Development of an Environment for 3D Visualization of Riser Dynamics

João Luiz Bernardes Jr., Clóvis de Arruda Martins

Escola Politécnica da Universidade de São Paulo

joao.bernardes@poli.usp.br, cmartins@usp.br

Abstract. *This paper describes the merging of Virtual Reality and Scientific Visualization techniques in the development of RiserView, a multiplatform 3D environment for real time, interactive visualization of riser dynamics. Its features, architecture, unusual collision detection algorithm and how UP was customized for the project are discussed. Using OpenGL through VTK, the software is able to make use of the resources available in most modern Graphics Acceleration Hardware to improve performance. IUP/LED allows for native look-and-feel in MS-Windows or Linux platforms. The paper discusses conflicts that arise between scientific visualization and aspects such as realism and immersion, and how the visualization is prioritized.*

1. Introduction

This paper describes the development of a multiplatform, tridimensional environment for real time, interactive scientific visualization of riser dynamics, making use of virtual reality techniques.

To understand this objective, risers and their use must first be explained in further detail. For now, suffice to say that risers are vital structures for the offshore oil industry, subject to strains of diverse (and often complex) natures. Risers, along with that industry in Brazil, are discussed in more detail below.

Due to the importance of these structures as well as to the strains they may be subject to during operation, risers have been the object of intense academic scrutiny for decades, resulting in large volumes of data from experimentation, analytical modeling and numerical simulation. The interpretation of all this data may be greatly facilitated through the use of scientific visualization techniques, especially if merged with virtual reality. This is the objective of this work.

1.1. Risers, Vortices and the Offshore Oil Production in Brazil

According to Ferrari (1999), risers are tubular structures which link the underwater oil well and the platforms or ships above water responsible for well-drilling or oil production. Risers are used in both activities. Figure 1a illustrates one of these structures connecting a platform to a well head.

During drilling, the drill is affixed to a *vertical riser*'s extremity and guided by devices on the well's head. The riser also transports drilling mud, a fluid responsible for, among other things, aiding in the riser's structural sustentation, cooling the drill and avoiding the backflow of hydrocarbons once they are reached. During production the riser is the means through which the hydrocarbons are transported from the well to the platform or ship. In this activity, both *rigid* and *flexible* risers may be used. For large depths, rigid risers may be used assuming a catenary shape, much like flexible risers. The curvature the riser assumes in this shape, and especially the variations of this curvature during operation, may be responsible not only for another strain on the riser's material, but also for fatigue effects.

Other strains come from the riser's own weight, internal hydrostatic pressure of the fluids it carries, and environmental phenomena such as waves, winds and currents which can cause the movement of the ship or platform (and thus of the riser) and originate vortices around the riser.

Jeong & Hussain (1995) affirm that the very definition of a vortex can be complex. For this paper, however, this simpler definition may be used: a vortex is a multitude of material particles rotating around a common center while immersed in a flow. Particularly for the problem of risers, vortices are shed around them, in the ocean's waters, behind the riser in relation to the current. Vortices may generate cyclic forces on a plane perpendicular to the current's direction which may be responsible for fatigue of the riser's material. This explains the importance of their study in relation to risers. And since vortices can be rather difficult to visualize based only on numerical data, Scientific Visualization techniques find here a prime application. Figure 1b shows a visualization of vortices being shed around a cylinder. One instant of the numerical simulation of the vortex street (sequence of vortices) is shown from above. The lighter areas of the flow spin clock-wise while the darker areas spin in the opposite direction.

Why, however, go to all this trouble to explore oil offshore? Because large reserves of this precious resource may be found under the oceans. Especially in Brazil. According to Petrobrás (2004), only less than twenty percent of all the oil currently produced in Brazil may be found on dry land. And this percentage tends to shrink even smaller as more fields are discovered offshore. If merely thirty years ago the bulk of offshore exploration was done at a depth of thirty meters, nowadays Petrobrás holds the world's record in production depth, at almost two thousand meters in the Roncador field. It was this pioneer increase in depth that allowed for the increase in the percentage of oil produced offshore as well as in overall production in the country. It is also what demands (and is made viable by) the intense research mentioned previously.

