Radboud Universiteit Nijmegen

Master Thesis

Atlantis 3D

Document : Thesis
Version : 1.0
Date : 27 March 2006
Document nr. : 546

Student
Author : Jeroen Broekhuizen
Email : j.broekhuizen@hef.ru.nl
Student nr. : 0219428
Education : Science of Informatics
Supervisor : Dr. Theo Schouten
Referent : Dr. Herman Geuvers

Experimental High Energy Physics
Supervisor : Drs P.F. Klok

Abstract

At CERN, a European laboratory for particle physics located near Geneva, Switzerland, scientists are currently searching for the so-called Higgs particle. To facilitate this, a circular 27km long accelerator is under construction in which particles will collide at high speeds. During such collisions many new particles are created. Data of collisions or events are collected by detectors and eventually stored in XML files. To view these events for physics analysis the event-display program Atlantis was created, to show events in various 2D specific data-oriented projections. Atlantis is written in Java.

The question whether it is possible to embed a 3D component into the application arose. This component should create realistic 3D images of detector elements and events in real-time, in such a way that users can for example 'walk' through the detector. Speed is of uttermost importance in this application, as speed is one of the strategy points of the Atlantis development group.

Certain limitations of both Java and the existing application made it impossible to display this data directly into the application window. One reason was that the application expects the 3D image in its custom buffer. To solve this problem a new algorithm has been inserted into JOGL, a graphical 3D library for Java, which does not draw into the application window but into the applications buffer instead. Unfortunately, custom buffers are not supported by the graphical libraries so an intermediate buffer, also called an off-screen buffer, is used as link between the graphical library and the applications buffer. During this research JOGL was used as graphical library, which makes use of the graphics hardware (via native OpenGL function calls) during the creation of the 3D images

Besides creating and displaying the images inside the application, it is necessary to build a good hierarchical representation of the objects (detector elements and events) so additional features can work as fast as possible. Features implemented during this research include object picking and hiding (groups of) objects. The hierarchical representation technique used in this research is the scene graph, a directed graph, which in combination with the visitor pattern is a very powerful tool for this class of visualization software. The resulting scene graph package can also be used by other Java based applications.

Foreword

As informatics student interested in computer graphics, I welcomed this research project about adding a fast 3D component to an existing application with open arms. I finally had to perform actual research & development in 3D graphics for a visualisation application.

Here I would like to take the opportunity to thank my family, girlfriend Julia and friends for their support during this research. Their encouragements helped me a lot to finish this research and thesis within the expected time frame. Further I would like to thank Peter Klok, Theo Schouten and Herman Geuvers for their help, insights and idea's during this research. Especially Peter Klok for the amount of time he spent with helping, arguing, idea's and of course for taking me along on business trips to Birmingham and Paris. Finally I would like to say thanks to the guys in my room for the pleasant discussions about Windows and Linux (and lots of other discussions!), and not to forget the rest of the people on the EHEF department for the nice atmosphere these months.

This thesis is intended for anyone interested in creating visualisation software via an intermediate 'buffer' in cases where it is not possible (either limited by an existing application, or by design) to display images directly on the screen. Knowledge about 3D graphics programming and Java is a pre, but is not required.

Table of Contents

1. Introduction

This thesis describes the research performed as part of obtaining a master degree in Informatics at the Radboud University Nijmegen (RU). The research has been performed at the Department of Experimental High Energy Physics (EHEF) of the RU.

The supervisor from the School of Informatics is Mr. Theo Schouten and referent Mr. Herman Geuvers. The supervisor from EHEF is Mr. Peter Klok.

Objectives

Main objective was the study of the implications, with respect to speed, of adding a realistic 3D graphics component to an existing, basically 2D, graphics application used in particle physics. Concerns with respect to the addition of such a 3D view were about the negative influence it might have on the overall application performance. High speed rendering (the method to generate and view 3D images on the screen in real-time) is the key topic of this research in order to have a minimal impact on the overall application performance.

Besides, a natural extension of the existing user interface to use the 3D new functionality to be added should be studied.

today to optimize the speed to display the images of which I will use and/or combine a fraction during this research to get to the final result.

Method

Today there are numerous methods available. First existing techniques to display 3D images inside existing applications were surveyed in literature (papers and web-sites), especially regarding rendering techniques and optimization algorithms and eventually combinations of those.

As there was limited time available, a small selection of techniques was made which should have the most impact on the rendering performance. This selection not only includes techniques for rendering but also techniques important for fast and transparent transfers of the rendering results to the application.

Various combinations of the selected techniques were tested to find an optimal solution. Using this solution a working sample view was added to the application to measure the impact on the performance of the application itself. This view includes detector geometry, event data and some transforms like zooming and rotation.

Previous research

An enormous amount of research has already been performed on the topic of 3D rendering and methods to increase its speed. Although much of these techniques could be used during this research, these will be not the main concern. Other influences might have greater impact on the overall performance.

As the view must be embedded into an existing application, usual techniques for high speed rendering can not be applied as those all render directly to the screen,

which is not possible in this case. For this application a fast method is needed for rendering to an invisible area which later can be accessed by the existing application to display the final image.

In this area little to no research has been done and as such the results of this research can be used as foundation for further research in 'off-screen rendering'.

What's next

Below an overview of the upcoming chapters:

2. *About Particle Physics* – Gives a short introduction in particle physics and the ATLAS project and concludes with a short explanation of the Atlantis program.
3. *Atlantis* – Atlantis is the package in which the new three dimensional view was added. In this chapter a more technical description and its limitations are explained.
4. *Java and 3D* – Java does not directly support native 3D libraries. Using special libraries makes it possible to use those 3D libraries. This chapter explains how and what implications this has for the application. Also JOGL is introduced as library for accessing OpenGL from Java applications.
5. *OpenGL* – The native library for 3D used during this research is OpenGL. An introduction of this library is given in this chapter. Besides, also a few OpenGL rendering techniques necessary for this research are explained.
6. *Performance measurements* – As explained before tests have been used to find the optimal rendering solution. This chapter explains the tests which were performed and gives the results.
7. *Improvements of JOGL* – The results of the previous chapters have been used to modify JOGL to use the techniques which resulted in the highest performance. In this chapter is explained which modifications have been performed.
8. *Scene graph rendering* – Having hardware support is important, but a good geometry hierarchy is equally important. This chapter introduces scene graphs and lists advantages it has for the application (and extensibility).
9. *Results* – Finally the results of this research and points for future research are listed.

2. About Particle Physics

Particle physics explores what matter is made of and what forces hold it together. The necessary tools for this exploration are accelerators, which accelerate particles to almost the speed of light, and detectors to measure what happens when such particles interact during collisions.

CERN

Late 1949 the French scientist Louis de Broglie proposed that the countries of Europe would collaborate and set up an European research facility for particle physics to achieve together what none off the countries could have achieved alone. As a result the "Centre Européenne de Recherche Nucléaire" (CERN) came into existence on September 29, 1954 [13]. Currently CERN is one of the worlds leading laboratories in the field of particle physics. It is located near Geneva on the Swiss-French border.

The research performed at CERN is pure scientific, probing the innermost constituents of matter to find out how our world and the whole of the Universe work. The aim is to understand the world around us in a better way; there are no direct technological or commercial objectives. However, the demanding research in this field pushes the borders of technology further and further and generates lots of "spin-off" technology.

