Graphical Visualization of a Scientific Space Mission

Prof. dr. ing. Dorian Gorgan

Dan-Adrian Avram

July 2012

Abstract

Visualization in the space sector is becoming increasingly important as space missions become larger, more complex and harder to control. Visualization can help simulate critical tasks, plan ahead and thus greatly reduce costs. This paper proposes a possible solution to create a 3D and 2D visualization library for space missions similar in their nature to ESA's Rosetta project, while motivating why the selected development technologies are advantageous to use in this context.

The 3D and 2D visualization techniques had to be integrated into the same software, so in this approach, a prototype software is created to show the extent to which the researched technologies can carry out the task. Visualizing a simulation of the Rosetta mission is the main focus of the prototype.

The result is a visualization library with core functionality and a prototype application that uses it(in the current approach they are both part of the same application, but a separation is possible and shall be done in future endeavours). Even though some functions are not implemented completely, the created library shows great promise for developing future visualization applications using the researched technologies. This is due to the proved versatility and flexibility.

Contents

List of Tables

List of Figures

Chapter 1

Introduction - Project Context

Work on this project started in June 2011, during my traineeship at the European Space Agency(ESA), European Space Astronomy Center(ESAC), near Madrid, Spain.

This project is associated with one of ESA's Missions - Rosetta. In brief, the mission's objective is to study a comet thoroughly by landing on it using an unmanned spacecraft. For more details about the mission please see section 1.1.

The project aims to successfully create a prototype for a 3D and 2D visualization interface that models important aspects of this mission. Moreover, it should be very easy to extend the functionality to similar missions in the future. Emphasis is placed on proven functionality that has been accomplished with the selected software technologies. However, a complete visualization tool for Rosetta is beyond the scope of this project - the main idea is to have a basis to continue further developments(using the knowledge gained and tools learned), if the solution proves to be viable.

Why is visualization necessary?

"Like good writing, good graphical displays of data communicate ideas with clarity, precision, and efficiency."
"Like poor writing, bad graphical displays distort or obscure the data, making it harder to understand or compare."

(Michael Friendly, Statistical Consulting Service and Psychology Department, York University. For more details see [1])

Scientific visualization and virtual reality are very important for developing functional prototypes quickly. Simulation and visualization are powerful tools that are time-saving and cost-effective. Space missions are becoming more complex and simulating and/or visualizing mission parameters and the overall flow can reduce risk and help overcome testing and evaluation challenges.

Moreover, visualization is also used in a variety of supporting roles of space missions including simulation and rehearsal of planned operations.

Another important use for visualization tools is in mission planning. Three dimensional modelling and visualization of the main protagonists and the environment is necessary in order to optimize the spacecraft's activities within a limited time frame.

Application Domain

The application domain is software development with a focus on Computer Graphics and Visualization. The data is coming from instrument measurements and other sources which are included in the domain of Space Science. Note that only test data is currently used. The current version of the prototype does NOT include the Map/Image handling component. However, when this will be included the domain will also cover some aspects of Image Processing.

However, visualization can be thought of as an integration of the following domains:

- **Computer Graphics**
  Efficiency of algorithms (CG) versus effectiveness of use (V).
- **Computer Vision**
  Mapping from pictures to abstract description (CV) versus mapping from abstract description to pictures (V).
- **Image Processing**
  Mapping from data domain to data domain (IP) versus mapping from data domain to picture domain (V).
- **(Visual) Perception**
  General and scientific explanation of human abilities and limitations (VP) versus goal oriented use of visual perception in complex information representation.
- **Art and Design**
  Aesthetics and style (AD) versus expressiveness and effectiveness of data (V) .

1.1 Mission Details

The project's main purpose is to be successfully used in ESA's Rosetta Mission. Rosetta is a spacecraft launched in 2004. After 7 years of Cruise phase with 4 planetary swing-bys and 2 Asteroid fly-bys it will enter into Deep Space hibernation in June 2011. Its final target is comet 67P/Churyumov-Gerasimenko. The estimated delivery is on 10 November 2014.

It is the first attempt to land on a comet. Scientists hope this will shed light on how life originated in our Solar System.

The name Rosetta comes from the Rosetta Stone which helped historians understand and decipher Ancient Egyptian. Similarly, ESA's Rosetta spacecraft will unlock the mysteries of the oldest building blocks of our Solar System - the comets. It will allow scientists to look back 4600 million years - when no planets existed and only asteroids and comets surrounded the Sun. The figure below is taken from [2] - consult this resource for more information about the mission.

Figure 1.1: Rosetta and Comet 67P

Rosetta has three components:

The Orbiter

This is the main spacecraft to which all other subsystems and equipment are attached. Dimensions: 2.8 x 2.1 x 2.0 metres.

The Instruments

- **ALICE** Ultraviolet Imaging Spectrometer
- **CONSERT** Comet Nucleus Sounding
- **COSIMA** Cometary Secondary Ion Mass Analyser
- **GIADA** Grain Impact Analyser and Dust Accumulator
- **MIDAS** Micro-Imaging Analysis System
- **MIRO** Microwave Instrument for the Rosetta Orbiter

- **OSIRIS** Rosetta Orbiter Imaging System
- **ROSINA** Rosetta Orbiter Spectrometer for Ion and Neutral Analysis
- **RPC** Rosetta Plasma Consortium
- **RSI** Radio Science Investigation
- **VIRTIS** Visible and Infrared Mapping Spectrometer

The Lander

After entering orbit around Comet 67P/Churyumov-Gerasimenko in 2014, the spacecraft will release a small lander onto the icy nucleus, then spend the next two years orbiting the comet as it heads towards the Sun. The lander weighs 100 kilograms.

Cost

The cost of the Rosetta project is around 1000 million Euros (including the scientific instruments funded by national agencies).

Chapter 2

Project Objectives and Specifications

2.1 Project Objectives and Motivation

The project's objective is to investigate the 2D/3D innovative software techniques for the visualization of the simulated planning elements coming from the system. The motivation behind this project is that visualization software shall provide the flexibility to take quick decisions to adapt to new scenarios - achieved by using the feedback from executed operations. The outcome of this endeavour is a software PROTOTYPE, not a complete application - it should demonstrate that a complete application can be built using the selected tools and provide a basic software library in order to make it easier to build visualization tools in this specific context. Future applications should make use of this library after it reaches a certain level of maturity.

Although there are some good visualization tools out there - a complete package that includes both the 3D and 2D visualization components mentioned before does not exist. Furthermore, most visualization in the space sector deals with Earth visualization. This project aims to solve this problem.

2.2 Functional Requirements

- The system shall allow the user to create a new simulation.
- The system shall allow the user to select a specific time interval for simulation.
- The system shall provide a function to pause the simulation
- The system shall allow only one active simulation

- The system shall provide graphics configurability options: changing the resolution, bpp etc.
- The system shall be able to display the following primary object types: Sun, Solar System(Cosmic) Objects, Spacecraft, Atmosphere, Mask.
- The system shall provide different kinds of masks corresponding to their respective instrument views. These masks will cut out a portion of the view.
- The system shall be able to display the following child objects: coordinate arrow, pointing arrow, spacecraft boresight, trajectory line, trail, region of interest, caption text, landmark, distance line.
- The system shall allow the user to plot data using charts while the simulation is running. The data shall be in sync with the 3D world.
- The system shall allow the user to hide/show both primary and child objects.
- The system shall provide a function to modify rotation and translation speed during runtime.
- The system shall also have a zoom function for both the main 3D world and the charts.
- The system shall provide functionality for creating new instrument views and chart plots.
- The system shall provide instrument views(secondary 3D views).
- The system shall display the current universe.
- The system shall allow authenticated users to edit the 3D scene and database.
- The system shall load trajectories, object creation data and most position changes using a database.
- The system shall have a user-friendly GUI with multiple visualization options: 3D Visualization - main scene and instrument views, 2D visualization - charts and maps

2.3 Non-Functional Requirements

- **Availability:** The system shall be operational at least 95% of the time. The down-time shall be used for updating the database and making changes to the 3D scene.
- **Efficiency:** The system shall be efficient enough to allow normal system operation on mid-end machines without great graphic processing power. Ideally the frames per second should not go below 60 with only one view. The minimum fps for one view

shall be 30 - for a benchmark system with an nVidia Geforce 9600GT M graphics card, 4GB DDRAM and an Intel Core 2 Duo CPU running at 2Ghz.

- **Flexibility:** The system shall be very easy to extend - developers will be able to use existing classes from the library without necessarily having access to the source code.
- **Portability:** The system shall be easy to port from one platform to another - there will be no required action, other than copying and running the necessary installation files and having a version of Java installed. The system shall run on Windows, Linux, and Mac OS X. Of course the platform should also have graphics drivers which enable 3D Acceleration.
- **Integrity:** Access shall not require authorization for regular use. However - a management area shall have restricted access because it will allow the administrator to change the contents of the scene and other critical factors.
- **Performance:** The start-up shall be at most 5 seconds. Ideally the frames per second should not go below 60 with only one window. The minimum workable fps will be around the 15 fps mark. 2D Visualization shall run smoothly, with no visible stuttering.
- **Reliability**: The software shall be reliable, and allow users to run it any time. The system shall allow for simulation restarts or for creating other ones, once the current simulation has been closed. Performance should not be affected negatively over time, but, in contrast, the system shall be updated when necessary - in order to improve reliability.
- **Reusability:** The software library shall be reusable and it shall aid the development of similar products(such as visualization software for a similar mission).
- **Robustness:** The system shall be very robust - exceptions and errors will be handled, the user will be notified of any problems and system operation can be resumed afterwards if a corrective action is taken by the user. Of course, robustness will be limited on a low-end machine which does not have good 3D acceleration capabilities - problems might appear with performance and perhaps older drivers.
- **Scalability:** The software shall run smoother on more powerful hardware. Moreover, visualization will be greatly improved by scaling up and providing several monitors to visualize different windows at the same time at a good resolution.
- **Usability:** The system is easy to learn and understand. After about 10-30 minutes the regular user shall be able to take advantage of most features. *Note:* Currently the UI is not complete, but a detailed usability study has been performed for a possible UI as a project for the ”User Interface Design” Course.

2.4 Technological Constraints

The primary constraint was to develop the product using Java, for the following reasons: the resulting product is cross-platform, it can easily be integrated into another(pre-existing) RPC application as a component. The consequence of this is that 3D software packages have been severely restricted - since most 3D development is NOT done in Java - the main reason for this is probably the lower speed compared to a language such as C++.

Another constraint was to use the Eclipse IDE as the main development tool. Furthermore, all selected libraries needed to be free and open-source. For more information please consult section 5.

2.5 Target users

The user base is represented by the ESA Rosetta Team. The product has no practical use outside of this user base, however it can be learned and used by most regular users. Of course the data only has meaning to people who are working in the team - or perhaps other mission stakeholders.

2.6 Problems encountered

Many problems have been encountered while selecting the technologies to develop the prototype. This is mainly due to the fact that Java had to be used as the main development language. Because of this, libraries which did not have native support caused problems while testing or were scarcely documented, thus needing to be abandoned.

Other problems steamed from the fact that a *Game Engine* was used as a developing environment, not a scientifically oriented visualization tool. The main problem is that a game usually runs in full screen mode, while a visualization tool usually runs a number of windows concurrently. This means that support for multiple windows is limited and harder to implement using a game engine.

Most other problems were performance related - striving to achieve a *good enough* performance for the prototype. This problem is partly still present in the current version of the prototype.

2.7 Project Structure and Organisation

The project was divided into three parts

1. Together with the supervisor(s) and subject matter expert(s) decide on the scope.
2. Research possible approaches and perform a comparative study.
3. Implement a prototype.

The project scope can be defined as "The work that needs to be accomplished to deliver a product, service, or result with the specified features and functions." - we basically decided on the fact that there were going to be two important components of the work effort: research available technologies and implementing a prototype using the selected technologies.

The product scope is usually defined as: "The features and functions that characterize a product, service, or result." - it has been decided and transformed into the form of functional requirements, which were described in a previous section. However, an important aspect is that the creation of a software library on top of the 3D and 2D libraries selected has been discussed and agreed on. This library should allow future visualization applications to be build much easier than using the standard libraries - the library should have functions tailored to the specific needs of visualization in the current context. Moreover, this library should also be useful for similar mission types(any similar mission - e.g.: Venus Express).

An important part of the project was finding suitable technologies to implement the required prototype given some constraints. The visualization can basically be broken into two categories: 3D Visualization and 2D Visualization. The 2D visualization, in turn, consists of two parts: chart plotting, map/image handling. The 3D Visualization shows the mission's scene: geometries, materials, interactions between objects etc.

A working prototype has been implemented to demonstrate that both the 2D and 3D visualization components can be integrated into the same application. The prototype also implemented most features at a minimum level. Again, the purpose of this project is to have a prototype implemented using some visualization technologies in order to demonstrate their power in creating a fully-fledged visualization tool - NOT a complete application.

Chapter 3

Bibliographic Research

Due to the nature of the project the bibliographic research is quite diverse and covers a lot of different aspects of the process that lead to the creation of the final prototype.

The information gathered from the bibliography helped to kick-start this project. From theoretical foundations and visualization algorithms and information to testing and validation, all topics have been researched in order to get a better idea of what needs to be done, why it needs to be done and how it needs to be done.

3.1 Sources On Visualization

Why visualization is needed is important to understand how to create a good visualization tool. A good deal of introductory information on this topic can be found in [1]. Moreover, useful information about deep space visualization and corresponding experiences are presented in [3]. More information about visualization used in the space field can be found in [4] as well as an example software that allows realistic simulation, presentation and evaluation of platform selection, flight profiles and range asset placement. Another good example of a visualization tool is the *Terrestrial Planet Finder* described in [5] - it addresses visualization issues that appear in that context.

Moreover, an important method of calculating the lighting for customly created objects is presented in [6]. Explanations are given on how Gouraud shading can be implemented in OpenGL. The solution ca be easily modified and applied in Java, using the rendering engine. More details on illumination models are presented in [7]. It is important to understand that the Phong illumination model has three primary components: ambient, diffuse and specular lighting components.

Finally, an important part of computer graphics are matrix operations which are used in

a lot of instances. How to rotate a vector into and another and how to rotate a vector along an axis with a specified angle are two important methods used in the application. The theoretical background for these operations is presented in [8].

Moreover, the *jMonkeyEngine* uses shaders for defining material properties. These are little programs that define how the data shall be processed by the GPU. For detailed information about the process and GLSL in general please consult [9]. The appendix can also be consulted for an example of a shader written for jMonkeyEngine3.

3.2 Sources On Technologies

The literature was searched for suitable graphics technologies, capable of building a 3D visualization interface for scientific missions. The constraints had to be taken into account and thus the use of *OpenGL* - the most popular graphics library, was abandoned as a possible solution. However, [10] was still used for good information regarding different graphics processes, since the graphics engine used is built on top of OpenGL. Afterwards, *VTK* was considered and proved to be a viable option. Using [11], test applications were created in order to test how easy it is to use this technology given the project's constraints. However, progress was not very fast and a lot of the tutorials we had access to including [12] and an official book(see [13]) were pretty out dated - though most information still applied to the current version of VTK, support for Java was poor. However, through the official VTK website(see [11]) one could sign up for an official mailing list. This is a plus, but still not very convincing.

The final technology used, *jMonkeyEngine3* was built for Java and all documentation was targeted towards Java programmers that were trying to create games or 3D visualization software. The official website is very resourceful. It can be accessed by going to [14]. Moreover, in [15] one can find all of the jMonkeyEngine classes explained in standard *JavaDoc* format. Furthermore, the official forums(see [16]) are also very useful as one can see the past problems that others have had with this technology, or post their own problem. Finally, very good articles can be found in [17] - both about the theoretical foundation and about actual implementation of functionality.

To see what other 3D graphics APIs were considered for use please check: [18] and [19] for lower-level APIs or [20], [21], or [22] for high-level APIs or game engines.

For the 2D image manipulation part *Java2D* and *ImageJ* were considered. The official web page for Java2D information is well written(see [23]). In [24] a lot of information can be gathered about ImageJ, its functionalities, advantages and disadvantages.

The charting libraries each have their respective official web pages with appropriate documentation: for *JFreeChart* this can be accessed at [25], while for *JChart2D* at [26]. There

is also a commercial version of officially written documentation for JFreeChart available in [25] - for a price.

Of course, since development was carried out using Java, the official source of documentation at [27] was continuously consulted .

3.3 Related Work

It is hard to find an application that specifically has the features that must be implemented in this prototype project. This is because different types of visualization are required to be integrated into the same application. Moreover, most space visualization software is mostly concerned with the Earth and not deep-space missions such as Rosetta. However, some applications have been found to do one job very well and have been a source of inspiration in particular areas.

3.3.1 Celestia

Celestia is a very complex 3D space simulator. It allows users to travel through the universe at any speed, in any direction and at any time in history. It can also display orbital paths of different objects including different types of planets and spacecraft. This is perhaps the best example of a popular 3D space simulator with different visualization capabilities.

A lot of add-ons are also being developed for Celestia including new types of objects, spacecraft, high-resolution textures and others. The *Celestia Motherlode* is a repository for such celestial objects. One of the sun textures available in the motherload has been included in this prototype - most items can be freely downloaded and used under some conditions. The 3D model of Rosetta is also available in [28]. These and other components which can possibly used in the future come as courtesy from [29] and the motherlode.

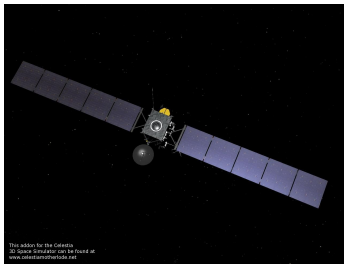

Figure 3.1: The Rosetta Probe - used in the current project

Fictional objects from popular Sci-Fi franchises can also be included in Celestia, as add-ons.