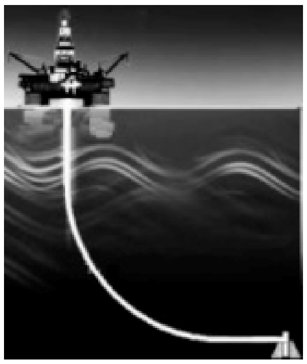	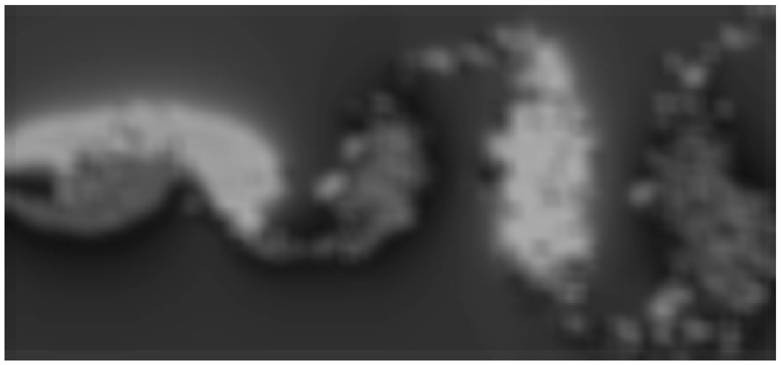
a) Riser	**b) Vortex Street**

Figure 1 - Risers and Vortices

1.2. Scientific Visualization and Virtual Reality

The research mentioned above, the volume of data it produces and the fact that the interpretation of this data may be rather complex justify the use of Scientific Visualization to aid in the treatment and interpretation of data related to risers.

Scientific Visualization, according to Schroeder (1998), is a branch of Computer Graphics concerned with the exploration, transformation and exhibition of data as images or other sensory forms to gain understanding and perception of said data. For this end, most visualization techniques make use of the human eye's excellent pattern recognition capabilities. Visualization is further characterized for handling data of at least three dimensions (and usually more) and for the interactive manipulation and transformation of data by the user, thus including the human element directly in the process.

One of the most widely used visualization techniques is color mapping, where distinct colors are assigned to groups of values a scalar variable may assume and then used to build a picture. The use of scalable glyphs to represent points and associated values or vectors, deformation of structures to show movement or strain as well as animation to add more dimensions to the visualization are also common techniques. These are the main artifices used so far in RiserView, although they are far from a complete listing of visualization techniques.

For multidimensional data such as the majority of the data treated by Scientific Visualization and more specifically, for time-varying spatial data such as the data associated with riser dynamics, the merging of visualization techniques with techniques used in virtual reality significantly enhance the capacity for the data's exploration and interpretation. Stereoscopy renders the tridimensional data that much more palpable and the scene-navigation techniques used in virtual environments make the data exploration a more intuitive process, akin to exploring a physical environment. So despite the fact that this work may not consist of a virtual environment according to the formal definition of the term (the user's capability to interact with the environment is severely limited, since the very objective of the software is to show the dynamic of risers without that sort of interference), the merging of RV and Visualization is an important aspect of it.

2. RiserView

It has been said above that RiserView consists of a tridimensional environment for the visualization of riser dynamics. How exactly it accomplishes that, however, is what will be explained in this section.

The main function of the software is to show the riser's configuration in space in each moment of time. The shape the riser assumes is strongly influenced by the strains it is subject to and even a quick visual inspection can show the points of higher curvature or where the regions where this curvature changes more abruptly once the riser can be seen in its entirety. RiserView may make use of stereoscopy to render a scene for each eye (making use of appropriate hardware) to allow a better exploration of the environment. Or, lacking stereoscopy hardware, the software may simply render conventional frames depicting the scene. The magnitude of variables such as tension or curvature may be also shown on the riser through color-mapping.

Risers are often used in groups, and RiserView is not limited to show a single riser, but is instead prepared to render large numbers of these structures.

The user must also be able to navigate through the scene, meaning zooming in or out of any of its elements, rotating the view or panning it. This is accomplished chiefly using a two-button mouse, aided by the keyboard. This guarantees that the user can see the entire configuration of risers or focus on a particular one, or a particular region of a riser, from whatever angle he needs.

Animation is used to show how these configurations change along the time dimension. The animation must be done carefully to make sure it allows the user to interact with the environment. Since the animation is handled by a loop always running in the software's background, and the application uses a similarly infinite loop to capture user-generated events, these loops must either be treated in different threads or implemented in a way that accounts for both. Another decision that must be made is which of these activities (rendering and animation or treatment of user events) must be prioritized in case both can't be satisfactorily processed. These decisions are discussed below, under "Project Decisions".