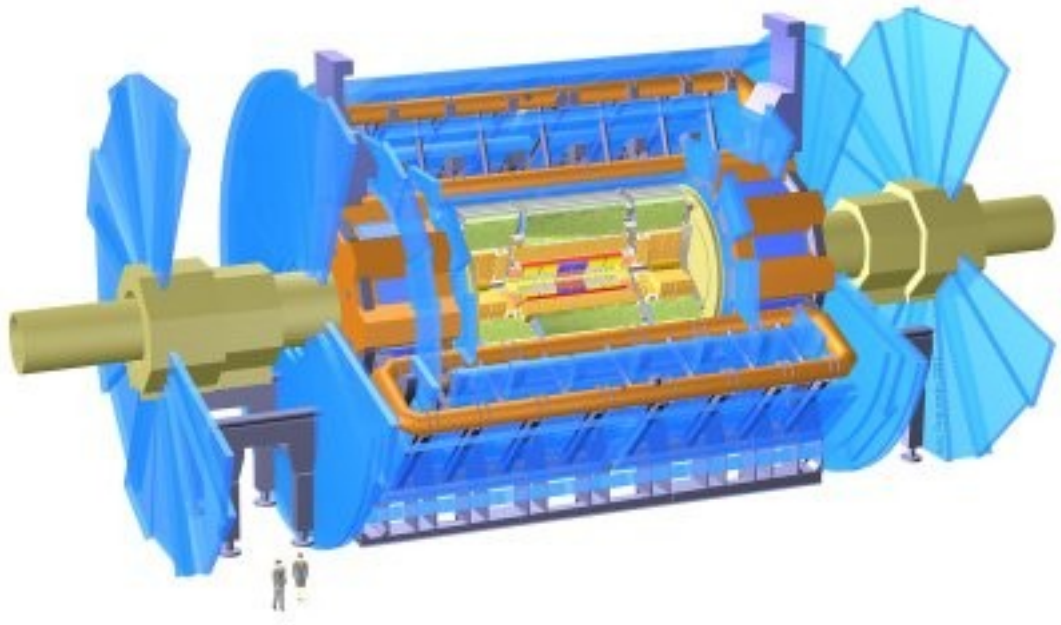


Figure 1: The ATLAS detector (currently under construction at CERN)

LHC, ATLAS and Atlantis

In the Large Hadron Collider (LHC) is a colliding accelerator currently under construction. When it is operational (expected end 2007) at the CERN laboratory, experiments will take place by colliding particles at extremely high speeds. During these collisions many other particles are formed and ultimately one hopes to find the Higgs particle [12] proposed by theoreticians, but so far never "seen". Special detectors positioned around the LHC collision points register these collisions and save the enormous amount of data collected into a database, from which via intermediate steps XML files are generated for later analysis. One of the main detectors in LHC is ATLAS, built by the ATLAS collaboration. EHEF is involved in this experiment as member of the collaboration. This collaboration consists of international groups (~100 groups and ~1000 scientists), including from The Netherlands (EHEF), United Kingdom and Switzerland.

During this experiment data of collisions and their decay products are stored for later analysis and visualisation. Since recorded collision are called events, programs to visualize event data are called an Event Displays. For the ATLAS detector the event display package Atlantis is used.

This package was originally developed in Fortran but was recently ported to the more flexible, modern and object-oriented Java, which also is platform independent. With Java more modern techniques became available like XML for data storage. As such this research will be limited to the Java language.

3. Atlantis

In the previous chapter the purpose of Atlantis was explained. In this chapter technical details and user perspectives of this application related to the research described in this thesis will be described. All details presented here where already implemented before this research started.

User orientation

Atlantis is written for physics analysis with it's users in mind. The strategy behind Atlantis is that it should work fast and consistently. This means that it should start up quickly and immediately respond to user input in a consistent manner. Different views of the event data must be controlled in the same way, including zooming, rotating and panning of the view. Also human perception is considered to facilitate an intuitive way of working with the program.

Atlantis currently provides seven types of two dimensional (2D) and three types of three dimensional (3D) projections of the data collected by the detectors. These 3D projections are non-typical and do not provide realistic images of the detector system and the event particles, but are very efficient for viewing specific physics properties in physics analysis. Figure 2 shows an example of one of the 2D projections and the user interface.

The Atlantis display area can be subdivided into a number of subwindows. Thus it is e.g. possible to view different projections of the same scene simultaneously. It also facilitates picking an element in one projection and showing this element in other projections.

An independent software review advised the Atlantis development group to integrate realistic 3D support into the application, meaning an insightful three dimensional representation of the detector and particles in such a way that images may be used for outreach and for analysis.

For ease of use and installation the new projection should be integrated and behave the same way as other existing projections do.

Buffered rendering

Certain events in Swing, a lightweight window environment for Java, require a redraw of the screen contents. In Atlantis this would mean that all visible geometry and event data must be detected and displayed every time such an event is received. Doing this would slow down the responsiveness on slower computer machines which does not comply with the original strategy.

The solution used is a buffered rendering technique. With this technique only in certain cases the visible geometry and event data have to be redrawn, like after a rotation of the projection or if an object's display property changes (for example its visibility). If a redraw is necessary, drawing is performed on an image located in memory instead of directly on screen. After the redraw the resulting image is displayed. This technique might seem slower at first, but the advantage of this technique is that in the cases where no complete redraw is necessary, only the image has to be displayed without performing any other operations.

Geometry to render

Atlantis uses an XML based storage system for initial settings, detector geometry and event data. At start-up the application loads this data into internal data containers (lists). This data later can be accessed by the different projections to be displayed. Usage of XML in combination with schema's can be used for automatic class generation, which currently is not used yet, but could eventually be used for automated loading and construction of the detector element instances and event data instances instead of manual parsing and instance construction.

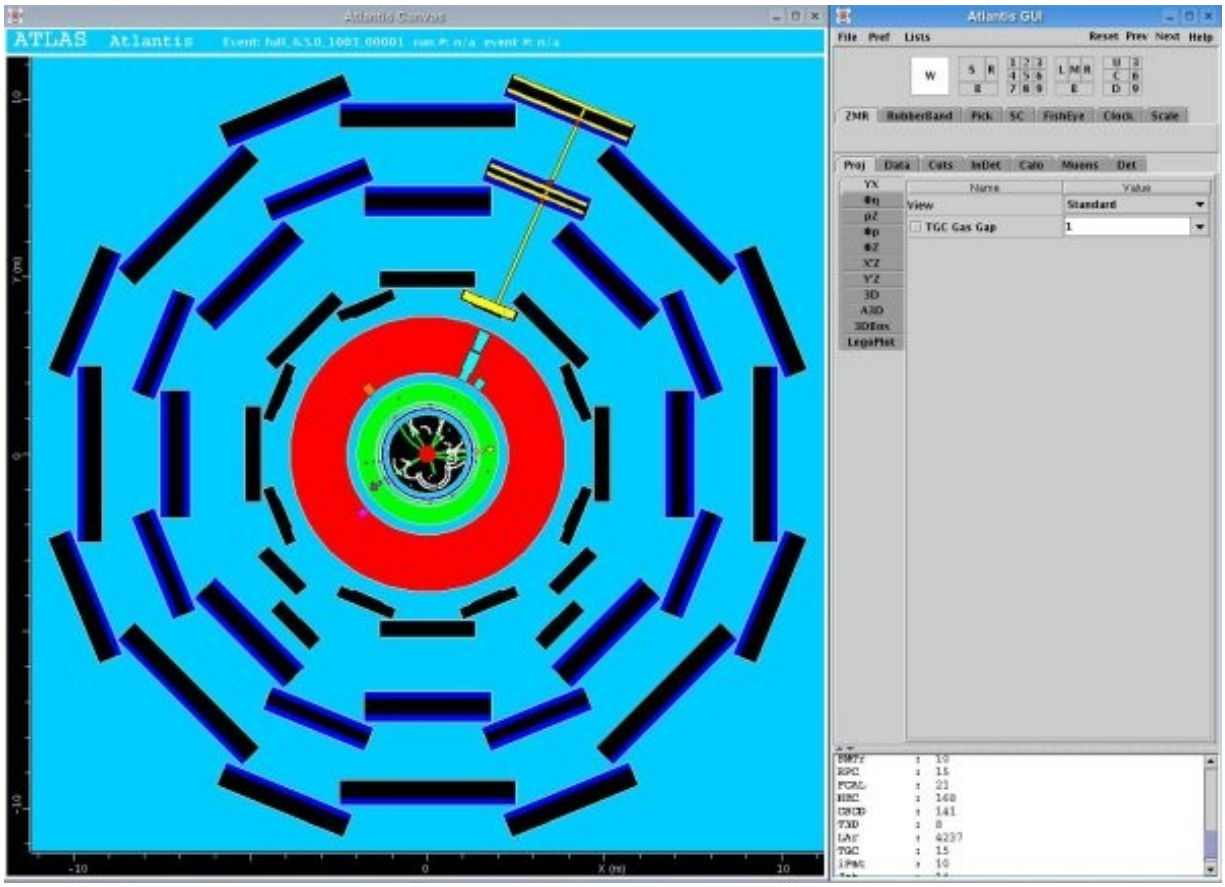


Figure 2: The final 3D detector as implemented according the changes in this thesis

The container interfaces supply methods to retrieve projection specific geometry lists to be rendered by that projection. These methods take into consideration if the data is/should be visible and compose a minimal list. For the new 3D projection a new method to these interfaces must be added to allow this projection to query the data. For this research no spatial subdivision algorithms, or any other techniques, were used to minimize the number of objects in the list. This kept the interface clean and simple to understand for the testing purposes.