3.3.2 JSatTrack

Is a satellite tracking program written in Java. It allows to predict the position of any satellite in real-time, past or future. For more information please consult the official resources at [30]. This was used in order to have a model for the image handling part. Unfortunately, this part of the project has not been implemented up to this point. However, it may still prove to be a powerful resource in the future, when and if the map handling component will be added to the prototype. Moreover, *JSatTrack* also has a friendly user-interface. This was used as an inspiration for developing part of the GUI in this project.

3.3.3 Other applications

A few other applications have been studied, but they will not be mentioned here due to confidentiality reasons - they are not publicly available. An important aspect to note is that the analyzed visualization tools were strongly geared towards Earth visualization and were missing a lot of functionality needed for deep space missions.

3.4 Sources On Testing

Some sources have been used to clarify and distinguish between the testing types: code level, functional and system-level testing. Great information is presented in [31]. Moreover, the type of testing can also be of three other types: white-box testing, gray-box testing and black-box testing. White-box testing refers to the internal components and workings of the software, while black-box testing refers to the software's functionality. Gray-box testing is a combination of both. Information on this topic and a comprehensive tutorial on manual testing is available in [32].

Chapter 4

Analysis and Theoretical Foundation

4.1 Conceptual Architecture

Since the most important function of the prototype is to work as a library for future projects, the conceptual architecture can be modelled using the **Layers** architectural style.

Each layer will provide services to the one above it and will use services from the one below it:

- At the lowest layer, we have the graphics driver which together with the Operating System provided the needed graphics capabilities to be used by the prototype(e.g.: 3D Acceleration).
- At the next layer, we have the *OpenGL API* which is a graphics library used to interface between hardware and software - we write low-level software that controls the functions of the GPU in order to draw the required graphics.
- Next, we have the *JMonkeyEngine 3(JME3)* library, which is actually a 3D Game Engine built for creating advanced graphics/games using Java. It is a high level-library and it is built on top of *LWJGL* - which is an *OpenGL* wrapper for Java.
- The following layer is the most important since it represents our prototype library - *RosettaViz*. This uses *JME3* functions in order to create useful features - primitive types for missions, flexibility etc.
- Next we need to use our created library in an actual application. For this, we create a database which will supply the system with required data for displaying the objects and their interactions on screen.

- Finally, we have our application GUI. This allows the users to interact with the system easily and to make changes to the database that can change the scene.

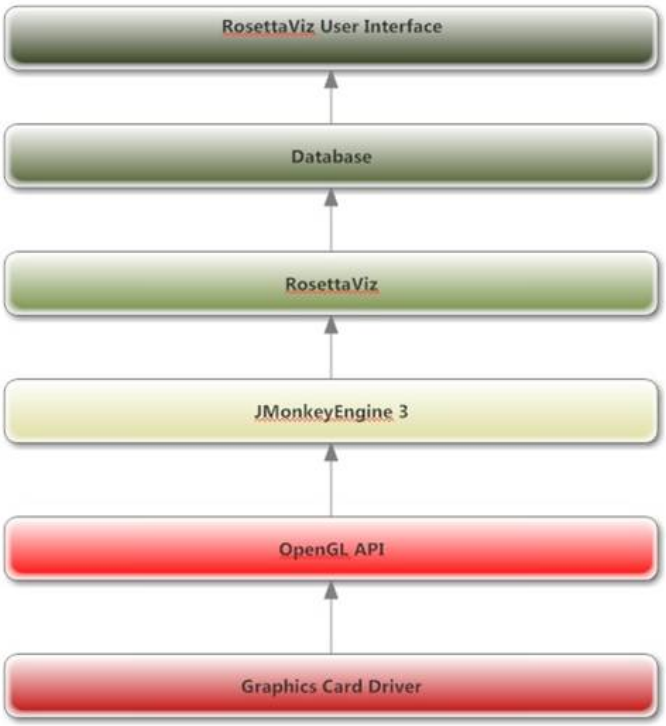

Figure 4.1: Layered System Architecture

4.2 Use Case Model

In the following the Use Case Model will be presented and each use case will be briefly described.

Use Case Descriptions

- **Use Case:** Create New Simulation
  Level: User-goal level
  Primary Actor: User
  Preconditions

  1. The application is running and the database is accessible.
  2. Another simulation has not been started/opened.

  Main Success Scenario

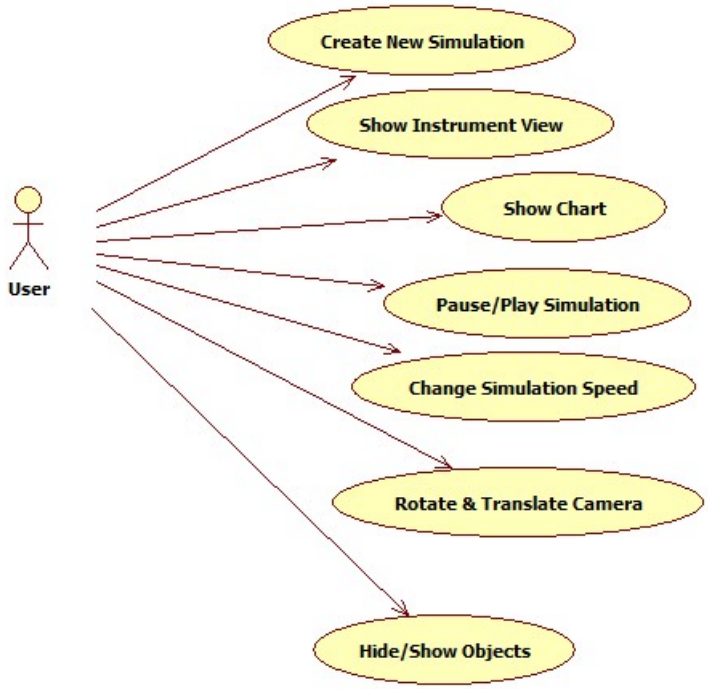

Figure 4.2: Use Case Model Diagram

1. The user requests to create a new simulation.
2. The system will display the date selection screen. The user can select a start and an end date for the simulation. Dates are submitted and system verifies their validity.
3. The system responds by opening the display settings panel.
4. The user will set display settings: resolution, fullscreen option, bpp etc.
5. The System responds by opening the 3D Visualization window.

Extensions

2a. Invalid date entry:

(a) The system responds by sending an error message.

(b) After the user acknowledges the error, he/she can retry step 2 from the basic flow.

2b. Date interval does not correspond to real data:

(a) The system responds by sending an error message and informing the user

of the available(valid) data.

(b) After the user acknowledges the error, he/she can retry step 2 from the basic flow.

2-5. Abort the simulation creation process:

(a) The system returns to the initial application state(all the set options are discarded and the simulation is not started).

4. Unsupported display settings:

(a) The system responds by sending a specific error message or by automatically adjusting graphics options to some lower quality settings which can be interpreted by the machine.

Postconditions

1. 3D Visualization window is displayed.

- **Use Case:** Show Instrument View
  Level: User-goal level
  Primary Actor: User
  Preconditions

  1. The application is running and the database is accessible.
  2. The simulation has been started.
  3. At least one instrument view must exist.

  Main Success Scenario

  1. The user goes to the instrument/offscreen view section.
  2. System displays available views.
  3. The user starts the operation by some action.
  4. The system responds and displays the view.

  Postconditions

  1. The selected instrument view is displayed in a separate 3D window.

- **Use Case:** Show Chart
  Level: User-goal level
  Primary Actor: User
  Preconditions

  1. The application is running and the database is accessible.

2. The simulation has been started.
3. At least one instrument view must exist.

Main Success Scenario

1. The user goes to the charts section.
2. System displays available charts.
3. The user starts the operation by some action.
4. The system responds and displays the chart.

Postconditions

1. The selected chart is displayed in a new 2D chart window.

- **Use Case:** Pause/Play Simulation
  Level: User-goal level
  Primary Actor: User
  Preconditions

  1. The application is running and the database is accessible.
  2. The simulation has been started.

  Main Success Scenario

  1. The user requests to pause the simulation.
  2. System pauses the simulation and waits for futher action from the user.

  Extensions

  1a. The simulation is already paused:

  (a) The user uses the same control.

  (b) The system resumes the simulation.

  Postconditions

  1. The simulation is paused.
  2. The simulation is running.

- **Use Case:** Change Simulation Speed
  Level: User-goal level
  Primary Actor: User
  Preconditions

  1. The application is running and the database is accessible.

2. The simulation has been started.

Main Success Scenario

1. The user requests to change speed: slower or faster.
2. System responds by adapting the speed.

Postconditions

1. The speed of the simulation changes.

- **Use Case:** Rotate and Translate Camera
  Level: User-goal level
  Primary Actor: User
  Preconditions

  1. The application is running and the database is accessible.
  2. The simulation has been started.
  3. The camera must exist.

  Main Success Scenario

  1. The user selects main window.
  2. The user requests to move and rotate the camera using the keyboard and mouse as input.
  3. System responds and applies the requested operation(s) to the camera.

  Postconditions

  1. The main camera is moved and rotated to the new position.

- **Use Case:** Hide/Show Objects
  Level: User-goal level
  Primary Actor: User
  Preconditions

  1. The application is running and the database is accessible.
  2. The simulation has been started.
  3. At least one object must exist.
  1. The application is running and the database is accessible.
  2. The simulation has been started.
  3. At least one object must exist.

Main Success Scenario

1. The user goes to the objects section.
2. System displays existing objects.
3. The user drills down to the specific object they want to change.
4. The system displays show/hide option.
5. The user selects the operation.
6. The system responds by modifying the corresponding object's visibility.

Postconditions

1. The selected object and its component(child) objects are shown or hidden from view.

4.3 Visual Object Structuring

In order to create and model all the needed object types their structure and properties had to be taken into account. First, and foremost, three main visual object types have been identified: Sun, Solar, Spacecraft(SPC).

It is important to note that all visible objects on the screen will have two important properties: a geometry which will define their structure and a material which will define their final appearance: textures, colors, shading etc.

Particle emitters are a bit different and have some constraints on the used geometry and material. An example of this is the *comet atmosphere*. It is created from a set of predefined effects and these entities are defined by a number of properties - these properties define the geometry, while the material needs to have a specific particle emitter texture applied(a different texture or material property cannot be applied).

1. **The Sun Object**

   Usually only one Sun object is used - because this library will mostly be used to model parts around our Solar System. It is the entity that emits the light in the scene. Thus, light will be modelled as coming from the Sun object's direction. In most cases, the Sun geometry is not important and it can be hidden from view. However, the light must be always turned on.

2. **Solar System Objects**

   These represent celestial objects such as planets, asteroids, comets, space dust, etc. They can have landmarks placed as points of interest. Furthermore, a region of

interest can be drawn on an object. In case of some objects they can also have particle affects attached: space dust, debris, atmosphere etc.

3. **Spacecraft Objects**

   This class of objects actually represents all types of man-made celestial objects: satellites, shuttles, rovers etc. These typically have an antenna boresight. Most man-made celestial objects also have some instruments used for testing, data acquisition and experimentation.

Differences between the object types can be extended further if required.

4.3.1 Component Objects

- **Coordinate Arrows**

  These arrows show the X, Y, Z axis with the origin in the current object. Arrows need to be rotated in their respective coordinates, in order to match the cartesian axis.

- **Pointing Arrows**

  These arrows have one end in the origin of one object. The other end points to a secondary object, we are interested in. For example, we can point towards the Sun so we always know where it is and subsequently remove it from view, if it is unnecessary.

- **Atmosphere**

  A special effect used to model a solar object's atmosphere. In case of the Rosetta mission, it can be used for the comet.

- **Boresight**

  The antenna boresight is the axis of maximum gain of a directional antenna. For most antennas the boresight is the axis of symmetry of the antenna.

- **Caption Text**

  This text can show an object's name or other important information. Orthographic projection is used to display the text like a 2D object. Rotation should not affect caption text. Moreover, the text shall move with the object.

- **Landmark**

  A landmark can be placed at a particular point on a 3D celestial object. We have the spherical coordinates at which we want to place this landmark, so we need to convert them to cartesian coordinates to draw this object.

- **Mask**

  A mask is a 2D object placed on top of all other objects in order to obscure some parts of the screen.

- **Distance Line**

  We just draw a line between the center of one object to the center of another. It must represent the shortest distance between the two points.

- **Trajectory Line**

  In some cases we want to see the whole trajectory of a SPC object for a particular time interval, to have a broader image on what is going on. We need to have enough points displayed in order to display a smooth continuous line.

- **Trail**

  The trail is simply the trajectory line displayed as the object progresses. In other words, it shows the path already traversed by a particular object. The trail must be updated in real time with the new position locations.

- **Region of Interest**

  This is a region which will be coloured differently on an object to indicate that there is some interest in it a particular point in time. Pixels can be drawn over the 3D object using another colour.

4.3.2 Flexibile Object Creation

The idea was to include all objects and their associated data(rotation data, position data etc.) inside a database or file and create them dynamically when the application starts, without having to specify this manually - in other words, whatever objects are found in the database are created at runtime and shown in the scene. Of course, the associated data is affected by the simulation time interval chosen by the user when they create a new simulation.

In order to achieve this dynamic and flexible creation of objects we need to create some data structures that get the required data from the data source and pass it on to the rendering engine to do its job.

Furthermore, the application should do a matching between what it finds in the data source and any class names included in the application - id does not care what type of object it is, as long as it exists, it will automatically create and display it in the scene.

This gives great flexibility to developers which can easily add their own content which will be treated in the same way as pre-defined objects.

4.4 Computer Graphics Projections and Spaces

Projections are used to transform a 3D scene into 2D screen coordinates. There are two important types of projections in computer graphics:

- **Orthographic Projections**
  Also called parallel projections. They use a square or rectangular viewing volume. Objects outside of this volume are not drawn. All objects that have the same dimensions appear the same side invariant of their location - there is no sense of perspective(depth). Orthographic projections are usually used to add text or 2D overlays on top of the 3D scene. The viewing volume is specified by the following clipping planes: far, near, left, right, top and bottom.

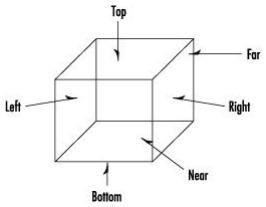

Figure 4.3: Orthographic Viewing Volume

- **Perspective Projections**
  This type of projection adds the effect that distant objects appear smaller then nearby objects. Objects in front appear close to their original size. The viewing volume is represented by a *frustum*. Realism for 3D animation and simulation is achieved by using this projection.

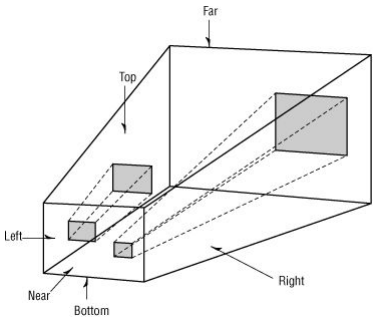

Figure 4.4: The Frustum

For more information on projections and general computer graphics concepts consult the first chapter of [33]. Figures 4.4 and 4.3 are taken from this resource.

Transforming 3D object coordinates to 2D sceen coordinates means going through the following process:

The object space coordinates are passed to what is known as a *vertex shader*(see section 4.5.4 for more information on shaders). Its position in projection space needs to be computed. This can be accomplished by transforming its position by the *WorldViewProjectionMatrix* - a combination of the *World*, *View* and *Projection Matrices*. For information on what a matrix is see section 4.7.

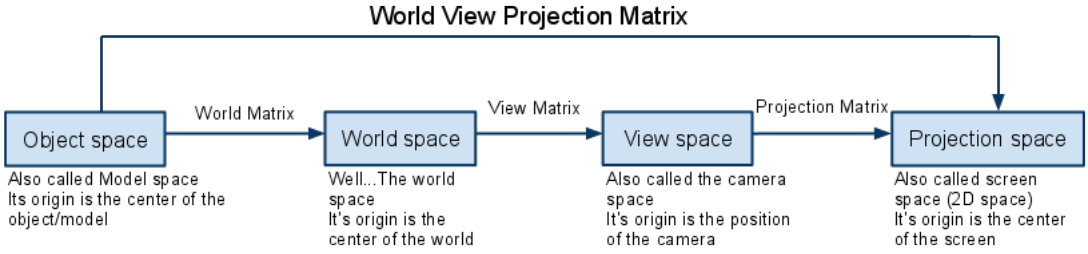

Figure 4.5: World View Projection Matrix (courtesy of [14])

4.5 JMonkeyEngine Theoretical Foundation

The most important concept used by the rendering engine is the *scene graph*. It represents the virtual space that the world happens in. The objects in the scene graph are called *spatials*. Spatial have a location(that can be translated), a rotation and a scale. Of course, three dimensions make up the virtual space - given by the classic X, Y and Z coordinates. The coordinate system is such that Y points upwards, X points to the right and Z points towards the user.

The basic operation that can be performed on any spatial are translation, rotation and scaling. Each operation can be applied simultaneously for one or more axis:

- **Translation**

  Moving the object, across one or more axis, to a new location in (X,Y,Z) coordinates

- **Rotation**

  Rotating the object, across one or more axis. Great care must be taken with cumulative applications of rotation operations, since the effect might be different than the one expected.

- **Scaling**

  Modifying the objects size on one or more axis. If the scaling is performed on all three components the spatial will be scaled uniformly.

Another important concept is that of *geometry*. In *JME3*, a geometry is the type of spatial that is actually visible on the screen. It has a *mesh* to define its form and a *material* to define its appearance.

Since the engine is using a scene graph, we must have nodes. A *node* is a spatial that can have other spatials as children. The children of the node are moved and rotated relative to their parent node. The scene graph has a *root node* to which all other nodes MUST be connected.

Nodes are used to organize the scene. Nodes can have other nodes as children. These nodes, in turn, can have other nodes and spatials as children. This helps organize the scene graph both visually and spatially. We can group objects together to move them(or apply some other operation), but we can also organize the scene graph logically.

