"Politehnica" University of Bucharest

Faculty of Computer Science

Diploma project

Extend the MonALISA client with 3D data visualisation models

Coordinator:
PhD-Professor Ing. Nicolae Ţăpuş (UPB)
Advisor:
PhD-Professor Iosif Charles Legrand (Caltech)

Student:

Lucian Muşat

Bucharest

- 2004 -

1. Introduction

MonALISA is a complex application created with the scope of monitoring grid activities.

It is composed primarely of a service component that is run on a computer on the net, and this service, by using several modules, monitores the intranet computers for their current activities and the reports to any client application that wants to see the data. The modules are wrappers on other tools made for monitoring specifical parameters or filters that transform the gathered data to a new meaning and make this available for the clients.

The client part of MonALISA has the role of representing in a human readable/viewable form the information gathered from several MonALISA services. So, there are several ways available to the client of visualizing the data.

1.1. Actual situation

Currently, MonALISA client has two world map visualisation modules, one for plane projection and one for sphere projection. As they use for representation the geographical world map, the client has attached an image file with a certain resolution. Both modules support zooming option and, when the area in view is very small, the viewable piece of world map looses its quality, as the pixels become larger and that creates an upleasant effect to the eye as the user cannot identify no more the location it is in.

The solution that is implemented is to have a set of several images with geographical world map, at different resolution and have the client run with one of these specified at start-up. The available resolutions are: 1024x512 (pixels), 2048x1024, 4096x2048 and 8192x4096.

When using a high resolution map, like the one of 4096 pixels, or 8192, the client uses large amounts of memory for storing the image, and so, the computer is run on must be a very powerfull machine.

A table of memory consumption is given below:

resolution	**memory consumption**
1024x512	16 Mb
2048x1024	64 Mb

4096x2048	256 Mb
8192x4096	1024 Mb = 1 Gb !!!!

1.2. Project's purpose

This paper proposes to introduce a new visualisation model for the MonALISA client, implemented using Java and OpenGL technology that will improve the performance of the application and the visualisation mode.

The way to increase the performance of the application is to unite the two modules that uses world map images, and, by doing that, decrease the memory consumtion, and then, implement an image changing algorithm so that to have the correct image for a certain zooming level, that has a 1 to 1 corespondence between an image pixel on application's map and a pixel on the screen.

For improvement of visualisation, I propose several new data visualisation models that function more like a set of plugins, in the way that, the user can choose what visualisation models to load, and then choose from a list the active one.

2. Technologies for 3D in Java

As the MonALISA framework became more known and used, problems appeared as different users were using it, and, as a direct action, the client application has suffered many modifications from it's initial form. One of this updates was the introduction of a 3d visualisation model, called Globe Panel, that allowed the user to view the data spread on a sphere covered by an image with geographical earth map. At that time, this was an excelent new method to visualize the data, but the drawbacks where that, in order to be able to use it, a user had to have installed on it's computer, besided the java runtime enviroment (jre), the Java 3D extension for displaying three dimensional graphics and that, for better looking earth map, the user had to have a powerfull computer, as the entire map has to be loaded into memory and used by this module.

2.1. Java3D

The technology used for Globe Panel is called Java3D and can be found at http://java.sun.com/products/java-media/3D/ and represents the Sun's proprietary implemetation of OpenGL extension in Java. For Windows platform, Java3D is available also in a flavour of DirectX implementation. This technology is not fully supported by Sun as the linux implementation for Java3D is available from Blackdown, http://www.blackdown.org/

Java 3D API is an application programming interface used for writing stand-alone three-dimensional graphics applications or Web-based 3D applets. It gives developers high level constructs for creating and manipulating 3D geometry and tools for constructing the structures used in rendering that geometry. With Java 3D API constructs, application developers can describe very large virtual worlds, which, in turn, are efficiently rendered by the Java 3D API.

The Java 3D API specification is the result of a joint collaboration between Silicon Graphics, Inc., Intel Corporation, Apple Computer, Inc., and Sun Microsystems, Inc. All had advanced, retained mode APIs under active internal development, and were looking at developing a single, compatible, cross-platform API using Java technology.

The Java 3D API draws its ideas from the considerable expertise of the participating companies, from existing graphics APIs, and from new technologies. The Java 3D API's low-level graphics constructs synthesize the best ideas found in low-level APIs such as Direct3D API's, OpenGL, QuickDraw3D, and XGL. Similarly, Java 3D API's higher-level constructs leverage the best ideas found in several modern scene graph-based systems.

Sumarization of Java3D:

- Java3D is a Java Application Programmer's Interface (API)
- Java3D is a set of standardized classes for use by Java programmers
- It enables authors to control shapes, animation, and interaction, PLUS rendering, input devices, and lots more
- It is (roughly) a Java-based superset of VRML.
- The start of Java3D

December 4 1995	Sun and SGI announce plans to develop a 3D API for Java. The partnership grows to include Apple and Intel
May 27 1997	Version 0.95 specification released - first public viewing
July 16 1997	Version 0.98 specification released
August 5 1997	Sun teaches course at SIGGRAPH 97

- Like VRML, Java3D is used to create a scene graph:
- A hierarchy of nodes and groups of nodes
- Nodes are instances of Java3D classes: shapes, groups, sounds, behaviors, etc.

2.2. JavaOne conference - alternatives for 3D graphics using OpenGL

This implementation of Java3D in MonALISA client is about 1 year old, and in this time, the java community has worked on bringing in java more powerfull implementations of 3D.

The 2002 and 2003 Java One conferences concentrated on new 3D technologies in java that could compete with c/c++ native implementations on different platforms. All this alternatives concentrated on bringing the power of OpenGL technology in java. OpenGL is the industry's most widely used and supported 2D and 3D graphics application programming interface (API).

At first Java3D was Sun’s solution for accelerated 3d graphics, but that meant a complex api that was not indicated for fast-rendering applications.

As always, a product cannot satisfy all customers, and cannot comply with the newest technologies on the market, and so, the java community, with Sun’s support, worked on different alternatives.

The most important ones are:

- gl4java,
- lwjgl,
- magician,
- jungle.

All this alternatives were presented at JavaOne's conferences in 2002 and 2003, the presentations being available at http://java.sun.com/javaone/ (main site), or dirrectly at

http://servlet.java.sun.com/javaone/resources/content/sf2002/conf/sessions/pdfs/3167.pdf (for 2002 presentation on OpenGL for Java), and

http://servlet.java.sun.com/javaone/resources/content/sf2003/conf/sessions/pdfs/2125.pdf (for 2003 presentation)

2.2.1. GL4Java (OpenGL for Java Technology)

The first alternative was gl4java, available at http://www.jausoft.com/products/gl4java/gl4java_main.html as a binding over natice OpenGL library of the underlying operating system. At the moment the development has stopped, last version dating from 11th December 2001.

The last available version is 2.8.2.0 and maps the complete OpenGL 1.3 API and the complete GLU 1.2 API to Java and integrates all managment functions, while using the Java-Native-Interface (JNI) and the JDirect-Interface of MS-JVM.

The GL4Java API maps native c++ OpenGL functions to the Java language and provides fast access to 3D accelerators from Java.

GL4Java maps the complete OpenGL 1.3 and GLU 1.2 API and implements window handle functions (native and java), while using the Java-Native-Interface (JNI) of Java or the JDirect-Interface of MS-JVM. Win32, X-Window, Mac.

GL4Java has been developed under Open Source Model since 1997 and has become over the years a serious and stable API.

The main programer of this library was Sven Gothel from Jausoft and his product, The GL4Java library is free software that can be redistributed and/or modified under the terms of the GNU Library General Public License as published by the Free Software Foundation; either version 2 of the License, or any later version.

GL4Java runs on GNU/Linux, Win32, Mac OS 9.x, Mac OS X and any Unix/X11/OpenGL.

How does GL4Java compare to Java3D ?

GL4Java is an OpenGL binding and provides nothing else than access to OpenGL/GLU commands.

GUI managment and toolkits like GLUT, texture managment, etc. are also provided.

Java3D is a multipurpose closed source scene graph API relying on DirectX and OpenGL.

Java3D provides parsers for loading objects, functions to manipulate scene objects, such as texturing, morphing and so on... Because of Java3D's multipurpose nature, it contains many functionalities and thus contains an important amount of code, wich many parts of it remain unused because they aren't in the field of purpose. This makes the API heavy and long to load. But then the display speed performs reasonnably. On the other hand Java 3D is much easier to use because it is a high level API. For example if you want to do a morph in Java3D, you create a morph node, load your geometry targets inside it, and set the morph amount between the targets.

Using plain OpenGL for a 3D morphing, you have to go through every vertex of your geometry and compute their x,y,z position yourself. Using plain OpenGL you have to code your own file parser and display your object, polygon per polygon, all this with your little hands ! This is an important work and it takes time. More, OpenGL is very specific and thus it's quite hard to learn, often requiring a good knowledge of math. But in the end you have coded only what you needed and your program is very small and fully optimized (this requires you use OpenGL intelligently, though).

Author's conclusion is: "*So if you need to have your project coded very quickly, if performance is not a top priority criteria and if you feel desesparate when doing math, use an avaiable well functional SCENEGRAPH environment, e.g. Java3D. If performance is critical (such as for games), if you have plenty of time, and you feel your shoulders are strong enough to face the big and malicious OpenGL monster, use plain OpenGL !*"

Implementations using GL4Java

An application that has been presented at Java One Conference in 2002 and that uses GL4Java is "JCanyon: Grand Canyon for Java" made by Kenneth Russell, Sun Microsystems, Inc., and can be found at http://java.sun.com/products/jfc/tsc/articles/jcanyon/

The jcanyon demonstration is a simple flight simulator which visualizes a large data set (roughly three hundred megabytes in size).

Other applications/libraries using GL4Java directly, or a derivation:

- gleem (http://www.media.mit.edu/~kbrussel/gleem/) : OpenGL Extremely Easy-to-use Manipulators, by Kenneth B. Russell
- Opale.Soya (http://opale.soya.tuxfamily.org/) : Opale.Soya is a 3D engine
- jFree-D (http://www.linuxstart.com/~jfreed/) : A free Java3D implementation using GL4Java by Jean-Christophe Taveau ! There is an JoGL and a GL4Java implementation !
- WebWinds (http://webwinds.jpl.nasa.gov/) : Interactive Data System. WebWinds will use GL4Java or an own OpenGL Java mapping based among others on GL4Java.
- MathModelica (http://www.mathcore.com/mathmodelica/) : MathModelica is a new kind of software tool for modeling, simulation, and visualization. It uses GL4Java for the 3D rendering.

2.2.2. LWJGL (Lightweight Java Game Library)

LWJGLis another 3d graphics library for java that can be found at http://www.lwjgl.org/ as is still under development, LWJGL version 0.9 alpha having been released on Tuesday, April 13th, 2004.

LWJGL is designed for New I/O, as it has additional support for audio (OpenAL)
and game input devices, also supports full-screen rendering, but does not support AWT and Swing integration and exposes pointers as longs, which destroys type safety.

The Lightweight Java Game Library (LWJGL) is a solution aimed directly at professional and amateur Java programmers alike to enable commercial quality games to be written in Java. LWJGL provides developers access to high performance crossplatform libraries such as OpenGL (Open Graphics Library) and OpenAL (Open Audio Library) allowing for state of the art 3D games and 3D sound. Additionally LWJGL provides access to controllers such as Gamepads, Steering wheel and Joysticks. All in a simple and straight forward API.

LWJGL is not meant to make writing games particularly easy; it is primarily an enabling technology which allows developers to get at resources that are simply otherwise unavailable or poorly implemented on the existing Java platform. The library developers anticipate that the LWJGL will, through evolution and extension, become the foundation for more complete game libraries and "game engines" as they have popularly become known.

LWJGL is available under a BSD license, as it's open source and freely available at no charge.

For the purpose of this paper, this implementation is not usefull as is it aimed at writing games, as it has no support for awt and/or swing, and that means that it can’t be integrated into a java frame based application.

2.2.3. Magician

Another binding over OpenGL, that was a front-runner in the cross-platform Java binding-for-OpenGL standardization race, but as from March 12, 1999 it was no longer supported and developed by Arcane Technologies, who owned the source code, and never made it public.

Some attributes of this bindings are:

- Swing and JFC 1.1 integration via a lightweight OpenGL-aware component
- Offscreen rendering support to work with Magician's standard heavyweight and new lightweight components
- Enhanced capabilities support, including the ability to enumerate available capabilities (or visuals, in X Windows terminology)
- Overlay support
- Runtime OpenGL version checking
- Some performance enhancements
- OpenGL 1.2 compliance
- Clean API
- Integrated with AWT and Swing
- Innovative composable pipeline (e.g. DebugGL)
- Did not support New I/O

2.2.2. Jungle

Binding over OpenGL for Java started by Ken Russel and Chris Kline. Russell is a Sun Microsystems employee working on the HotSpot Virtual Machine with many years of 3D experience. Kline works for Irrational Games and also is very experienced with 3D graphics.

Here are some of its characteristics:

- Supports OpenGL 1.4 and vendor extensions
- Integrates with AWT and Swing
- Designed for New I/O
- Clean, minimalist API
- Supports composable pipeline (e.g., DebugGL)
- Open source
- Written almost entirely in Java programming language
- AWT Native Interface, WGL and GLX bound into Java programming language using GlueGen

When developers started working in collaboration with Java Gaming Initiative, Jungle has been adopted as JGI's OpenGL binding, and has been renamed to "Jogl" (Java bindings for OpenGL).

It is, in its actual form, open source (modified BSD license), and available from https://jogl.dev.java.net

3. Project specifications

First make an evaluation of 3D technologies based on OpenGL available at the moment, and find a solution for replacing/complementig the Java3D implementation currently available for the MonALISA client.

Second, see the impruvements that can be done using this technology and some of the problems that appeared during development.

3.1. Evaluation of 3D technologies

- **OpenGL provides a procedural model of graphics**
  This closely matches many of the algorithms and methods graphics programmers have used historically. The procedural model is at once intuitive and straightforward for many accomplished 3D graphics aficionados.
- **OpenGL provides direct access to the rendering pipeline**
  This is true with any of the various language bindings, including most Java bindings. OpenGL empowers programmers to directly specify how graphics should be rendered. One doesn't just *hint* and *request* as with Java 3D, one *stipulates.*
- **OpenGL is optimized in every imaginable way**
  OpenGL is optimized in hardware and software and targeted platforms ranging from the cheapest PCs and game consoles to the most high-end graphics supercomputers.
- **Vendors of every kind of 3D graphics-related hardware support OpenGL**
  OpenGL is *the* standard against which hardware vendors measure their graphics technology, bar none. As Microsoft has joined with SGI in the Fahrenheit initiative, it has become increasingly obvious to many that this is in effect Microsoft's indirect acknowledgment that OpenGL won the API wars for 2D and 3D graphics.
- On the other hand, nothing is perfect. OpenGL, and certainly Java-OpenGL bindings, do have some significant shortcomings:
- **The strengths of the procedural approach to graphics programming are simultaneously a weakness for many Java programmers**
  For relatively new programmers, many of whom may have received their first formal programming instruction in Java using object-oriented methodologies, OpenGL's procedural method does not mesh well with an object-oriented approach and good engineering practice.
- **Many vendors' OpenGL optimizations are meant to decrease hardware choice**

  It is in each vendor's best interest to build proprietary extensions and make proprietary optimizations to sell more of its own hardware. As with all hardware optimizations, you must use accelerator-specific OpenGL optimizations with the

understanding that each optimization for one platform diminishes portability and performance for several others. Java 3D's more general-purpose optimizations mostly aim to maximize the portability of Java 3D applications.

- **OpenGL's exposure of the inner details of the rendering process can significantly complicate otherwise simple 3D graphics programs** Power and flexibility come at the price of complexity. In the fast development cycles of today's technology world, complexity is in and of itself something to be avoided where possible. The old adage about bugs is true: the more lines of code, the more bugs (in general).