Some types of data are stored with only 2D information. This data can thus not be displayed in the new projection as the third dimension is not yet available. When this information becomes available the changes necessary in order to display this new data should be kept to a minimum.

Stereo rendering

Three dimensional objects are seen differently through the two different human eyes. This cue is called stereo parallax [5]. Together with other cues people are better able to distinguish objects in a three dimensional environment. Stereo parallax is used in most stereo rendering algorithms.

Rendering in stereo most of the time needs special hardware to be usable. For instance special glasses have been developed to make stereo viewing work. Other techniques require very high frame rates (>100 frames per second) of the screen it is displayed on. Overall these techniques require rendering in full screen as otherwise the brain can not interpret the stereo pictures correctly. Bottom line is that expensive special hardware or a full-screen application is needed to make stereo rendering useful.

Both of these points do not fit within the strategy of the Atlantis development group to keep Atlantis simple and consistent. Therefore stereo rendering will not be investigated and used during this research.

4. Java and 3D

As stated in the Introduction, the main objective of this project is to research the implementation of a realistic 3D projection into the Atlantis application. To do this there are several possibilities available:

1. Write a custom 3D software library or
2. Use an existing Java package for 3D rendering.

The first option, writing a software renderer, is not feasible any more. Current desktop computers, in particular the graphics hardware, are often optimized for 3D rendering using for example OpenGL or DirectX. These graphics libraries make optimal use of hardware transformations and rasterization, as will be discussed later. In case hardware is not available, optimized software drivers will transparently take over the tasks which otherwise would have been performed by the graphics hardware.

Currently there exists numerous packages which add hardware optimized 3D support to the Java platform, including Java3D, JOGL and GL4Java.

Lightweight & heavyweight Java

Java GUI components can be classified as either lightweight or heavyweight. A heavyweight component is associated with its own screen resources also known as a peer. The lightweight components "borrows" its resources of its ancestors, and thus has no resources of its own and thus "lighter". Swing is a library consisting of only lightweight components written entirely in the Java Application Programming Interface (API).

It is possible to mix both kinds of components but this has some drawbacks like z-order fighting between components which heavyweight components always win. For a good integration in Atlantis this drawback is unacceptable as other subwindows must be able to draw its contents on top of the 3D projection.

To fully support the lightweight Swing components and it's z-order (the order in which the child-windows are drawn on the screen) one may not directly render OpenGL scenes to the screen like the heavyweight AWT components, a collection of classes to create user interfaces, do as discussed above. To circumvent this problem an off screen buffer must be used in which all rendering takes place.

The functionality of the heavyweight components is not relevant for this research and as such will not be discussed in this thesis.

OpenGL Java packages

Java3D is a well known package among 3D Java Game programmers. It offers a complete set of classes to ease the development of 3D graphics applications. Advantage of using Java3D over the other packages is the availability of ready-to-use functionalities. Main disadvantage of the package is little to no custom optimization possibilities and dependency on the developers of Java3D for new features to become available.

These two disadvantages can be fairly easily overcome by using the Java binding for OpenGL (JOGL) library. This library offers low level access to the graphics package OpenGL in Java, so program specific optimizations can be easily implemented. A drawback of JOGL is the lack of helper functionalities which are provided in Java3D.

Both packages have support for heavyweight and lightweight environments by using respectively an AWT canvas component or an off screen buffer. Recent versions of Java3D and JOGL support the pbuffer extension for off screen rendering.

Off screen rendering has many usages in today's applications. Sample usages are reflections by rendering the inverted world, procedural textures, blurring, etc. Though it is not yet used as necessary for this research.

As JOGL provides low level and flexible access to all OpenGL features necessary for this research, features which were not fully supported in Java3D at the time of this research, therefor the JOGL package has been chosen as underlying 3D library. This low-level interface also allowed to easily perform comparisons between the different methods for off-screen rendering.

Brief JOGL explanation

As Java is a platform independent programming language it has no direct access to the OpenGL function library available on most platforms. The Java Native Interface (JNI) framework can be used to invoke methods written in another native language like C/C++ while maintaining platform independency. With JNI it is thus possible to integrate OpenGL in Java and this is exactly the way JOGL has implemented it.

To support platform independent OpenGL access the developers of JOGL used the bridge pattern [9] to separate the platform dependent code from the actual Java code. A concrete factory class (abstract factory pattern [9]) is used to create the OpenGL context for each supported platform.

The solution JOGL provides for the lightweight swing environment can be accessed through the `GLJPanel` class. This panel can be added to an existing window enabling OpenGL support to the application. When the panel is created the programmer can invoke the same OpenGL commands as for a heavyweight environment. Underwater the panel uses an off-screen surface as it may not access the window frame buffer directly (would be heavyweight otherwise).

As explained above Atlantis uses a buffered rendering strategy. The JOGL panel class does not recognize this method and directly outputs it's data to the window. However, Atlantis also outputs it's buffered image to the window, thus overwriting the generated 3D event data. To prevent this, new methods have been implemented in JOGL as part of this research to support rendering to a buffered image, like the one used in Atlantis (see chapter 6).

Summary

The decision to use JOGL was based on its flexibility with respect to being able to easily add new functionality to the library, which is more difficult for Java3D. Besides, it supports the lightweight component interface Swing.

5. OpenGL

In this chapter OpenGL is described. Knowledge of this topic is necessary for a better understanding of the rest of this document. However, only the parts that are of interest for this research are discussed. For a complete reference of the OpenGL API see [1,2].

OpenGL & Extensions

Open Graphics Library (in short OpenGL) is a platform independent library of functions used by graphical applications, mostly for 3D imaging and CAD programs, to draw objects. It functions as a software layer on top of the graphics hardware. As time moves on, graphics hardware grows more sophisticated and gets more useful features. To keep up with these new improvements the OpenGL Architecture Review Board (ARB) tries to add new functionalities to OpenGL either through directly changing the API or by adding extensions.

Extensions mostly start as vendor specific extensions like the `SGIS_multitexture` extensions, which makes it possible to apply multiple images at the same time to an object. When multiple vendors agree to support an extension it will most likely be named `EXT_multitexture` (EXT may only be used when at least two vendors support the extension). In case the ARB approves the extension it will be named `ARB_multitexture`. With new revisions of OpenGL some widely used (and hardware supported) extensions will be integrated in the API. For example, the OpenGL Shader Language (GLSL) extensions were recently incorporated in OpenGL version 2 (specifications appeared in October 2004).

To use an extension in applications one must verify that the extension is available on the current hardware. This can be done using a standard OpenGL API function call which queries the graphics driver for a list of available extensions. As developer you are thus dependent on the extensions a graphics hardware vendor puts in their drivers. It is not possible to add your own extensions to these drivers. During this research I noticed this also with the PBO extension (will be explained later in this chapter) which is not available on ATI graphics hardware.

Using OpenGL

The model used for interpreting OpenGL commands is based on the client-server model. When a client program issues an OpenGL command it will be executed by the OpenGL server located in the graphics hardware driver. On both sides state variables are used to keep track of the current OpenGL state, of which most reside on the server side. These states together form the OpenGL rendering context (context in short).

Creation of such a context must be done by the window manager of a platform which manages the framebuffer. Window managers that are OpenGL compatible supply a small API to control the creation of this context. For example MS Windows has the WGL extension and the X Window System has support for OpenGL through GLX. During the creation of the context the complete or part of the framebuffer will be assigned to the context for rendering. Generally this is the client area of a window.

All contexts reside on the OpenGL server and thus a program must connect to it, which can also be done via the window manager. The server can have any number of contexts available, but one thread may only have one active context at a time and vice versa. Only after creation and connection to a context, issuing OpenGL commands will have effect on the state and/or the output.

Brief history of off screen rendering in OpenGL

In the early years of OpenGL rendering to an off-screen buffer was not possible. These days a programmer had to render the scene to the fixed-size frame buffer and perform a slow copy operation from this frame buffer to for example a texture object.

Soon it became apparent that this method was not sufficient for real-time applications like games or simulation software. For this reason the pixel buffer (pbuffer) and soon thereafter the bind-to-texture extensions were invented. The combination of these two extensions allowed an application to render directly to an off-screen pbuffer and use it as an image, all completely hardware accelerated.