Although the focus of the visualization are the risers, several other elements may be used to compose the scene and serve as reference when studying them. RiserView allows for rendering of the water's surface, which may also be animated showing the behavior of the waves. Since waves are often an input to the determination of the dynamics of both risers and ship or platform, their exhibition may aid in the understanding of riser dynamics. The ships and platforms may also be shown, as rigid body models imported from 3D Studio which may also have movement. Likewise, other structures, static or not (for instance buoys or the well's head), may be imported and added to the scene as rigid bodies. The ocean floor with its own relief is another static element of the scene. Finally, vortex shedding and their progression in time may also be shown.

RiserView must also detect collisions between risers. Unlike in most virtual environments, the user cannot collide with any of the elements of the scene, so he may reach points to observe the risers that could otherwise be blocked. The risers are attached to the ships or platforms, so there's no need to detect collision between them. And although ships could collide with each other, this is more easily predicted and avoided than collision between risers. Collisions with the waves or the ocean bottom make even less sense. Collisions between risers, however, may occur in reality and have considerable impact on the riser's integrity. The software detects these events and shows them clearly to the user, pausing the animation when a collision occurs and highlighting the area where it does.

This environment, however, has no dynamic model of risers and performs no numeric simulations. Its function is to show the results of previously-run simulations, instead. Thus, the input for RiserView is actually the output of one or more programs responsible for calculating the dynamics of risers, sea surface, ships, vortices etc. Figure 2 illustrates RiserView's functions through its inputs and outputs.

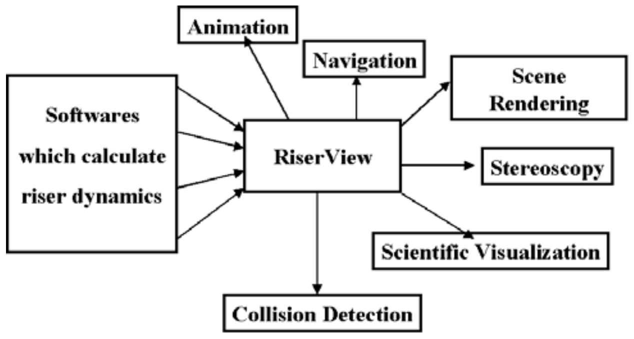

Figure 2 - RiserView

2.1. Project Decisions

The features listed above for RiserView were all defined in the initial phase of software design and later translated into use cases and non-functional requirements. Some questions arose before this translation, however, that had to be addressed.

One of the first issues to be considered is related to the animation framerate. Each scene may be composed by several distinct elements, each resulting from distinct numerical simulations. Although risers may influence each other based on their proximity (due to the formation of vortices for instance), distant risers need not be simulated together. Likewise, different programs may be responsible for determining the dynamics of the water's surface or of the ships. Each of these simulations has its own optimum time step and they are not necessarily equal. Due to constraints on development time, however, in this first version of RiserView no interpolation of positions between time steps was developed (although it is one of the future works discussed in section 4). The animation framerate, then, is limited by the time steps of each simulation. In fact, with no interpolation, the software works best when the dynamics of every element in the scene is described using the same time step. In the initial phase of development this was obligatory, but later allowances were made to work with the smallest time step available, although some discrepancies may still occur if the time steps are not multiples of each other.

Another question that has already been mentioned refers to the animation loop and the detection of user-generated events. Instead of making use of independent threads for these processes, RiserView opts for handling them in a single main loop. This loop prioritizes the rendering of the scene, using only the time left after the rendering to treat user interaction. For every frame rendered, however, at least one user event is treated. For scenes with a high polygon count and framerate this may result in jerky interaction and poorer immersion. The alternatives, however, reducing the framerate or rendering an incomplete scene, were deemed less appropriate. Section 4 discusses other strategies to deal with this issue.

One important decision that had to be made at the beginning of the project regards conflicts that may arise between the scientific visualization and realism. Although in 1987 Upson already predicted that these two elements are certainly not mutually exclusive and many techniques used for the increase of realism may aid in the interpretation of scientific data, sometimes conflicts do happen. Color Mapping, for instance, negates the use of (or is hampered by) the use of realistic textures on the object of study. Another non-realistic technique often used in Scientific Visualization is the exaggeration of deformations or movements to facilitate their perception. Although RiserView strives for realism, allowing the use of textures, anti-aliasing and user-defined colors, for instance, whenever it clashes with the visualization the latter is prioritized.