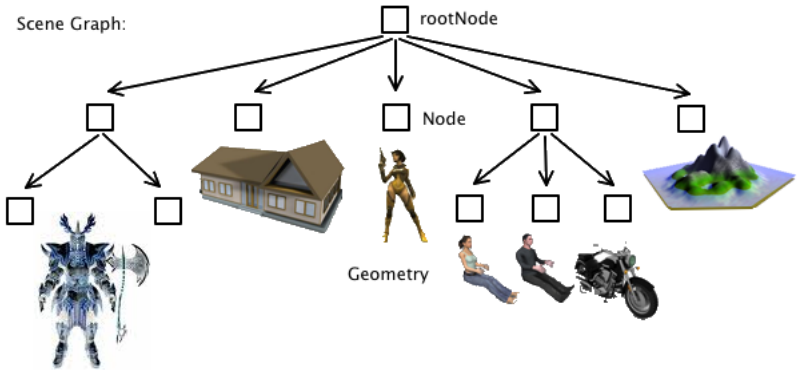

Figure 4.6: JME3 Scene Graph Example (courtesy of [14])

A visual representation of a scene graph with connected nodes, including both spatials and geometries is presented in figure 4.6.

A very important operation in computer graphics is *culling*. In *JME3* culling consists in hiding spatials and nodes from view. In other words, some geometries will not be rendered by the graphics card. The culled geometries are still processed as part of the scene graph but all rendering related operations are suspended.

There are three cull states: *always*, *never* and *inherit*. The *always* value means that the specific entity will be always culled. *Inherit* means that all the children of the current node shall inherit its culling property - we shall get to a point in the hierarchy where one of the states must be always or never. As its name implies, the *never* value means that the node will never be culled.

For more information please consult JMonkeyEngine3 *Scene Graph for Dummies*, accessible in [17].

In the following sections, some theoretical aspects of the scene graph will be presented. It is important to note that although there is actually one large scene graph, it can logically

be broken into two parts:

- General, which describes the workings of the whole scene up to the abstraction of a visual object
- Object, which describes the scene graph at the level of a visual object and further in depth

4.5.1 General Scene Graph Structure

We need to consider what kind of visual information we want to store at a *general*(global) level. First and foremost, we need lighting so we can see the objects in our scene. Next, considering the fact that we need secondary or offscreen instrument views, we need to represent them at some level as well - in order not to interfere with our main window.

Finally we can start to add the actual objects(Sun, Solar, Spc and more).

4.5.2 Object Scene Graph Structure

First, we need to consider the fact that not all component objects should be rotated with the primary visual object(Sun, Solar, SPC) it is associated to. We need to model two kinds of behaviours - component objects that rotate and translate with the primary object, or ones that only translate with the object. Next, we just need to add the component objects to their corresponding locations - with or without rotation. These objects will be siblings on the same hierarchical level, beneath the primary object. One exception is the caption text which theoretically is independent of the object and only needs to translate along with it or another node.

4.5.3 Geometry Properties

Each geometry has a polygon mesh that defines the geometry's structure. There are three types of meshes that can be used in *JME3*:

- **Shape**
  Shapes are built-in simple meshes such as cubes and spheres. They can be used to build more complex shapes and are built-in into the engine.
- **3D Model**
  JME3 provides the functionality of importing 3D models and scenes created with a modelling tool directly into your application.

- **Custom Mesh**
  A custom mesh must be created, if some input data is used to build the geometry(not imported from another program) programmatically. Each mesh can have one of multiple modes which represent primitives differently:

  1. **Hybrid**
     Represents a mode in which a combination of other modes is used in other to interpret the data.

  2. **Points**
     Each vertex is a single point on the screen. A point is the simplest primitive. Note that the point size can be changed.

  3. **Lines**
     Each pair of vertices defines a line segment. A batch of lines should consist of an even number of vertices, one for each end of the line segment.

  4. **LineStrip**
     A line segment is drawn from the first vertex to each successive vertex - used for creating continuous lines. It can be thought of as a *connect-the-dots* parallel.

  5. **LineLoop**
     Used to build a closed line - just like *LineStrip*, but the first and last vertices are connected to form a loop.

  6. **Triangles**
     Every three vertices define a triangle. Can be used to construct a solid polygon with the triangle as the primitive building block. The triangle is the simplest solid polygon. and rasterization hardware processes triangles much faster then any other polygons.

     An important characteristic of triangles is called winding. It refers to the order in which the triangle vertices are connected - it can be counter clockwise(front facing) or clockwise(back facing). This is important because, by default, the game engine is culling the back faces for optimization, so great care must be taken when connecting vertices to form triangles. The back faces shall be inside of the object and thus will usually not need to be rendered.

  7. **TriangleFan**
     Triangles fan out from an origin and they share adjacent vertices - this represents a group of connected triangles that fan around a central point.

  8. **TriangleStrip**
     Vertices are shared by triangles along a strip. A lot of time can be saved by drawing using this mode if the shape we wish to draw has several(or more) connected triangles. This method has some advantages over drawing the triangles

separately. First, only one extra vertex is required for each new triangle - saves storage space. Secondly, this approach yields better performance and higher bandwidth savings.

All except Hybrid modes are equivalent to using the seven *OpenGL* geometric primitives used to draw solid geometry, namely: *GL_POINTS*, *GL_LINES*, *GL_LINE_STRIP*, *GL_LINE_LOOP*, *GL_TRIANGLES*, *GL_TRIANGLE_STRIP*, *GL_TRIANGLE_FAN*. For more details on this topic please consult the *OpenGL SuperBible*.

Meshes can be created using *VertexBuffer*s as input data. These are buffers for storing the position, the colours, the normals, texture coordinates and others.

4.5.4 Material Properties

Materials are needed because shapes cannot be rendered if their surface properties are not known. A colour and/or texture must be applied to a geometry in order to make it visible. Each material must have a material definition - properties which define its appearance. The rendering engine comes packed with a number of very useful definitions, but others can be created as well.

Both *illuminated* and *unshaded* material definitions are supported. Unshaded materials look very abstract, while Phong illuminated materials look more naturalistic.

There are two types of unshaded materials used in this project. First, standard non-illuminated materials, used for simple coloring and texturing. Also supports transparency. This is used for arrows, lines and other such objects which do not require complex illumination. Illumination makes a scene look much better, but it also adds a great burden of work for the GPU since lighting uses a lot of resources. Secondly,the *particle* material definition which is used in combination with texture masks - used to create particle emitters.

The *Phong illumination model* is a linear combination of three components: ambient, diffuse and specular. The ambient component is an approximation to global illumination - indirect illumination, such as reflections from walls. The ideal diffuse reflectors work according to *Lambert's cosine law* which states that: reflected energy from a small surface area in a particular direction is proportional to the cosine of the angle between the direction and the surface model. A specular component is usually present if the surface is shiny or very smooth. The Phong reflection model can be mathematically stated as being a combination of the three components described above: $I_v(\lambda) = I_a k_a(\lambda) + I_i k_d(\lambda)(L \cdot N) + I_i k_s (R \cdot V)^n$. Please consult [7] for a complete description of the model.

In our game engine, illuminated materials require a light source that must be attached to at least one of their parent nodes. They are darker on sides that face away from the light source. They use the Phong illumination model. This *Lighting* material can be used

together with *DiffuseMap*, *SpecularMap* and *BumpMap* textures. It supports shininess, transparency and diffuse, specular and ambient colours.

The available light source types are *SpotLight*, *PointLight*, *AmbientLight* and *DirectionalLight*. The Sun will have a directional light, since this light has no position, only a direction and can be thought of as being *infinitely* far away - since the Sun is actually very far away, this is perfect to simulate the lighting conditions.

Every material in JME3 uses shaders. The material definitions are actually shader programs. These programs specify which vertex and fragment shader to use.

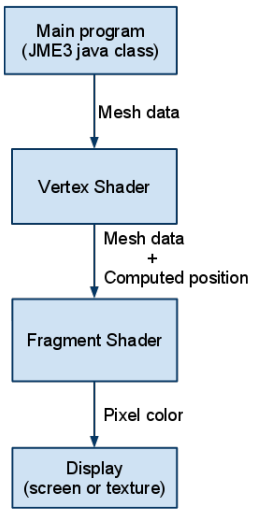

Figure 4.7: JME3 Shaders (courtesy of [14])

The vertex shader is executed once for each vertex in the view and computes the screen coordinates for the vertex, while the fragment(pixel) shader is executed once for each pixel on the screen and computes the colour of the pixel. A simplified call stack is shown in figure 4.7. For an example implementation of a simple shader please consult appendix A.

4.6 Gouraud Shading

This method is used to compute the shading for *custom mesh* objects. The normals of these objects must be computed in order to use this shading method.

It is also called smooth shading - gives a more realistic touch to the object. This is because the illumination has gradual variations. In order to compute the shading we only need to

calculate the normals of the vertices of our 3D object. The lighting part is automatically handled by *OpenGL* and thus by the used rendering engine - once the normals are known. We need to calculate *the average of the normals of the polygons adjacent to the vertex*. To calculate a vertex's normal it is enough to make the arithmetic average of the normals of the polygons adjacent to the vertex. Of course, in our case a polygon is actually a triangle and the calculation of such a normal is explained in section 4.7.1.

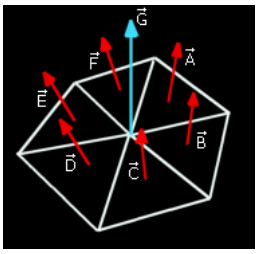

Figure 4.8: Gouraud Shading

For more information on the whole process consult [6].

4.7 Mathematics

In this section a description of the most important mathematical(algebraic) formulae is given. Almost all operations that will be presented are vector operations. These are extensively used in a visualization context.

A 3D Vector is an element of 3D space. It combines the number of elements in the vector space - 3 in this case: (X, Y, Z). The three values combined represent two important aspects: magnitude and direction.

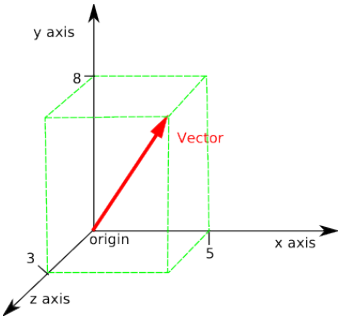

Figure 4.9: A Vector in 3D Space

Another important mathematical concept in 3D visualization is the matrix. A matrix is actually a two-dimensional array with rows and columns. Matrices can be multiplied and add together, but also by vectors and scalar values.

The figure below is an example of a 3x3 matrix.

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

4.7.1 Vector Operations

- **Addition, Subtraction And Scalar Multiplication**

  If we have two vectors, $v1 = (x_1, y_1, z_1)$ and $v2 = (x_2, y_2, z_2)$ then the resulting addition vector will be:

  $$a = (x_1 + x_2, y_1 + y_2, z_1 + z_2) \tag{4.1}$$

  Similarly, to subtract v2 from v1:

  $$s = (x_1 - x_2, y_1 - y_2, z_1 - z_2) \tag{4.2}$$

  Finally, we can multiply by a scalar which means we multiply or divide the vector, depending on the scalar's value. For example, if the scalar is equal to $\frac{1}{2}$ then we multiply all the vector's components by this value: $\frac{x}{2}$, $\frac{y}{2}$, $\frac{z}{2}$.

- **Computation of Vector Length**

  In order to compute the length of a 3D vector we must use the following formula:

  $$|v| = |(x, y, z)| = \sqrt{(x^2 + y^2 + z^2)} \tag{4.3}$$

- **Normalizing a Vector**

  Normalization means converting vectors into unit vectors. This can be done by dividing each of its three components - x, y, z to the magnitude.

  $$\begin{aligned} nx &= \frac{x}{|v|} \\ ny &= \frac{y}{|v|} \\ nz &= \frac{z}{|v|} \end{aligned} \tag{4.4}$$

The normalized vector will have the components *(nx, ny, nz)*.

- **Dot Product**

  The Dot Product calculates the cosine of the angle between two vectors. It is used in lighting calculations and other areas. The result is a scalar.

  It can be computed in 3D space between two vectors *v1(x1, y1, z1)* and *v2(x2, y2, z2)* like so:

$$P(v1, v2) = (x1 \cdot x2 + y1 \cdot y2 + z1 \cdot z2) \tag{4.5}$$

  The value of the cosine of the angle between the two vectors is equal to the following:

$$\cos(\theta) = \frac{P(v1, v2)}{|v1| \cdot |v2|} \tag{4.6}$$

- **Cross Product**

  The cross or vector product, is an operation on two vectors. The cross product of two vectors produces a third vector which is perpendicular to the plane in which the first two lie.

  If the angle between the vector is θ we can express the product like so:

$$v1 \times v2 = |v1| \cdot |v2| \sin(\theta) \tag{4.7}$$

  We denote the cross-product as C:

$$C = v1 \times v2 \tag{4.8}$$

  The components of the cross-product are calculated as follows:

$$\begin{aligned} C_x &= A_y B_z - A_z B_y \\ C_y &= A_z B_x - A_x B_z \\ C_z &= A_x B_y - A_y B_x \end{aligned} \tag{4.9}$$

- **Compute a Surface's Normal**
  A surface normal, or simply normal, to a surface at a point P is a vector that is perpendicular to the tangent plane to that surface at P. A surface normal for a triangle can be calculated by taking the vector cross product of two edges of that

triangle. The order of the vertices used in the calculation will affect the direction of the normal. For a triangle with vertices a,b,c, the vector $U = b - a$ and the vector $V = c - a$ we have the normal $N = U \times V$.

- **Conversion from Spherical to Cartesian Coordinates**
  The spherical coordinate system uses three parameters to represent a point in space:

  1. A radial distance (r) from a point of origin to a point in space
  2. A zenith angle θ from the positive z-axis
  3. The azimuth angle ϕ from the positive x-axis

  On the Earth, r = radius of the Earth, θ = latitude, ϕ = longitude. The origin is the center of the Earth.

  The Cartesian coordinate system is the standard x, y, and z coordinate system that is commonly used to represent a point in space.

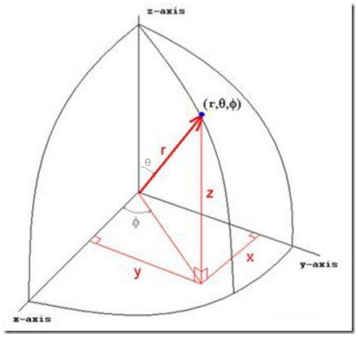

Figure 4.10: Spherical And Cartesian Coordinates

$$\begin{aligned} x &= r \sin\theta \cos\phi \\ y &= r \sin\theta \sin\phi \\ z &= r \cos\theta \end{aligned} \tag{4.10}$$

4.7.2 Matrix Operations

- **Rotation Matrix From Angle And Axis**
  Given a normalized vector $v = (v_x, v_y, v_z)$, the matrix of rotation by an angle of θ

in v's direction is given by:

$$R = \begin{pmatrix} \cos\theta + v_x^2(1-\cos\theta) & v_x v_y(1-\cos\theta) - v_z\sin\theta & v_x v_z(1-\cos\theta) + v_y\sin\theta \\ v_y v_x(1-\cos\theta) + v_z\sin\theta & \cos\theta + v_y^2(1-\cos\theta) & v_y v_z(1-\cos\theta) - v_x\sin\theta \\ v_z v_x(1-\cos\theta) - v_y\sin\theta & v_z v_y(1-\cos\theta) + v_x\sin\theta & \cos\theta + v_z^2(1-\cos\theta) \end{pmatrix} \tag{4.11}$$

- **Rotation Matrix That Rotates a Vector Into Another**
  We call the first vector *start*, and the second vector *end*. We are dealing with normalized vectors.

  We know that $\cos\theta = start \cdot end$ and $\sin\theta = |start \times end|$. Let:

$$\begin{aligned} v &= start \times end \\ c &= start \cdot end \\ h &= \frac{1-c}{1-c^2} = \frac{1-c}{v \cdot v} \end{aligned} \tag{4.12}$$

  Thus, using the formula 4.11 we get:

$$R(start, end) = \begin{pmatrix} c + hv_x^2 & hv_x v_y - v_z & hv_x v_z + v_y \\ hv_x v_y + v_z & c + hv_y^2 & hv_y v_z - v_x \\ hv_x v_z - vy & hv_y v_z + v_x & c + hv_z^2 \end{pmatrix} \tag{4.13}$$

  For more details, please consult [8].

Chapter 5

Technological Considerations

In this chapter the technologies that were chosen, and why, shall be briefly discussed. Due to some restrictions a great number of technologies had to be discarded as viable solutions. The main restriction was to use *Java* as the development language for the prototype. Not many 3D software packages use *Java* - most are based on languages like *C++* because of the greater speed. Moreover, *C++* is the de facto standard in this industry. But, of course, there are several advantages when using Java. These are just a few:

- The final application will be truly cross-platform(no need to build for specific architecture).
- Unlike other languages, Java has a garbage collector which makes life easier for the developers.
- It comes with the *AWT* and *Swing* GUIs.
- It has a standard way of formatting your comments and displaying them using a nice look and feel using *Javadoc*. This is great because most applications developed using Java shall have a very similar looking documentation - so most developers will be used to it.

In the following, the development environment shall be presented as well as technological comparisons for all the three main areas: 3D Visualization, 2D Dynamic Chart Plotting, 2D Map Handling.

5.1 Development Environment

- **Operating System**
  Both *Linux* and *Windows* were used in the development process - because the con-

sidered technologies run on both environments(and more) without problems.

- **Integrated Development Environment**
  A *Java IDE* was used due to increased productivity, easier bug tracking and other features. The two most popular IDEs for developing in Java are *Netbeans* and *Eclipse*. There was a restriction to use Eclipse, because the team wanted to integrate the prototype into their Eclipse RCP(Rich Client Platform). This was a disadvantage considering the fact that the 3D engine that was selected in the end had a custom SDK(Software Development Kit), based on the Netbeans IDE. Thus, it would have been easier to develop using the official *JMonkeyEngine3 IDE*.

- **Database**
  For the database, *MySQL* was the chosen technology, because it can be freely used, is very popular and powerful. On Windows a *WAMP(Windows Apache MySQL Php)* package was used.

5.2 Technological Comparisons

A number of technologies have been considered for all the different areas of the visualization interface. All selected libraries are free to use - at least for non-commercial development. A summary of the results is presented below.