Unfortunately the pbuffer has never been very popular among the OpenGL programmers for several reasons. One of these reasons was the requirement of unique contexts for every pbuffer. As explained above OpenGL maintains all of its states in contexts. Switching between different contexts is an expensive operation as the current state must be saved and is overwritten by the new state. Another problem with pbuffers is that they can get lost, for example during screen resolution changes. When this happens the application has to recreate the buffers and switch contexts again. For cross-platform applications another reason is the fact that the pbuffer extensions are not platform independent. It started on Linux as an GLX extension which later was ported to Windows and extended with the bind-to-texture extension. Thus using pbuffers leads to writing different code paths for each supported platform.

As a solution to these problems several vendors released the Framebuffer Object (FBO) extension [5]. At creation time one or more logical buffers, like the colour and depth buffers, are attached to a framebuffer object. After binding a framebuffer object to the current context the attached buffers will be used as source and destination during rendering operations. This new extension solves most of the problems related to the pixel buffers, including context switching and platform dependent code paths.

Rendering methods in OpenGL

When there is an active context, a program can render scenes to the buffer associated with this context. OpenGL supports only simple primitives: points, lines, triangles and polygons. By combining these primitives complex three dimensional representations can be created.

Rendering scenes involves sending information about the primitives to the graphics hardware which then can process it to output a representation of the primitives to the framebuffer. A primitive is built up from one or more points with several attributes, like position, normal, colour, etc. Sending this information about the primitives to the graphics hardware can be done using several methods which range from slow to very fast.

The first method, generally learned first, is the use of the `Begin/End` pair, also called immediate mode rendering. The `Begin` command takes one argument: the type of the primitives (point, line, etc.) which are described till the next `End` command. Inside the `Begin/End` pair you can use other OpenGL commands to set attributes of the points describing the primitive. Since these attributes must be set again for every frame, this method can be slow compared to other techniques depending on that specific situation (static or dynamic attributes, amount of attributes, etc.).

A (possibly slightly) faster way of rendering is the use of display lists. When a new display list is created you can fill it with a batch of OpenGL commands. Current video drivers tend to optimize them which results in high frame rates. Due to the overhead of setting up display lists it is not always a faster technique compared to immediate mode. Display lists also describe static primitives only. So for a dynamic scene the display lists must be recreated completely whenever a change occurs. Therefore this method is only useful in static scenes.

A faster way of rendering is the use of so called vertex arrays and its range extension (VAR). In a preprocessing step the vertices of the scene are stored in an array in RAM. Rendering of all the primitives stored in this array can be invoked with only one function call. As current GPU's work asynchronous, they can process the primitives while the CPU can continue processing the application, for example update the array for the next frame (the contents of the array are copied into the memory of the graphics hardware). VAR's can greatly speed up rendering of a scene and fully support dynamic scenes.

Although VAR's can speed up rendering, this method still has some important draw-backs. For example using a VAR requires a semaphore-like construct to perform efficient memory management. Recently a new method was introduced which handles the VAR and memory management for the programmer: vertex buffer object (VBO). Memory allocation and management are now completely implemented and optimized in OpenGL drivers and reside either in video memory or in RAM depending on how it is set up. Rendering from a VBO is invoked the same way a vertex array is. It has the additional advantage over VAR that you can specify that the VBO contains static only or dynamic data. Based on this information OpenGL can improve performance even better.

Reading back rendering results

As discussed before, rendering takes place on an off screen surface. In order to display the results there must be a way to read information back from these surfaces. In OpenGL this is implemented in the `glReadBuffer` function. This function reads all pixel colour values from the surface and stores it in a buffer supplied as argument to this function.

Based on the current platform, hardware and internal surface format this function can be very slow and stalls the graphical processing unit (GPU) from performing other 3D related operations until the read back has been finished. Transfer rates are dependent on the used hardware, for example the PCI-Express bus reaches much higher transfer rates then the AGP bus [6].

NVidia wrote the OpenGL extension 'Pixel Buffer Object' (PBO) [17] supporting asynchronous read backs from surfaces without any stalls. This way a program can request a read back and in the meantime perform some other processing

jobs until it really needs the pixel data. The extension guarantees that at that moment the pixel data is necessary, it will be available for use. However, it still may stall in case the operation was not finished yet.

Summary

OpenGL is a widely known graphics library used for visualisation and simulation software. With OpenGL complex scenes can be rendered by combining a multitude of primitives. A sample scene, as used in this research, would contain the detector elements and event data.

Graphics hardware vendors make new functionalities of their hardware available to the programmers through OpenGL extensions. During this research a selection of these extensions were measured for their influences on rendering speed, including the FBO as off-screen rendering surface and the PBO for a (potential) faster read-back from this off-screen surface.

6. Performance measurements

During this research the performance of a selected set of important techniques necessary for the new 3D projection has been tested. The techniques and the corresponding test results are listed in this chapter.

Finding the fastest code path

As speed is on of the most important concerns for Atlantis, multiple speed tests have been performed to determine which technique or combination of techniques results in the fastest rendering path. A specific test program has been written to perform these tests.

Vertex buffers have been excluded from the tests as all geometry is static (which is also true for the Atlantis geometry), which means that the geometry will not change during the run of the application. Therefore there will be no speed improvements compared to display lists (which can only contain static geometry). So, only display lists are used during the tests.

The test application is written in Java and uses either the modified or unmodified (see chapter 7) version of JOGL to render two Stanford models: a bunny and a dragon. The test application allows a user to configure the (combination of) techniques to be used during the upcoming test. The following techniques are supported by this application: pbuffer, FBO and PBO (only available on NVidia hardware). During the test the selected model is rendered 500 or 1000 times to the window via the off-screen buffer. At the end of the test the best, worst and average rendering- and read back times are displayed.

The results of the tests are listed in appendix A. These results show that using a framebuffer object is slightly faster than using a pbuffer even though both are hardware accelerated. An interesting point is that the use of a PBO can even slow down rendering considerably on fast graphics hardware instead of improving. This probably is caused by the fact that the test application immediately expects the pixel data after invoking the read call. The overhead of the synchronization plus a stall for immediate reading results in a slight performance worsening.

So, the best off-screen solution for Atlantis is the use of an FBO as framebuffer and a synchronized read back of the pixel data (thus not using the PBO extension).

Vertically flipping an image

In hardware the images are drawn up-side-down (e.g. the top of the image is at the lower left corner). Unfortunately in the Java environment this is the other way around: the image starts in the upper left corner. To display the image in a correct manner the program has to flip the image vertically. Flipping images can be implemented with different algorithms:

1. inverse scale using matrix, very slow,
2. up-side-down drawing of picture, faster,
3. copying buffer with for-loop, fastest.

Method 1 uses the `AffineTransform` class which transforms images using affine matrices while preserving image quality and correctness. Applying such a matrix involves a lot of matrix computations which in this case degrades performance. Using this method with the bunny test with FBO takes 33.7 seconds for 500 frames.

A considerably faster method uses the `Canvas.drawImage` method to flip the image. Flipping the image is done by supplying the function with two rectangular regions. The first region contains the region of the screen where the image should be drawn. In this case this region was supplied in an up-side-down manner <0,height,width,0>. The second region contains the part of the image that should be displayed on screen. The complete picture must be visible, thus the region will be <0,0,width,height>. The combination of these two regions make `drawImage` paint the image in the upper left corner of the window and correctly flipped. This method is much faster than method 1 and only takes 11.9 seconds for the bunny test of 500 frames.

The last method uses a for-loop to flip the image and then draws it also with the `drawImage` method. Instead of directly reading the pixel data to the image it will be stored in a temporary buffer. The loop then copies the data row wise from this buffer to the image in opposite direction and thus flipping the image. This is the method already used in JOGL. During the test this method needs 10.3 seconds.

Java versus C++ benchmarking

Early versions of the Java runtime environment were slow when compared to other languages like C++. Since that time a lot has changed and Java programs became much faster. Unfortunately a lot of people still assume Java to be slow. Numerous benchmarks [8] have been performed which show that recent Java versions can compete or are faster than C++ when used for certain algorithms. Mainly this is caused by the Just-In-Time (JIT) compilation to native code that is done in real time. This JIT compiler can generate code for the processor it is currently running on, while a C++ compiler has to generate native code at compile time with support for various hardware architectures.

The above mentioned benchmarks are mostly related to numerical aspects and do not address the Java Native Interface (JNI) which is used by the JOGL package. The use of native code inside Java programs could influence the code generated by the JIT compiler. To test if a wrapper around native OpenGL indeed slows down Java a benchmark has been run. The results of this benchmark are listed in appendix B. These show that C++ is indeed slightly faster, but this difference in speed is negligible.