As you can see from the pros and cons for OpenGL-based approaches, Java-OpenGL is strong in many of the areas in which Java 3D is weak. OpenGL gives programmers the low-level access to the rendering process that Java 3D explicitly avoids, and OpenGL is currently available on far more platforms than Java 3D (Magician aside). But this flexibility comes with a potential price: programmers have a lot of room to optimize, which conversely means they have a lot of room to screw things up. Java 3D has more built-in optimization and an easier programming model that may prove particularly useful for programmers new to Java, 3D graphics work, or networked and distributed graphics programming.

3.2. Improvement of 3D part in MonALISA client

Improvement of 3D part in MonALISA client and the graphical navigation in 3D can be done by taking in account two factors:

- the performance of the application when using this module, and
- the ease of use and understanding for the client of the data values that are represented.

Performance is a key element for any application, and, as the industry for 3D hardware is developing permanently, so are the 3D solutions for improving graphics on home computers. OpenGL is such a solution and its implementation in java can rival with the native c/c++ implementations.

Also, there are several academic papers regarding algorithms for rendering maps at different levels of detail, and one of them is the ROAM (Real-time Optimally Adapting Meshes) who’s ierarhical structure was the base for the algorithm developed in this paper.

These different levels of detail maps allow a better and easier use of the location of different visible data on the client.

3.3. Theoretical aspects

Because is 3D graphics, the algorithms used in this paper uses a lot of mathematics and phisics, mainly vectors operations and other 3D space computations.

The 3D representation of the module is done by considering an eye (or camera) and a view frustrum, starting from eye position, and following an direction, taking in account a normal for up direction.

Every object in this space is referenced to the eye and it's view frustrum, and the 2D representation viewable on the screen is realised by using a specific projection: perspective projection.

The other projection is orthographic, or parallel projection, because it involves no perspective correction. There is no adjustment for distance from the camera made in these projections, meaning objects on the screen will appear the same size no matter how close or far away they are.

Traditionally this type of projection was included in OpenGL for uses in CAD, or Computer Aided Design. Some uses of orthographic projections are making 2D games, or for creating isometric games. To setup this type of projection use the OpenGL provided glOrtho() function.

Although orthographic projections can be interesting, perspective projections create more realistic looking scenes, so that's what I will use. In perspective projections, as an object gets farther from the viewer it will appear smaller on the screen- an effect often referred to as foreshortening. The viewing volume for a perspective projection is a frustum, which looks like a pyramid with the top cut off, with the narrow end toward the user.

There is a few different ways the view frustum can be setup, and thus the perspective projection. The first is as follows:

void glFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far)

where left and right specify the x-coordinate clipping planes, bottom and top specify the y-coordinate clipping planes, and near and far specify the distance to the z-coordinate clipping planes. Together these coordinates provide a box shaped viewing volume.

Using glFrustum enables you to specify an asymmetrical frustum, which can be very useful in some instances, but isn't what you typically want to do. For a different solution OpentGL has the Utility Library:

void gluPerspective(GLdouble fov, GLdouble aspect, GLdouble near, GLdouble far);

where fov specifies, in degrees, the angle in the y direction that is visible to the user; aspect is the aspect ratio of the scene, which is divided by the height. This will determine the field of view in the x direction.

This functions are standard functions for OpenGL, and their available in any implementation, on any platform.

A first problem that appeard was that, on certain platforms, at least linux, the support for glu* functions was defectuos, although that gl* functions worked fine. That was because there are 2 separate libraries for this 2 sets of functions and, although the basic OpengGL library named „gl" was properly setup, the „glu" was not, and so, for implementation I used the glFrustum projection function.

Data representation in this modules is done by representing several geometric shapes, as pies or spheres. To decrease the number of OpenGL calls, the positioning of this objects is memorized in internal structures. The computation is based on using a position and a direction vector, so, for representation, 3D math computation is used. Also, camera movement in 3D space is not left to OpenGL, but is done manually by using structures to memorize position, direction and normal, and functions for computation, that are called inside of user interaction functions.

4. Description of used technologies

The programming language chosen for this project's development was Java. Java is a high level programming language, started and developed by JavaSoft, The business unit of Sun Microsystems that is responsible for Java technology. Some of the most important characteristics of this language are:

- simplicity, by eliminating operators overloading, multiple inheritance and all "features" that can misslead the user in writing cluttered code.
- robustness, that means to remove the sources of frequent errors in programming, the corrupted pointers and memory leaks by implementing an automatic memory memory management and emploing a garbage collector that reclaims the memory occupied by an object once it determines that object is no longer accessible. A Java application that has passed the compilation phase behaves well on execution in the sense that it does not "break the system".
- totally object oriented and by that completely rule out the procedural programming style.
- ease for network programming
- security, is the most secure programming language available at the moment, by assuring strict applications security mechanisms through: dynamic checking of code for dangerous sequences, enforcing of strict policies for applications run on remote computers.
- is neutral from architectural point of view
- portability, Java is a platform - independent programming language, the same application can run, without any modification, on different systems, as Windows, UNIX, Linux or Macintosh, a thing to consider for internet related developer firms because it causes substantial savings by not having to implement the same thing on each platform.
- compiled and interpreted
- high performance
- is multithreaded
- dynamicity

- is inspired from C and C++, going from these ones to Java is done without much effort.
- allows easy creation of web documents with animation and multimedia.

4.1. Jogl (Java binding for OpenGL)

The 3D technology I have chosen to develop the project is JoGL.

JoGL is designed to provide hardware-supported 3D graphics to applications written in Java. It is part of a suite of open-source technologies initiated by the Game Technology Group at Sun Microsystems and integrates the best ideas from LWJGL, Magician, and GL4Java while remaining fast, clean, and easy to use.

JOGL provides full access to the APIs in the OpenGL 1.5 specification as well as nearly all vendor extensions, and integrates with the AWT and Swing widget sets.

OpenGL is the prevailing industry API for developing 2D and 3D graphical applications. It can be considered the successor to the formidable Silicon Graphics IRIS GL-library which made so popular SGI workstations as the predilect platform for scientific, engineering and special effects development. SGI put into OpenGL a great deal of their expertise to make an easy-to-use, intuitive, portable and network aware API for the future. At the same time SGI for realize the importance of open standards. Several hardware and software makers took part in OpenGL's specification and stand behind it. Thanks to this, OpenGL applications can be ported quite easily to virtually any platform in the market, from windows, to Linux system, high-end UNIX workstations, right up to mainframe supercomputers. The ARB (Architectural Review Board) oversees OpenGL specifications, accepting or rejecting changes and proposing conformance tests.

In contrast to the old IRIS GL-library of SGI, OpenGL is by design, platform and operating system independent. It is aware of the network, so it is possible to separate the OpenGL application into a server and a client which actually renders the graphics. There is a protocol to move through the network OpenGL commands between server and client. Thanks to its OS independence, server and client do not have to run on the same type of platform. Quite commonly the server may be a supercomputer running a complex simulation and the client a simple workstation

mostly devoted to the graphical visualization. OpenGL allows developers to write applications that can be easily deployed across many platforms.

Above all OpenGL is a streamlined, high-performance graphics rendering library and there are many graphic accelerator cards and specialized 3D cards that implement OpenGL primitives at the hardware level. At first, these advanced graphic cards used to be very expensive and only available for SGI stations and other UNIX workstations. Things changed rapidly and thanks to Silicon Graphics' generous licenses and driver development kit there are more and more OpenGL hardware for PC users.

To achieve OpenGL's hardware independence, commands for windowing tasks as well as commands for obtaining user input were excluded. This is not a serious drawback to using OpenGL, but it is possible to combine OpenGL with other flexible programming libraries that would handle windowing tasks and obtain user input. Furthermore, OpenGL does not provide any commands for describing complex models (molecules, airplanes, houses, birds, etc.). In OpenGL there are available only the most primitive geometric objects (points, lines, and polygons). The developer has to construct his/her own models based on these few simple primitives. There are OpenGL-related libraries that provide more complex models, and any user can use these libraries to build their own.

JOGL is the Sun supported set of Java class bindings for OpenGL. It supports integration with the Java platform's AWT and Swing widget sets while providing a minimal and easy-to-use API that handles many of the issues associated with building multithreaded OpenGL applications. Jogl provides access to the latest OpenGL routines as well as platform-independent access to hardware-accelerated offscreen rendering ("pbuffers"). Jogl also provides some of the most popular features introduced by other Java bindings for OpenGL like GL4Java, LWJGL and Magician, including a composable pipeline model which can provide faster debugging for Java-based OpenGL applications than the analogous C program.

Jogl was designed for the most recent version of the Java platform and for this reason supports only J2SE 1.4 and later. It also only supports truecolor (15 bits per pixel and higher) rendering; it does not support color-indexed modes. Certain areas of the public APIs are more restrictive than in other bindings; for example, the

GLCanvas and GLJPanel classes are final, unlike in GL4Java, and the GLContext class is no longer exposed in the public API. These changes have been made to keep the public API simple and because most of the programming errors that have been seen with earlier Java/OpenGL interfaces, in particular GL4Java, have been related to subclassing the OpenGL widget classes and performing manual OpenGL context management. Several complex and leading-edge OpenGL demonstrations have been successfully ported from C/C++ to Jogl without needing direct access to any of these APIs. However, all of these classes and concepts are accessible at the Java programming language level in implementation packages, and in fact the Jogl binding is itself written almost completely in the Java programming language. There are only about fifty lines of handwritten C code in the entire Jogl source base; the rest of the native code is autogenerated during the build process by a new tool called GlueGen, the source code of which is in the Jogl source tree.

Some of GlueGen properties are:

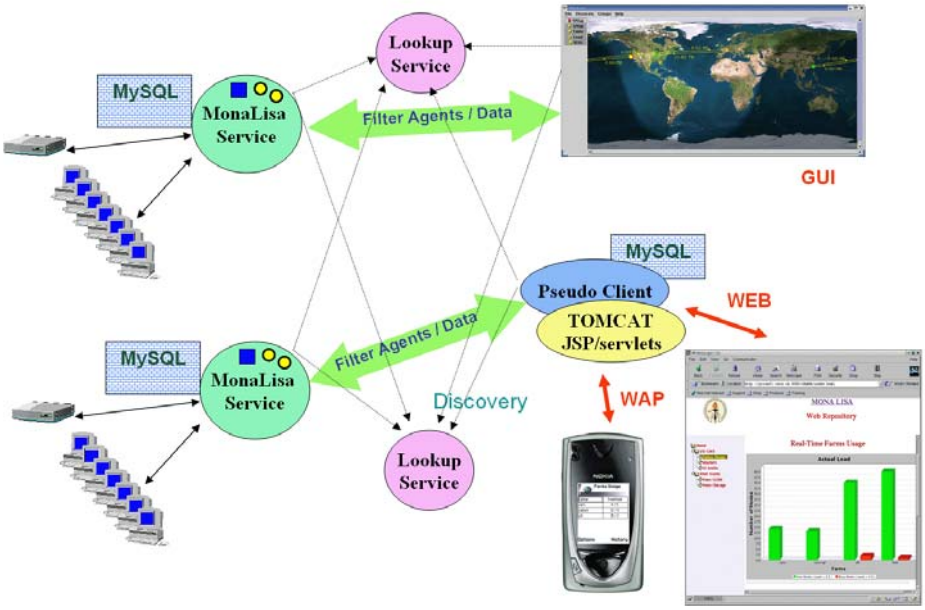
- Parses C header files using ANTLR
- Generates intermediate representation expressing primitive types, function prototypes, structs, unions and function pointers
- Autogenerates Java programming language and JNI code
- Powerful enough to bind AWT Native Interface back into Java programming language
- Enabled Jogl to be written in Java programming language instead of C
- Open source

5. MonALISA

An essential part of managing a global Data Grid is a monitoring system that is able to monitor and track the many site facilities, networks, and the many task in progress, in real time. The monitoring information gathered also is essential for developing the required higher level services, and components of the Grid system that provide decision support, and eventually some degree of automated decisions, to help maintain and optimize workflow through the Grid. The agent-based MonALISA

(Monitoring Agents in A Large Integrated Services Architecture) system, was developed based on the DDSA[1] framework.

A service in the DDSA framework is a component that interacts autonomously with other services through dynamic proxies or agents that use self-describing protocols. By using dedicated lookup services, a distributed services registry, and the discovery and notification mechanisms, the services are able to access each other seamlessly. The use of dynamic remote event subscription allows a service to register to be notified of a selected set of event types, even if there is no provider to do the notification at registration time. The lookup discovery service will then automatically notify all the subscribed services, when a new service, or a new service attribute, becomes available.



The code mobility paradigm (mobile agents or dynamic proxies) used in the DDSA extends the remote procedure call and the client server approach. Both the code and the appropriate parameters are downloaded dynamically into the system. Several advantages of this paradigm are: optimized asynchronous communication and disconnected operation, remote interaction and adaptability, dynamic parallel execution and autonomous mobility. The combination of the DDSA service features and code mobility makes it possible build an extensible hierarchy of services capable of managing very large Grids, with relatively little program code.

[1] Dynamic Distributed Services Architecture

MonALISA was built as a prototype implementation of the DDSA based on JINI technology. The JINI architecture federates groups of devices and software components into a single, dynamic distributed system; functionality that the future Open Grid Services Architecture (OGSA) (http://www.globus.org/) will need to include. JINI enables services to find each other on a network and allows these services to participate and cooperate within certain types of operations, while interacting autonomously with clients or other services (The Openwings Project, http://www.openwings.org/). This architecture simplifies the construction, operation and administration of complex systems by:

1. allowing registered services to interact in a dynamic and robust (multithreaded) way;
2. allowing the system to adapt when devices or services are added or removed, with no user intervention;
3. providing mechanisms for services to register and describe themselves, so that services can intercommunicate and use other services without prior knowledge of the services' detailed implementation.

MonALISAalso includes WSDL/SOAP (World Wide Web Consortium, http://www.w3.org, The Glue Web Services Pacakage http://www.themindelectric.com/) bindings for all the distributed objects, in order to provide access to the monitoring information from other types of clients and to facilitate a possible future migration to the Open Grid Services Architecture.

5.1. The Monitoring Service

MonALISA is an ensemble of autonomous multi-threaded, self-describing agent-based subsystems which are registered as dynamic services and are able to collaborate and cooperate in performing a wide range of monitoring tasks in large scale distributed applications, and to be discovered and used by other services or clients that require such information.

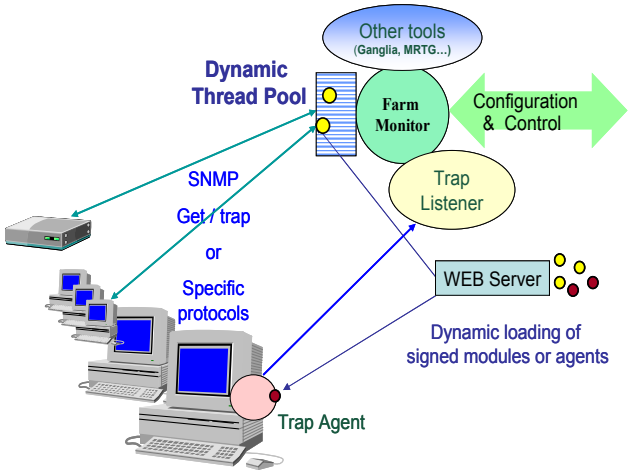
MonALISA is designed to easily integrate existing monitoring tools and procedures and to provide this information in a dynamic, self describing way to any other services or clients. MonALISA services are organized in groups and this attribute is used for registration and discovery.

5.1.1. The data collection engine

The system monitors and tracks site computing farms and network links, routers and switches using SNMP (The Net-Snmp Web Page, http://www.net-snmp.org/), and it dynamically loads modules that make it capable of interfacing existing monitoring applications and tools:

- Ganglia Monitoring tool, http://ganglia.sourceforge.net/;
- MRTG monitoring tool. http://www.mrtg.org;
- Hawkeye monitoring tool, http://www.cs.wisc.edu/condor/hawkeye/.