5.2.1 3D Visualization Technologies

The following 3D technologies were considered:

- **VTK**
  The Visualization ToolKit is a free and open source library for visualization, 3D computer graphics and image processing. Natively it is a C++ library, but it also provides interpreted interface layers for Tcl/Tk, Java and Python. It is geared towards scientific visualization rather than for entertainment purposes. It is cross-platform and runs on Linux, Mac, Windows and Unix platforms. It is a more complete package and also provides charts and other functionality.

- **OpenGL**
  Open Graphics Library is the industry standard for high performance graphics. Almost all graphics and visualization packages that do not focus on video game development build on OpenGL.

- **JOGL**
  It is a Java binding for the OpenGL API.

- **LWJGL**
  The Lightweight Java Game Library provides access to very powerful cross-platform libraries such as OpenGL, OpenCL(Open Computing Language) and OpenAL(Open Audio Library). It also provides support for diverse input devices such as Gamepads, Steering Wheels and Joysticks.

- **JMonkeyEngine3**
  JME3 is a java-based 3D game engine. It is very well documented, provides an SDK and comes with all of the latest features in the industry: shaders, lighting, physics, special effects, asset system, texturing, GUI, networking and others. JME3 is built on top of LWJGL.

- **Java3D**
  Is a scene graph based 3D API originally developed by Sun Microsystems. However, official development has stopped some years ago. As a consequence, a lot of modern features available in other engines or APIs are missing from this package. It is cross-platform.

- **Other game engines: Unreal Engine, Unity**
  Unreal Engine is developed by Epic Games. It was first illustrated in the first-person-shooter game Unreal. It mainly focuses on first-person-shooter games. Its core is written in C++ and it has a high-degree of portability. Uses DirectX on Windows and OpenGL on Mac OS X.

  Unity 3 is also a game development tool. It is also very advanced and many different types of games have been successfully developed with it. It is however an authoring tool, but it also allows scripting. It supports multiple platforms and devices.

Due to the restriction of carrying out the development in Java, the Unreal Engine, Unity 3 and OpenGL were ruled out. Next, there were two other types of libraries low-level: JOGL, LWJGL and high-level: VTK, Java3D and JMonkeyEngine3. The low-level libraries were not used mainly due to timing constraints.

Java 3D is not nearly as powerful as the other two libraries, possibly because official development was stopped. Its user community is not nearly as wide as VTK's or JME's. These reasons and a brief research of applications developed with this technology clearly outlined its limits and inferiority with respect to the other two libraries. As a consequence, Java 3D was also discarded.

Finally it comes down to VTK vs JMonkeyEngine3. The two libraries are shortly compared in table 5.1.

VTK's main disadvantage is that it does not use Java natively and documentation on VTK Java development is very poor in comparison with the standard C++ documentation, or the JME3 documentation. However, it is specifically designed to create scientific applications rather then games which is an advantage. Moreover, there is a stable, final version,

	VTK	**JME3**
Dev. languages	Tcl/Tk, Java and others	Java
Native Language	C++, does not run natively in Java	Java
Type	Scientific Visualization	Game Engine
Version	Stable	Beta Stage(In development)

Table 5.1: VTK vs JMonkeyEngine3

which is constantly updated. Another plus is that it also includes chart plotting and some image handling tools.

On the other hand, JME3 is the only 3D game engine that runs natively in Java and at the same time, is competitive with respect to other non-Java based rendering or game engines. It also has some disadvantages: the first is the fact that it is currently only in Beta development stage - not a stable release. Secondly, it is a game engine - geared towards creating full-screen 3D games, not multi-window scientfic applications. That is why multi-window support and a few other features are limited.

5.2.2 Chart Plotting Technologies

The following charting libraries were deemed promising:

- **JFreeChart**
  Is the most popular free chart library available for Java. It includes a large number of professionally designed charts. However, it is mainly aimed for creating static charts, or charts that do not update very often. It is also very well documented.

- **JChart2D**
  It is a minimalistic real-time charting library. It has been designed with dynamic charts and precision in mind. However, the look and feel is average and there are not so many chart types to choose from. It is available for free.

	JFreeChart	**JChart2D**
General	Actively developed. Widely used.	Developed by one person.
Dynamicity	Poor	Great, real-time, multithreaded
Look & Feel	Professional and customizable	Average

Table 5.2: JFreeChart vs JChart2D

5.2.3 Image(Map) Handling Technologies

In the current version of the prototype image or map handling libraries have NOT been implemented. However, some candidates have been also considered for this part. The main alternatives are presented below:

- **ImageJ**
  Is a Java based library that focuses primarily on image processing and analysis. It is a complex application and has some considerable overhead. It takes some work to include it into Netbeans or Eclipse with all functionality - including plug-ins.
- **Java2D**
  Is included in the JDK. Aimed for advanced 2D graphics and imaging. It is a powerful library, but everything needs to be implemented manually. So, it is lower level than other software packages which already include some functionality.

5.3 Other Technologies

- **GUI Tool**
  Inside Eclipse, a plug-in called *WindowBuilder Pro* was used to develop the GUI. Specifically, the Swing components of this tool were used for designing the GUI. It is a graphical editor and other *Java Beans* can be added to it. *jCalendar* has been added for selecting dates.
- **Tools for 3D Background Generation**
  In order to generate a proper background without weird affects when applied to a 3D space, a cube map was used to display the background(by contrast, sphere mapping had unwanted effects in some parts of the scene and was not used).

  In order to generate the starfield wallpaper *Bryce* was used. A cube map has six parts(is six sided) and *Bryce* was used to generate a different image for each side of the cube(by rotating the star field accordingly). *ATI CubeMapGen* was used to generate the complete cross-shaped image formed by the 6 sides of the unfolded cube. An example of a resulting image is shown in figure 5.1.

5.4 Final selection

- **3D Computer Graphics Library**
  JMonkeyEngine3 was chosen in the end as the 3D package to be used for developing this prototype. Short test cases of simple applications have been developed using both VTK and JME3. However, VTK development was harder because of a lack of

Figure 5.1: Star Field Cube Cross

proper documentation on development in Java. JME3 also seemed quite powerful, and some of its problems with the windowing system were not known at the time - actually when work on this project began, JME3 was in Late Alpha development stage. The risk was maybe higher, but the first JME project was established back in 2003, so it has been continuously developed and improved.

- **Chart Plotting Library**
  Charts have been created using both libraries. JChart2D is definitely the faster option, however its look and feel is very minimalistic and looks poorly designed in comparison with JFreeChart. The current version of the application uses a JFreeChart test chart. Although a final decision has not been officially taken, JFreeChart will continue to be used unless the performance will be affected to a point where it becomes clear that a more lightweight, real-time plotting library is needed.

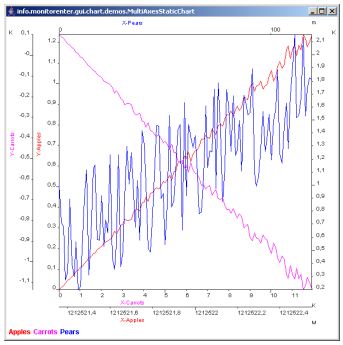

Figure 5.2: JChart2D L&F

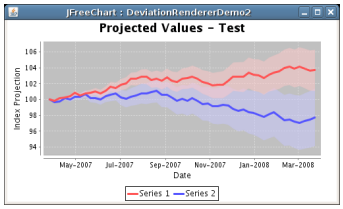

Figure 5.3: JFreeChart L&F 1

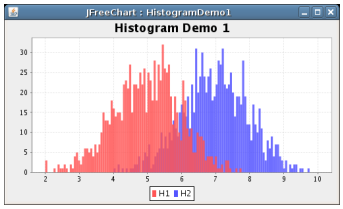

Figure 5.4: JFreeChart L&F 2

- **Image handling library**
  A final decision has not been taken yet. ImageJ might save some time, because it has

some useful functions including zooming and drawing regions of interest. However, it is a complex application, and only a few of its features are actually needed.This is not necessarily bad since the more functions might be required later, but developing a library from scratch using Java2D will provide exactly the required functionality. Of course, if a better library is found at some point in time, it makes sense to start using it, if work on this part will not have already started.

Chapter 6

Detailed Design and Implementation

6.1 System diagrams

- **Package Diagram**

  The package diagrams shows all the system's packages and their dependencies. UML package diagrams help to organize and arrange model elements and diagrams into logical groups. Developers can use the packages to model the physical packages or namespace structure of the application. This can also help explain the system's architecture from a broad view.

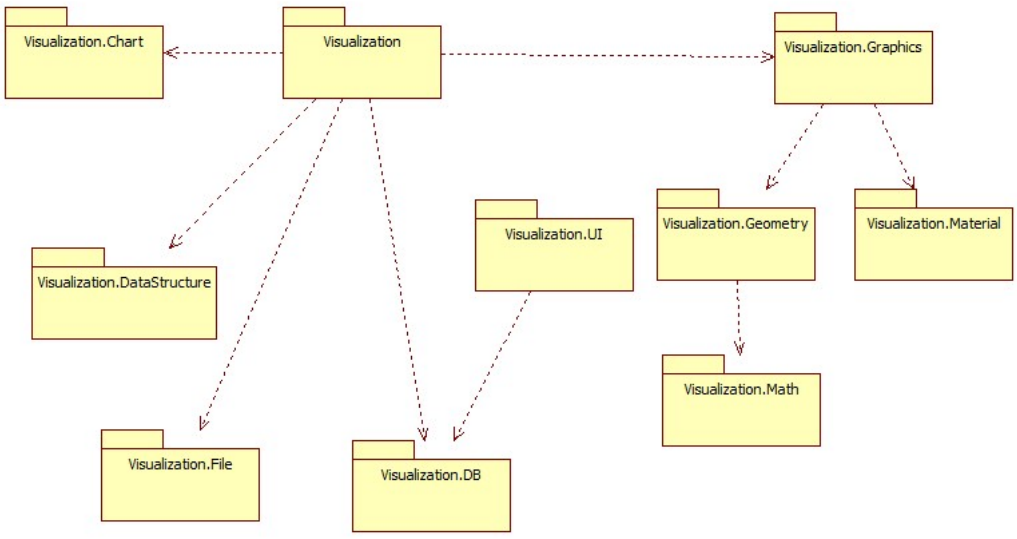

Figure 6.1: The Package Diagram

A brief description of the involved packages is presented below. Note that a shorthand version of the names is used. The actual package names are preceded by *esa.esac.Rosetta.Visualization*.

- **Visualization**
  This package contains global tools and variables which are needed to initialize the *Asset Manager*, helps the handling of multiple 3D windows, initializes the scene graph nodes, the renderer and others.
  Next, another class, called the *Environment* is actually a 3D view/window controller. It manages the environment by managing the corresponding 3D windows. Finally, the most important component of this package is the *Universe* class. This class starts the 3D world, draws all the objects on the screen and updates their properties such as position and visibility.
- **Visualization.DataStructure**
  This package contains all of the data structures that supply parameters to the visual object creator factories, depending on what is being read from the database. For more information please see section 6.2.
- **Visualization.DB**
  This package contains the database connection and all database related operations: these mostly consist in reading parameters for creating and positiong a 3D object.
- **Visualization.File**
  This package was used before the database was introduced in the application. It is no longer used. However, it could be reintegrated into the application to supply some additional method for reading object data. A few types of files can be read. Identification is done by file extension, so as long as the file extension is correct the system will know how to read that particular type of file. A factory file creator is used so other types of readers can be easily added later.
- **Visualization.UI**
  This package contains all the user interface components in which either charts, 3D worlds or menus and other controls can be drawn.
- **Visualization.Chart**
  Includes all of the chart related components and all types of charts currently drawable.
- **Visualization.Graphics**
  All of the different objects that need to be drawn on the screen, from Spacecraft to lines, are included here.
- **Visualization.Geometry**
  Includes all types of geometries that can be created. Starting from an abstract

type to concrete object geometries.

- **Visualization.Material**
  Includes all types of materials that can be created. Starting from an abstract type to concrete object materials.
- **Visualization.Math**
  Includes all the math related functions not present in the *JME3* math package. Mainly helps construct some geometries.

- **Deployment Diagram**

  A simple deployment diagram is presented below. The user's machine must have the graphics drivers from their card's manufacturer and Java installed in order to run it. The MySQL database is accessed via JDBC. All database related operations are performed in the *Visualization.DB* package. All the artefacts on the user's machine are included in the diagram - to emphasize the fact that the actual application is built on top of the other tools.

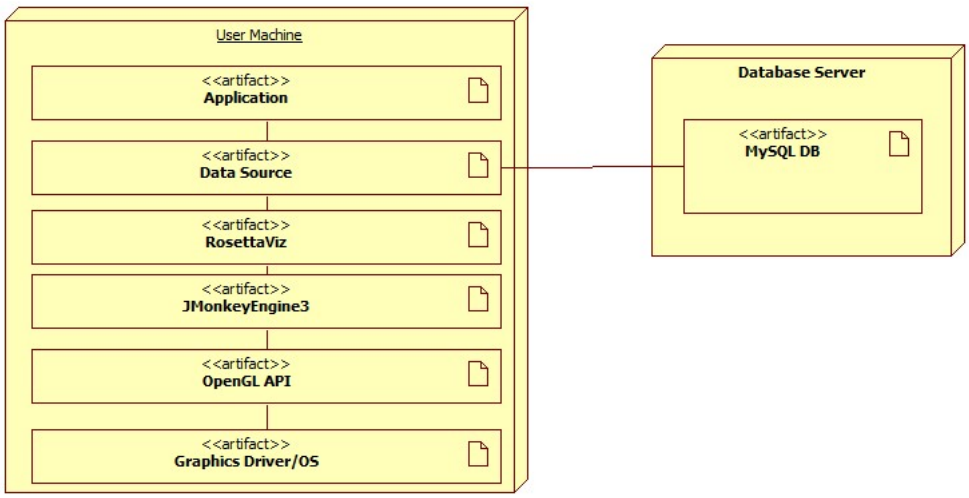

Figure 6.2: Deployment Diagram

6.2 Data Structures

Since all the objects are created, from the database, when the application is started there has to be a way to efficiently feed the data(the needed parameters) to the object constructors. The parameters are logically broken into object parameters(define general object properties such as a name, perhaps a file that defines the object's geometry etc.), material parameters, shape data, position data and others.

- **Object Parameters**
  The object parameters define a name, the geometry type, an optional filename if the object is a 3D model created with a modelling tool such as *Blender*, shape data - optional as well, a reference to the material parameters and the object's scale with respect to the original size(which is 1.0 on all axis by default).

- **Material Parameters**
  These parameters will determine the appearance of the object. It will create a material using a name, type and texture. Of course, the texture is optional - not all materials have a texture applied to them. Moreover, the type of the material actually determins the JME3 shader to be used - unshaded, lighting, etc.

- **Shape Data**
  This data is used for creating custom objects without having a model created with a modelling tool. The total number of points and triangles must be supplied along with the whole list of points and triangles. The way triangles are defined is important. Example: Triangle of points (1,2,3) is not the same to (3,2,1). - back facing triangles may be culled.

- **Position Data**
  Includes all the positions in time of a particular object as well as the precise date and time of each movement.

- **Additional Parameters**
  Used for lines and other objects. Defines the colour of the lines as well as the width and start point.

6.3 Data Model

The data model is used to create the 3D world and also supplies the data used by charts. The *EER*(Enhanced Entity Relationship) model is shown in figure 6.3. The *rosetta_vo* entity has the main parameters for a visual object. The name, geometry and positions are specified either by using some file or by using additional entities which contain the required data, captions to display their name or other important information, the radius - an optional field for spherical objects only and the scale factor.

rosetta_mat specifies which material parameters to use(ambient, diffuse and specular color) - *rosetta_mat_params*, an optional texture and gives a material name. This material is associated of course to a particular object. The material can be of course used by many objects.

If a *custom geometry* was chosen then the shape data must be supplied in order to generate the object. The required data is found in the *rosetta_vo_data* entity. Moreover, the vertices

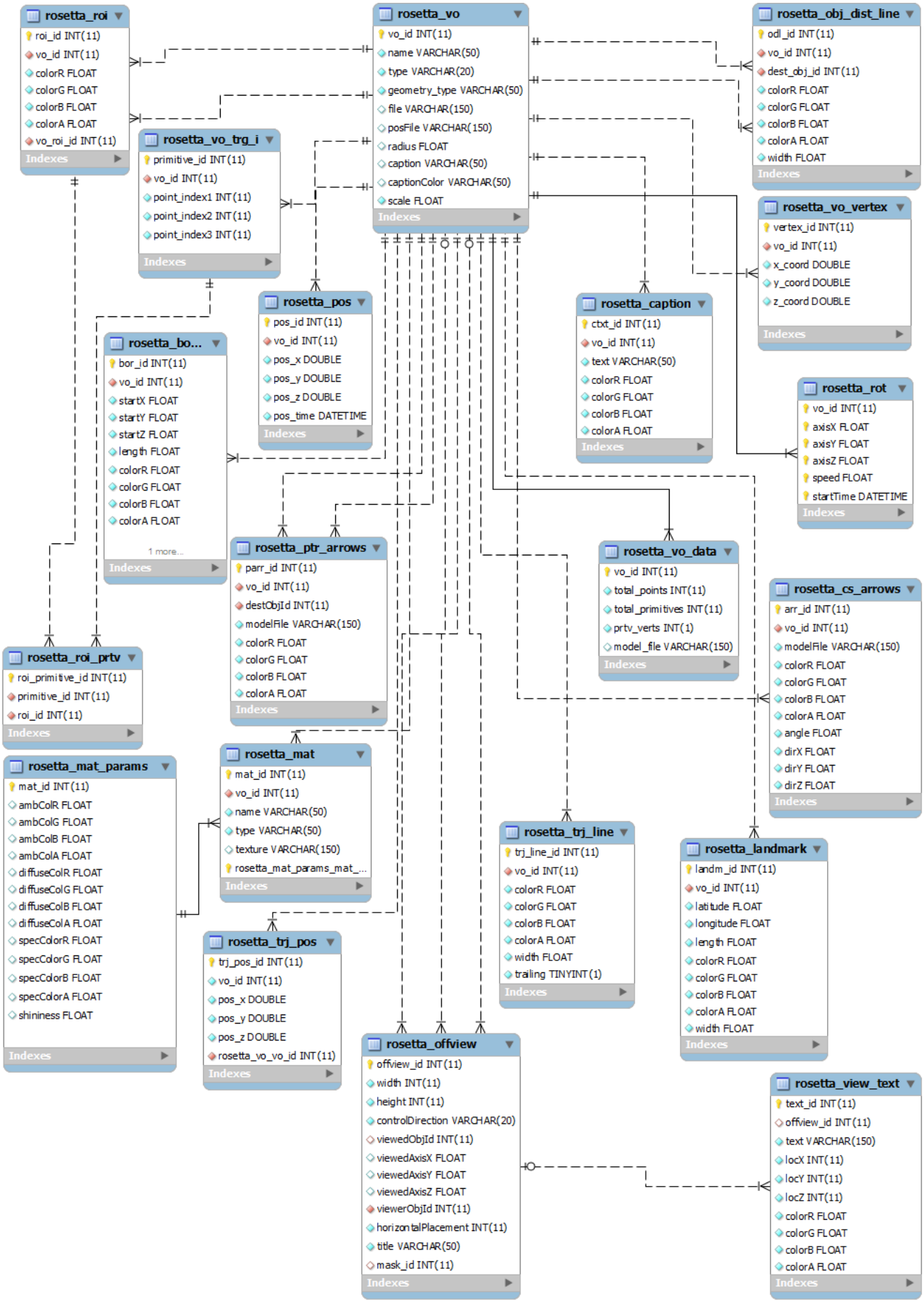

Figure 6.3: Data Model

are found in *rosetta_vo_vertex* and their interconnection in groups of triangles is found in *rosetta_vo_trg_i*. Perhaps a more flexible entity would allow the user to specify if the primitive blocks used are triangles or four point polygons. Currently, only triangles are supported.