Summary

Results from the tests performed for this research, showed that using a FBO in combination with synchronized read-back to a Java image has the best performance. As hardware stores its images in bottom-up manner, this image must be flipped in order to show up correctly in the Swing environment. The fastest way to do this is using the row-wise copying method as explained above.

The last test showed that using C++ instead of Java shows only negligible improvement. Therefore it is not recommended to use C++ and make the application more complex and platform dependent.

7. Improvements of JOGL

To enable JOGL to render to an image and improve its performance, some changes to the JOGL library have been made as part of this research. The how and why of these improvements are discussed in this chapter.

The improvements

To improve the standard JOGL distribution with off-screen facilities the following adjustments have been made:

1. off-screen rendering to buffered image,
2. use new techniques whenever appropriate.

Ad 1. Off-screen rendering to buffered image

As described in an earlier chapter, JOGL must be adapted to store its rendering result in an external image, instead of writing it directly to the window. To solve this problem a new class in the JOGL class hierarchy has been added with this functionality. At start-up the program can tell JOGL which image to use for storing rendering results.

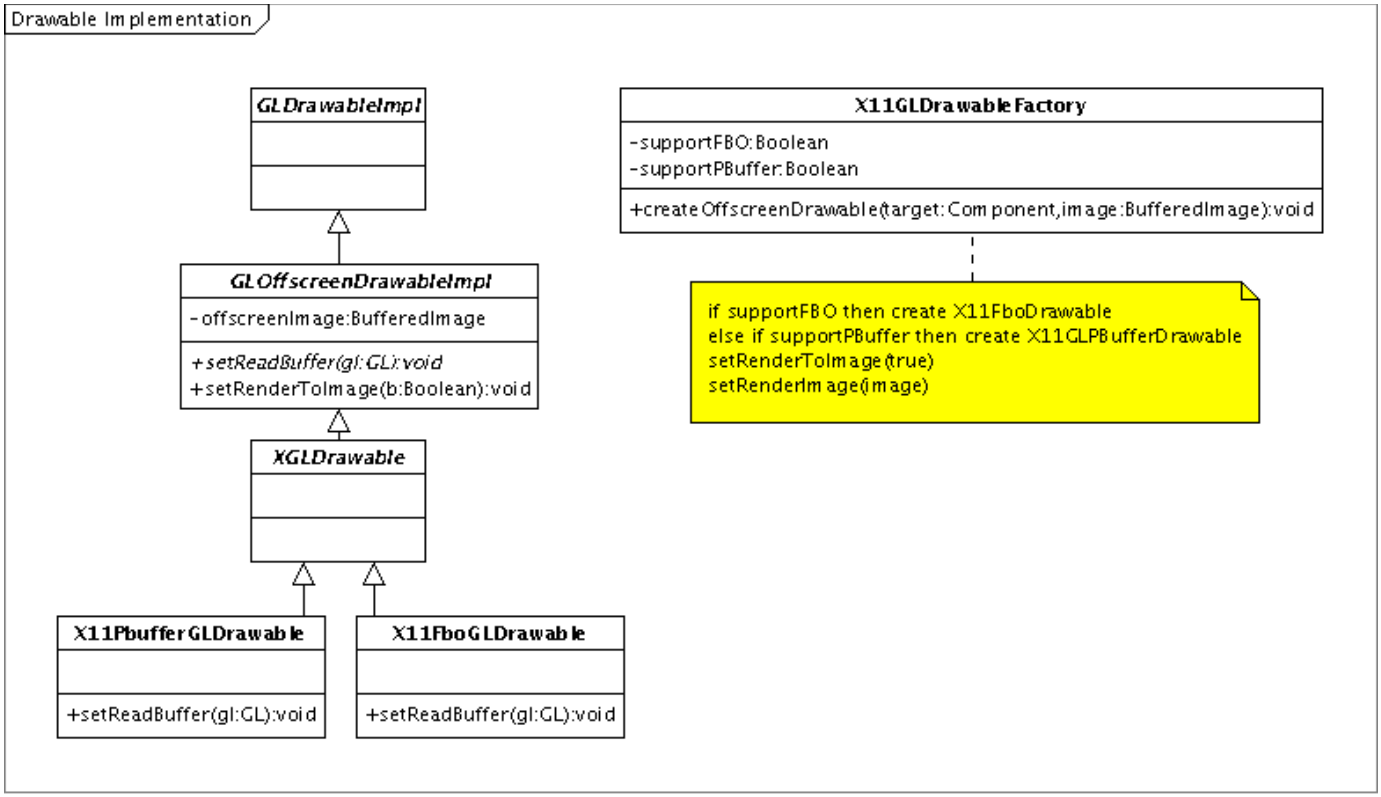


Figure 3: UML diagram of the GLDrawable Implementation

In chapter 4 was explained how OpenGL is used to read back pixel information to an image. However, one small remaining problem is that off-screen buffers, pbuffers and FBO's have different locations in memory. So, there must be a way

to tell OpenGL where to look for the pixel information. Therefore an abstract function `setReadBuffer` has been added that inheriting classes must implement, as can be seen in figure 3. The only task of this function is to give OpenGL the location of the pixel information of this class framebuffer. The `X11FboGLDrawable` class for example tells OpenGL to use its FBO as framebuffer for reading back the pixel information.

Atlantis may use the factory class to create an off-screen buffer class (it does does not know whether it is a pbuffer or a FBO). This factory automatically is supplied with the image where the result of the rendering process must be stored.

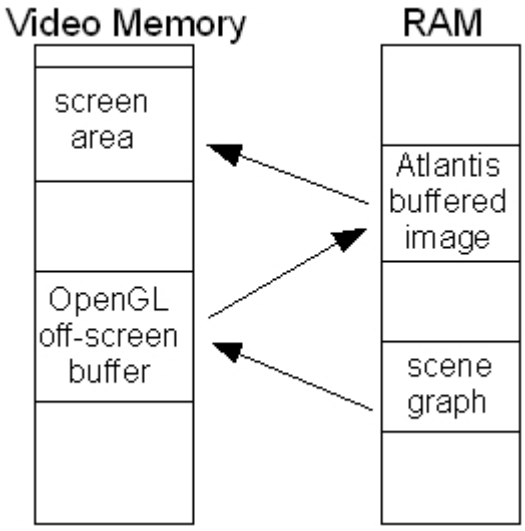


Figure 4: Data flow between buffers

Figure 4 schematically shows the image data flow throughout the application and OpenGL to finally appear on screen. As will be discussed in the next chapter, the detector elements and event data will be stored in a scene graph. First these objects are rendered via OpenGL in an off-screen image resulting in a full 3D representation of the detector with the event data. This image is then read back into Atlantis' own buffered image till the application has drawn everything. Finally this buffered image is displayed on screen (under water graphics cards reserve a special area of the video memory in which applications can write their images, which is then send to the monitor).

Ad 2. Use new techniques whenever appropriate

The first title of this section was 'Use new techniques whenever possible' but that is not true. Sometimes older techniques are still fast enough and easier to implement. An example is the use of a VBO or the older display list. As described above, Atlantis only uses static data, so display lists are as fast or faster then using VBO's. This is partially caused by the necessary state changes in OpenGL to make a VBO active. The code required for creating and using VBO's is also substantially more complex to write. Thus for Atlantis pre-generated display lists are used for rendering all geometry. Another example we've seen before is the PBO extension which in our case is slower than the original synchronous method.

For rendering to an off-screen surface JOGL uses the FBO extension by default if it is supported on the current platform. If this is not the case a pixel buffer (or on very old hardware the good old bitmap) is used as an alternative. As can be seen in figure 3 a derived class `XGLFboDrawable` is created to add support for FBO's in JOGL, which is totally transparent to the JOGL user as the factory returns a reference to the `GLOffscreenDrawableImpl` base class.

The tests in the chapter 6 have shown that using a PBO can slow down the reading of pixel data from the off-screen surface to an image. For this reason, and the fact that it is not a widely supported extension, the modified JOGL does not use it.

Summary

By inserting a new off-screen class into the Drawable class hierarchy full off-screen rendering support was added to the JOGL package. This new class uses the best solution found during the tests: FBO whenever available on this current platform and definitely no PBO will be used, even if it is available (could be made as an user option later), as it may slow down the application. In case a newer technique is not available on the hardware, a fall-back mechanism is used (implemented during this research) for selecting an older technique which is supported by the hardware for usage during rendering.