The core of the monitoring service is based on a multi-threaded system used to perform the many data collection tasks in parallel, independently. The modules used for collecting different sets of information, or interfacing with other monitoring tools, are dynamically loaded and executed in independent threads. In order to reduce the load on systems running MonALISA, a dynamic pool of threads is created once, and the threads are then reused when a task assigned to a thread is completed. This allows one to run concurrently and independently a large number of monitoring modules, and to dynamically adapt to the load and the response time of the components in the system. If a monitoring task fails or hangs due to I/O errors, the other tasks are not delayed or disrupted, since they are executing in other, independent threads. A dedicated control thread is used to stop properly the threads in case of I/O errors, and to reschedule those tasks that have not been successfully completed. A priority queue is used for the tasks that need to be performed periodically. A schematic view of this mechanism of collecting data is shown in the next figure:



This approach makes it relatively easy to monitor a large number of heterogeneous nodes with different response times, and at the same time to handle monitored units which are down or not responding, without affecting the other

measurements. The number of threads necessary to monitor a complete site is dynamically adjusted, and very dependent on the response time for each node, which is related to its load as well as to the quality of the network connections.

5.1.2. Data Storage

The collected values are stored in a relational database, locally for each service. The JDBC framework in JAVA offers the flexibility to dynamically load any driver and connect to virtually any relational database. A normalized scheme is used to store the result objects provided by the monitoring modules in indexed tables, which are themselves generated as needed, dynamically. As data is becoming older, the values stored in the database are compressed by evaluating the mean values on larger time intervals and at the same time keeping the fluctuation range for each parameter.

5.1.3. Registration and discovery

Each MonALISA service registers with a set of JINI Lookup Discovery Services (LUS) as part of a group and having a set of attributes. The LUSs are also JINI services and each one may be registered with the other LUSs. If two LUSs have common groups any information related with a change of state detected for a service in the common group by one is replicated to the other one. In this way it is possible to build a distributed and reliable network for service registration and this mechanism allows dynamically adding or removing LUSs from the system. Any service should also provide for registration the code base for the proxies that other services or clients need to instantiate for using it. This mechanism is used to make sure that the right proxies are used for each service while different versions may be used in a distributed organization at the same time. The registration is based on a lease mechanism responsible to verify periodically that each service is alive. In case a service fails to renew its lease it is removed from the LUSs and a notification is sent to all the services or clients that subscribed for such events.

The monitored client services at a given site being monitored by MonALISA use the Lookup Discovery Services to find all the active MonALISA services running as part of one or several group “communities”. Selection based on a set of matching attributes is also possible. The discovery mechanism is used for notification when

new services are started or when services are no longer available. The communication between interested services or clients is based on a remote event notification mechanism which also supports subscription.

The client application connects directly with each service for receiving monitoring information. To perform this operation it first downloads the proxies for the services it needs, from a list of possible URL specified as an attribute of each service. This procedure allows each service to correctly interact with other services.

5.1.4. Predicates, Filters and Alarm Agents

Clients can get any real-time or historical data by using a predicate mechanism for selecting or subscribing to selected measured values. These predicates are based on regular expressions to match the attribute description of the measured values a client is interested in. They may also be used to impose additional conditions or constrains for the interested values. For the historical data the predicates are used to generate SQL queries. The subscription requests will create a dedicated thread, to serve each client. This thread will perform the matching test for the client predicates with the measured values in the data flow and is responsible to send them to the client as compressed serialized objects. Having an independent thread per client allows sending the information they need, fast, in a reliable way and it is not affected by communication errors which may occur with other clients. In case of communication problems these threads will try to reestablish the connection or to clean-up the subscriptions for a client or service which is not anymore active.

Monitoring data requests with the predicate mechanism is also possible using the WSDL/SOAP binding from clients or services written in other languages. The class description for predicates and the methods to be used are described in WSDL and any client can create dynamically and instantiate the objects it needs for communication. Currently the Web Services binding does not provide the functionality to register as a listener.

Application or clients may also use Agent Filters to receive the information they need. These agent filters are modules which can be dynamically deploy to a service and they perform a local data processing (by subscribing with a predicate to the data flow) and return back the processed information. As an example, such filters are used to compute the aggregate IO traffic in a farm, or to provide the number of

nodes which are free. The same thread used for handling the predicate subscription is used for sending the filtered results back to each client.

Dynamically loadable alarm agents, and agents able to take actions when abnormal behavior is detected, are currently being developed to help with managing and improving the working efficiency of the facilities, and the overall Grid system being monitored.

5.2. MonALISA Clients

The MonALISA clients use the discovery mechanism to find all the active services from a list of user defined groups. There are two types of clients that we are going to present in the next sections.

5.2.1. Graphical Clients

The graphical client is developed as a Web Start (Java Web Start, http://java.sun.com/products/javawebstart/) application and it can be easily started and used from any browser.

Any MonALISA services can provide its own GUI to any client as a complex proxy marshaled as an attributed to the service. This GUI is used to communicate back with the service and plot the requested values. MonALISA provides flexible access to real-time or historical monitoring values, by using a predicate subscription mechanism or dynamically loadable filter agents. These mechanisms are used by any interested client to query and subscribe to only the information it needs, or to generate specific aggregate values in an appropriate format. When a client subscribes with a predicate to certain values the GUI will be automatically updated every time a new value matching the subscription is collected.

Graphical user interfaces allow users to visualize global parameters from multiple sites, as well as detailed tracking of parameters for any component in the entire system. The graphical clients also use the remote notification mechanism, and are able to dynamically show when new services are started, or when services become unavailable. Dedicated filers are used to provide global views with real time updates for all the running services.

5.2.2. Pseudo-Clients

A generic framework for building "pseudo-clients" for the MonALISA services was developed. This has been used for creating dedicated Web service repositories with selected information from the groups of MonALISA services. The "pseudo-clients" use the same LUSs to find all the active MonALISA services from a specified set of groups and subscribes to these services with a list of predicates and filters. These predicates or filters specify the information the pseudo-client wants to collect from all the services. It stores all the values received from the running services in a local MySQL database, and uses procedures written as Java threads to compress old data. A Tomcat (The Jakarta Project, http://jakarta.apache.org/) based servlet engine is used to provide a flexible way to present global data and to construct on the fly graphical charts for current and customized historical values, on demand. Dedicated servlets are used to generate Wireless Access Protocol (WAP) (WAP Forum, http://www.wapforum.org/) pages containing the same information for mobile phone users. Multiple Web Repositories can easily be created to globally describe the dynamic services running in a distributed environment.

5.3. Administration of Services

MonALISA also provides a secure mechanism (SSL with X.509 certificates) for dynamic configuration, using a dedicated GUI, of farms/network elements, and support for other higher level services that aim to manage a distributed set of facilities and/or optimize workflow.

It allows reconfiguring any monitoring services by adding new nodes, network elements or clusters and at the same time to dynamically loaded into the system any new monitoring module as needed. It also allows stopping or suspending any monitoring module. Adding dynamically new monitoring modules is important for debugging and understanding the way certain applications perform.

The Administration interface connects to a service using Remote Method Invocation over SSL. X.509 certificates for trusted administrators are imported in the keystore of each service and they are used to establish a SSL connection based on client authentication.

The administrative GUI can be stated automatically from the global web start client if started by an administrator. When the administrator loads is private key into

the global GUI client it automatically gets administrative rights on the services that imported his certificate in the trust keystore for services.

5.4. Automatic service updates

MonaALISA is currently deployed on many sites and maintaining and updating such applications may require a significant effort. For this reason it has been developed a mechanism in MonALISA which allows automatically updating the monitoring service. A dedicated thread is used to periodically check for updates of the distribution. Alternatively a remote event notification can be used to notify only selected services to perform an update. When such an event is detected, the service will trigger a restart operation.

When a MonALISA service is started, it is using the web start mechanism to describe an application and all its dependencies and constrains into a XML file (jnpl). This will perform an automatic download of all the packages which were updated and will check all the necessary constrains to run the application. All the files downloaded in this way must be digitally signed by a developer for which certificate is imported in the trust keystore. This can be done when the MonALISA service is used for the first time.

All the running services, as well as the services which may be started after an update was done will run the last "published" version and this is done in a secure way.

Users may start a MonALISA service with the auto update flag switch off.

6. Design

My development is related only to data visualisation on world map, although the MonALISA client contains other panels[2], but those do other kinds of processing on the received data. At the moment, there are two such panels that represent the world map as a background for data representation. The initial purpose of my

[2] from now on, the terms **panel** and **visualisation module** will be used interchangeable

development was only to replace the 3D panel that was realised using Java3D technology, because of the new alternatives for 3D graphics that appeard in the last years, but, as the other panel, called “Map Pan” was similar in construction with this one, I found a way to make a single panel, by reuniting the two, but without loosing any functionality of the two, in fact, adding some new one.

The next step was to reunite the two clients, and I’ve done that by using a plugin like system. This means that I sepparated the data rendering from the world rendering, objects that did not depended on received data. The object that is used to reder the data is specified in the main class of the client, so that means that my module doesn’t know how the data is going to be represented, only what it should call. For this reason I compare this system with a plugin system. But more about that later on.

The module I developed can be though as having two parts:

- one that is independent of MonALISA and is related to optimizations and visual improvements.
- and one that is related to data visualisation and the new models I’ve implemented.

6.1. Performance improvements

As I stated above, the problem the current client faced was the lack of high resolution texture for zooming to small areas of the world map. The solution was to provide a higher resolution map of the world at the cost of high memory consumption. This solution is good for important presentations as it gives a pleasant to the application, but the ordinary user cannot enjoy it, only from pictures. So, a new method had to be found.

The thing that the two panels have in common is the world map, that is viewable as a plane image for the “Map Pan” module, and wrapped on a sphere for “Globe Pan”. The important thing is that both panels support a zooming function that allows to get closer to the map, and so, study localized data, and also see a smaller piece of the map. But, as the viewable area of the map becames smaller, but the window’s width remain the same, the viewable pice of image will be streched to cover the full window, and, by doing so, each pixel on this image will be repeated several times on x and y axis.

During the development of the project I had the occasion and the interest to study several terrain rendering algorithms for a future implementation in MonALISA client. One of these algorithms is ROAM (Real-time Optimally Adapting Meshes), and the paper about it can be found at http://www.llnl.gov/graphics/ROAM/ .This algorithm uses a ierarhical structure, a tree, to improve the visualisation of the terrain closer to the user, by constructing child nodes[3] and so increase the detail level for the zone the parent node is responsible for.

From this study I came up with the ideea of dividing the image that was loaded on the map in smaller pieces and then, for the area in the field of view, load the smaller images at the correct resolution. This means that the client applications has to have available several sets of smaller images at different resolutions.

The difference between a 1024x512 resolution image and a 2048x1024 resolution image is that, when both images are scalled at the dimensions of the second image (that means the width and height of the first image are each multiplied by a factor of 2), one pixel from the first image, on any to two pixels on the second image. This is called that the second image is a better resolution image. What I want to do is, when the client sees the pixels of the current image too big, that meaning that, a pixel of image occupies more than one pixel on the screen, I change that image with a number of smaller images at a higher resolution. For the example above, if the current slice[4] from the map has a resolution of 1024x512, I will changed it with four slices of 2048x1024 resolution, so that the number of pixels on x and y axis remain the same, but the detail level is increased.

Because an image valours as 1000 words bellow is the visual effect:

[3] From now on, the term **node** will be related to an element of the texture tree that contains an piece of the image

[4] Further on I make the convention that **slice** represents a piece of the entire image

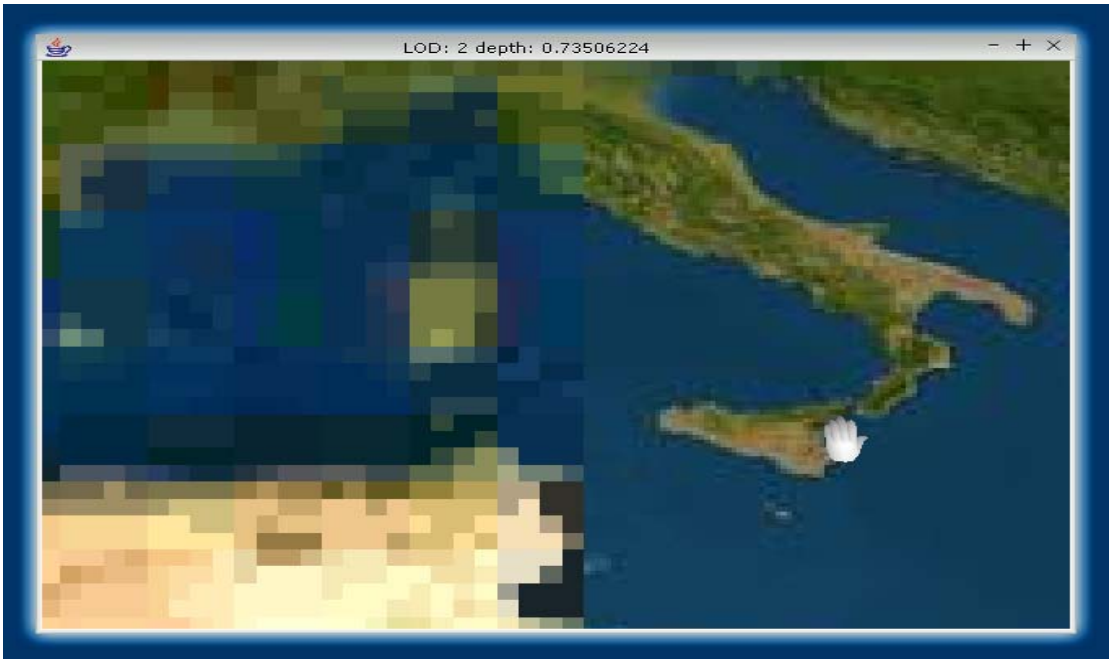


fig.: slices at different resolutions: on the left low resolution, on the right high resolution

The method I used in obtaining the smaller images is presented bellow.

6.1.1 SlicePic application

Small application used to generate a set of smaller images from a bigger one.

I choose to present it here because the structure, the organisation of the image files is important for the module and this application is creating it.

To be able to find the correct higher resolution slices for a given slice, I structured, as adviced by one of my leading professors, the slices on a tree like structure, starting from a base directory, called map0. Inside it there are image files, and, if an image file has a corresponding directory (with the same name as the file), then it may have inside slices at a higher resolution. The recurent process can continue for as long as images are available.

As input there is an image, a world map image, starting at -180 degrees west longitude and ending at 180 degrees east longitude on x axis, and starting at 90 degrees north latitude and ending at -90 degrees south latitude on y axis. The image can have any type Java can load (GIF, JPEG or PNG). Because of the ierarhical structure, the user must specify to which level of detail the image corresponds, and then supply the characteristics of each level (the number of slices the picture is divided in on x, and, respectively, on y axis, for each level).

The application loads the image file, generates the required directory structure, and then saves the slices into a special file format, described in implementation section.

6.1.2 Texture algorithm

This is the one responsible for showing the image at the right resolution.

fig: as the area into view becomes smaller, the level of detail increases

It does that by constructing a tree, each node in the tree represents a slice. All the children of a node put together cover the same area as the node, as they are higher resolution replacements for specified zones in the slice. That means that a node can have a variable number of children, from zero to maximal number of slices on x multiplied by maximal number of slices on y for the current level, and, for the missing children, it is its duty to put a piece of its image into the area covered by the missing children.