Furthermore, object's positions are stored in the *rosetta_pos* entity. Of course, every object has its own data - if required; some objects might not need to translate and consequrntly will not need the date and time of each movement. Similarly *rosetta_rot* specifies the object's rotation along all axis and its speed.

Secondary views are also modelled here. The coordinates of the *viewer object* and of the *viewed object* are supplied, as well as a window size and title, an optional mask and other properties.

All of the component objects have an associated entity that contains all the necessary data to create the object. The database is queried for each of these object types and their corresponding 3D counterparts are created.

6.4 SceneGraph Implementation

As mentioned before the scene graph can be logically broken into two parts. Moreover, any JME3 scene graph must have a root node from which all others descend. Any node can have one or more children - either spatials or nodes. The general scene graph nodes can be thought of as global nodes and they can be accessed and used in a few different parts of the application. The other nodes are internal to that object and only instances of a particular object can access them. In the following sections, a short description of how these two parts of the scene graph were actually implemented is presented.

- **General Scene Graph**

  First, all of the global nodes were considered. Then they were included with the other global tools and interconnected in the very beginning of the creation of the *Universe* - in the application initialization method of this class.

  The *SunLight Node* was connected directly to the *Root Node*. All objects that are visible on the screen must therefore be a child of the *SunLight Node*. All other children of the Root Note shall not appear in the scene - because of lack of lighting.

  Next, a *View Node* was added in order to add some objects which should only be visible in an instrument(secondary) view. An example of this is *the mask*. The mask needs to be drawn on top of everything in an instrument view, while serving no purpose in the main 3D window.

At the next level, the *Sun Node* was attached. Here the visible Sun is added - with a geometry and an applied material.

Finally, the *Solar* and *SPC* nodes are attached to the *Sun Node*.

This part of the scene graph's structure is shown in the figure below:

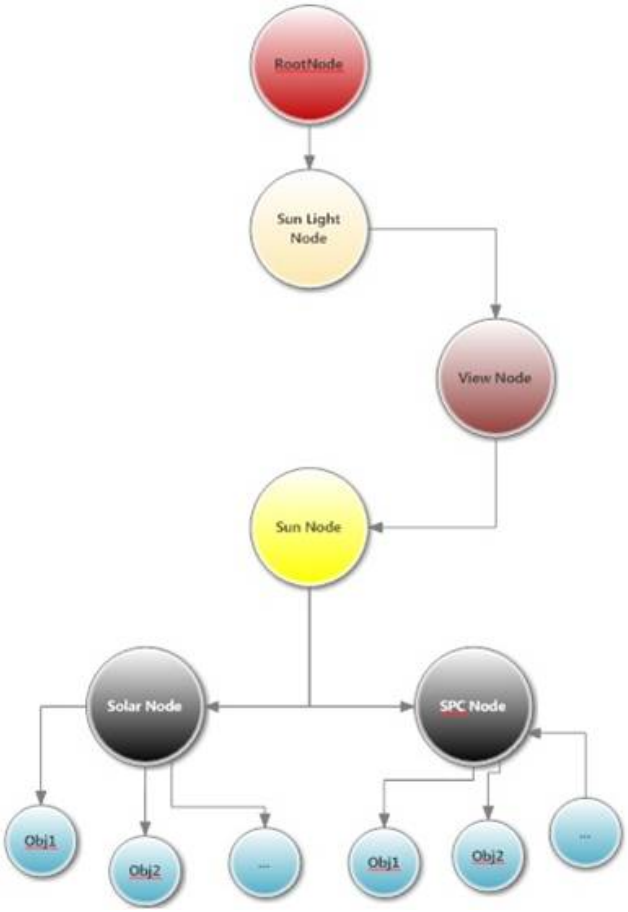

Figure 6.4: General Scene Graph Part

The nodes can be added using JME's *attachChild* method. This is how a scene graph can be easily constructed from a root node, which is available in a *JME SimpleApplication*.

```
// construct node hierarchy

rootNode.attachChild(GlobalTools.sunLightNode);
GlobalTools.sunLightNode.attachChild(GlobalTools.viewNode);
GlobalTools.viewNode.attachChild(GlobalTools.sunNode);
GlobalTools.mainViewNode.attachChild(GlobalTools.sunNode);
```

```
GlobalTools.sunNode.attachChild(GlobalTools.solarObjectNode);
GlobalTools.sunNode.attachChild(GlobalTools.spcObjectNode);
```

- **Object Scene Graph**

  This part of the scene graph is the structure that defines all the possible properties of *SPC* or *Solar* objects. These objects can have other objects as components - arrows, landmarks, boresights, regions of interest and more. The detailed structure of the object scene graph part is shown in figure 6.5. There are two basic operations

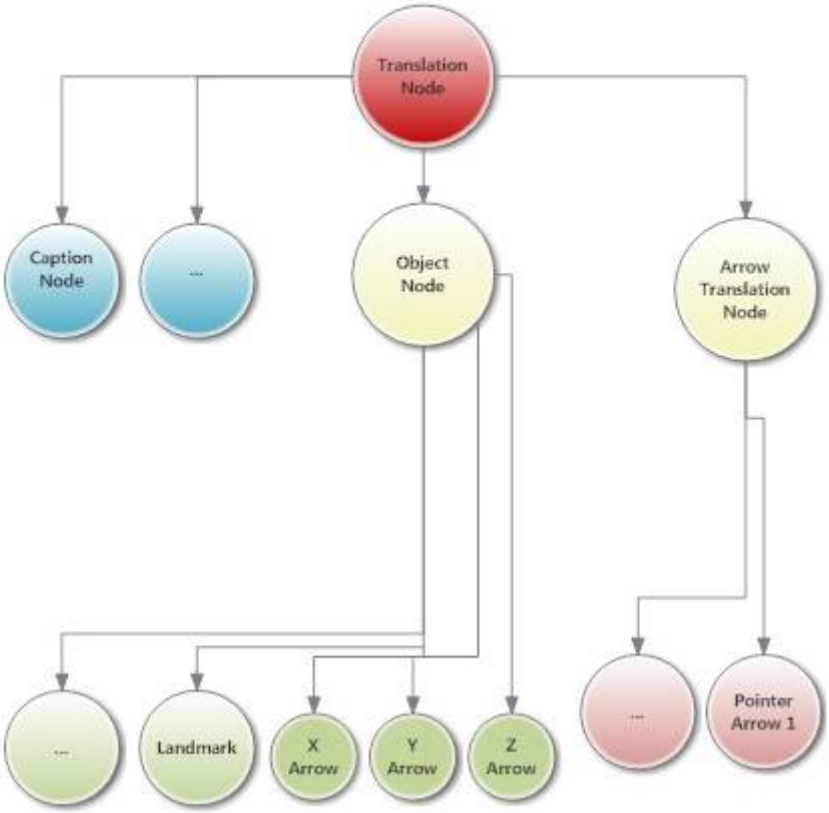

Figure 6.5: Object Scene Graph Part

which are performed consistently on visual objects - translation and rotation. A *Translation Node* was introduced that is used to apply translation to the SPC and Solar objects and also to their respective component objects. The rotation operation is not applied at this level!

An Object Node has been added and rotation can be performed at this level. Any operations performed here will be inherited by the children of this node which are all the component objects' nodes of the current SPC or Solar object.

A sibling of the previous node is the *Arrow Translation Node* which is used for

positioning directional arrows, which point from the center of the current object to a specific destination, which is usually represented by some other entity such as the Sun or the comet. This is necessary because these arrows have a separate rotation type, independent of the object's rotation.

Moreover, there are some objects that do not need to be rotated at all. An example of this is the *Caption Text*. It is always placed above the object and translates along with it, but it is similar to a 2D billboard - it does not need to rotate.

6.5 Visual Object Implementation

The object parameters are set in an abstract class called *AbstractObject*. All visible objects must inherit this class, be it a 2D mask, a particle effect such as an atmosphere or a regular 3D object. The component objects are just that - they are added using composition in Sun, Solar or SPC objects. The main components are briefly explained below:

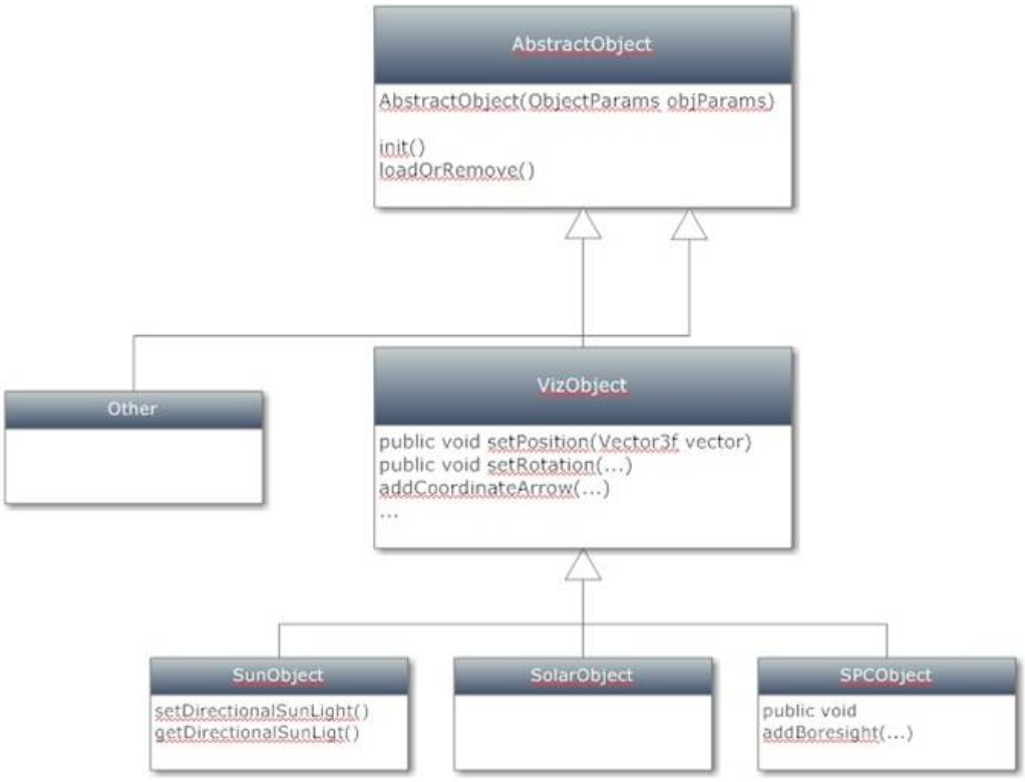

Figure 6.6: Visual Object General Structure

- **AbstractObject**
  Represents a very abstract visual object type with few parameters. It is used to con-

struct the object nodes and, set the Geometry(structure) and Material(appearance) using their respective factories. Moreover, this entity supplies a way of loading or removing objects from the scene.

- **VizObject**
  Is used by all primary 3D objects. It adds more detail to the objects. It includes all common properties of *SunObject*, *SolarObject* and *SPCObject*. The following common component objects can be added here:
  - *Arrows(3D Arrows)*
    - *Coordinate System Arrows*
      Represent the X, Y and Z axis for the object which contains them.
    - *Pointing Arrows*
      Represent arrows which point from the object to another destination point, usually another object which has some meaning
  - *Caption Text*
    Represents text that is mainly used as a label for the object's name
  - *ROI(Region Of Interest)* - only partially implemented
    A region of an object is created in a particular colour to indicate that there is something to monitor in that region.
  - *Lines*
    - *Distance between objects*
      A line that shows the shortest distance between the origin of two objects.
    - *Trajectory line(trailing or not)*
      A trajectory line that shows the whole path a spacecraft traverses for the simulated time period. Moreover, the trajectory line can also be turned into a trail which shows the path from the beginning until the current point in time.

  New component object types an be easily added. More details on component objects including how they are created can be found in section 6.5.3.

- **SunObject**
  Is used for the Sun(figure 6.7). It specifies sun geometry and light source and position. A directional white light source is used. The actual light source is added to the *SunLight Node*, while the 3D Sun object(geometry and material) is added to the *Sun Node*. The physical Sun object can be turned on or off or moved very far away, outside the view of the scene, but the white light will always shine from the Sun's direction, even if we cannot see the Sun.

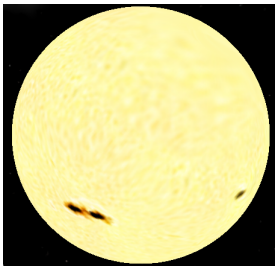

Figure 6.7: The Sun

- **SolarObject**
  Represents cosmic objects - comets(figure 6.8), asteroids etc.

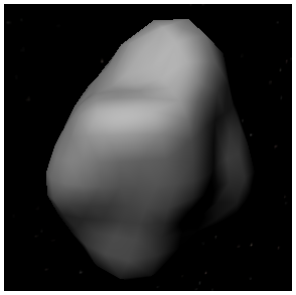

Figure 6.8: Comet 67P

It can also have *Landmark*s added as a component object.

- **SPCObject**
  SPC objects represent spacecraft objects (figure 6.9 shows Rosetta).

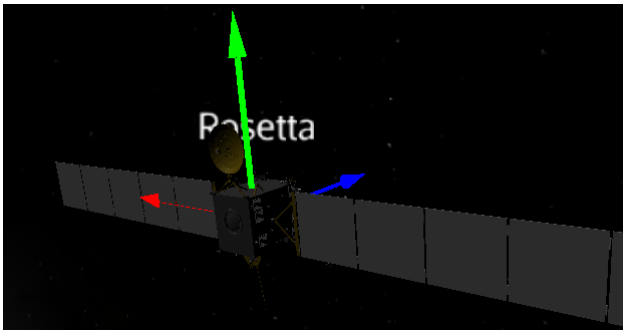

Figure 6.9: Rosetta SPC with X(Red), Y(Green), Z(Blue) Arrows

It can also have a *Boresight* added as a component object.

The Solar and SPC objects are not very differentiated in the current version of the prototype. However, based on the specific needs more components and properties can be specifically added to each type of object or to both.

6.5.1 Geometry Creation

All objects must have a geometry. The geometry defines, the mesh or overall structure of the object and it must also have an associated material which defines the object's final appearance.

The idea is to have a geometry factory that calls different geometry creators depending on the object parameters received from the database. The overall structure for this part of the system is shown in figure 6.10.

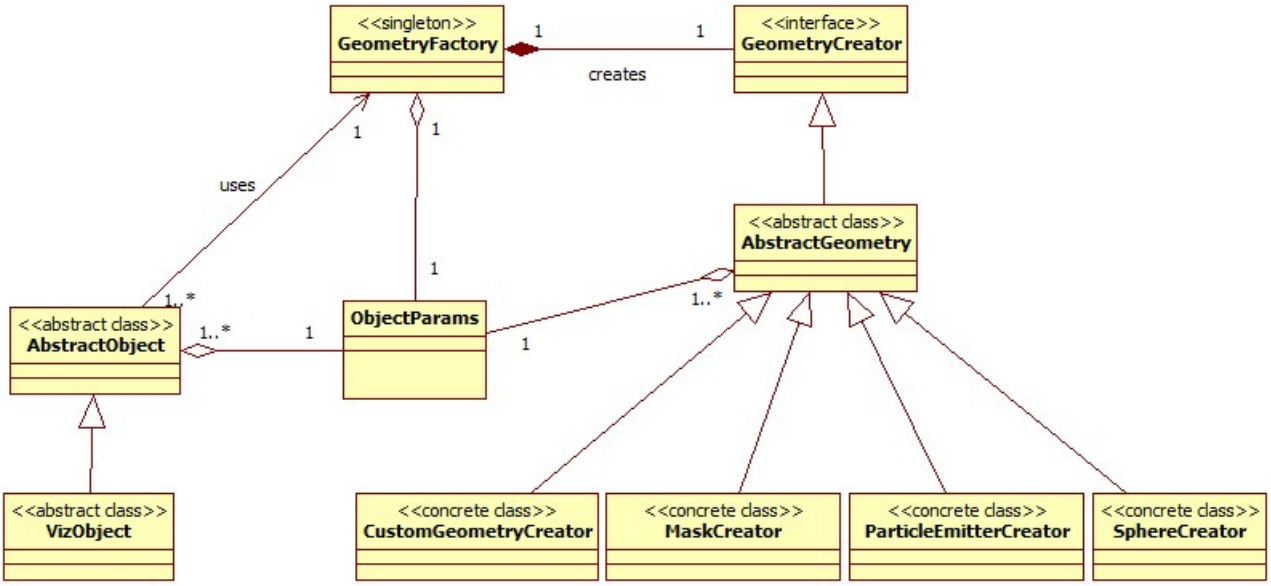

Figure 6.10: Geometry Creation Class Structure

The components and their functions are detailed below:

- **Geometry Creator**
  Is an interface for creating all geometry types. All creators must implement the methods specified here. This provides flexibility - all creators will have similar structure and new ones can be easily added.