Finally, since early on in the project a non-functional requirement was established, regarding RiserView's platform: It must be compatible with both MS-Windows and Linux windowing systems. Both platforms enjoy ample use in riser-related applications thus justifying this requirement. This impacted heavily on the selection of an APIs for the project.

2.2. Architecture

RiserView's software architecture was based primarily in the classical Model-View-Controller (MVC) pattern, popularized with Smalltalk 80, according to Veit & Herrmann (2003). This pattern is used to isolate data and algorithms related directly to the problem from those related to the user interface. The Model class encapsulates the problem-related data, which may be associated to several distinct Views, distinct user interface elements each responsible for showing a different visualization of the Model's data and accepting user input. Each View is also associated with a Controller responsible for the communication between Model and View.

Figure 3 is a simplified class diagram (for a UML reference see Booch et al., 1999) showing the implementation of the MVC pattern within RiserView. The class named after the application serves as a container for the RVModel, RVView and RVControl classes. RiserView has a single view, the rendered scene, and thus a single control. RVControl handles the user input, first captured in RVView through IUP's callbacks, and prompts modifications in the view when they result from user input. The controller contains the main loop and is responsible for informing the model of changes in simulation time, thus modifying the model. Another way in which the controller modifies RVModel is through user inputs (for instance adding elements to the scene). Since all model changes are prompted by this single controller, there is no need for the former to notify the latter of those changes, unlike other MVC implementations. RVModel notifies RVView directly of its changes, however. This notification is encapsulated within VTK's visualization pipeline (see below). RVInteractorStyle is a helper class to RVView responsible for capturing events related to scene navigation and interpreting them. This makes modifying the navigation style merely a matter of replacing or extending this helper class.

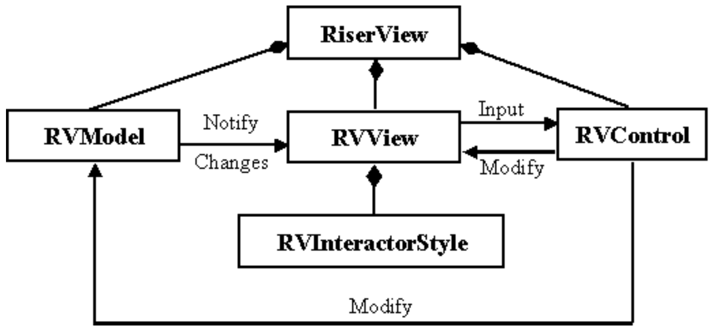

Figure 3. RiserView implementation of MVC

The RVModel class is the application's most complex class, since it encapsulates all the distinct scene elements. Figure 4 illustrates this class through another simplified class diagram.

The scene elements mentioned previously are encapsulated within the Riser, RV3DSModel, RVWaves, RVBottom and RVVortex classes. RVObject is a generalization of these classes containing the basic functionality of scene elements and simplifying the process of adding new classes of elements to the system.

The Riser class is actually a generalization of RiserFreq and RiserTime, containing basic riser functionality (chiefly its graphical representation as a composition of cylindrical segments). RiserFreq handles risers whose dynamics is described in the frequency domain, i.e. through the amplitudes and phases of a harmonic series. RiserTime handles risers described in the time domain. The object factory design pattern, as described by Gamma et al. (1997), is used to create objects of either class of riser through the RiserFactory class. The relationship shown on Figure 4 between these classes represents RiserFactory responsibility of creating either class of riser.

RVVortex models vortices in the scene. In this first version of RiserView, vortices are represented only within horizontal planes at distinct heights. A single object contains the representation for all the vortices in the scene. RVWaves and RVBottom represent, respectively, the water's surface and the ocean's floor. RV3DSModel is the class responsible for importing rigid body models of ships, platforms, buoys, well heads and other structures from 3D studio files. These structures are further described by a separate file determining their position and orientation in each time step.

While a scene may be composed by only a single object of each of the RVVortex, RVBottom and RVWaves classes, the description of each riser and each rigid body model is encapsulated in a distinct object of the appropriate class (and is also described in a separate file). A scene may be composed of any number of these objects. The RiserList and RV3DSModels classes are collections of risers and rigid body models represented in the scene. They extend the RVObjectList class which encapsulates basic collection functionality in RiserView's context.