An example of the things I explained above is given in the next pictures:

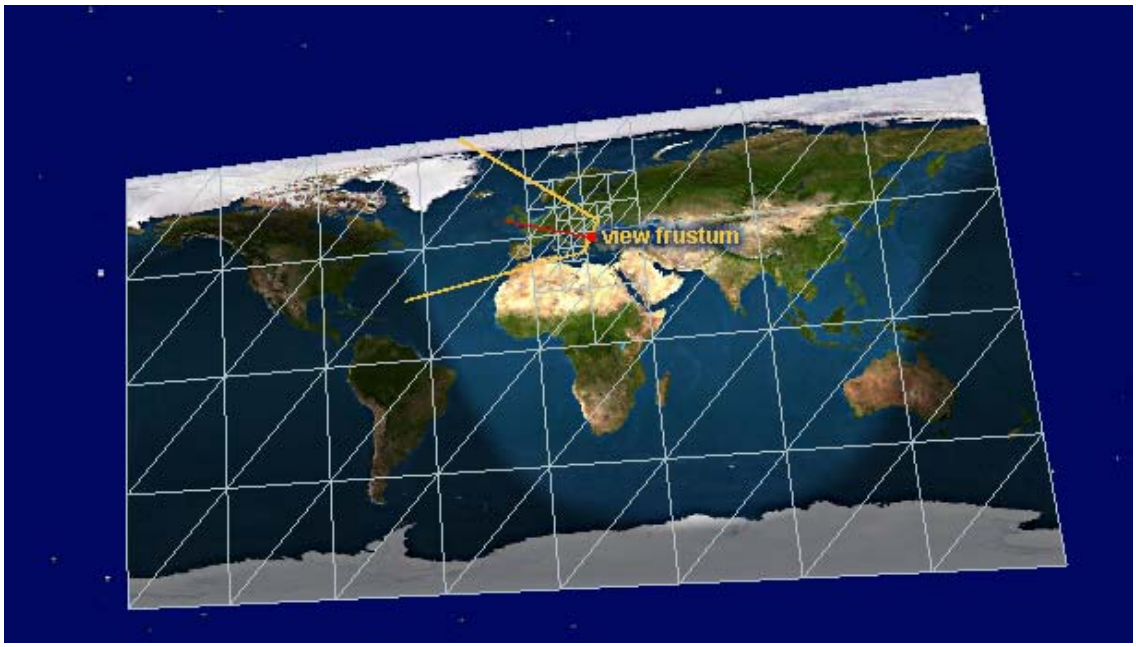


fig: world image is cut into smaller pieces, and them are also cut into even smaller pieces

There were several problems I encountered implementing the algorithm, but more about them later on.

The class Texture is responsible for memorizing the tree structure of the images. It is responsible for several actions:

- generating child nodes and setting the right texture to load
- loading the texture[5] from file
- setting the loaded texture to the corresponding node
- draw the node
- unload textures that are not use, because they are not in the view frustum or because the current level of detail does not require them.

Some of the actions seems strage to be put separatelly but it is so because they are run from separate threads, as I'll show in the implementation section.

Because JoGL does not permit more than one thread to access the rendering object (instance of GL class, accessible as a final variable from the GLDrawable class), the operations related to OpenGL data manipulation are done by the drawing thread.

Therefore, the actions done by the drawing function (and thereby the drawing thread), are, in the order they are executed:

1. check for new loaded from file textures available to set into OpenGL memory.

[5] texture is an image loaded into OpenGL memory, so it is used interchageably with image

- try to remove an element from the texturesToSet list;
- if one is available, generate OpenGL texture from the byte array read from the file;
- set the new texture for the node;
- check to see if the parent texture may be removed based on the fact that all child nodes have their own texture.

2. check for available textures to unload from OpenGL memory. This is done by removing the elements stored in texturesToUnSet list, and call a OpenGL specialized function for deleting a texture.
3. draw the tree of textures. The drawing process is started from the root element, that represents the hole world map, but has no asociated texture.

Before describing the algorithm for drawing, I must explain the mechanism used by OpenGL to render an image from a file in 3D space. First the image must be loaded from the file, with an utility that recognizes the file type and can obtain an array of octets grouped in sets of 3, to form a color, as each octet represents a nuance of the color: red, green, blue. The length of the array is equal with the width of the image multiplied by the height and by 3, and represents the colors of each pixel in the picture. An OpenGL color can be coded on 3 of 4 octets, but the fourth represents an alpha value, meaning the intensity of the color, and was of no interest to me. The second step to render the image is to allocate an OpenGL texture id for the new texture, by calling glGenTextures, function that tells OpenGL to asociate an integer with a structure representing the texture and its properties. Third step is to copy/move (depending on the implementation) the array of octets to OpenGL memory using glTexImage2D function.

OpenGL's memory space is made of dynamic memory (in RAM), and available video card's memory, for 3D video cards. To increase the rendering speed, OpenGL, uses much of the card's video memory to store heavely used textures so, at some moment, a texture can be in system memory or video memory.

The drawing algorithm starts at the root node's children and tries to draw each one's texture. A node will be rendered (its texture drawn on screen), if it passes a visibility test. This test checks to see if a texture is visible based on its four corners world coordinates being in the visible frustum, or positioned in such a way that the view frustrum's intersection with the texture is not empty.

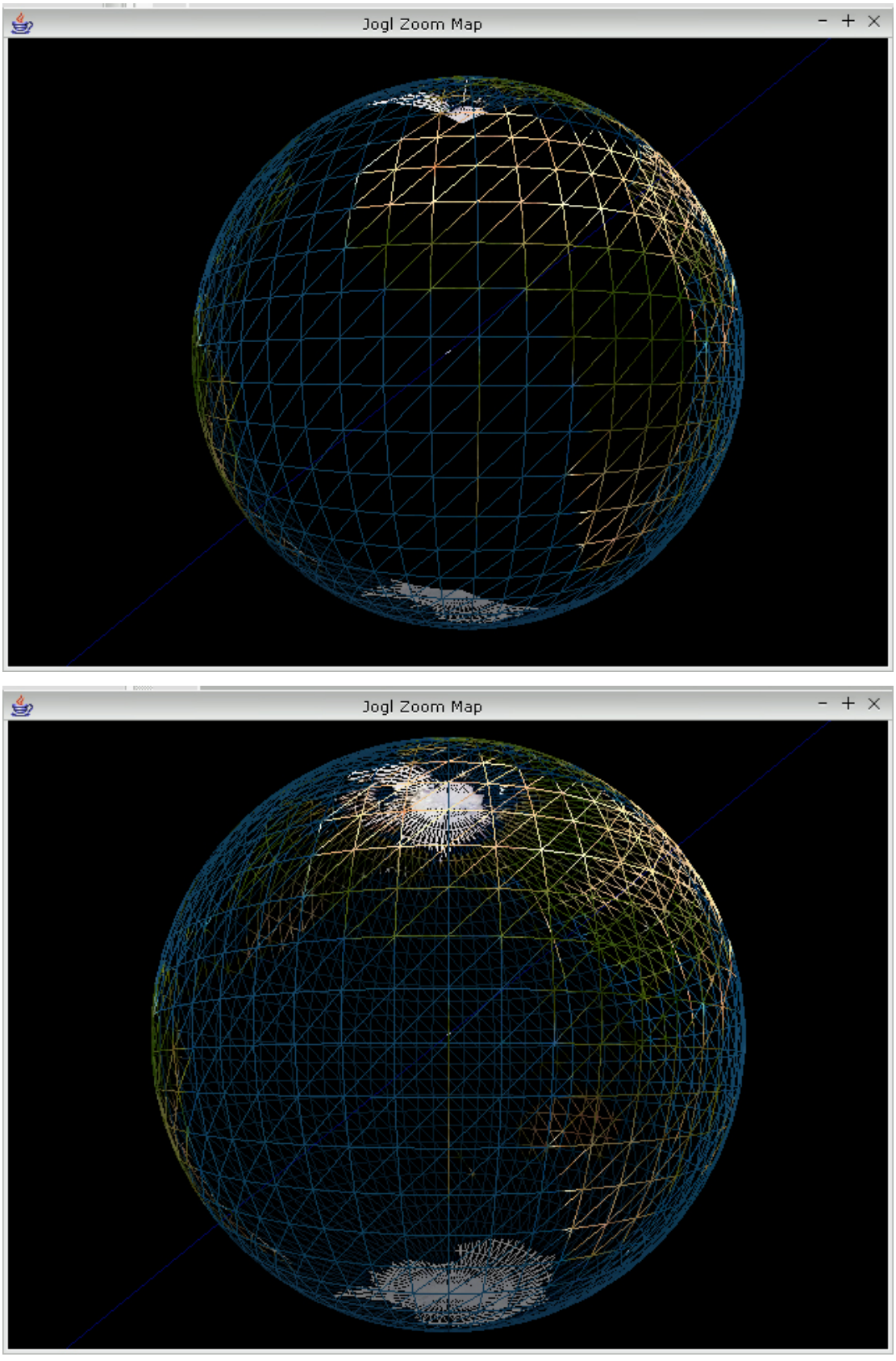


Fig: visibility test in action, top picture shows visibility test on, for bottom one is off

As it can be observed comparing the two pictures, the back of the sphere in the first image is not fully visible, only some small portions, around the margins of the sphere, whereas, in the second image the front and the back are clearly visible. To further more explain the pictures, the map is drawn using only the lines that unite the corners of the textures, not the full images.

If a node passes the visibility test, the next check is to see if it has children. If it does, set each existing child to self draw, for the ones that are missing, draw its piece of the texture coresponding to the area that should be covered by the child. If the node has no children, then it can draw itself on the world.

When the visible area of the map changes, as a response to user input (translation, rotation or zooming), and after a small period of time has passed, to confirm that the user is satisfied with the new position (the time to wait is called IDLE_TIME), a thread is assigned to check if new textures should be loaded.

The mechanism used is done by traversing the tree, to see each node if it's at correct level of detail. The level of detail is computed as the height in the tree the current node has. This is the current LOD (level of detail) and this is compared with the desired LOD, that is computed based on the distance from the texture to the eye.

Depending on the result of the comparation, there are several decisions that are made:

- if current LOD is greater than the desired one, then the level of detail for this node must be decreased, by deleting it and its children, and, by doing that, reverting to its parent LOD, which is lower that the current one, because of the way the level was considered (as the height of the node in the tree);
- if current LOD is lower than the desired one, it means LOD must be increased for this texture, and this is done by creating the child nodes that are in the visible area, and setting higher resolution images for them. Before trying to create the child nodes, a checking is done to see if image files are available for the next level of detail, by searching for the coresponding directory of the curent node's image file. If the directory exists, visible children are created and their textures are set for loading from file, and their current texture id is set to parent texture id, to be able to draw themselves at any moment. The LOD algorithm is repeated for each new created child, and, so, the child could create, on his turn, other childen, or can decide that it should be deleted, in this case setting itself to an invalid state.

In the case that no children could be set, or because there are no available files any more, or the children are invalid, the node must set its texture for loading, if it is not already set.

- if current LOD is equal to desired LOD, then, if the node has children, delete them, because it does not need a more detailed image, and, if it doesn't have already its texture set, or set for load, set it for loading. The difference between set and set for load is that set means the texture has already been loaded into OpenGL memory and is in use by the node, whereas "set for load" means that the node uses its parent texture and now should have its own texture.

As soon as a texture is set for loading, there is another thread that is awaken, if it was sleeping, that will do the actual loading from file of the images and providing them into a form that OpenGL can use. This is the third thread, and it runs untill the texturesToLoad list is empty. From that moment waits to be notified that new textures should be loaded. The notification comes from the thread that responds to user interaction, the second one as described above.

The loading process follows the next steps: an object is accuired from the textureToLoad list. A node is identified from object's properties, and is checked to see if it's texture still should be loaded from file (a situation can appear when, the LOD changed fast from lower LOD to higher LOD and the other way around, so, by the time this thread decided to load the file, the node who's image is set for loading, is no longer available and is marked as DEREFERENCED). Assuming the node is valid, its coresponding file name is computed and loaded, shadow is applied over the image, and the initial object, with this updates, is introduced into the texturesToSet list. The object is removed from textureToLoad list, as it has been introduced into the other list.

After all new textures are loaded, a refresh is called for the window, so that the new textures become visible.

Because Texture class has functions that are called from three different threads, the access to its textures tree is synchronized, because each thread can modify the data, and each node has a property that indicates its status.

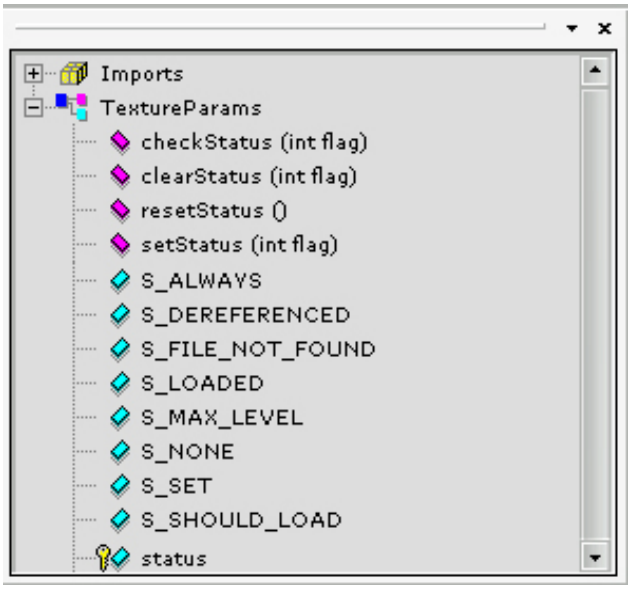


fig: status property, its getters and setters and the available flags

The status of a texture evolves as follows:

Depending on the result of the comparation, there are several decisions that are made:

- at creation time it becames S_NONE,
- if this texture is on the first level of the map, it should never be unloaded, so that there could always be a texture ready to be drawn, so the flag S_ALWAYS is set,
- if LOD changed and a node has to load its texture from file, it's status is set to S_SHOULD_LOAD,
- but if a node should increase LOD, and it has no asociated directory with higher resolution images, it's status is flagged to S_FILE_NOT_FOUND,
- if a node became invalid, but there still are some references to it, it's status is S_DEREFERENCED,
- if image was loaded from file, node's status becames S_LOADED
- if file was not found, status is set to S_FILE_NOT_FOUND,
- if image is transferred into OpenGL's memory, texture's status becames S_SET; this is the final and desired status for a texture.

6.2. Visualisation extensions

This section of the paper refers to the way the data gathered from farms or vrvs nodes is represented on client application.

As I mentioned before, the challenge I faced was to improve the current visualisation module. I did this by allowing a higher resolution of the world map and by reuniting the two modules: “Globe Pan” and “Map Pan”. The resulting panel is called “3D Map” and is realized using JoGL technology.

6.2.1. 3D Map Panel

This panel can show world map as a plane projection or as a sphere projection.