- **Abstract Geometry**
  Is an abstract geometry class that must implement the *GeometryCreator* interface.

It has basic geometry properties and has object parameters which must be set by concrete classes which inherit this class.

- **Concrete Creators**
  These creators actually create different types of geometry, depending on the object parameters that are supplied to the factory and subsequently, to the adequate creator.
  - The *CustomGeometryCreator* creates a geometry from a given list of points and triangles.
  - The *MaskCreator* is responsible for creating the 2D geometry for a mask that needs to cover an instrument view.
  - The *ParticleEmmiterCreator* is used for creating effects such as a comet atmosphere.
  - The *SphereCreator* is used for creating spheres - mainly used for generating planets.
- **Geometry Factory**
  The *GeometryCreator* class is the one that deals with delegating geometry creation responsibilities to the correct concrete *GeometryCreator*. A hash table with the following structure *("geometryType","className")* that contains all existing geometries and their corresponding classes is built into the factory when it is first created. This information is used when comparing the received data from the *ObjectParams* and determining if there is a geometry creator that can handle the task. Otherwise, an exception is thrown.
- **Object Params**
  Are the object parameters read from the database which include the geometry type which is supplied to the factory for choosing a particular creator.
- **Abstract Object and Viz Object**
  Are the abstract definitions of all objects in the world. For the differences between them as well as for more details please see section 6.5. From here the *ObjectParams* are passed to the *GeometryFactory*.

This approach allows for great flexibility because it gives other developers the ability to create their own geometry types using only the documentation or some provided template.

```
Example:
A developer wants to add an asteroid geometry type to his application.
He just needs to create his own AsteroidGeometryCreator class and add
the corresponding values in the hash table: ("asteroidGeom", "package.
AsteroidGeometryCreator."). Of course, his class needs to implement the
GeometryCreator interface and to extend the AbstractGeometry class.
```

6.5.1.1 Geometry Types

In this section the implementation of geometry types is briefly explained.

- **Custom Geometry**
  In order to create a complete solid geometry with Gouraud shading we need to set three vertex buffers for our mesh. First, we set the *Position* buffer which holds all of the object's vertices. Secondly, we set the *Index* buffer which defines how the vertices are connected into triangles. Lastly, we need to set a *Normal* vertex buffer. In order to do this, the normals had to be computed using the technique described in section 4.6.

  The following steps were taken in order to compute the normals:

  1. All vertex normals are reset(set to 0).
  2. For each triangle, its three vertices are stored.
  3. Two co-planar vectors are created using two sides of the selected triangle.
  4. The vector product of these two vectors is computed next.
  5. The result is normalized. What we have now is the triangle normal.
  6. The number of connections for each vertex of the triangle is increased.
  7. We add the current normal to the vertex buffer.
  8. Repeat steps 2-7 until all triangles have been visited.
  9. Average the triangle normals stored in the buffer to obtain the vertices' normals.

- **Mask Geometry**
  The current solution only allows the creation of square/rectangle masks. To do this, eight vertices are used. The four outer points are the top-left, top-right, bottom-left and bottom-right most points of the camera. The four inner points define the size of the mask and how much will be visible after it is applied. Triangles are used to connect the area between the inner and outer vertices, and the interior area is left unmasked. An orthographic projection is used to project a 2D mask on top of the 3D scene.

- **Sphere Geometry**
  The *Sphere* is a bult-in JME3 shape. In order to create a sphere one needs to supply the z and radial samples, the radius, if the sphere should have an interior and if the slices should be even.

- **Particle Emitter Geometry**
  Particle emitters have a lot of properties which define their appearance and the current implementation has the parameters set to create the *Atmosphere* object.

These include the number of triangles, gravity, low life and high life, velocity, image size and so on. All particle emitters will have a similar geometry but the texture can be different and this can change the appearance drastically. An improvement would be to have some particle emitter parameters which are read from a database(like for the rest of geometry and material creation) and apply them independently, for each particle emitter.

6.5.2 Material Creation

Materials are created in a similar way to geometry. This time, a material factory is used to delegate the material creation task to the responsible creators by using the information gathered from the *MaterialParams* - an entity that contains information that defines a particular material. The overall structure for this part of the system is shown in figure 6.11. The components and their functions are detailed below:

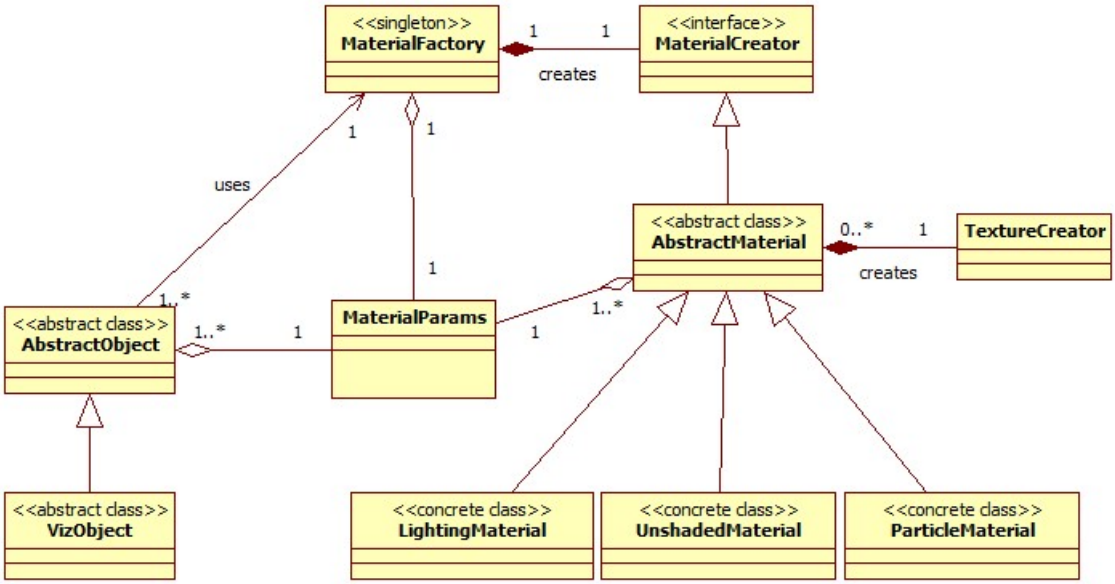

Figure 6.11: Material Creation Class Structure

- **Material Creator**
  Is an interface for creating all material types. All creators must implement the methods specified here. This provides the same flexibility as for the geometry.
- **Abstract Material**
  Is an abstract material class that must implement the *MaterialCreator* interface. It has basic material properties and has material parameters which must be set by

concrete classes which inherit this class. In addition, one of the parameters represents the location of a diffuse texture that is to be applied. The texture value is optional and can be *null*.

- **Concrete Creators**
  Use the material parameters supplied by the factory to create the class which is responsible for the generation of the selected material type.
  - The *LightingMaterial* creates a material that is displayed after lighting calculations are performed. The object will be shaded.
  - The *UnshadedMaterial* does not take lighting effects into account when computing the material's appearance. The material will be unshaded and will be displayed exactly in the color that is specified by the material parameters.
  - The Particle *ParticleMaterial* is used for setting the type of material used by particle effects. Different types have different properties.
- **Material Factory**
  The *MaterialFactory* class is the one that deals with delegating material creation responsibilities to the correct concrete *MaterialCreator*. A hash table similar with the one used for geometry creation has been implemented: *("materialType","className")*. This time the data is checked against the current *MaterialParams* instance.
- **Material Params**
  Are the material parameters supplied in the database for each material. Material properties include type, an optional texture and the combination of ambient, specular and diffuse colours, as well as shininess.
- **Texture Creator**
  If the texture parameter is not null, the system tries to create the corresponding texture and apply it to the material. All material types have this property since it is included in the *Abstract Material*.
- **Abstract Object and Viz Object**
  Similarly to the case of the geometry the material parameters are passed on to the *Material Factory*.

6.5.2.1 Material and Texture Types

There are two main materials used in this project: *Unshaded* and *Lighting*. In order to use them one must set the appropriate material definition(shader) file. For more information about how these files are created and used please see appendix A.The difference is that only diffuse colour is set for the unshaded material, while a combination of diffuse, specular and ambient colours plus an optional shininess value is used to compute the appearance

of an illuminated material. A third type of material is the *Particle* material which is used in conjunction with a texture to create a particle emitter effect.

The first two material types can also have an optional texture. The *DiffuseMap* method is used to display textures.

6.5.3 Component Object Creation

As mentioned before the component objects are part of either *Sun*, *Solar* or *SPC* visual objects. Each is different from the other and they all have their own properties. A brief description of all of the available component objects plus the Mask and the Atmosphere is given in the following sections.

6.5.3.1 3D Arrow

A model of a 3D arrow has been created in *Blender*. It has also been aligned and then exported to a format readable by JME3. The arrow is used for modelling two types of components: cartesian coordinate system arrows - local to each object, or pointing arrows. Each arrow has a 3D model(the one mentioned before is used), a color, an origin and a direction. In this case, the origin will always be the origin of the object to which this arrow is added, regardless of its type.

- **Cartesian Coordinate System Arrows**
  Each of the primary object types can have its local 3D cartesian coordinate system

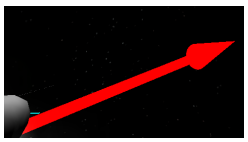

Figure 6.12: The X Axis

  arrows displayed. We use the rotation matrix operation discussed in section 4.7.2 to align the arrows along each of the primary axis: X, Y, and Z. By default, the 3D arrow model is aligned exactly on the Z axis. Therefore, a rotation with $\frac{\pi}{2}$ radians is performed on Y and X in order to obtain the other two axis. An example of the X cartesian coordinate arrow is shown in figure 6.12. The standard colours are used for the three axis: red for X, green for Y, blue for Z.

- **Pointing Arrows**
  For creating the pointing arrows the start and destination object's position vectors are used. The arrow changes its position in time due to the change of the object's

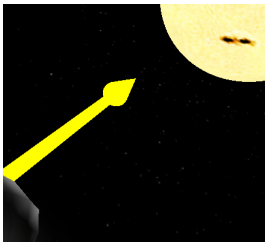

Figure 6.13: Arrow pointing towards the Sun

positions - thus the pointing arrows must be continuously updated.. We need to rotate the start vector into the end vector as described in section 4.7.2.

An example is shown in figure 6.13. The arrow points from the comet towards the Sun. This is useful because we always know where the light comes from. The arrow's rotation is independent to that of the comet and will always point to the Sun, regardless of where in the world we place the comet. Another pointing arrow is used to point from Rosetta to the comet.

6.5.3.2 Boresight

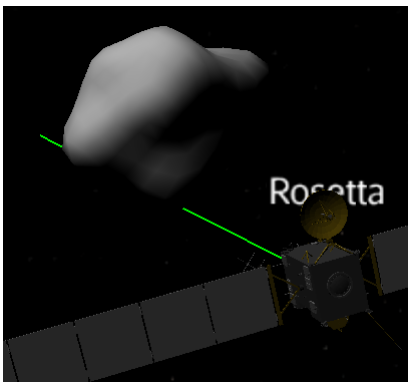

Figure 6.14: Rosetta's Boresight

The boresight can only be added to the *SPC* objects. It is represented by a line(see figure 6.14 for an example). Currently, the boresight is set to point in the negative Z direction. It starts from the object and follows its vector a very long distance - it passes through the comet and beyond. Therefore, it is just a line created using the current object as a starting point and a direction and length to follow.

6.5.3.3 Caption Text

In order to create the caption text we first need a font(which is available in JME3) and colour. Then, the text is positioned above and to the middle of the object - its position is fixed with respect to the object. In order to let JME3 know that this is actually a 2D

Figure 6.15: Rosetta's Caption

object we use a special control, called a *billboard control*. A control is a type of behaviour that can be associated to a spatial. As its name suggests, this control draws all graphics associated to this particular node as a billboard. Billboards, can also be images and other objects and can be placed anywhere in the scene.

The text *Rosetta* is shown in figure 6.15 above the spacecraft. It will always be in the same position with respect to Rosetta.

6.5.3.4 Landmark

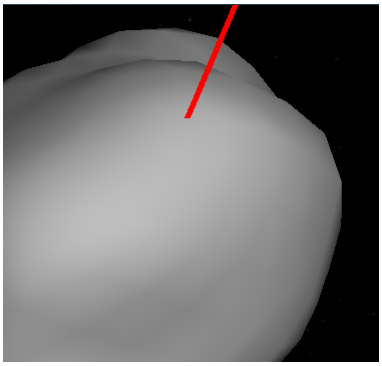

Figure 6.16: A red landmark on Comet 67P

The landmark can only be added to *Solar* objects. It is drawn as a line protruding out of a 3D object, at a particular point(see figure 6.16). In order to make this work, we need

to supply what in Earth coordinates is known as *latitude* and *longitude*, in addition to the start position, color, length and width. The line's direction vector is computed by using the spherical coordinates and transforming them into cartesian coordinates. JME3 provides a math library in order to do such computations. However, the intricacies of how this works can be found in section 4.7.1.

6.5.3.5 Distance Line

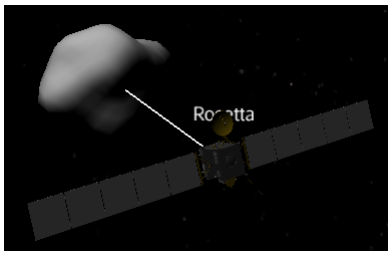

Figure 6.17: White distance line between the two objects

This line draws the shortest distance between two objects, with the start and end points in the origin of the first, and respectively last object. In the current implementation, the line is added to the object that has the start point. Perhaps, an improvement would be to make the line independent of both objects. The line is created between the two points and updated continuously, whenever one of the objects changes position.

In figure 6.17 a white line is displayed, showing the distance between Rosetta and comet 67P.

6.5.3.6 Trajectory Line

Its behaviour can change depending on its type. There are two types of such lines available for use. Each of these has its own uses. Both of them are shortly described in the following sections.

- **Normal**
  The full spacecraft trajectory is loaded in the 3D world. It shows the full path the spacecraft will take from the beginning to the end of the time period specified. Note, that the full time period available is currently displayed(see figure 6.18) - not just the selected interval(which can be a smaller time frame). A number of the points of the original trajectory are used to plot this line using a list of positions. The number of points is far less then that of the actual simulation, because not much detail is needed to create a line. For the geometry's mesh, a *LineStrip* mode is used.

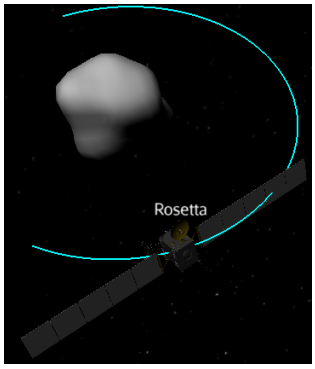

Figure 6.18: Rosetta's Trajectory (Blue Line)

- **Trail**
  Is just a trailing trajectory line, but instead of loading all the positions mentioned

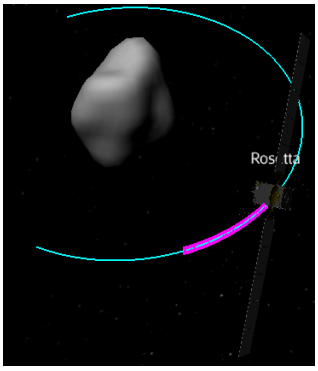

Figure 6.19: Rosetta's Trailing Trajectory (Pink Line)

above, the line is only made up of the points from the position at the beginning of the current simulation, until the current time period's position. It updates in real time, showing how the spacecraft advances and what progress it has made. It is displayed over the normal trajectory line, in another colour and with a larger width.

The example shown in figure 6.19 shows that the trail can start at any point on the full trajectory line, depending on the selected simulation time interval.

6.5.3.7 Region of Interest

The current implementation is just a test for the ROI to confirm it can be successfully integrated in the application - which is the main point. A *test case* of points is currently used to draw an unshaded irregular region on the comet. This region is not pleasing to the eye(see figure 6.20). It is a bit scaled, in comparison to the original object so that the region will protrude on the edge of the object - otherwise the region can not be seen.

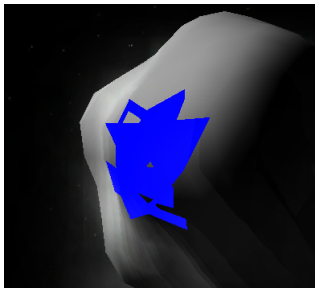

Figure 6.20: Region Of Interest on Comet 67P

6.5.4 Other Object Creation

These objects are considered to be independent of all other objects, including *VizObject*s. Note however that the atmosphere is associated with the comet and is attached to its node. The objects are presented in the following sections. Of course, they also need to be added in the scene after the *SunLight Node* or in the *View Node* if they should only appear in secondary views.

6.5.4.1 Atmosphere

Figure 6.21: Comet Atmosphere(Test)

The comet atmosphere has been implemented as a particle emitter. A JME3 texture defines the overall shape of the particles. This can be: debris, flames, smoke, mosquitoes, leaves and butterflies - the *smoke* texture was used to create the effect in figure 6.21. The particle emitter has many properties and the values were chosen by testing the effect of each property. Note however, that this is not a final implementation, more work is needed in order to transform the atmosphere into a more appropriate shape. The resulting effect can be observed in figure 6.21.