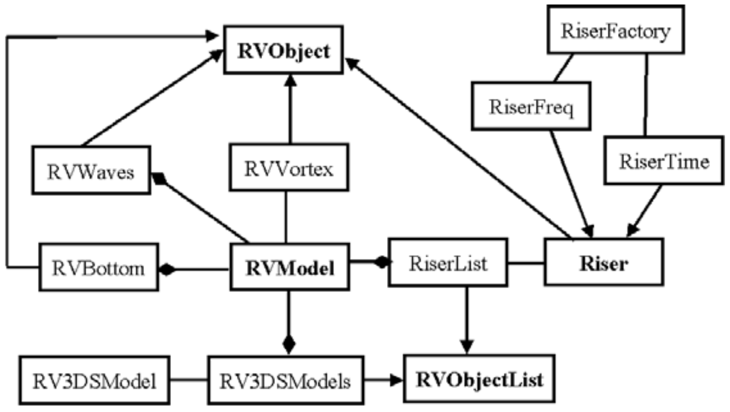

Figure 4 - The RVModel simplified class diagram

Aside from the MVC and Object Factory patterns implemented directly by RiserView, many other design patterns such as Observer and Strategy are used indirectly, implemented in VTK.

2.3. Collision Detection

As explained before, the only collisions of interest in this particular environment are the ones happening between risers. In RiserView, risers are discretized as collections of cylindrical segments, so it is the collisions between these segments that will actually be detected.

The collision between risers in RiserView have a defining characteristic which is rather unusual for virtual environments: the position of individual risers in every moment of time is known a priori, since riser dynamics is pre-calculated and is actually one of RiserView's inputs. This does not mean that the interaction of all risers with each other is known ahead of time, since each riser's movement may be repeated after distinct time periods. It makes it possible, however, to find enveloping curves for the movement of each riser and thus find the segments that may never collide with each other.

RiserView makes use of this feature to pre-process each riser as it is added to the scene, checking which of the new riser's segments have a possibility of colliding with segments belonging to any of the other risers already in the scene. A collection of pairs of segments with a possible collision is built, and in run-time only the collision between these pairs of segments is tested. This artifice greatly speeds up and reduces the complexity of the collision detection algorithm. Even in the worst case scenario, where the enveloping curves for the movements of distinct risers completely overlap, this still considerably reduces the number of collision checks when compared with the brute force method, i.e. checking each segment against every other segment.

Another simplifying factor of collision detection between risers is that the riser's exact geometric shape (save for the discretization in segments) is known. Each segment is a cylinder, so instead of detecting collisions between each polygon that comprises the riser, it is only necessary to detect the collision between each cylindrical segment.

But actually, according to Eberly (2000b), detection of intersection between bounded cylinders is a rather expensive process, ill suited for real time collision detection. Instead, RiserView works with capsules for collision detection. A capsule is the set of points at a fixed distance from a line segment. It can be visualized as a cylinder capped by a couple hemispheres, instead of a couple of circles. Since in this application the capsule's "caps" will only intersect with segments from the same riser (except in very atypical riser configurations) and collision is only tested between segments belonging to different risers, capsules may be safely used.

Eberly (2000a) affirms that collision detection between capsules is reduced to merely checking if the distance between the line segments originating the capsules is smaller than the sum of their radii. If squared distances (and squared radii) are used instead, calculating this distance means minimizing the squared distance function between two points, each belonging to a distinct segment. Eberly (2000a) also shows that if the minimum distance between the lines would occur between points outside the segments, the distance between the segments will necessarily be the distance between one of the segment extremities and the other segment. Based on these facts RiserView implements its algorithm for exact collision detection between cylindrical riser segments, not dependant of the number of polygons in which each segment is discretized and less costly than detection of collision between individual polygons.

The algorithm makes use of the problem's intrinsic temporal and spatial coherence (the fact that the time steps are always much smaller than the riser's periods of vibration and thus the position of the riser segments doesn't change abruptly from step to step) and does not check against the possibility of segments moving completely through each other between time steps.

Collision detection and its related data structures are also encapsulated by RVModel but weren't shown in Figure 4 to avoid making the diagram overly complex and reducing its readability.