This image shown bellow presents the initial form of the panel, as a plane projection:

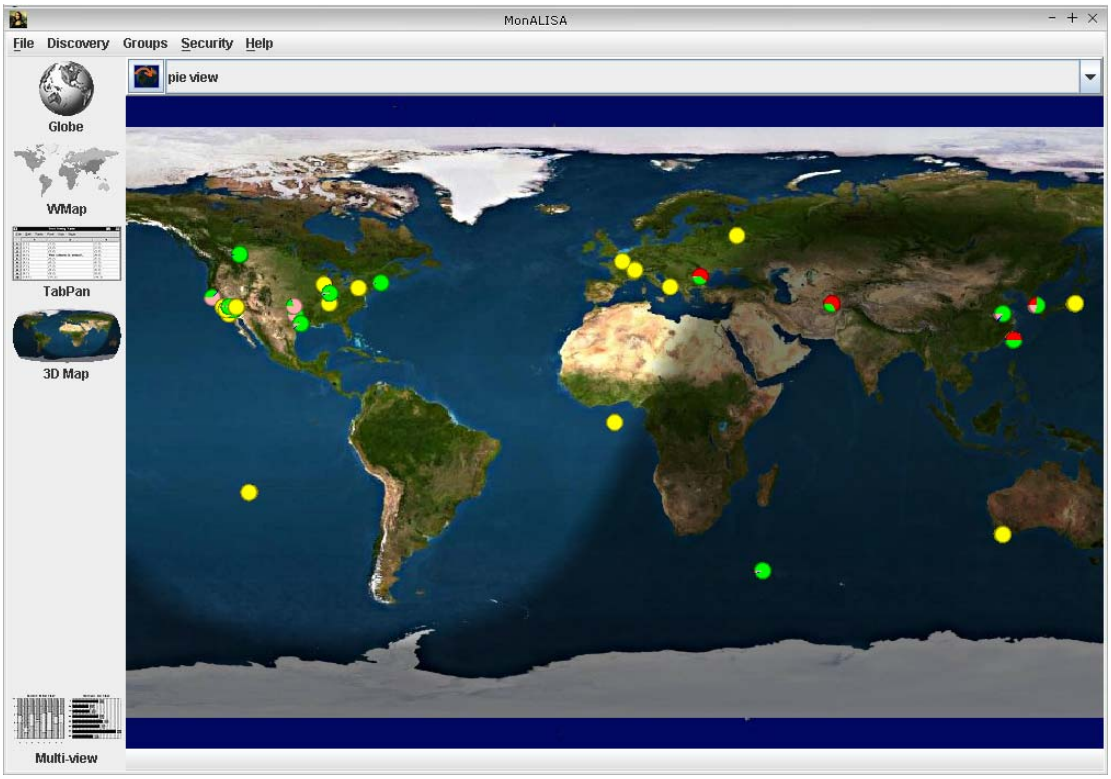


fig: world map as plane projection in 3D Map panel

When the visualisation mode is changed, using the button available on the upper toolbar, the map is projected on a sphere, and it looks like in the next image:

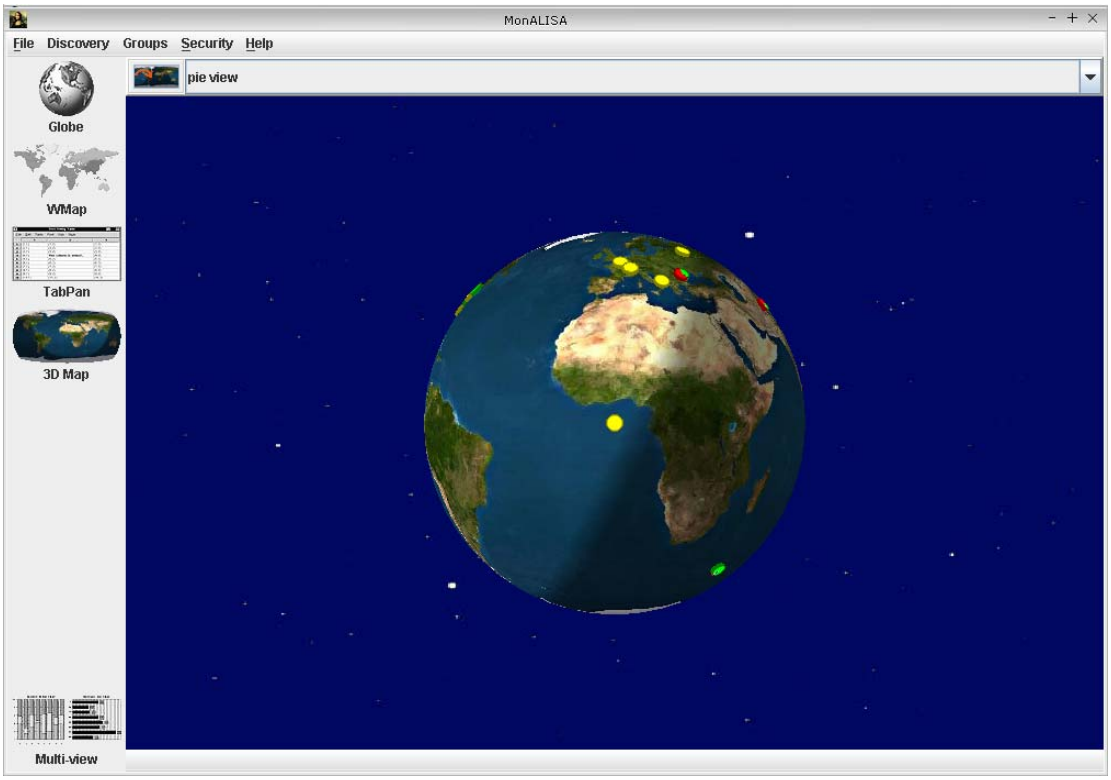


fig: world map as sphere projection in 3D Map panel

The transformation between this two projection is visible to the client and that creates a pleasant 3D effect to the eye. During the transformation all 3D objects present on the scene are updated and repositioned so that the shape trasition be smooth, and phisical correct. The next set of images shows this process, starting from plane and ending to sphere in a series of 7 steps, that capture different moments of the transition to better emphasise the action. The reverse process is also possible, and is executed in inverted order of the steps presented in the image.

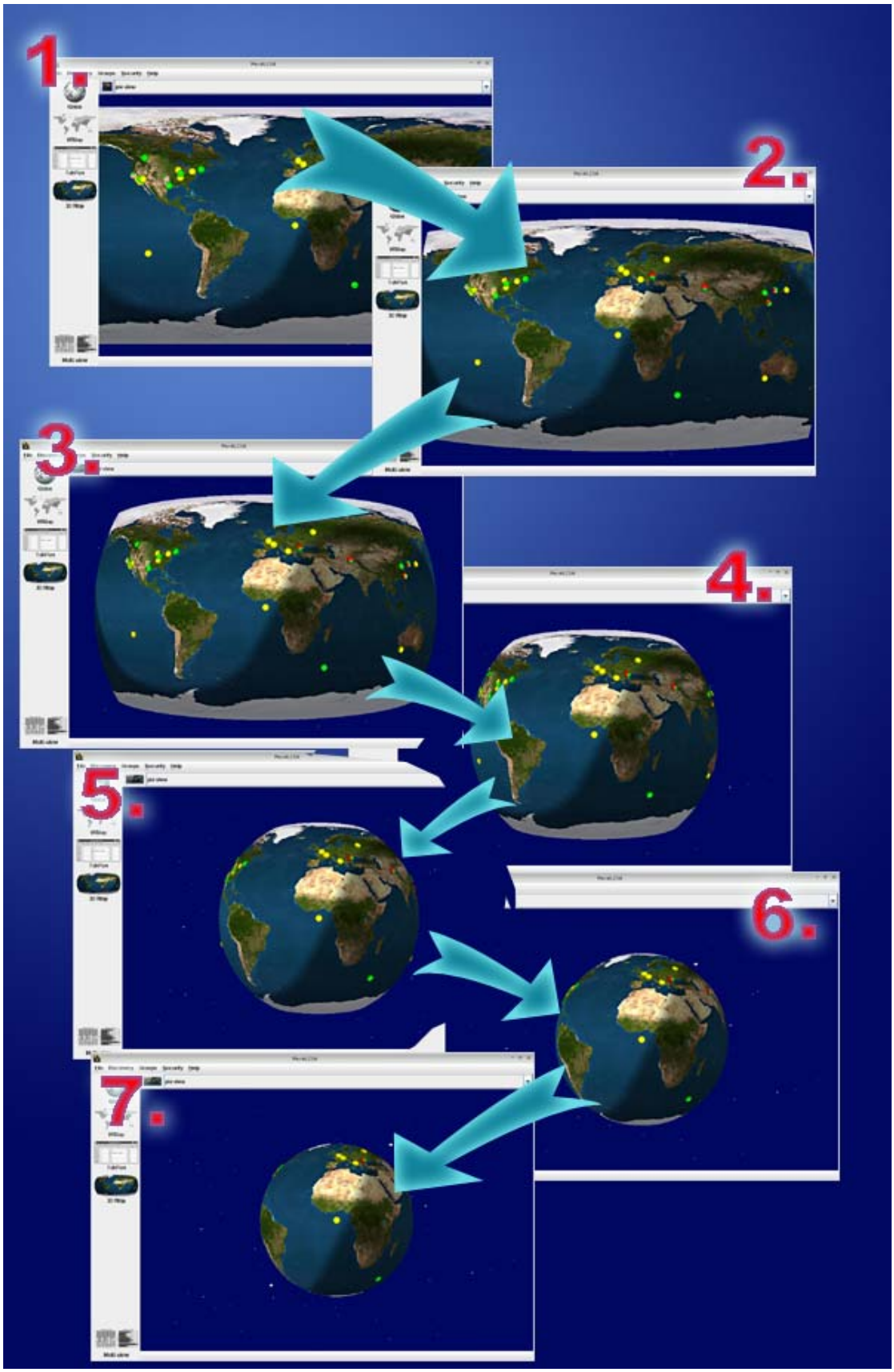


fig: transformation of 3D Map from plane projection to sphere projection

This transformation is mathematically possible by using a small trick: that a plane is actually a sphere but with an infinite radius. By doing so, this process actually means gradually to decrease the radius and recompute map position based on the new radius. The measurement used to change the radius value is the angle between the center and one extreme on x axis of the map. The formula for obtaining the radius from this angle is:

Radius = MAP_WIDTH * 45 / PI / angle,

because the map folds on the sphere, but covers only an amount, for ecuatorial circle, it represents an arc, who's length is angleR * Radius, but, from construction of angle, angleR is twice the angle, and arc's length is half the map width. This explanation is also presented in a graphical form:

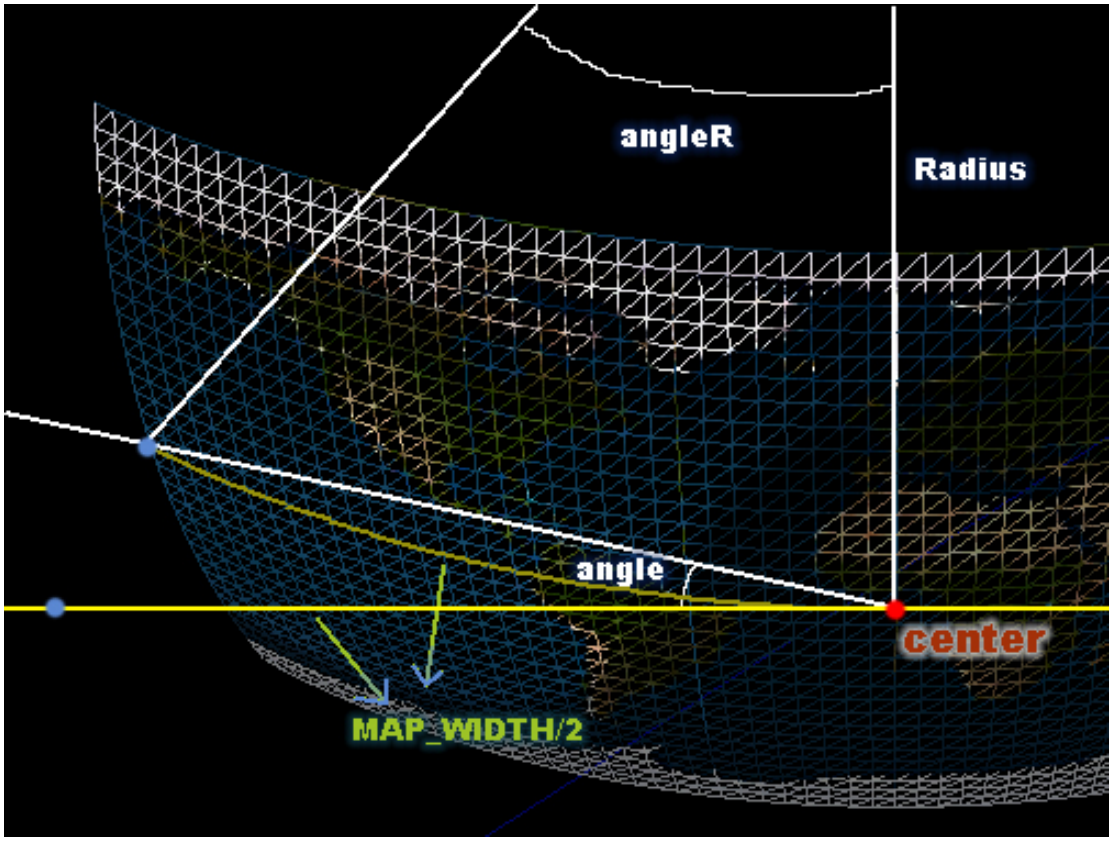


fig: all elements neccesary to compute radius based on angle

6.2.2. Grids

The world map is broken into pieces. Each such piece has four corners that are memorized into a grid, by their indices. A texture expands on one or more pieces on x axis and y axis, depending on its position in the tree. The grid is a static 2d grid, meaning that is allocated at program start up, as an matrix with fixed width and

height, containing 3d points. The texture is described as having a starting index in the grid, for the top-left corner, and thus, having a corresponding 3d point, and an width and height, both of them measured in units, not distances, so that the other corners are computed as value of the grid on index plus width, or height, or both, as the case requests it, for the top-right, bottom-left and bottom-right corners. This grid is allocated at the beginning because it is used on sphere projection, to give the shape a smooth aspect. The optimal number of points to create the sphere is 32 on up-down axis, and 64 for left-right axis, as the angle is 360 degrees.

As the texturing algorithm supports unlimited levels of detail, the grid soon becomes insufficient for more than 4 levels, so that, for a node that has its width or height (in units) equal to one, a dynamic grid is allocated as a matrix of 3d points. Its dimensions are given by the number of child nodes to be created on x and y axes. The values inside the grid are computed based on the four corners values. This is done by creating a equation for each axis (x, y, z) using the four corners values and indexes i and j in the grid, so that,

- for i=0 and j=0 value of 3d point is upper left corner,
- for i=nx, j=0 value of 3d point is upper right corner,
- for i=0, j=ny value of 3d point is lower left corner,
- for i=nx, j=ny value of 3d point is lower right corner.

The equation is, for each axis:

lt*(nx-i)/nx*(ny-j)/ny + rt*i/nx*(ny-j)/ny + lb*(nx-i)/nx*j/ny + rb*i/nx*j/ny,

where lt is left-top component on the current axis, same for rt (right-top), lb (left-bottom), rb (right-bottom), nx is the number of points on x (number of textures + 1 because a texture is between 2 points), ny is on y.

The result is an interpolation between the starting and ending points of the parent texture, but this does not affect the map drawing as all the generated points are inside one of the 2 triangles generated by the four corners, and points on matrix borders are also on a segment formed by two adjacent corners.

An example of how such a grid looks like is given in the picture below:

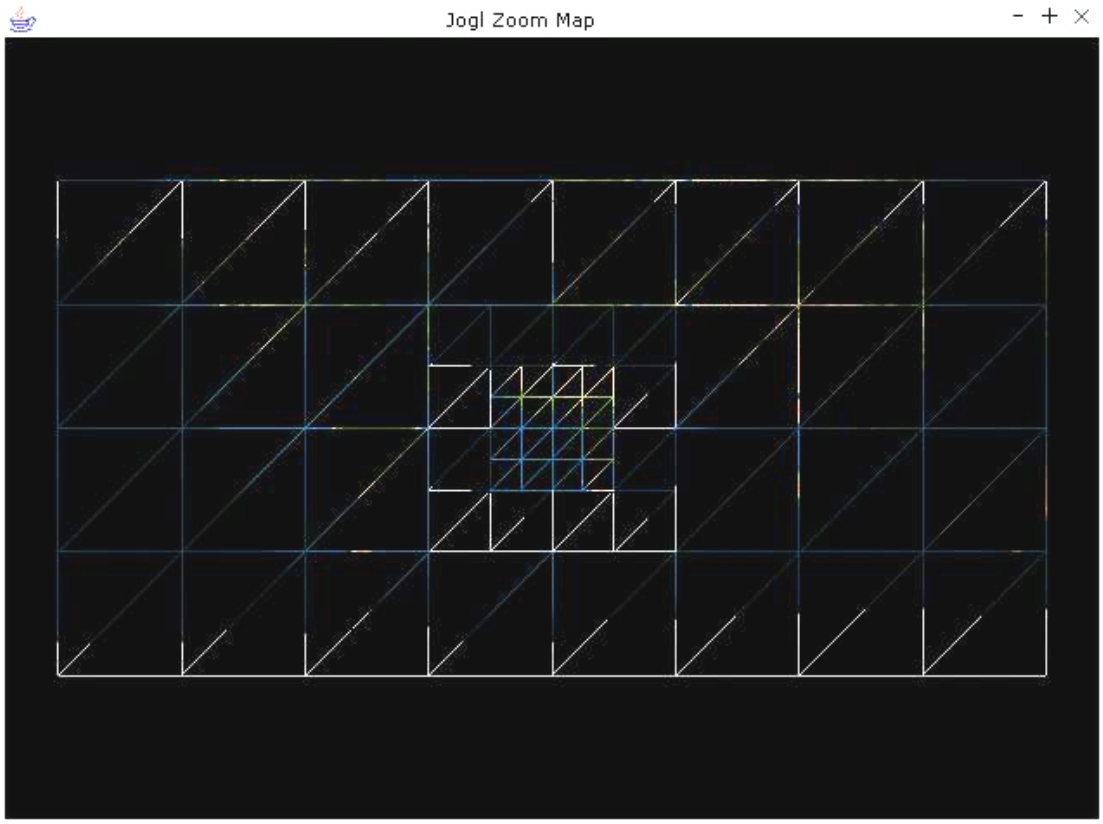


fig: 3 levels of detail visible that are using the static grid

6.2.3 Eye

The decision of changing the level of detail is based on a depth value from eye position to the texture. In this section I'll describe how this value can be computed.