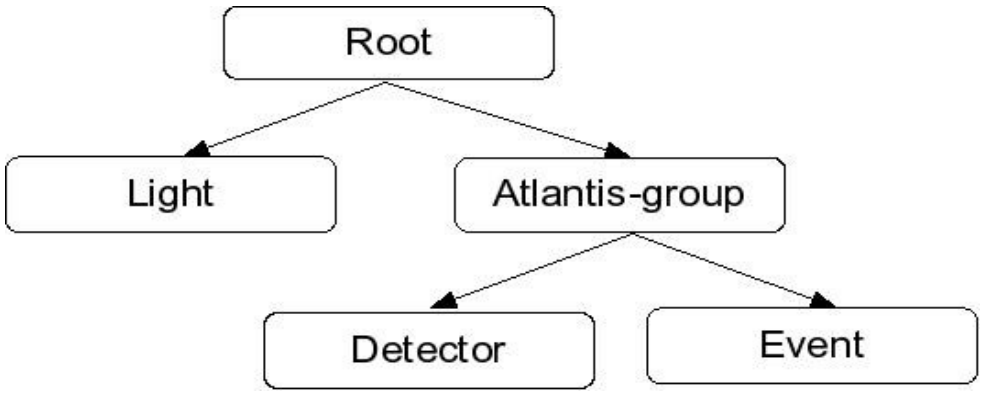
8. Scene graph rendering

In the previous chapter is explained how JOGL was improved to render as fast as possible in the Atlantis environment. To facilitate the rendering of 3D scenes, a scene graph package has been developed for this research. In this chapter the usage and implementation of the scene graph is discussed.

What is a scene graph?

The scene graph implemented during this research is a directed graph. This kind of graph is a data structure with exactly one parent and an arbitrary number of children. The top node is called the root and a node without children is called a leaf. Numerous nodes can be used to form a hierarchical representation of the scene [15].

Often a car is used to explain the use of a scene graph. The frame of the car is the root of the car. The four wheels are children of the car and are thus stored in a group node with as parent the car. This way a complete representation of the car can be build. In the following picture a scene graph is shown more specifically for this thesis subject.



In computer graphics, scene graph nodes can have different types. For every attribute or object a new node is constructed, for example light, geometry and transform nodes as can be seen in figure 5. Both 'Root' (it is just another name) and 'Atlantis-group' are group nodes; the only nodes in a scene graph which may have [0..n] children as described above (group nodes are the composite nodes of the Composite Pattern [9]), all other nodes are leafs. The 'Detector' and 'Event' nodes in this figure contain the geometry of this scene graph, which are lit by the 'Light' node.

The visitor pattern [9,16] is a widely used pattern in scene graphs because it is useful for separating algorithms from the nodes. Using several implementations of a visitor, multiple algorithms can be executed on nodes without any necessary changes to the node interfaces. To use the visitor pattern only one function must be added to the node, a function that accepts the current visitor (this is possible as all visitors should be inherited from the abstract base class `Visitor`). This pattern is used for example to implement geometry picking in Atlantis. A `PickVisitor` is implemented which traverses the scene graph in search of the node which has been clicked on.

A disadvantage of using scene graphs is that they must be generated manually. There is not yet an algorithm which determines what hierarchical structure is necessary for a given implementation. This is also the case for the Atlantis package, so the graph hierarchy is built manually.

Existing scene graph packages

Currently there are only a few alternative scene graph packages available for Java. The two best known packages are Java3D and Open Inventor (OI) where OI is mostly targeted at scientific solutions.

Both packages offer a rich set of classes for rendering geometry, lights and a multitude of other types of objects using a scene graph. Both packages have their advantages, but also some disadvantages. Advantages include ready to use classes for all kinds of operations including rendering, object culling and picking. When speed is involved both packages have to perform redundant calculations as they are written for usage by all kinds of application types. Therefore, the decision to implement a custom scene graph library which is focused on speed only, was made as part of this research.

Using the RUN scene graph

The separate RUN package that has been developed during this research contains a functional (but still far from complete) and optimized scene graph library developed for use in the Atlantis software, although other Java based programs should be able to use it too. In appendix C an UML class diagram of the RUN package is included. In that diagram only the important classes and methods are included to simplify the understanding of the behaviour of this package.

The `Node` class is the base class for scene graph nodes. The inherited nodes `Group` and `Geode` are currently the only node types necessary to build the complete Atlantis scene graph.

The group node is not a visible entity in the projection but is used to group certain child nodes. As an example all box detectors are grouped together in one group. Advantage of this kind of grouping is the ability to easily hide or perform an operation on a *group* of nodes. Hiding all box detectors can thus be done by simply hiding their group node, instead of iterating over all nodes in the scene graph to hide the box detectors.

Geometry of the event data and detectors are stored in geode nodes (geode is an abbreviation for 'geometry node'). Construction of this geometry is encapsulated in the `Shape` class hierarchy. Due to the lack of 3D information of certain detectors and event data in the Atlantis package currently only box- and trapezoidal shape detectors and simple event tracks are supported.

During the creation of a shape only the geometry is created. After the scene graph is fully created, the graph can be compiled. During this compilation process the geometry is stored in a display list for fast rendering (see chapter 5 for the reason). The result is that per geode only one function call is necessary to display the geometry.

Picking objects, like detector elements, is an important tool in the Atlantis application during analysis. Picking can be implemented in different ways. For example the build in OpenGL selection mechanism can be used for selecting

geometry sent to the graphics hardware. Unfortunately this mechanism only supports a maximum of 64 objects which is far from enough (typically there are >1000 detector elements). Another method is to pick objects based on colours. Every object should be rendered with a unique colour after which OpenGL can be queried to determine which colour is under the mouse pointer (this all happens in a special OpenGL buffer, so those coloured objects will not be visible for the user). Unfortunately this technique also can not be used. It only works for screens with a resolution of at least 24 bits. Otherwise the colours returned by OpenGL might not be the same as the ones originally supplied with the object. The method which works and is used in the RUN package shoots a ray from the mouse pointer into the scene. Using [11] we can determine if the ray hits the object and determine the distance. The object with the closest distance must have been the object which the user wanted to select.

Summary

A scene graph is a data structure used often for a hierarchical representation of objects, in this case the detector elements and event data, to display. Already existing scene graph libraries offer a rich set of classes, but as they are written with all sorts of applications in mind, they regularly have to perform unnecessary operations and thus waste valuable CPU time. During this research a custom scene graph (as a directed graph with 1 parent and [0..n] children) is implemented. This scene graph eases development of certain additional features, like object picking, object hiding, through the use of the Visitor Pattern. These features are implemented as different visitor classes which can all be applied on the scene graph nodes. This method makes it possible to easily traverse the graph and perform operations on (a subsection of the) nodes.

9. Results

Using the results and implementations of the previous chapters the new realistic 3D projection has been embedded into Atlantis and can be seen in figure 6 on the next page, reusing the existing functionality and user interface to simplify the usage of this projection for users. With this projection it is possible to view the ATLAS detector and simple event data. Using the POV-Ray program high resolution images of 3D scenes can be generated which can be used for PR, other public events and websites.

After a discussion with Julius Hrivnac (LAL, Orsay, France) was decided to display the geometry in two passes. The first pass renders solid geometry and during the second pass a wireframe (only the edges of the detector) is rendered on top of the solid geometry. The result is a much clearer image of the detector in which the detector elements can be distinguished better then without the additional wireframe. This extra pass does incur a slight penalty to the rendering speed, but does not outweigh the better visual perception.

This projection uses the scene graph and improved JOGL for rendering the detector and event data geometry to the off-screen buffer of Atlantis. After rendering Atlantis can display its custom interface widgets on top of this image the same way it is done on the existing projections.

Due to the current data layout used in Atlantis it is not yet possible to quickly detect which detector elements are hit by an event. It is thus not yet possible to hide all nodes without a hit. On the other hand it is possible to select a subset of the detector elements and hide the rest. This functionality can be easily extended in the future with other operations.

The tests and implementations have shown that it is possible to render fairly quickly to off-screen buffers, but it will stay slower than directly rendering to a window for the next couple of years although most graphics card vendors spent more time on off-screen rendering during the last five years. The modifications done inside JOGL resulted in a JOGL implementation which is at least 5 times as fast as the standard Java3D implementation. Without usage of the new FBO feature, it still is 4.7 times as fast. It is also more generally usable in other (scientific) applications which need 3D graphics embedded into it then the original JOGL implementation.