2.4. User Interface

RiserView's interface is quite simple. Most of the application's area is occupied by the rendered animation. All commands (except for the ones related with scene navigation) are accessible through a menu bar, while the commands most often used during the exploration of the scene may all accessed by keyboard shortcuts, mouse clicks or dragging or through a toolbar. There is no status bar, but the toolbar also shows the current simulation time. All other important information (for instance collisions or legend boxes for color mapped variables) are shown within the scene.

Initially, scene navigation was done in a way akin to viewing the scene as an object to be manipulated, as if the scene was contained within a snow globe that could be turned this way and that to view different parts of it. It was observed that this went against the intuition of many users, who expected to navigate the scene as if the user was moving inside the environment, much like in a first-person game or traditional virtual environments. The navigation style was then changed to behave in that manner and increase the interface's usability.

The user faces only one constraint when navigating the scene. To make it harder for her to "get lost" while navigating, the user is always in either a vertical or a horizontal plane, not allowed to "look up" or down. There are no boundaries to stop the user from leaving the scene implemented in this version, but there is a command to bring her back to a default position that allows the visualization of all scene elements.

2.5. APIs

Aside from the standard C/C++ libraries, RiserView makes extensive use of two APIs: VTK and IUP/LED. The Visualization Toolkit (VTK) encapsulates the use of OpenGL to render the scenes using the object-oriented paradigm and implements several scientific visualization techniques and algorithms. The Portable User Interface Toolkit (IUP/LED), according to Levy (1996), is a user interface system composed of a virtual toolkit (IUP) and a window specification structured language (LED).

Before the beginning of the software construction phase (see below), several alternatives were studied to fulfill these roles. Although a description of this selection process and the alternatives considered is not in the scope of this paper, the main reasons why VTK and IUP/LED were chosen are mentioned. Both are freely distributable as open source and are compatible with the GNU General Public License (GPL). Both are fully compatible with both MS-Windows and Linux/X-Windows platforms, greatly reducing the work involved in making RiserView portable. Both have excellent documentation, complete with code samples and are maintained and updated regularly and have been for many years (actually, VTK is even distributed, optionally, as a "nightly release"). Aside from these characteristics, each API has several interesting features.

VTK implements a decentralized structure, the visualization pipeline, to process the data transformations intrinsic to scientific visualization. In this pipeline each node is responsible for determining when it must be re-calculated. According to Law et al. (2001), this makes VTK much easier to parallelize.

IUP/LED allows the specification of windows through an abstract layout (using a boxes-and-glue paradigm) that not only makes interface building much simpler but also maintains the layout while the windows change in size. This toolkit also allows the use of a native look-and-feel (instead of a fixed one) in both MS-Windows and Linux.

3. Unified Process

RiserView's development was guided by the Unified Process (UP), as described by Jacobson et al. (1998). UP, however, is actually a framework that must be adapted for each project. According to Smith (2002), both the Inception and Elaboration phases may be greatly reduced for small scale and low risk projects. In the development of RiserView these phases were nearly eliminated, resulting merely in a rough sketch of the architecture, pre-selection of the APIs to be used, the break-out of the project in iteractions and a time table for the main project activities. The Transition phase was not within the scope of this work either. That being the case, the project was broken in the following three construction iteractions:

i. The first priority was to render each class of scene element, based on files describing their dynamics. In this first iteraction they were rendered separately. Scene navigation, lighting, textures and anti-aliasing were also implemented in this iteraction, which dealt almost exclusively with VTK.

ii. In this second iteraction the user interface was built with IUP/LED, implementing the MVC pattern as described above. The distinct scene elements were integrated within RVModel. Most use cases were implemented in this iteraction, where IUP and VTK were integrated.

iii. Finally, collision detection was implemented.

The artifacts generated in each of these iteractions where maintained together, incrementally. Therefore, there isn't a set of artifacts for each one, but instead a single set of artifacts with the end result of the entire project.

One final simplification regards the Analysis stage. According to Jacobson et al. (1998), the Analysis model doesn't always need to be maintained or it may actually be completely eliminated, or rather absorbed by the Requirements and the Design stages (especially Design). With that in mind, there is no distinct Analysis stage in the development iteractions of RiserView.

4. Results and Conclusions

One of the first consequences, for testing, of the use of UP in the project was that the use cases already represented an excellent guide for black-box testing of the system. During the project itself, the use of UP was greatly simplified by the customizations made but still resulted in a sleeker architecture. To be truthful, part of the first iteraction was made before UP was chosen and applied and once it was, the existing class structured was greatly improved upon by the use of design patterns and generalizations that were made visible when the process was used. Another advantage of this formal approach to the development was a well-rounded documentation.