For a node, the aproximative center's position is computed as average values on x, y, z of the corners positions, and then, the depth, by definition, is considered as the projection of vector starting from eye to center of texture, on vector eye direction. This formula has been chosen for its propertie on plane projection: when eye direction is perpendicular on map, every texture has the same depth, and, consequently, same LOD. When eye direction is at an angle to plane, a nicer effect appear, as closer textures have greater LOD than distant ones. This is also visible for sphere projection.

But depth is not sufficient for computing the correct LOD, as it depens also on application's window dimensions. For that, an array with standard computed depth values must exist that is compared with the current one.

The formula for standard depths is:

D = width / 2 / tan(FOV_ANGLE) * MAP_WIDTH / Xpt_max,

where

- width is window dimension from left to right in pixels,
- FOV_ANGLE is the view frustum angle
- MAP_WIDTH is the width of the world map in 3d space
- Xpt_max is a constant equal to 2 and means that the change of resolution will be when the dimension of a texture pixel is greater or equal to 2 (screen pixels).

D is then used to compute the depths with the interation:

for each i=0 .. depthZ.length – 1 do

depthZ[i] = D/ resolutionX[i+1]

Explanatory image for how depth algorithm functions:

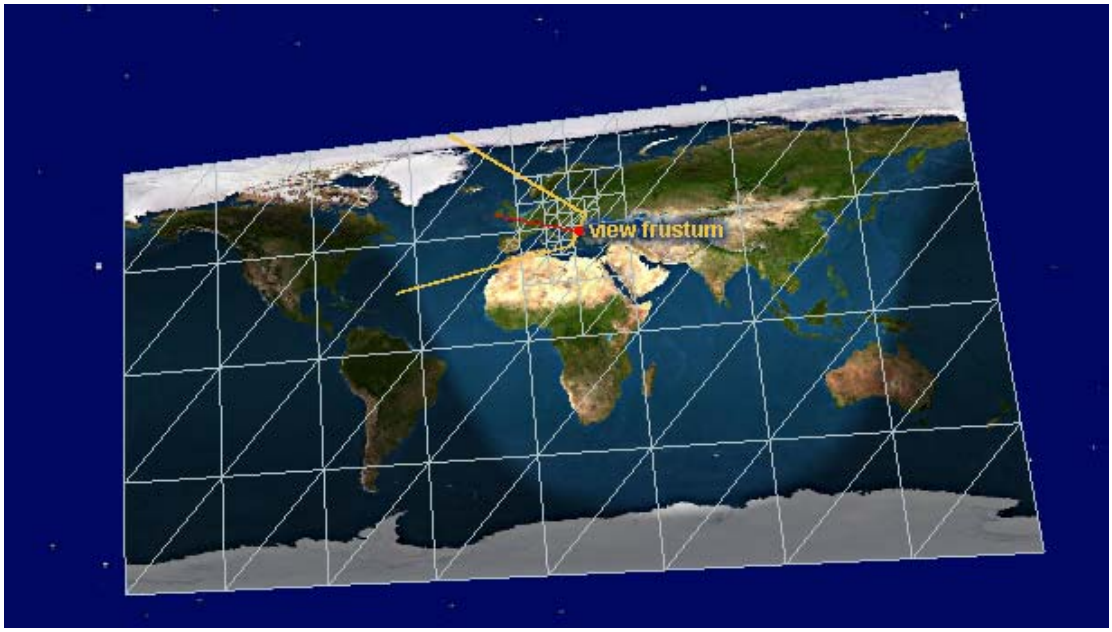


fig: eye position and the LOD generated

6.2.4 Data representation – plugins system

Because the 3D Map module is a replacement/complementary module for the other ones, it borrowed their models of data representation, meaning, for farm client, the pie-charts organisation, and, for vrvs client, the half-spheres organisation.

At the moment, as menitoned above, there are two clients that share one or more common panels. The method used to do that is by constructing a base class, that manages background related visual details, and then deriving it and rewrite neccesary methods to draw the data as needed.

My solution is a little more general, as I propose to use only one module for both clients, and, only when the client is run, the module is informed what data

representation model will use, by suppling from Main class an object that knows how to represent it. I call this the “plugins system” because that’s how a plugin functions.

The implementation allows for more such objects to be provided, and then, at run time, through user interaction, be able to change from one representation to another one.

A problem for existing representations is that, when more monitorized MonALISA services have close global locations, their representations may overlap, generating an unpleasant impresion to the user.

Inspired by a suggestion of a more experienced coleague, I choosed to represent the pie-charts (for farm client), on a three dimensional direction when two or more of them are overlapping.

The resulting representation, called “OnTop”, can be admired in the left side of the picture bellow, and compared with the clasic solution:

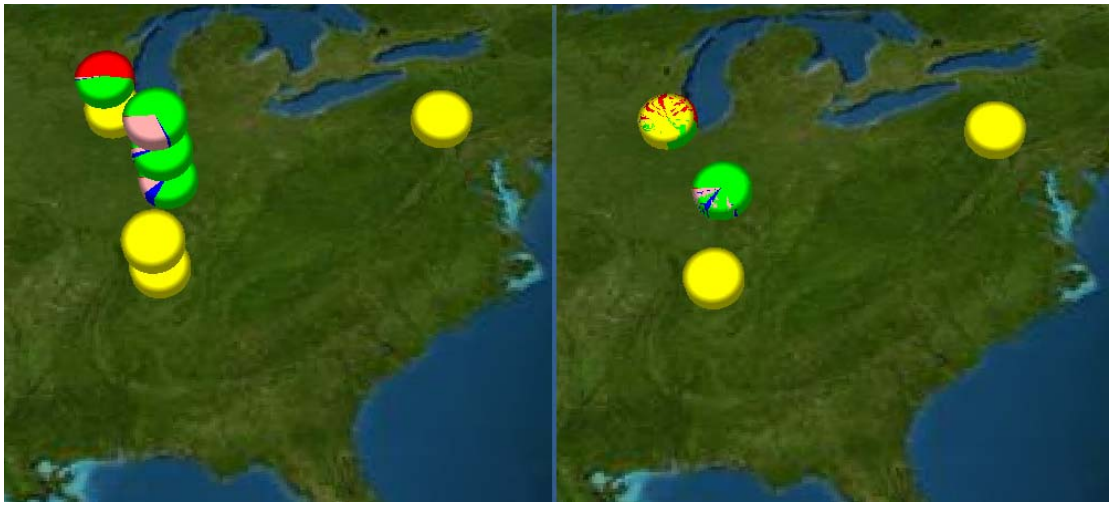


fig: OnTop representation versus the clasic representation, there are visible benefits

7. Implementation

I will present the most interesting parts of the implementation of various components of the whole project.

7.1 SlicePic application – image files naming convention

This application generates smaller image files and gives them unique names, when saving in a tree like structure of directories. The function to do this is "doPieces".

```
private static void doPieces( int level, String prefix, String currentDirectory, int
startX, int startY, int width, int height)
{
 if ( level<nlevels ) {
  width /= nx[level];
  height /= ny[level];
  currentDirectory += System.getProperty("file.separator")+"map"+level+prefix;
  File f = new File( currentDirectory);
  f.mkdir();
  for( int y = 0; y<ny[level]; y++)
    for( int x = 0; x<nx[level]; x++)
      doPieces( level+1, prefix+"_"+(y+1)+"."+(x+1), currentDirectory, startX+x*width,
startY+y*width, width, height);
  } else {
   //create the slice
   //get the pixels array from picture
   //transform in 3 byte array
   //save it to file under prefix+".oct"
   bi.getRGB( startX, startY, width, height, pixels, 0, width);
   for(int y = height-1, pointer = 0; y>=0; y--)
    for(int x = 0; x<width; x++,pointer+=3) {
     data[pointer+0] = (byte)((pixels[y*width + x] >> 16) & 0xFF);
     data[pointer+1] = (byte)((pixels[y*width + x] >>  8) & 0xFF);
     data[pointer+2] = (byte) (pixels[y*width + x]        & 0xFF);
    }
   String fileName =
currentDirectory+System.getProperty("file.separator")+"map"+level+prefix+".oct";
   try {
    System.out.print("Saving file "+fileName+" ... ");
    FileOutputStream fos = new FileOutputStream(fileName);
    fos.write( data);
    fos.close();
    System.out.println("OK");
   } catch (FileNotFoundException e) {
    System.out.println("Error");
    e.printStackTrace();
   } catch (IOException e1) {
    System.out.println("Error");
    e1.printStackTrace();
   }
  }
 }
```

The call is ***doPieces(0, "", System.getProperty("user.dir") + System.getProperty("file.separator") + "bin" + System.getProperty("file.separator") + "images", 0, 0, global_width, global_height)***, where global_* are the dimensions of the full image.

The first part of the function generates the directory structure to where the file is to be saved by calling itself recursivelly.

The second part writes an stream of octets, representing pixels, into a file who's name has the format:

- starts with "map" and a number representing the level this texture is created for,

- for each level until and including current one, add "_" and a number representing position on that level on y (specifies the line), then a "." and a number representing position on that level on x (specifies the column),
- the filename ends with the extension that is ".oct"

To easily be able to view an oct file, I created a windows application called OctViewer that receives as input parameter the name of the file and draws the image into a window. This application is very easy to use because it can be associated with the file type and by simply duble clicking the file the image will be available.

7.2. Texture algorithm

The implementation for the texture algorithm is a clean and powerfull one, in my opinion, as I structured my properties into a base class, called TextureParams, and some general methods, while the class Texture contains two types of functions:

- static ones that define a global action, on the tree, such as drawTree or as a response to user-interaction method, such as zoomChanged,
- and private ones, that are called only with a valid texture object and define actions available onto a texture.

7.2.1. Constructors

A texture object can be created by suppling one or more parameters, as it has available several constructors:

```
public Texture() {}

/**
 * constuctor to initialize a node in the tree
 * @param tex_id opengl id for texture
 */
public Texture( int tex_id)
{
 this();
 texture_id = tex_id;
}

/**
 * constuctor to initialize a node in the tree
 * having associated the world positions and dimensions
 * @param tex_id opengl id for texture
 * @param wX world position on x axis -> indice on x for array
 * @param wY world position on y axis
 * @param w width on map ( world map ) -> number of divisions covered by this texture
 * @param h height on map
 */
public Texture( int tex_id, int wX, int wY, int w, int h)
{
 this( tex_id);
 nWorldX = wX;
 nWorldY = wY;
 nWidth = w;
 nHeight = h;
```

```
}

/**
 * constuctor to initialize a node in the tree
 * having associated the world positions and dimensions
 * and a given status
 * @param tex_id opengl id for texture
 * @param wX world position on x axis
 * @param wY world position on y axis
 * @param w width on map ( world map )
 * @param h height on map
 * @param status texture loading indicator
 */
public Texture( int tex_id, int wX, int wY, int w, int h, int status)
{
 this( tex_id);
 nWorldX = wX;
 nWorldY = wY;
 nWidth = w;
 nHeight = h;
 this.status = status;
}

/**
 * constuctor to initialize a node in the tree
 * having associated the world positions and dimensions<br>
 * and the slice coordinates in texture
 * @param tex_id opengl id for texture
 * @param wX world position on x axis
 * @param wY world position on y axis
 * @param w width on map ( world map )
 * @param h height on map
 * @param lT left position in texture
 * @param bT bottom position in texture
 * @param rT right position in texture
 * @param tT top position in texture
 */
public Texture( int tex_id, int wX, int wY, int w, int h, float lT, float bT, float
rT, float tT)
{
 this( tex_id, wX, wY, w, h);
 leftT = lT;
 bottomT = bT;
 rightT = rT;
 topT = tT;
}
```

7.2.2. Load file

The function responsible for loading an image from file is loadTextureFromFile and it uses an Java specific optimisation in the way that, for a file name, it first searches into an Hashtable of weak references to images data, that has as **key** the slice identifier and as **value** the weak reference to a slice's byte array. Information is available from this hastable when the hard reference, the node reference, is null-ed, and is maintained for as long as there is available memory so that the garbage collector does not need to write something over.

```
public static byte[] loadTextureFromFile( String fileName, byte[] data, int level)
{
 SoftReference sr;
 sr = (SoftReference)texturesData.remove(id);
 //check to see if weak reference still points to something
 Object obj;
 //if ( wr!= null && (obj=wr.get())!=null ) {
 if ( sr!= null && (obj=sr.get())!=null ) {//from memory
  loadedFrom = 1;
```

```
   data = null;//free previous allocated array
   data = (byte[])obj;//return the referent of weak reference
  } else {//else (re)load the image data from file
   loadedFrom = 0;
   try {
    ClassLoader myClassLoader = JoglPanel.class.getClassLoader();
    BufferedInputStream bis = new
BufferedInputStream(myClassLoader.getResourceAsStream(fileName));
    ...
    if ( data == null )
     data = new byte[file_length];
    bis.read( data);
    bis.close();
   } catch (...) {}
   if ( data!=null ) {//if there is something to refer to
    sr = new SoftReference(data);
    texturesData.put( id, sr);
    //data will not be null'ed now because it will be few moments later
   }
  };
  return data;
 }
```

Thread that calls this function is TextureLoadThread that extends class Thread and calls inside the main function the Texture's static function loadNewTextures. This thread is commanded by the Idle Thread, through zoomChanged function, run from a TimerTask object. An important element in this function is the visibility test, which is not perfect, but behaves well under normal parameters.