As little to no research has been performed in the area of off-screen rendering this thesis can be used as foundation for future research. After more investigation perhaps off-screen rendering can be used much easier and better then it is currently possible.

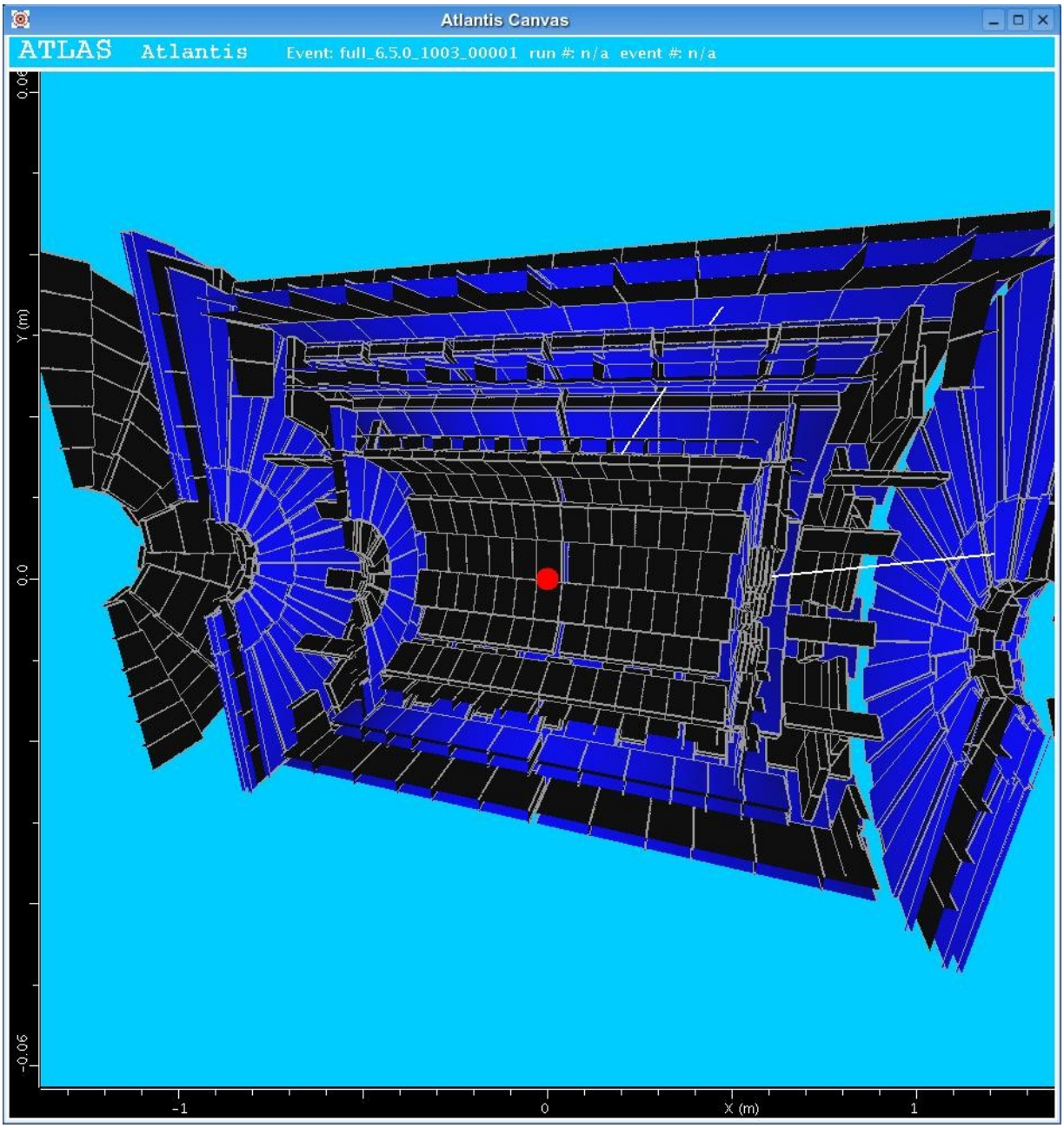


Figure 6: The final 3D detector as implemented according the changes in this thesis

Future research

Much research still can be performed in order to improve the performance or applicability of this new 3D projection. Below a few points are listed which might improve the performance of the 3D projection or its usage and can be topics of further research.

- *Scene graph serialization*: currently the scene graph is constructed every time Atlantis is started. Serialization of this scene graph to and from a file can speed up the construction, as lots of calculations can be skipped. Result of this can be a shorter start-up time of the application.
- *Automatic scene graph generation* – using XML with schema's allows a program to automatically generate instances of certain classes defined in the XML file. This way it will be possible to define the scene graph in the XML data file combined with a schema. Then no manual Java coding process is necessary any more and the scene graph can be changed without recompiling the application.
- *General scene graph* – currently only the new 3D projection uses the scene graph to represent the data. It is also possible to use the same kind of graph for the other 2D projections, resulting in a common application structure. Though such a change would have a tremendous effect on the complete application structure.
- *Culling techniques*: culling techniques can speed up rendering tremendously if used correctly. These techniques try to cull away as much invisible geometry as possible leaving us with as less geometry as possible to render.
- *Clipping geometry*: with clipping techniques (for example the build-in clip planes in OpenGL) geometry can be cut in such a way that the inner detectors become better visible.
- *Using transparency*: allowing users to make detectors with a hit transparent would result in better insight where exactly this hit was detected inside the detector.

Appendix A

Benchmark: JOGL versus modified JOGL

In this test various rendering techniques are tested to determine which combination is the fastest way for off screen rendering. The measurements are performed and listed in milliseconds. Please consult the legend on the next page for explanations of the table headers. JOGL used for these tests was downloaded at October 14, 2005.

	Test Framework	***Triangle model***	***Total time***	***Min. time***	***Max. time***	***Avg. time***	***Fps***
System: ATI Mobile x300 PCI-E, Intel Centrino 1,6GHz							
512x512, fbo	1	B	10,312	15	32	20	49
512x512	1	B	10,424	15	32	20	48
512x512, fbo	2	B	10,219	15	32	20	49
512x512	2	B	10,312	15	32	20	49
512x512, fbo	2	D	41,312	15	32	20	24
512x512	2	D	41,390	31	47	41	24
System: NVidia 7800GT PCI-E, AMD64 3200+							
512x512, fbo	1	B	2,531	0	16	4	198
512x512, fbo,pbo	1	B	2,578	0	16	4	194
512x512	1	B	2,563	0	16	4	195
512x512, pbo	1	B	2,641	0	16	5	189
512x512, fbo	2	B	2,390	0	16	4	209
512x512, fbo,pbo	2	B	2,438	0	16	4	205
512x512	2	B	2,391	0	16	4	209
512x512, pbo	2	B	2,437	0	16	4	205
512x512, fbo	1	D	7,719	0	16	7	130
512x512, fbo,pbo	1	D	7,671	0	16	7	130
512x512	1	D	7,781	0	16	7	129
512x512, pbo	1	D	7,719	0	16	7	130
512x512, fbo	2	D	7,375	0	16	7	136
512x512, fbo,pbo	2	D	7,484	0	16	7	134
512x512	2	D	7,422	0	16	7	135
512x512, pbo	2	D	7,500	0	16	4	133

	Test Framework	*Triangle model*	*Total time*	*Min. time*	*Max. time*	*Avg. time*	*Fps*
NVidia 6800 AGP 8x, AMD 2600XP							
512x512, fbo	2	B	6,953	0	16	13	72
512x512, fbo,pbo	2	B	2,438	0	16	4	205
512x512	2	B	6,964	0	16	13	72
512x512, pbo	2	B	2,437	0	16	4	205
512x512, fbo	2	D	20,031	15	32	19	50
512x512, fbo,pbo	2	D	7,484	0	16	7	134
512x512	2	D	20,075	15	32	19	50
512x512, pbo	2	D	7,500	0	16	4	133

Legend:

Test Framework	1 : standard JOGL code 2 : changed library
Triangle Model	B: Bunny model with 69451 triangles (with 500 tests) D: Dragon model with 202520 triangles (with 1000 tests)
Total time	Total time necessary for both rendering to the off screen buffer and capturing/displaying image on screen.
Min. time	Minimum time necessary for capturing and displaying one frame.
Max. time	Maximum time necessary for capturing and displaying one frame.
Avg. time	Average time necessary for capturing and displaying one frame.
Fps	Average frames per second displayed during test.

