 21st Century". Al Gore. 31 Jan 1998. Los Angeles. Retrieved: 15 Dec 2004. <http://www.digitalearth.gov/speech.html>

[21] Digital Earth. <http://gai.fgdc.gov/>

[22] Geospatial Applications & Interoperability (GAI). <http://gai.fgdc.gov/>

[23] ESRI. Retrieved: 4 Dec 2004. <http://www.esri.com>

[24] MultiGen-Paradigm. <http://www.multigen.com/>

[25] There.com. Retrieved: 4 Dec 2004. <http://www.there.com/>

[26] Reddy, Martin. "Perceptually Modulated Level of Detail for Virtual Environments". University of Edinburgh: 1997.

[27] "Heuristic". WhatIs.com. Updated: 03 Jan 2003. Retrieved: 02 Mar 2004. <http://searchcio.techtarget.com/sDefinition/0,,sid19_gci212246,00.html>

[28] Ross A. Finlayson. "VRML 2.0 EAI FAQ". Retrieved: 28 Jun 2004. <http://www.frontiernet.net/~imaging/eaifaq.html>

[29] World Wide Web Consortium. "Embedded object". Retrieved: 28 Jun 2004. <http://www.w3.org/TR/2001/PR-MathML2-20010108/appendixh.html>

[30] blaxxun Developer Information. "VRML 2.0 External Authoring Interface (EAI)". Retrieved: 28 Jun 2004. <http://developer.blaxxun.com/developer/contact/3d/eai/index.html>

[31] Genesis : Search for Origins | JPL | NASA. "Extrapolation". Retrieved: 29 Jun 2004. <http://www.genesismission.org/glossary.html>

[32] NCAR Graphics. "Extrapolation". Retrieved: 29 Jun 2004. <http://ngwww.ucar.edu/ngdoc/ng/ngmath/natgrid/extrapolation.html>

[33] Morten Granlund. "Rezerver - Rez to Servlet Converter".

[34] Jack Shirazi. "JAVA Performance Tuning". 1st ed. O'Reilly & Associates. Sebastopol: 2000.

[35] Aschehoug og Gyldendals store ettbinds leksikon. "Jorden".

[36] Taylor Hamish. "Advanced VRML Glossary". Retrieved: 15 Oct 2004. <http://www.macs.hw.ac.uk/~hamish/9ig2/glossary.html>

[37] "JCanyon -Grand Canyon Demo". Delevopers.sun.com. Retrieved: 8 Dec 2004. <http://java.sun.com/products/jfc/tsc/articles/jcanyon/>

[38] "Dot-com". Wikipedia. Retrieved: 17 Dec 2004. <http://en.wikipedia.org/wiki/Internet_bubble>

[39] "Federal Geographic Data Committee". Retrieved 17 Dec 2004. <http://www.fgdc.gov/>

[40] "daemon". Microsoft Windows 2000 Server Documentation. Retrieved: 15 Mar 2004. <http://www.microsoft.com/windows2000/en/server/iis/default.asp?url=/windows2000/en/server/iis/htm/core/iigloss2.htm>

[41] "Geospatial". Webster's New Millennium™ Dictionary of English, Preview Edition (v 0.9.5). Retrieved: 13 Dec 2004. <http://dictionary.reference.com/search?q=geospatial>

[42] Sun Microsystems. Java Technology. <http://java.sun.com/>

[43] John R. Hubbard et. A. Huray. "Data Structures with Java". International Edition. Person Education Inc. USA: 2004.

[44] Callaway, Dustin R. "Inside Servlets, Server-Side Programming for the Java Platform". USA: 1999.

[45] "About the Web3D Consortium".Web3D.org. Retrieved: 20 Dec 2004. <http://www.web3d.org/contact/about.html>

[46] Chris Thorne. Rez - Multiresolution terrain management tool. <http://old.ping.com.au/3map/rez/>