7.2.3. Visibility test

The method is

```
private boolean textVisible( int x, int y, int nx, int ny)
```

and is called with starting indexes of the top-left corner, o a grid, static or dynamic, and the width and height the texture has (for dynamic grid these are always 1). The algorithm is composed of two parts:

- at 0 step, the first visibility test has the role of reducing complexity, as is intended to be simple, and to avoid duplicate points for sphere projection, one on each side of the sphere by hiding the back of the sphere:
- as pseudo-code, it looks like:

if (globeRadius!=-1) then

dist(p,e) must be less or equal to sqrt(d(e,C)^2-gR^2)

where gR is the virtual globe radius, p is point on texture, e is eye position, C is sphere center. The virtual globe radius is a small trick used to make a smooth transition of the center of the map, from plane projection, when is centered in (0,0,0) on world coordinates, to sphere projection, when sphere center is at (0,0,0) and map's center is at (0,0,gR). This radius is zero, when real globe radius is infinite, and starts to increase to a maximal value, as the other one decreases to the minimal value, final sphere radius, when both radiuses are equal one with the other and have the value:

Radius = MAP_WIDTH / (2 * PI)

- the second part uses a more complex and computation intensive algorithm
- what it does: determines if a point is in the visible frustum by going through several steps:

1. construct the eye direction vector d and normalize it
2. construct the eye normal vector n and normalize it;

   - reprezents the y axis of the screen

3. construct the second axis of the projection plane (screen):

   m = d x n => m has the direction to the right (x axis),

   - m is already normalized as cross product of two normalized vectors

4. construct the vector to the studied point, from eye: p
5. compute p projections on the n,m axis:

   pn = n.p (dot product between the two vectors)

   pm = m.p (dot product between the two vectors)

6. compute dimensions of visible frustum for the distance to the point:

   6.1 compute the distance from point to the plane:

   dp = d.p (dot product)

   6.1.0 if all 4 points(corners) have projection <=0 then texture is invisible

   6.2 compute fx=fm, fy=fn, the limits of the frustum:

   fx = dp*tg(alpha/2)

   fy = fx/aspect

   where alpha is the view frustum angle and aspect is the raport between width and height of the window

7. check if pn is contained in (-fy,fy) and pm in (-fx,fx)

   => transformed in: consider the four points that make the texture

   if at least 1 point's pm is smaller than its fm and at least 1 point's pm is greater than its -fm and same condition for n axis, then this texture is visible

   The code:

```
float []coord;
coord = Globals.points[y][x];
VectorO Vp1 = new VectorO(coord[0], coord[1], coord[2]);
Vp1.SubstractVector( JoglPanel.globals.EyePosition);
coord = Globals.points[y+ny][x];
VectorO Vp2 = new VectorO(coord[0], coord[1], coord[2]);
Vp2.SubstractVector( JoglPanel.globals.EyePosition);
coord = Globals.points[y][x+nx];
VectorO Vp3 = new VectorO(coord[0], coord[1], coord[2]);
Vp3.SubstractVector( JoglPanel.globals.EyePosition);
coord = Globals.points[y+ny][x+nx];
VectorO Vp4 = new VectorO(coord[0], coord[1], coord[2]);
Vp4.SubstractVector( JoglPanel.globals.EyePosition);
```

```
  coord = Globals.points[y+ny/2][x+nx/2];
  VectorO VpC = new VectorO(coord[0], coord[1], coord[2]);
  VpC.SubstractVector( JoglPanel.globals.EyePosition);
  //0. small visibility test
  if ( JoglPanel.globals.globeRadius!=-1f ) {
   float vgR = JoglPanel.globals.globeVirtualRadius;
   float dmin, dc;
   VectorO eye_center = new VectorO(0f,0f,-JoglPanel.globals.globeRadius+vgR);
   eye_center.SubstractVector(JoglPanel.globals.EyePosition);
   dc = (float)eye_center.getRadius();
   //the small visibility test functions only for outside of sphere view
   if ( dc>JoglPanel.globals.globeRadius ) {
    dmin = (float)Math.sqrt(dc*dc-
JoglPanel.globals.globeRadius*JoglPanel.globals.globeRadius);
    if ( VpC.getRadius()>dmin && Vp1.getRadius()>dmin && Vp2.getRadius()>dmin &&
Vp3.getRadius()>dmin && Vp4.getRadius()>dmin ) {
     return false;
    }
   };
   //0. test points
  }
  //1. eye direction vector
  VectorO Vd = JoglPanel.globals.EyeDirection;
  //2. eye normal vector
  VectorO Vn = JoglPanel.globals.EyeNormal;
  //3. second axis
  VectorO Vm = Vd.CrossProduct(Vn);
  //for each point repeat:
  //5. projections on axis
  float pn1, pm1;
  pn1 = (float)Vn.DotProduct(Vp1);
  pm1 = (float)Vm.DotProduct(Vp1);
  float pn2, pm2;
  pn2 = (float)Vn.DotProduct(Vp2);
  pm2 = (float)Vm.DotProduct(Vp2);
  float pn3, pm3;
  pn3 = (float)Vn.DotProduct(Vp3);
  pm3 = (float)Vm.DotProduct(Vp3);
  float pn4, pm4;
  pn4 = (float)Vn.DotProduct(Vp4);
  pm4 = (float)Vm.DotProduct(Vp4);
  //6. view frustum
  //6.1 dp
  float dp1;
  dp1 = (float)Vd.DotProduct(Vp1);// if (dp1<0) dp1=-dp1;
  float dp2;
  dp2 = (float)Vd.DotProduct(Vp2);// if (dp2<0) dp2=-dp2;
  float dp3;
  dp3 = (float)Vd.DotProduct(Vp3);// if (dp3<0) dp3=-dp3;
  float dp4;
  dp4 = (float)Vd.DotProduct(Vp4);// if (dp4<0) dp4=-dp4;
  //6.1.0 negative projections
  if ( dp1<=0 && dp2<=0 && dp3<=0 && dp4<=0 )
   return false;
  //6.2 frustum
  float fm1, fn1;
  fm1 = dp1*(float)Math.tan(Globals.FOV_ANGLE/2f*Math.PI/180f);
  fn1 = fm1/JoglPanel.globals.fAspect;
  float fm2, fn2;
  fm2 = dp2*(float)Math.tan(Globals.FOV_ANGLE/2f*Math.PI/180f);
  fn2 = fm2/JoglPanel.globals.fAspect;
  float fm3, fn3;
  fm3 = dp3*(float)Math.tan(Globals.FOV_ANGLE/2f*Math.PI/180f);
  fn3 = fm3/JoglPanel.globals.fAspect;
  float fm4, fn4;
  fm4 = dp4*(float)Math.tan(Globals.FOV_ANGLE/2f*Math.PI/180f);
  fn4 = fm4/JoglPanel.globals.fAspect;
  //7. check if at least 1 point is well positioned
  if ( pm1>fm1 && pm2>fm2 && pm3>fm3 && pm4>fm4 )
   return false;
  if ( pm1<-fm1 && pm2<-fm2 && pm3<-fm3 && pm4<-fm4 )
   return false;
  if ( pn1>fn1 && pn2>fn2 && pn3>fn3 && pn4>fn4 )
   return false;
  if ( pn1<-fn1 && pn2<-fn2 && pn3<-fn3 && pn4<-fn4 )
   return false;
  return true;
```

7.2.4. Texturing problem

A problem this algorithm had was that for slices of map situated behind the eye. As the picture below shows the algorithm increased the LOD for images that wheren't visible, so behaving incorrect.

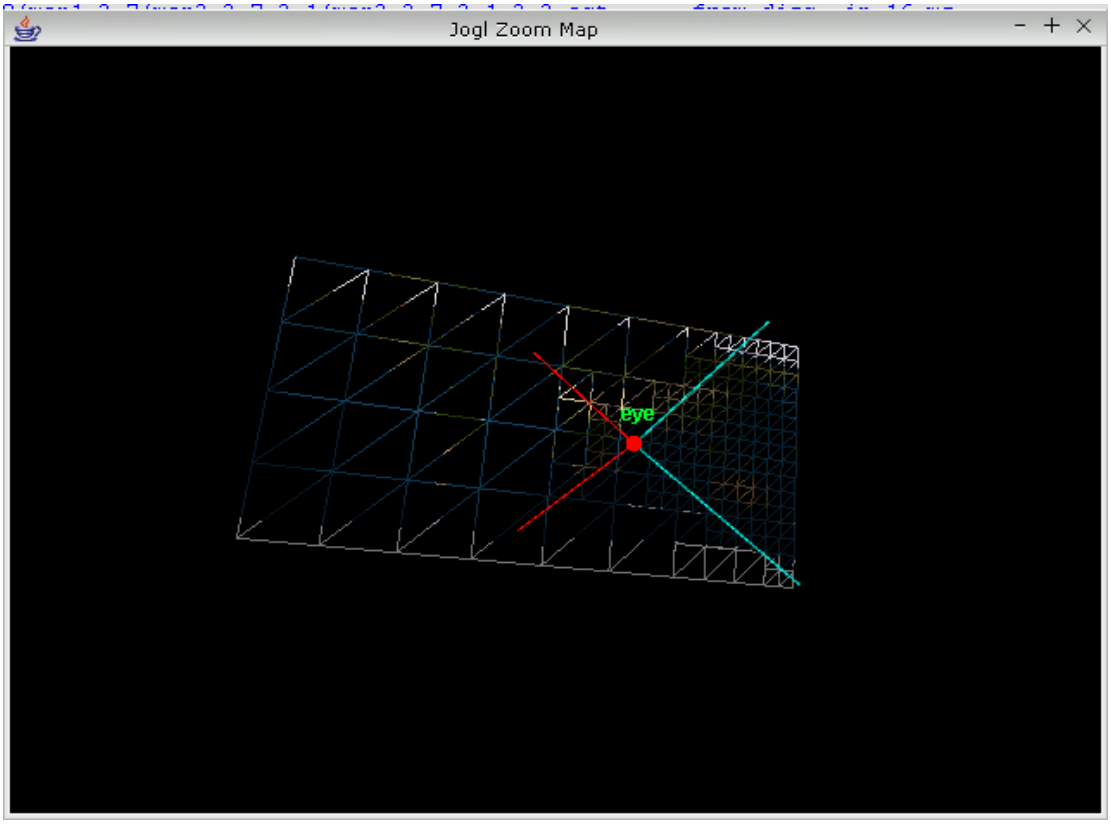


fig: misbehavior for visibility test

Solution stated that slices with negative depth from the eye be ignored (step 6.1.0 in above algorithm), resulting the correct output, as in the next image:

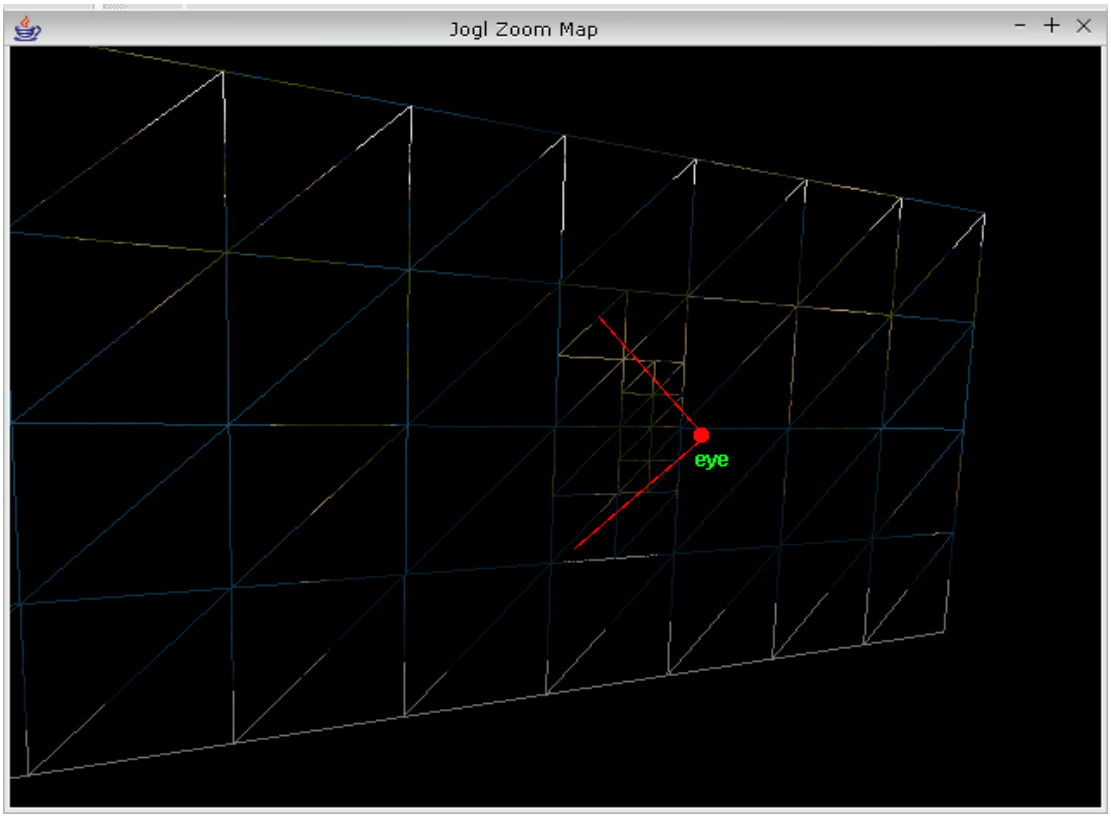


fig: visualisation of the algorithm after corect implementation

7.3. Grids

All points in the static grid are on a sphere, but they are specified only by an index for x and one for y, and a width and a height. These dimensions generate a 2d position inside a dreptunghiular area starting at (0,0). To get from this system to position in the 3d world, a tranformation is made with the global function point2Dto3D. This function first computes plane coordinates, and then, if it's the case, transforms them to sphere coordinates. Considering xp, yp, zp, plane coordinates, and xs, ys, zs, sphere coordinates with a radius of R and a virtual radius vR, then:

ys = R sin(yp/R),

xs = R cos(yp/R) sin(xp/R),

zs = vR – R + Rcos(yp/R) cos (xp/R),

where xp is in [-MAP_WIDTH; MAP_WIDTH], yp in [-MAP_HEIGHT; MAP_HEIGHT] abd zp = 0.

The code looks like this:

```
/**
 * computes the 3D coordinates of a point that is on the grid at (x,y) position
 * @param x position on grid on x axis
 * @param y position on grid on y axis
```

```
   * @return an float array with 3 coordinates for 3D
   */
  public static float[] point2Dto3D(int x, int y, float[] coord)
  {
   if ( coord == null )
    coord = new float[3];
   float px, py, pz;
   float aux;
   px = -(float)MAP_WIDTH/2f + divisionWidth*x;
   py = (float)MAP_HEIGHT/2f - divisionHeight*y;
   pz = JoglPanel.globals.globeVirtualRadius;
   coord[0] = px;
   coord[1] = py;
   coord[2] = pz;
   if ( JoglPanel.globals.globeRadius != -1 ) {
    aux = py/JoglPanel.globals.globeRadius;
    points[y][x][1] = JoglPanel.globals.globeRadius*(float)Math.sin( aux);
    aux = JoglPanel.globals.globeRadius*(float)Math.cos( aux);
    points[y][x][0] = aux*(float)Math.sin( px/JoglPanel.globals.globeRadius);
    points[y][x][2] = pz - JoglPanel.globals.globeRadius + aux*(float)Math.cos(
px/JoglPanel.globals.globeRadius);
   };
   return coord;
  }
```

The dynamic grid needs only its value to be recomputed, when a corner's value is changed, not through transformation, but using interpolation, as mentioned in the design section. The function to do that is computeDynGrid that initialize the grid, if is null and calls initDynamicGrid to set the correct values, based on the four corners. These functions can be found in TextureParams class.

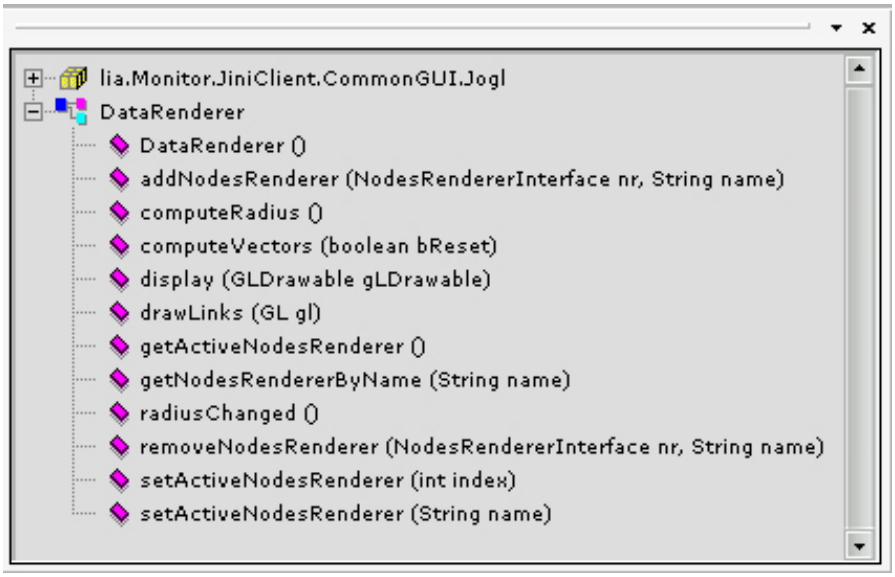
7.4. Plugins mechanism

The data visualisation components are organized, as stated above, into a plugins system, because the visualisation is a black box that accepts a limited sets of commands, and, based on that, represents the data. The black box and commands are represented by a Java interface, called NodesRendererInterface, that exports a function, drawNodes that has as input parameters, the 3d drawing object, list of nodes, and a list of context properties that should be respected for the graphics generated with this function to gracefully integrate with the rest of the 3d world.