6.5.4.2 Mask

The mask is used to hide some of the field of view of an instrument that is used to look in some particular region. In figure 6.22, the mask is the border around the 3D world. This

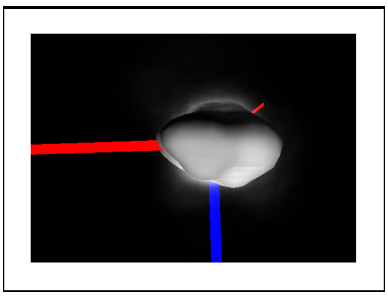

Figure 6.22: Border mask in Rosetta's view of the comet

is just for demonstrative purposes - the real masks shall have different shapes. However, only square type masks have been implemented in the current software. The mask uses an *ortho node* - which means that the projection for this object is orthographic - which is used for 2D objects, not perspective. Moreover, must always be on the instrument view - culling property is set to *never*. It is also moved on the z axis to appear on top of all the other 3D objects.

The square mask's geometry is created by connecting 8 vertices into triangles such that the inner square formed by the inner four points is always revealed/unmasked and the rest is behind the mask. An algorithm should be devised to generalize this process and allow the creation of other shapes such as a diamond or hexagonal shape.

6.6 GUI Implementation

The GUI has been designed using an editor called *WindowBuilder Pro* which comes as a plug-in for Eclipse IDE. Normal *Swing* components can be added by using a drag and drop method. A separate library was used for the date selector.

The GUI has the following parts:

- **Main Window**
  This window has the main menu as well as the user controls(see figure 6.23). The

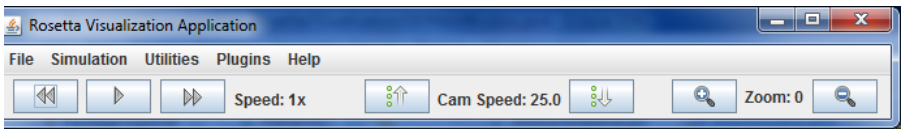

Figure 6.23: The Main Menu

user can create a new simulation, or if the simulation is started, the user can use the recorder buttons to play/pause, speed up or speed down the simulation. The speed is conveniently displayed in a label. The user can also control the speed of the

camera with two additional controls. There is also a camera zoom function which can be changed using the other buttons.

Furthermore, chart or offview windows can be created using the *Utilities* menu. Finally, the user can use the Show/Hide menu in order to turn the visibility of (most) objects on or off. Note that all show/hide, chart and offview related menus are generated dynamically, based on the data that is retrieved from the database.

Moreover, this class(*MainWindow.java*) contains the *main* method which starts the application.

- **New Simulation Dialog**
  This dialog box(figure 6.24) has two date selector controls. These allow to select

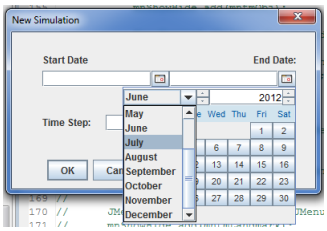

Figure 6.24: New Simulation Dialog

a time interval between a start and end date, for the simulation. Moreover, it also has a time step which specifies if the simulation will start at its original speed, with all data, or if the default speed is greater than the normal(greater than 1). This speeds up the application start-up since data is actually skipped each time the value is increased, by the corresponding amount - value n, means every n-th value is read.

- **Main Simulation Window**
  This is the main 3D window(see figure 6.25). It has been included in a JFrame,

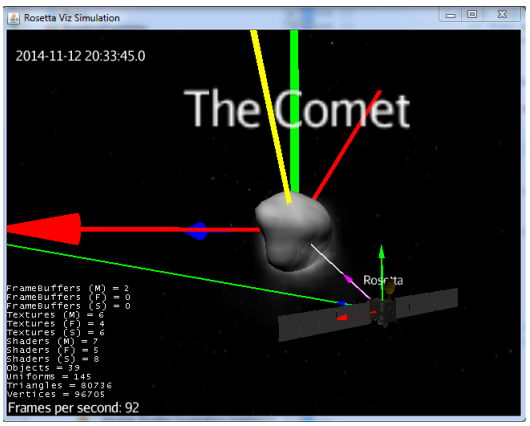

Figure 6.25: Main Visualization Window

line any other component. The resolution can be changed in the display settings dialog, which is an automatically generated dialog(JME generates it to allow for some graphics configuration, it can also be turned off if default values are specified).

- **Chart views**
  A JFrame that contains a JFreeChart chart component. For more details about the chart window please see section 6.7. The chart in figure 8.7 is more thoroughly described in the previously mentioned section.

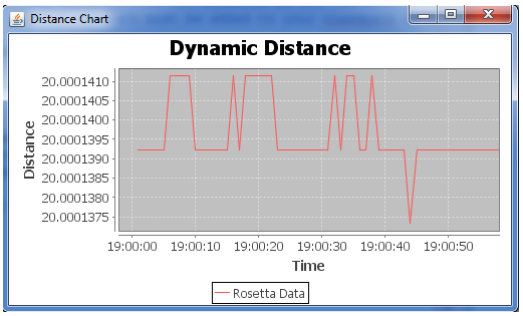

Figure 6.26: Distance plot between comet 67P and Rosetta

- **Offviews(Instrument views)**
  For an example see figure 6.22.

  Similar to the main simulation window, these views display parts of the 3D scene from another perspective - an instrument on board of the spacecraft. However, note that there is no direct camera control on these views. The camera only changes with the object - the spacecraft rotates, so the instrument rotates too, into another position. For more information please see section 6.8.

6.7 Chart Implementation

A chart has been added to the application in order to showcase the software's ability to integrate the different technology and be able to show the same meaningful data by using different visualization techniques. The created chart is plotting in real time the *distance* between the comet and the Rosetta spacecraft. It shows the distance on the Y axis and the time on the X axis. The chart is displayed in figure 8.7. If the speed of the simulation is increased or decreased the plotting speed of the chart is also updated. It runs on a separate thread and the *sleep* function is used to control the plotting speed. Moreover, if the simulation is paused/resumed the chart also pauses or resumes when necessary. Finally, the chart can also be zoomed in and out on one or more axis. The *JFreeChart* library allows for easy customization of the plot, but the look and feel is currently fixed.

A possible extension would give more customization power(with respect to the look and feel) to the user.

6.8 Instrument Window(View) Implementation

The secondary 3D views show the field of view of a scientific instrument. The camera of secondary views cannot be manipulated directly by the user. An environment was created that stores all secondary views. When adding such a view one has to specify the viewer object(the instrument or object where the camera is placed) and the destination object. Then the camera is simply tilted toward the direction of the destination object. Of course, it has to be updated continuously in order to reflect movement changes of the two objects. However, the destination must not necessarily be a visual object(VizObject), but can also be an axis. If the parameters that define this axis are null then the software automatically skips this part and sets the destination object's direction for the camera. Otherwise, the direction is set to the specified axis, regardless of the specified destination object. Other parameters used in the creation of these views are the width and height of the window which specify the resolution of the 3D window. An example of such a view can be seen in figure 6.22.

In addition to the views mentioned above, the software also provides inner views, which instead of being created in a new window are created inside the main window, as a viewport - the size can be specified. These are currently not used, because they are not that useful and were actually created as an intermediate step towards the creation of secondary stand-alone views. However, they might prove useful later on and have not been excluded from the project.

Chapter 7

Testing and Validation

7.1 Testing

Software testing is necessary in order to assess the quality of an application and more specifically is a method of checking that:

1. **The software meets the specified requirements.**
2. **The software works as expected.**

Testing is also used to identify any defects, bugs or other errors that might be normally overlooked. The problems that have been found by testing can be fixed and help create a more robust software.

Three main types of testing have been performed in the development of this application. *Code Testing* - which refers to testing at code level, *Functional testing* - which is used to test the functionality of performed tasks and *Application Testing* or system-level testing which is used to test the application as a whole after the modules that perform individual tasks have been integrated into a single, larger, application module.

A performance evaluation has also been performed, however only one machine has been used to test the recent version of the application. In order to get a more exact performance evaluation the application should be tested on a larger number of similar lower-end machines, in order to factor in variance and establish a minimally acceptable performance level. Moreover, the hardware specifications of the machines that achieve this level of performance will be considered the *minimum system requirements*.

Validation has also been performed on some controls, where it was required. Because most of the GUI controls that would be used to supply input is not currently implemented, validation is only applied in a few instances. More details in section 7.3.

7.1.1 Code Testing

Code testing has been carried out manually. Test cases have been created to incrementally test the components of the system. This form of testing is a type of *white-box testing*.

If one performs testing on the structural part of an application then that method of testing is known as *White Box Testing*. It is usually done by Developers or Test Engineers.

The goal of manual testing is to ensure that the code does what it is supposed to do and is error free. Testing was performed first at unit level and once all units of a module were confirmed to be working, the whole module was tested for errors. Please note that there were no unit testing classes created. The code was manually tested by supplying valid, significant input parameters and observing the output results. Each method and class was tested this way until it was confirmed to work correctly. The input data shall be restricted to only valid results, so no testing is necessary for other values. Since this is a visualization application the outputs of a test case were easily observed visually in most cases.

A *trial and error* approach was also used in some cases. This is usually a highly-criticized approach because the testing is not structured and arguably - random. However, in some cases this approach might actually save time if the solution to the problem is intuitive.

In order to achieve a more robust code testing - unit test classes can be created with a tool such as *JUnit*. If not for all the components, it should be used to test the main parts of the system. This also allows automation of testing. Classes and methods can be tested repeatedly just by running these unit tests with different input parameters.

7.1.2 Functional Testing

Testing at this level is performed on a task by task basis. The system's required functionality is tested, module by module, to make sure that each part works correctly or to find and correct any errors.

The following tasks have been tested:

- **Creating a New Simulation**
  The first step is to activate the new simulation control. Afterwards, the time interval must be selected as well as a time step. If all is well, in the third step the display settings can be selected, otherwise the supplied input dates are invalid/not found. The display settings are selected and the system opens the main 3D window after this step(if the current machine supports the minimum system requirements).

  A test on this task also proves the functionality of its subtasks: *Allowing the user to select a time interval for the simulation* and *providing graphics configurability options*.

- **Only One Active Simulation**
  If a user tries to create more then one active simulation it will not work(the system will restrict this behavior). However, a problem is that if the current simulation is closed and the main menu remains opened, the user cannot create another simulation without restarting the whole system.

- **Displaying Sun, Solar System(Cosmic) Objects, Spacecraft, Atmosphere, Mask Objects and Their Subcomponents In a 3D Scene**
  This test is done by creating a new simulation. Instances of all of the required object types are in the main scene, with the exception of the mask, which is only visible in instrument view windows. The functionality of each object types is tested: the sun gives the light, the Solar object(the comet) can have a landmark as well as the other common components, while the Rosetta spacecraft can have a boresight in addition to the other components.

  The atmosphere is also placed at the comet's location. More testing is necessary to achieve a better result.

  The component objects each serve their purpose: the coordinate arrows show the cartesian coordinate system, each pointing arrow points to a destination object, a distance line is drawn between the comet and the spacecraft, a boresight in the z direction is added to the spacecraft, a landmark is placed on the comet, a trajectory line shows the spacecraft's full trajectory while a trailing trajectory line shows the spacecraft's trajectory from the starting point until the current point, a region of interest is drawn on the comet in other to test its functionality and finally caption text is added to the comet and the spacecraft to test that it works correctly.

- **Displaying Different Mask Types**
  This functionality is only partially completed. All the masks that can be created in the current version of the application are square or rectangular masks. If an instrument has a circle, hexagonal or diamond field of view the currently available masks will not suffice.

- **Adding and Viewing Charts**
  Charts that can plot any number of parameters can be added to the application. The charts can be displayed at the same time that a simulation is running. The plotted data is synchronized with what is happening in the 3D scene - the charts are updated in real - and simulated time. Once the user clicks the control that corresponds to opening a particular chart - it will be displayed in a new window.

- **Hiding and Showing Objects**
  Hiding and showing objects works for both primary object types and their component objects. Each component object can be turned on or off independently.

- **Changing Camera Rotation and Translation Speed**

Two controls used for increasing or decreasing camera movement speed can be successfully used to carry out this task. However, the rotation functionality was not implemented - it became clearer that it is not useful, as the default rotation suffices.

- **Adding and Viewing Instrument Views**
  Different instrument views can be added to the application. They can be displayed once the simulation is running. Optionally, these views can be partially covered by a mask. Moreover, they can also display the name of the instrument that is used to view the scene. Once the user clicks the control that corresponds to opening a particular instrument view - it will be displayed in a new window.

- **GUI - Integrating Other Tasks**
  The application GUI integrates all of the other functionalities presented - it holds all the controls to do the required tasks and provides the frames that contain the 3D scenes and charts. All controls are tested by observing the behaviour and a check is performed against the expected outcome.

- **Load All Objects Using a Database**
  All of the objects are loaded automatically at runtime, including all their properties and position, rotation and scale information. The parameters that are used to create the objects are read from a database. Moreover, the visualization library that was part of the scope of this project successfully transforms these parameters into the required 3D and 2D entities.

The modules have later been integrated into the same application using a bottom-up approach.

7.1.3 Application Testing

This is a type of *black box testing* which means testing the functionality of an application as opposed to its internal structures and workings. At this level of testing all modules must have been integrated somehow in the same application. This integration is part of the white box testing. Furthermore, tests are performed at application level.

At the system testing level all parts that were integrated into the application are tested to verify that all elements function properly - as a whole. For example, multiple instrument view windows should be able to be open at the same time that the application is running. All functionality of both the main and instrument windows should be intact. Moreover, multiple 2D charts can be created and should also function properly and be in sync with the 3D data. Another example of system testing is when showing or hiding different components - changes are reflected both in the main and secondary windows.

The main window is also navigable and the scene can be zoomed in and out and the camera movement speed can also be changed. All of the actions mentioned were tested as a whole

and were proved to work correctly.

7.2 Performance Evaluation

Performance evaluation for this project has two main aspects: start-up speed and general application speed - which can be measured in *fps*(frames per second). Tests have been carried out on a benchmark machine. The system's hardware specifications can be consulted in section 8.1.

The start-up time is a bit long due to a very large database with millions of records that are used to plot the spacecraft's position. The start-up time should however, be less then a minute in general(in 90% of cases), and about 10-15 seconds for smaller datasets.

The fps rate is generally above average if only the main visualization window has been opened. In this state the fps rate should be greater then 30 - if it is not, chances are that you are using a low-end machine to run the application, and need to upgrade. The fps rate in this state has surpassed the value of 100 by using the benchmark computer - this is great, considering that the fps rate that video games are said to run great at is about 60.

However, the fps are reduced each time a new 3D or chart window is opened due to the additional processing power that is required. If too many of these windows are opened the application may reach a very low fps rate, which will make caring out useful tasks impossible. A guideline is to usually open at most 2 or 3 extra windows. This should be enough for most scenarios and the system should still run at an average or workable fps rate.

7.3 Validation

Some constraints are placed on a number of controls(where needed) in order to only allow valid data to be passed to them. For example, the simulation speed is restricted to be at minimum *1x*. Slower speeds are not useful in any way since we are working with a huge set of data.

The only input validation needed is at the new simulation dialog box, when the supplied time interval is checked against the database to see if the data is found. The display settings dialog only shows valid values, so no further validation is necessary.

In the case that the application is extended with a full GUI that has CRUD functionality, a lot of validation controls will be needed to make sure that the data that is placed inside the database is valid - otherwise there will be problems that might crash the system.

Chapter 8

User's Manual

8.1 System Requirements

The software requires a decent machine in order to run normally. A good graphics card with 3D Acceleration is required in order to run smoothly. The benchmark computer has the following specifications:

- **CPU:** Intel Core 2 Duo, 2Ghz
- **CPU Architecture:** x64
- **Video Chipset:** nVidia GeForce 9600GT M, 512MB memory
- **RAM:** DDR2 PC2-5300, 4GB(2x2GB slots)
- **HDD:** WDC WD3200BEVT-22ZCT0, 320GB capacity, 5400 RPM
- **OS:** Windows 7, 64 bit

A computer with lower hardware specifications is not recommended to run this application. Although it may work to some extent, it will most likely perform poorly. A main factor that influences performance is the video card, so even if the other components are not as powerful as the above, performance may be impacted in a positive way.

Furthermore, a monitor that supports at least *800x600* resolution is recommended(the default window size is *640x480* + space needed for the menu), but it will not suffice if multiple windows are opened. For the full functionality a higher resolution, or, even better, multiple monitors are required to take full advantage of the windowing system. Please note that both the higher resolution and the number of windows opened can drastically affect performance - these features require more sophisticated hardware.

Finally, about 300MB of hard drive space is required(including the database files and extra assets).

8.2 Installation

In order to install the application one must first deploy the database on a MySQL server using the supplied SQL file(s). If you are using a *WAMP* or *XAMPP* server, please note that the dataset is large and may not be imported correctly with *phpMyAdmin*. A more robust tool is the *MySQL WorkBench*. Of course, the data can also be imported manually.

The next step is to copy the software files: assets + executable *.jar* file. In order to run the *.jar* the Java Runtime Environmnet(JRE) must be installed on the system. It can be downloaded from the official Oracle website: `http://www.oracle.com/technetwork/java/index.html`.

8.3 Using the Application

How to use all features of this application shall be described in the following section. The details provided can make the application very easy to use even for non-expert users.

1. **Main Menu**
   Once the application has started you are greeted with the main menu. A new simulation can be started by going to the *File* menu and choosing the *New Simulation* option. In addition, on the toolbar beneath the menu there are three sets of controls

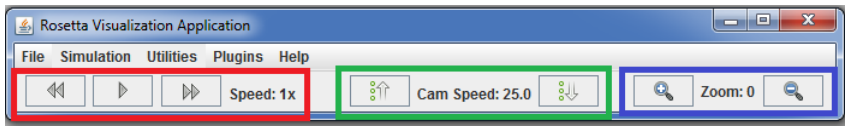

Figure 8.1: Main Menu. Simulation Speed Controls - Red; Camera Movement Speed Controls - Green; Zoom Controls - Blue.

   used to accomplish the following tasks: changing the simulation speed, changing the camera movement speed, zooming in and out. All of these tasks shall be detailed later on.