As mentioned before, since VTK uses OpenGL, RiserView's performance is considerably improved by running on systems with graphics acceleration hardware, since most of this hardware nowadays is compatible with OpenGL. This could be clearly noticed during the stress testing of the application. What consisted a large number of polygons for testing was largely a function of the graphics card used. The use of hardware-based anti-aliasing was disappointing however, and RiserView maintains an option for handling anti-aliasing through software that despite a considerable drop in performance when activated does show noticeable results.

Using VTK and IUP/LED made the task of building a portable application much simpler. It was actually harder to install Linux and then the necessary libraries than to work on RiserView's portability. A single line of code had to be provisioned for, the line responsible for the interface between these two APIs. One drawback, however, was that a satisfactory manner of rendering the scene in full-window mode using IUP and VTK together wasn't determined, what makes the hardware needed for stereoscopy more expensive.

When compared to brute force collision detection, RiserView's algorithm was always superior even in the worst case scenario. The cost for this superiority, the pre-processing overhead, was hardly noticeable during testing even for large numbers of risers. The detection of collision between capsules was much faster than between polygons, using and algorithm only slightly harder to implement. The possibility of further optimizations of the collision detection algorithm was not checked, however.

One interesting observation made during development and testing was how much trading the "snow globe" navigation style for the more traditional first person style used in virtual environments improved this aspect of usability. Using stereoscopy also makes for a much better exploration of the visualization.

Several future works may derive from this first version of RiserView to improve upon it. Below are listed several suggestions that occurred during the project but weren't implemented due to time constraints.

Interpolation of positions between time steps, to free the framerate from the constraint of the simulation steps, is one of the first improvements to come to mind.

A more difficult proposition is to study whether the priority given to rendering instead of to the handling of user generated events in the main loop is the best solution. It has been noted that it may result in jerky interaction. Perhaps a middle ground, along with adaptive framerate and level of detail would be a better solution.

Making use of VTK's suitability to parallelization, a parallel version of RiserView could maintain a similar architecture and handle much more complex scenes with higher quality.

And although when it conflicts with visualization it is set aside, realism is still a requirement for the application. Realistic scenes are often assimilated and interpreted much more easily. This aspect of the environment could still be considerably improved upon, since time constraints in this first version set a limit on what could be done. Even the possibility of making use of augmented reality techniques, for instance using image reconstruction for some of the rendered structures, does not seem far-fetched.

Finally, RiserView's simple interface in this first version could certainly use improvements. While many of these improvements are suggested for future implementations in the design document, no formal usability or interface studies and tests were made so far and these could possibly yeld further improvements.

References

Booch, G. et al., The Unified Modeling Language User Guide, Addison-Wesley, 1999.

Eberly, D. (2000a) "Distance Between Two Line Segments in 3D", http://www.magic-software.com.

Eberly, D. (2000b) "Intersection of Cylinders", http://www.magic-software.com.

Ferrari, J. A., Hydrodynamic Loading and Response of Offshore Risers, 1999. PhD Thesis - Imperial College of Science, Technology and Medicine, London.

Gamma, E., Design Patterns: elements of reusable object-oriented software, Addison-Wesley, 1995.

Jacobson, I. et al., The Unified Software Development Process. Addison-Wesley, 1999.

Law, C. et al. (2001) "An Application Architecture for Large Data Visualization: A Case Study", In: IEEE 2001 Symposium on Parallel And Large-Data Visualization and Graphics, IEEE, p. 85-92.

Levy, C. H. et al. (1996) "IUP/LED: A Portable User Interface Development Tool", Software: Practice and Experience, June, p. 737-762.

Petrobrás (2004), http://www.petrobras.com.br.

Smith, J. (2002) "A Comparison of RUP and XP, White Paper", Rational Software.

Schroeder, W. et al., The Visualization Toolkit, 2nd. Ed, Prentice Hall, 1998.

Upson, C. et al. (1987) "The Physical Simulation and Visual Representation of Natural Phenomena", Computer Graphics, v. 21, n. 4, p. 335-336.

Veit, M. & Herrmann, S. (2003) "Model-View-Controller and Object Teams: A Perfect Match of Paradigms", In: 2nd. International Conference On Aspect-Oriented Software Development, p. 140-149.