Appendix B

Benchmark: Java versus C++

The results below are from a speed comparison between Java Swing and C++ using the Windows GDI platform. JOGL used for these tests was downloaded at November 23, 2005 (uses the JSR 231 specification).

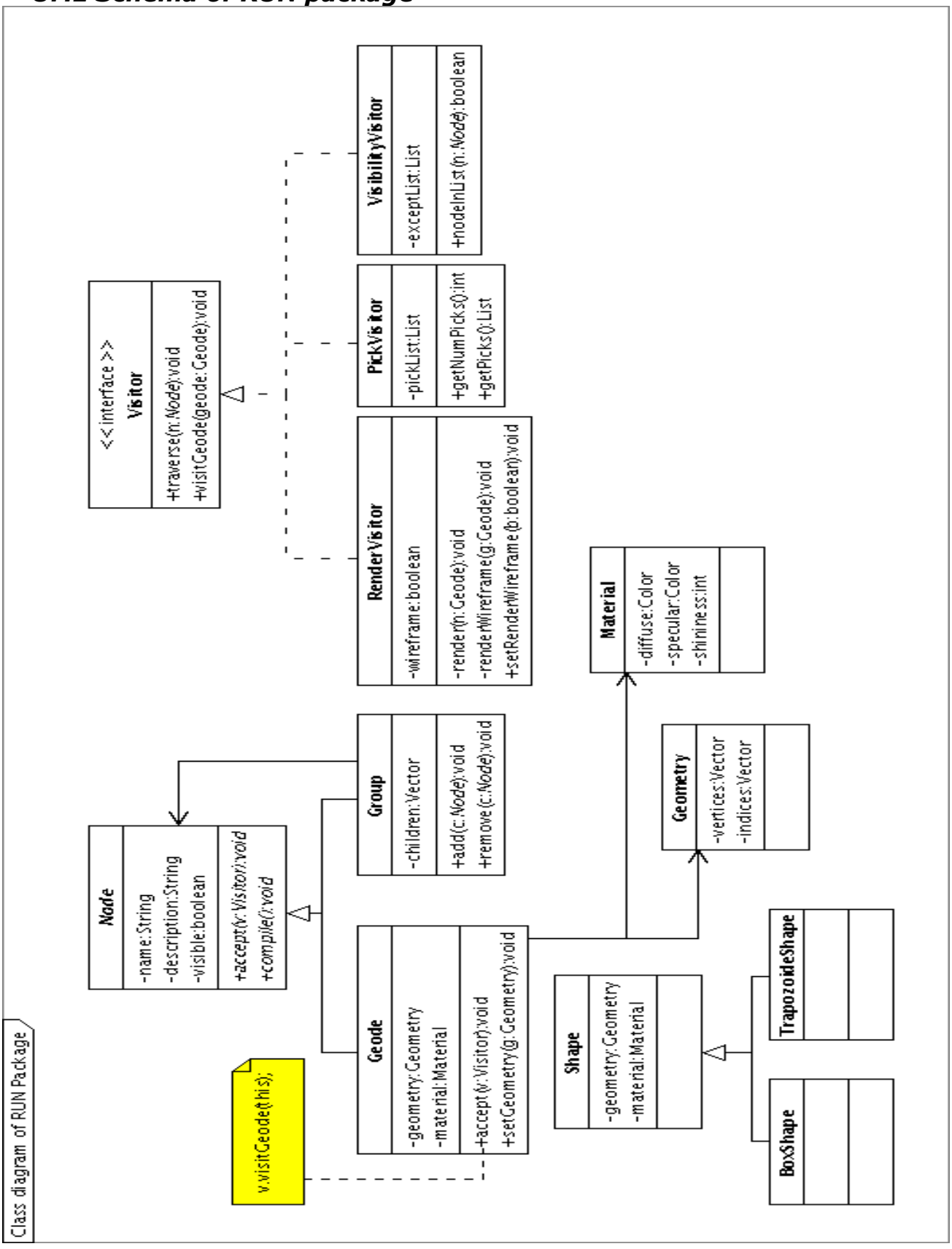
	Test Framework	***Triangle model***	***Total time***	***Fps***
System: ATI Mobile x300 PCI-E, Intel Centrino 1,6GHz, 1.25GB RAM				
512x512, fbo	Java	B	7,328	49
512x512	Java	B	9,375	53
512x512, fbo	C++	B	6.766 (6,640)	74
512x512	C++	B	9.375 (7,125)	53
512x512, fbo	Java	D	21,859	46
512x512	Java	D	22,140	45
512x512, fbo	C++	D	21,094 (20,906)	49
512x512	C++	D	21,719(37,486)	46

Legend:

Test Framework	Java : compiled Java code (JBuilder 2005 Foundation) C++ : compiled C++ code (Microsoft VS.NET 7.1, Standard)
Triangle Model	B: Bunny model with 69451 triangles (with 500 tests) D: Dragon model with 202520 triangles (with 1000 tests)
Total time	Total time necessary for both rendering to the off screen buffer and capturing/displaying image on screen.
Fps	Average frames per second displayed during test.

Appendix C

UML Schema of RUN package



References

Articles & books

[1] Mark Segal and Kurt Akeley,*The OpenGL Graphics System: A specification* (version 2.0, October 22, 2004), http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf

[2] Tom Davis, Jackie Neider and Mason Woo, *OpenGL Programming Guide, Second Edition* (also called the Redbook), Addision-Wesley Publishing Company, http://www.gamedev.net/download/redbook.pdf

[3] Louis Bavoil, *OpenGL VBOs vs Display Lists: Data Types*, November 4, 2005, http://www.sci.utah.edu/~bavoil/opengl/vbo/data_types/

[4] Ikrima Elhassan, *Fast Texture Downloads and Readbacks using Pixel Buffer Objects in OpenGL*, Technical brief, Nvidia, http://download.nvidia.com/developer/Papers/2005/Fast_Texture_Transfers/Fast_Texture_Transfers.pdf

[5] Emil Person, *Framebuffer Objects*, ATI Technologies Inc. (included in the ATI SDK Oktober 2005)

[6] J.M. Bull, L.A. Smith, L. Pottage and R. Freeman, *Benchmarking Java against C and Fortran for Scientific Applications*, Edinburgh Parallel Computing Centre, 2001

[7] *Trail: Java Native Interface* (September,2005), http://java.sun.com/docs/books/tutorial/native1.1/

[8] J.P. Lewis and Ulrich Neumann, *Java versus C* (January 2005), http://www.idiom.com/~zilla/Computer/javaCbenchmark.html

[9] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Professional Computing Series, Addison-Wesley, 1995

[10] Neil A. Dodgson, *Autostereoscopic 3D Displays*, August 2005, IEEE Computer Society

[11] Thomas Möller and Ben Trumbore, *Fast, minimum storage Ray/Triangle Intersection*, Journal of Graphics tools, 2(1):21-28, 1997

[12] Simon Hands, *Ripples at the Heart of Physics*, Theory Division, CERN, 1993, http://www.phy.uct.ac.za/courses/phy400w/particle/higgs5.htm

[13] LHC Large Hadron Collider, Cern Publication, June 1990

[14] Tomas Möller and Eric Haines, *Occlusion Culling Algorithms*, November 9, 1999, Gamasutra, http://www.gamasutra.com/features/19991109/moller_haines_01.htm

[15] Avi Bar-Zeev, *Scenegraphs: Past, Present and Future* (Januari 31, 2006), http://www.realityprime.com/scenegraph.php

[16] Jeremy Blosser, *Java tip 98: Reflect on the Visitor design* (January 31, 2006), http://www.javaworld.com/javaworld/javatips/jw-javatip98.html

[17] NVidia, *White Paper, Using Pixel Buffer Objects (PBO) (October 15, 2006)*, http://developer.nvidia.com/attach/6427

Websites

Atlantis - http://atlantis.web.cern.ch/atlantis/
ATLAS – http://atlas.ch/
CERN - http://public.web.cern.ch/Public/Welcome.html

JOGL - https://jogl.dev.java.net/
Java3D - https://java3d.dev.java.net/
Open Inventor - http://www.tgs.com/

OpenGL in Java - http://csdl.computer.org/dl/mags/cs/2005/01/c1051.pdf
OpenGL extensions - http://www.opengl.org/resources/features/OGLextensions/
GLX Specifications - http://www.opengl.org/documentation/specs/glx/glx1.3.pdf

Poseidon for UML - http://gentleware.com/index.php