The interface:

```
public interface NodesRendererInterface
{
	public void drawNodes( GL gl, Hashtable hnodes, Hashtable graphicalAttrs);
}
```

The class accountable for 3D Map panel's drawings is DataRenderer that extends the ZoomMapRenderer, and, as the name suggest, first one has the role of data rendering, while the second one background rendering, and, in particular, the texture LOD mechanism.



DataRenderer has a list of objects that implemented the interface and can draw the data for the nodes. A node here means a farm or a vrvs node. The functions are capable of selecting an active nodes renderer, remove it, return the current active one, add a new one.

The code of interest is listed below:

```
 //list of available nodes renderer objects that implement NodesRendererInterface
 private ArrayList NodesRendererList = new ArrayList();
 private ArrayList NRNameList = new ArrayList();
 //TODO: remove the two arrays, make a hashtable
 //current active nodesrenderer
 private int activeNodesRenderer = -1;
 //sets the active nodes renderer
 public synchronized void setActiveNodesRenderer( int index)
 {
  if ( index<0 || index >= NodesRendererList.size() )
   return;
  activeNodesRenderer = index;

JoglPanel.globals.mainPanel.comboNodesRenderer.setSelectedIndex(activeNodesRenderer);
 }
 //gets the active nodes renderer
 public synchronized NodesRendererInterface getActiveNodesRenderer()
 {
  if ( activeNodesRenderer!=-1 )
   return (NodesRendererInterface)NodesRendererList.get(activeNodesRenderer);
  return null;
 }
 //adds a nodes renderer object to the list
 public synchronized void addNodesRenderer( NodesRendererInterface nr, String name)
 {
  for( int i=0; i<NodesRendererList.size(); i++)
   if ( (NodesRendererInterface)NodesRendererList.get(i) == nr )
    return;//already added
  //else add this new nodes renderer
  NodesRendererList.add( nr);
  NRNameList.add(name);
  //also set active nodes renderer
  activeNodesRenderer = NodesRendererList.size()-1;
  JoglPanel.globals.mainPanel.comboNodesRenderer.addItem(name);

JoglPanel.globals.mainPanel.comboNodesRenderer.setSelectedIndex(activeNodesRenderer);
 }
 //removes a renderer from the list
 public synchronized boolean removeNodesRenderer( NodesRendererInterface nr, String
name)
 {
```

```
  //change the active nodes renderer
  NRNameList.remove(name);
  return NodesRendererList.remove(nr);
 }
 public synchronized int getNodesRendererByName(String name)
 {
  for( int i=0; i<NRNameList.size(); i++)
   if ( ((String)NodesRendererList.get(i)).compareTo(name)==0 )
    return i;
  return -1;
 }
 public synchronized void setActiveNodesRenderer( String name)
 {
  for( int i=0; i<NRNameList.size(); i++)
   if ( ((String)NRNameList.get(i)).compareTo(name)==0 ) {
    activeNodesRenderer = i;

JoglPanel.globals.mainPanel.comboNodesRenderer.setSelectedIndex(activeNodesRenderer);
    return;
   };
 }
```

It is main function's duty to set the available data rendering objects for node, and an call example for the farm client is:

```
JoglPanel jogl = new JoglPanel();
jogl.renderer.addNodesRenderer( new FarmNodesRenderer(), "pie view");
jogl.renderer.addNodesRenderer( new OnTopNodesRenderer(), "on-top view");
```

The OnTop nodes rendering alternative implements the interface and then, when called it renders a node based on previous nodes positions.

For that, it uses a list (Hashtable), to remember the treated nodes. The algorithm used is:

1. iterate through the global nodes
2. if the node has graphical attributes, then extract the positioning information and
3. begin iterating through local nodes
4. compare global node with each local node
5. if distance from one local node to the global node is smaller than 2 multiplied by the radius supplied as an graphical attribute, then the global node should be drawn on top of the local one. For that, in the list of local nodes there is an value at what height this node should be put, as there can already be other nodes.
    a. each node that is put on top of another one increments that one's counter
    b. this node is not stored in the local nodes as it gives no other new information
6. else the global node is inserted into local nodes list and its counter is set at value 1

The code:

```
  //get radius
  Object obj;
  obj = graphicalAttrs.get("NodeRadius");
  if ( obj == null )//no radius specified
   return;
  float radius = ((Float)obj).floatValue();
```

```
  //System.out.println("radius = "+radius);
  rcNode node, checked_node;
  VectorO []vectors;
  Hashtable htComputedNodes = new Hashtable();
  boolean bNodeTreated = false;
  Map.Entry entry;
  VectorO[] cnVectors;//checked node vectors
  //for each node, see if falls on top of another one, already cheched
  //and the draw it accordingly
  //for that I'll use an Hashtable to remember the checked nodes and
  //their properties: the postion and direction, the next level to put
  //a node that falls on current one, etc
  //so, hastable contains an array of two objects: the rcNode and its properties
  //one is  the key, and one the value
  //there is only one property that must be memorized: the number of nodes already
drawn on top of first one
  //including this one, because this can be multiplied with the radius to obtain an
height
  for ( Enumeration en = hnodes.elements(); en.hasMoreElements(); ) {
   node = (rcNode)en.nextElement();
   bNodeTreated = false;
   obj = graphicalAttrs.get(node);
   if ( obj!=null ) {//this node has its attributes computed, so draw it
    vectors = (VectorO[])obj;
    //check with each already treated nodes
    for( Iterator it=htComputedNodes.entrySet().iterator(); it.hasNext();) {
     entry = (Map.Entry)it.next();
     checked_node = (rcNode)entry.getKey();
     int levels = ((Integer)entry.getValue()).intValue();
     cnVectors = (VectorO[])graphicalAttrs.get(checked_node);
     if ( (float)vectors[1].distanceTo( cnVectors[1] ) < 2*radius ) {
      //distance from one node to another one is smaller than radius, so draw one on
top of other
      VectorO vNewPos = new VectorO(cnVectors[1]);
      VectorO vLevel = new VectorO(cnVectors[0]);
      vLevel.MultiplyScalar(2*levels*radius);
      vNewPos.AddVector(vLevel);
      //draw components
      drawNode( gl, vectors[0], vNewPos, radius);
      //update first's node infos
      entry.setValue( new Integer(levels+1));
      bNodeTreated = true;
      break;
     }
    };//end for
    if ( !bNodeTreated ) {//that means that this node falls over no other node
     //so draw it apart, but remember it on the hashtable
     //draw components
     drawNode( gl, vectors[0], vectors[1], radius);
     //add to htComputedNodes
     htComputedNodes.put( node, new Integer(1));
    }
   }
  };
```

8. Tests and evaluation

The first purpose of this paper was to find an alternative to Java3D. This purpose was achieved, but there was an initial test that has been done to compare JoGL with Java3D. The test consisted of an application that has two modules, one for

each technology, that were doing the same thing: draw the world map in sphere projection and several pies on it. An image of the test application is given below:

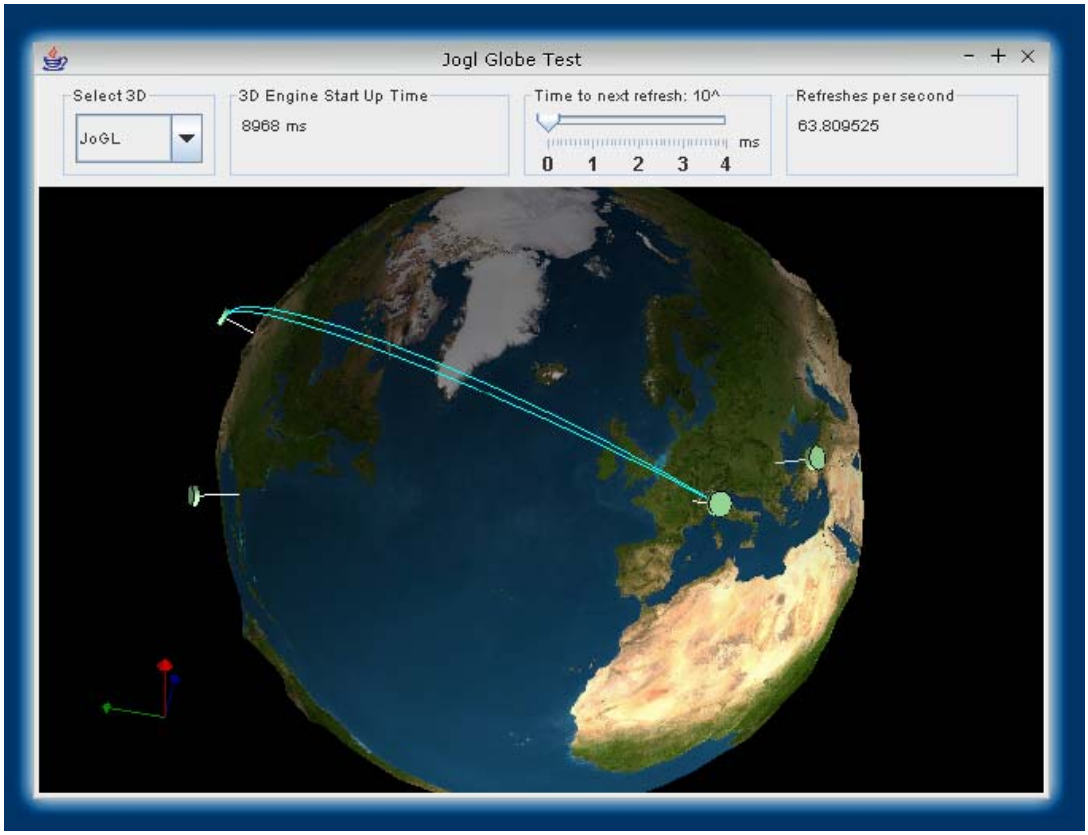


fig: JoGL module of the test application

The use of the program was to see the number of frames each rendering module could have per second, and the start-up time.

Results, although a little bit hazardous, shown that the JoGL engine functioned better than Java3D both on start-up time and on FPS (frames per second). The start-up time though, is not a decisive factor as it depends on Java's speed of loading images from file, and others, and so, only the bare engine start-up could not be measured.

This is an expected thing, because Java3D is a higher API, and overloads the OpenGL's calls with its calls, while JoGL permits plain OpenGL functions calls, and thus removing a layer of functions, at the cost of greater work for the programer.

Another test applications were developed to study the 3D map, optimal realisation for better performance and visualisation. I would remember here the "Map 3D" set of application that studied how the world map would look if it would be applied on it an height map. The visual effect is, for example, that mountains drawn on the geographical map, were at a higher level than the areas where plain was.

For better understanding a picture is provided:

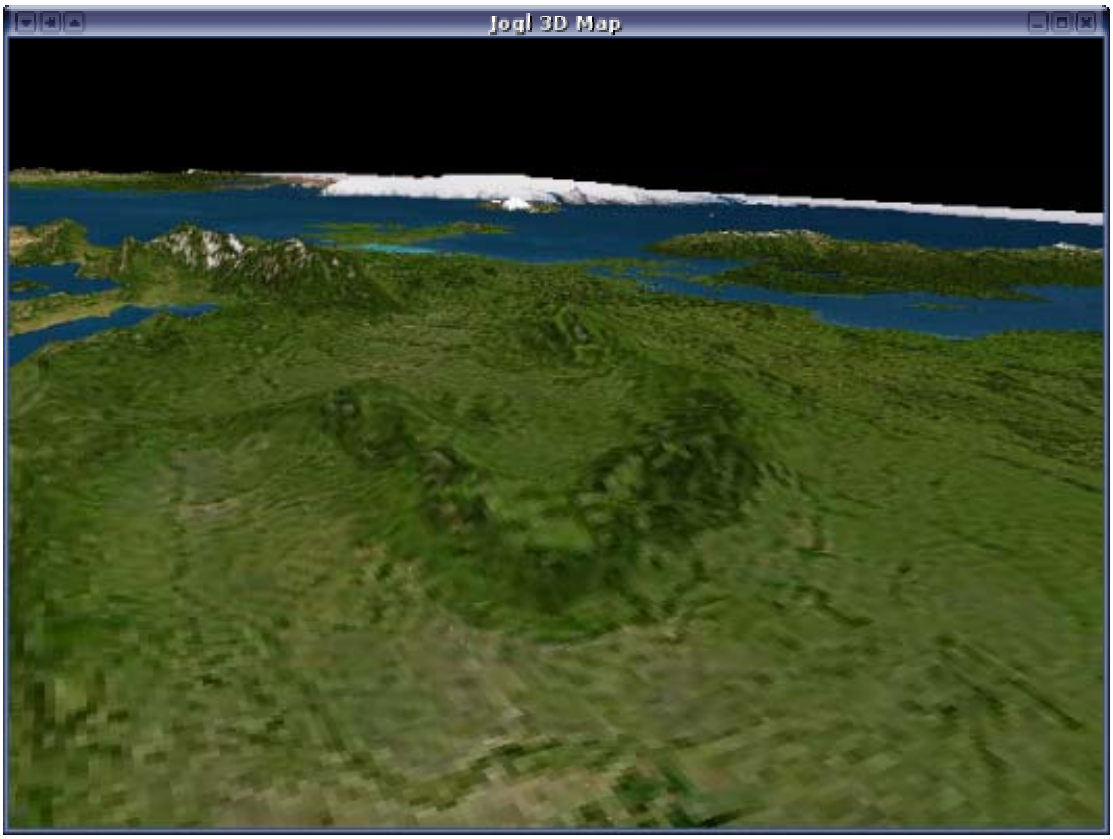


fig: higher points on world map are easily spotted with this representation

At the moment this solution was not implemented because the performance was decaying rapidly for an increased LOD.

9. Conclusions

The purpose of this project was to implement a module in MonALISA client to improve the 3D part of it. I think it did reach its goal as the new module not only replaces the Java3D implementation, but also the 2D map panel, and, by doing so, it substantially decreases the memory consumption.

What this module brings new is, from performance perspective, an increased level of detail for world map, at a much lower memory consumption price, as it does not exceeds 50 Megs, comparing to values of up to 1 Gigs of memory for the 8192x4096 map. The value is constant, but a little higher than the equivalent for smaller resolutions, mainly 1024x512 and 2048x1024, because of the internal structures that must exist.

From visualisation perspective, the transformation plane-sphere is ones of the atractions of this module, but also the new farms nodes renderer, OnTopNodesRenderer, brings a greater clarification level as the farms no more overlap on the map, because of vertical groupping.

JoGL proved the right alternative for 3D graphics as it doesn't need any addintional software installation, and runs on any platform that implemented OpenGL extension. The major platforms that provide support for this are: Windows, Linux, Solaris, Macintosh.

The mechanism of different level of detail for a portion of the world map that depends on zoomming level is very extensible, in that the user can have an increased LOD only for his areas of interest. An example is shown below, where only Switzerland map is shown at a higher LOD.

Bibliography

- H.B. Newman, I.C. Legrand, J.J. Bunn, “A Distributed Agent-based Architecture for Dynamic Services” CHEP 2001, Beijing, Sept 2001, http://clegrand.home.cern.ch/clegrand/CHEP01/chep01_10-010.pdf
- MonALISA web page http://monalisa.cacr.caltech.edu
- JavaOne site http://java.sun.com/javaone/
- JoGL site https://jogl.dev.java.net
- H.B. Newman, I.C. Legrand
  A Self-Organizing Neural Network for Job Scheduling in Distributed Systems
  CMS NOTE 2001/009, January 8, 2001
  http://clegrand.home.cern.ch/clegrand/SONN/note01_009.pdf
- Java Web Start, http://java.sun.com/products/javawebstart/
- Nehe site http://nehe.gamedev.net/
- OpenGL site http://www.opengl.org
- Mark Duchaineau
  "ROAMing Terrain: Real-time Optimally Adapting Meshes"
  http://www.llnl.gov/graphics/ROAM/