2. **New Simulation Window**
   In the newly created dialog box, you must choose a start and an end date. The simulation contains data from *November 14, 2012 00:00:00 hours* to *November 17, 2012, 00:00:00 hours*. Make sure that you specify a time interval contained in the

aforementioned period. Otherwise, an error message will appear and you will be able to retry this step. Make sure the dates are validated by clicking inside the box - an invalid format will be shown in red while a valid format will be shown in green. An additional *time step* parameter is used to specify how much data to include in

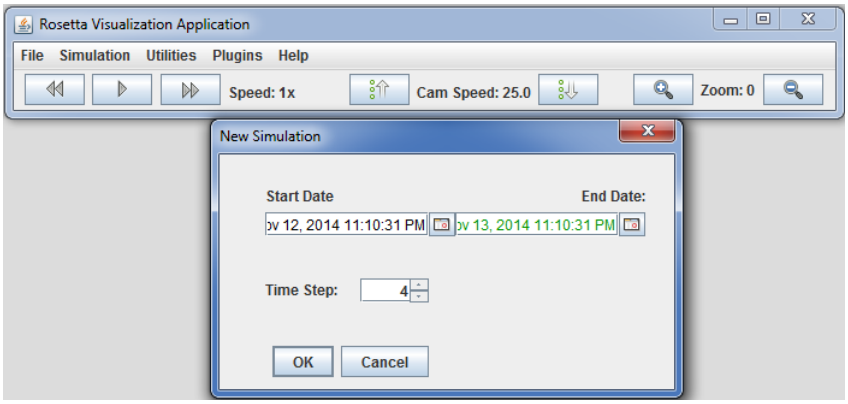

Figure 8.2: New Simulation - Date And Time Step Selection

the simulation. The simulation will have n times less data and thus will be n times faster, where n is the selected value. Click *OK* to continue.

3. **Display Settings Dialog Box**
   Next, the display settings dialog box shall appear on the screen. Here you can select the resolution, bits per pixel, enable V-sync and other options. Please note that the true fullscreen mode is NOT supported. The window can be maximized later, after it is opened(but this will not get rid of the menu bars). Click *OK* to continue.

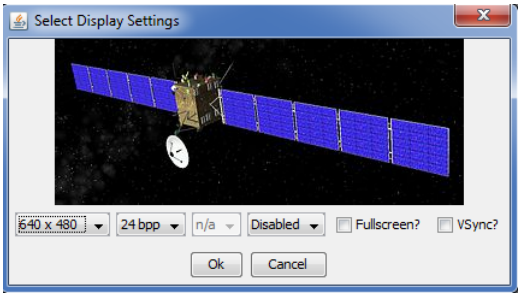

Figure 8.3: Select the Display Settings

4. **Main Visualization Window**
   The main visualization window opens. Now, a number of tasks can be performed. First, you can move around the world using the W,A,S,D keys and the mouse. You can hold the left mouse button in order to drag(rotate) the camera.The camera is a classic *FPS*-type(First Person Shooter) camera that is included in so many games.

5. **Changing the Simulation Speed**
   The speed can be changed using the recorder type controls on the left side of the toolbar(red area in figure 8.1). The minimum speed is *1x*. There is no current maximum speed limit set, but the application will start to run sluggishly if the speed is too great(in the number of tens of thousands).

6. **Changing Camera Movement Speed**
   In addition, you also have the option of changing the movement speed of the camera by using the two up and down arrow buttons on the main menu(see green area in figure 8.1. The default speed value is 25 and it is a good speed to move about at medium distances. A higher number means a greater speed. A negative number is used to reverse controls.

7. **Zooming In and Out**
   Furthemore, you an also zoom the main scene - in and out, using the zoom buttons on the main menu. The default zoom factor is 0. You can go from -100 to 100. These buttons are found in the blue area in figure 8.1.

8. **Hide/Show(Add/Remove) Objects**
   From the simulation you can also hide and show most objects in the scene. In order to hide or show a *VizObject* click on the bold-font menu option that has its name to remove and add it back. In order to remove and add subcomponents you need to go to the regular font menu which will open a new menu with the children nodes of the object. Next, some children also have children of their own, or grandchildren with respect to the object nodes. Everything in the list can be turned on/off. By default, all objects are visible and this can clutter the screen - this is because the application is a prototype used to showcase the features that can be built using the development technologies, not a final product.

   An example of adding(showing) and removing(hiding) the Rosetta spacecraft is shown in the figures below.

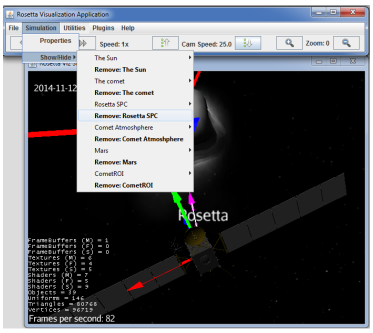

Figure 8.4: Before the Remove Command

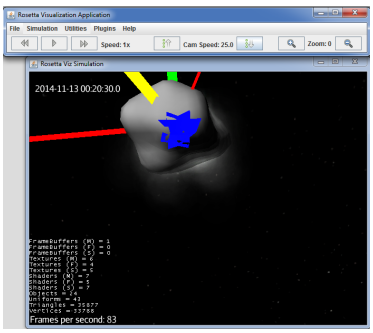

Figure 8.5: After the Remove Command

9. **Adding a chart** From the *Utilities* menu you can add a chart. Just click on the

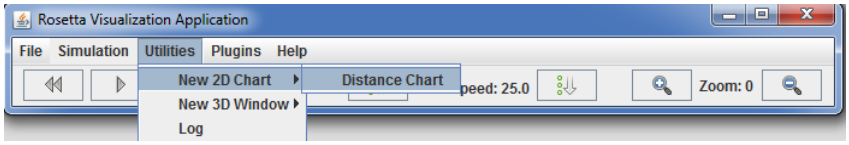

Figure 8.6: Add a chart from the menu

appropriate item and a new window will open that contains a chart. It will run in parallel to the main simulation and will use the same data in real-time to carry out the required plotting. If you are done with it, you can close it and go back to doing some other action.

The result of opening the distance chart window can be seen in the figure below: The

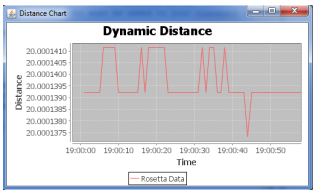

Figure 8.7: Example Chart

chart can also be zoomed in and out in order to see the data more clearly. To do this right click on the chart and choose your preferred zooming method. Alternatively, you can also drag up or down on a small portion of the plot and the chart will automatically focus and zoom in on it.

10. **Adding another(instrument) 3D Window**
Finally, the last action that can be taken is to display instrument views. Go to the Utilities menu, offview section and select the desired view. It will open in a new

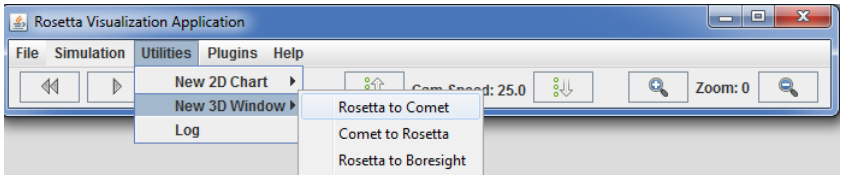

Figure 8.8: Add a chart from the menu

window and display whatever it is required to display, all features included. Any number of windows can be opened, but note that every new window requires more hardware resources to be displayed at a descent fps rate. If the application becomes sluggish close some windows to improve performance.

An example of a window is shown in the figure below. In this case the *Rosetta to Boresight* option was selected.

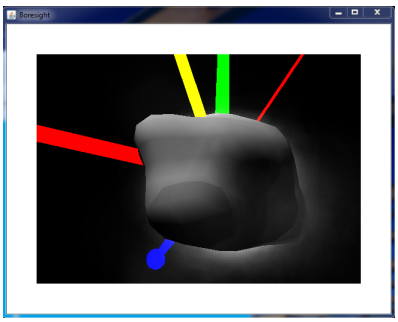

Figure 8.9: Rosetta's View Along the Boresight

Chapter 9

Conclusions and Future Work

9.1 Conclusions

This project's main mission was to find suitable visualization technologies and integrate them into one cross-platform prototype which can be integrated into a larger Rich Client Platform. Space mission visualization is important because some problems may become more evident, certain scenarios can be replayed a number of times and planned ahead of time and decisions can be taken quickly based on the available data. It can help the overall mission planning and consequently, reduce risks and cost.

Moreover, an important aspect was developing a visualization library that allows a robust and flexible handling of space mission visualization. The library provides the functionality of creating a 3D visualization of such a mission by just using some parameters which are read from a database. This is very powerful, since it allows stakeholders to change the contents of the scene just by editing the database - a completely different mission could be visualized just by supplying different(corresponding) data, without having to write another line of code. Of course, the previous holds only if the other mission doesn't use other types of objects or behaviour. Otherwise the new entities must be created, but the library can be easily extended and allows developers to add new types of objects and behaviour by following some simple guidelines - much easier then creating everything from scratch. What is currently included the library can be thought of as the *core functionality* needed for this specific type of visualization.

The visualization tools that were selected during this endeavour have proven their versatility and ability to carry out the required tasks during practical use. They were successfully integrated in the same Java desktop application. Furthermore, they assisted in creating the visualization library and also helped develop a *Rosetta* mission prototype application that uses the previously created library - all in one application.

Most of the features specified in the functional requirements were successfully implemented in this prototype. Some, such as the mask or region of interest were not implemented completely so to speak, but a good basis for future extensions is included. Moreover, there was not enough time to create full GUI with all needed functionality - including database manipulation. Perhaps, the most important feature that has been successfully implemented, is the fact that the whole 3D scene, all charts and secondary windows, all behaviours and data are actually some parameters in a database that the created visualization library transforms into actual usable objects at runtime.

Given the obtained results, the project has successfully completed its objectives. The researched development technologies were used to create a visualization library included in a prototype application that can be used to visualize the Rosetta space mission - and other, similar missions. The software can also be easily extended with new features, in order to provide the flexibility for the software to be used in other projects. Even if the application will not be used in its current form, it provides the basis for embarking on the creation of a fully-fledged space mission visualization tool.

9.2 Future Work

A future improvement that should provide greater flexibility in new object creation, or loading some particular set of objects, geometries and materials would be to create an *XML* file that has the name of each class and the corresponding entity name specified. This would allow developers to read the *XML* file and store the values in the internal hash table used for the geometry and material factories. Thus, the system would know exactly what type of objects it can create and how to create them.

Future extensions include the creation of new objects and types of behaviour and fixing the problems(or polishing) of existing ones. Some important features are listed below:

- The particle emitter should be created from database as this would provide a variety of possible effects at the user's fingertips, not only atmospheric affects. A good example of a needed effect is space dust.
- The region of interest should be extended to, perhaps, a shaded material that looks less rigid and the appropriate part of te object data should be supplied as input - in order not to have the missing triangles that are currently seen in ROI object.
- Another useful extension is to diversify the current field of view masked used on instrument views. New shapes should be added such as: circular/oval, hexagonal, diamond and others.
- A useful object that was not included in the current prototype is a cone-like figure, but with the end in the shape of the instrument field of view - the same shapes

used for the masks. This would allow to create these c̈oneswẅith the origin on the spacecraft and associate an instrument to each of them. By giving them different colours one will be able to see what area each instrument covers at a particular point in time.

- Improving the reading of very large data sets to reduce start-up time.
- Creating a complex GUI which allows the super user to make changes to the database from the application. Good validation techniques are necessary to make sure that the data that makes it to the database is correct.
- Optimizing the 3D operations and the overall integration of the different technologies in order to achieve better performance is also an important aspect to consider in the future.

Bibliography

[1] G. Domik, "Introduction to Visualization," University of Paderborn, the first chapter from the course was used.

[2] "ESA, Rosetta Mission Overview," http://www.esa.int/SPECIALS/Rosetta/index.html.

[3] S. M. R. P. John Wright, Scott Burleigh and M. Maruya, "Visualization Experiences and Issues in Deep Space Exploration."

[4] B. W. C. Sarah C. Daugherty, "The Mission Planning Lab: A Visualization and Analysis Tool," Wallops Systems Software Engineering Branch, NASA Wallops Facility, Wallops Island, VA, USA.

[5] M. W. L. Ken Museth, Alan Barr, "Semi-Immersive Space Mission Design and Visualization: Case Study of the "Terestrial Planet Finder" Mission," California Institute of Technology, Pasadena, CA 91125.

[6] "OpenGL Lighting Tutorial - Spacesimulator.net," http://www.spacesimulator.net/wiki/index.php?title=Tutorials:OpenGL_Lighting, accessed in: 2011 and 2012.

[7] D. Gorgan, "Graphical Processing Systems - Lighting Models," Technical University of Cluj-Napoca, only the course on lighting was used.

[8] J. F. H. Tomas Moller, "Efficiently Building a Matrix to Rotate One Vector to Another," *Journal of Graphics Tools*, vol. 4, no. 4, 2000.

[9] "An Introduction to the OpenGL Shading Language," Jan. 2008.

[10] "OpenGL - The Industry Standard for High Performance Graphics," http://www.opengl.org/, accessed in: 2011 and 2012.

[11] "The Official Visualization Toolkit Website," http://www.vtk.org/, accessed in: July, 2011.

[12] L. S. A. William J. Schroeder and W. Hoffman, "Visualizing with vtk: A tutorial," *IEEE Computer Graphics and Applications*, 2000.

[13] K. M. Will Schroeder and B. Lorensen, *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, 3rd ed. Kitware Inc., 2002.

[14] "jMonkeyEngine 3.0," http://jmonkeyengine.com/, accessed in: 2011 and 2012.

[15] "jMonkeyEngine 3.0 - JavaDoc," http://jmonkeyengine.org/javadoc/, accessed in: 2011 and 2012.

[16] "jMonkeyEngine 3.0 - Forums," http://jmonkeyengine.org/forums-index/, accessed in: 2011 and 2012.

[17] "jMonkeyEngine 3.0 - Documentation and Tutorials," http://jmonkeyengine.org/wiki/doku.php/jme3#tutorials_for_beginners, accessed in: 2011 and 2012.

[18] "JOGL - Java Binding for the OpenGL API," http://jogamp.org/jogl/www/, accessed in: July, 2011.

[19] "Home of the Lightweight Java Game Library," http://www.lwjgl.org/, accessed in: July, 2011.

[20] "UDK - The Unreal Development Kit," http://udk.com/, accessed in: July, 2011.

[21] "Unity - Game Engine," http://unity3d.com/, accessed in: July, 2011.

[22] "Java 3D API," http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138252.html, accessed in: July, 2011.

[23] "Java 2D API," http://java.sun.com/products/java-media/2D/index.jsp, accessed in: July and August, 2011.

[24] "ImageJ API," http://rsbweb.nih.gov/ij/, accessed in: July, 2011.

[25] "JFreeChart Library," http://www.jfree.org/jfreechart/, accessed in: 2011 and 2012.

[26] "JChart2D Library," http://jchart2d.sourceforge.net/, accessed in: 2011.

[27] "The Official Java 6 Documentation," http://www.oracle.com/us/technologies/java/overview/index.html.

[28] "The Celestia Motherlode," http://www.celestiamotherlode.net/, accessed in: 2011 and 2012.

[29] "Celestia," http://www.shatters.net/celestia/, accessed in: 2011 and 2012.

[30] "JSatTrack - Java Satellite Tracker," http://www.gano.name/shawn/JSatTrak/, accessed in: July, 2011.

[31] J. E. Bentley, "Software Testing Fundamentals - Concepts, Roles and Terminology," *SUGI 30 Proceedings*, no. 141-30.

[32] "Manual Testing," Prasanna.

[33] G. S. Richard S. Wright, Nicholas Haernal and B. Lipchak, *OpenGL: SuperBible: Comprehensive Tutorial and Reference*, 5th ed. Addison-Wesley Professional, 2010.

Appendix A

A Simple Shader Example

The following information has been obtained by courtesy of *jMonkeyEngine.org*.

In the following sections the vertex and fragment shader are used in order to render a solid colour on a test object.

- **Vertex Shader**

```
//the global uniform World view projection matrix
uniform mat4 g_WorldViewProjectionMatrix;
//The attribute inPosition is the Object space position of the vertex
attribute vec3 inPosition;
void main(){
    //Transformation of the object space coordinate to projection space
    //coordinates.
    //- gl_Position is the standard GLSL variable holding projection space
    //position. It must be filled in the vertex shader
    //- To convert position we multiply the worldViewProjectionMatrix by
    //by the position vector.
    //The multiplication must be done in this order.
    gl_Position = g_WorldViewProjectionMatrix * vec4(inPosition, 1.0);
}
```

- **Fragment Shader**

```
void main(){
    //returning the color of the pixel (here solid blue)
    //- gl_FragColor is the standard GLSL variable that holds the pixel
    //color. It must be filled in the Fragment Shader.
```

```
    gl_FragColor = vec4(0.0, 0.0, 1.0, 1.0);
}
```

- **Material Definition File**
  In the next code snippet the vertex and fragment shader colours are used inside a material definition file that is created. Its purpose is to apply a solid material to an object. The *SolidColor.vert* and, respectively, *SolidColor.frag* files can be the two code snippets shown above.

```
MaterialDef Solid Color {
    //This is the complete list of user defined uniforms to be used in the
    //shaders
    MaterialParameters {
        Vector4 Color
    }
    Technique {
        //This is where the vertex and fragment shader files are
        //specified
        VertexShader GLSL100:   Common/MatDefs/Misc/SolidColor.vert
        FragmentShader GLSL100: Common/MatDefs/Misc/SolidColor.frag
        //This is where you specify which global uniform you need for your
        //shaders
        WorldParameters {
            WorldViewProjectionMatrix
        }
    }
    Technique FixedFunc {
    }
}
```