Autotag:

Applying Tags to Virtual Objects in Real-Time Visualization Systems

Renato D. Prado, Alberto Raposo, Luciano P. Soares
Tecgraf, DI (Departamento de Informática)
PUC-Rio (Pontifícia Universidade Católica do Rio de Janeiro)
Rio de Janeiro, Brazil
e-mail: {rdprado,abraposo,lpsoares}@tecgraf.puc-rio.br

***Abstract*—A main objective of large industrial-engineering departments is implementing integrated information systems to manage their projects' life cycles. Particularly, most industrial engineers in the oil and gas industry use 3D geometric models, which they create using CAD systems to interact with the information systems. This is possible because modern CAD systems have evolved from drawing programs to collaborative design tools that combine geometric modelers with specialized tools for tasks such as engineering-document management, physical-plant documentation, and computer-aided visualization. Important information that needs to be associated with CAD objects in these scenarios is the name of the objects, essential for tasks such as operation management, for instance. Despite its importance, there is not yet a well-established solution for the presentation of objects' names in CAD real-time visualization. This paper presents a solution to generate textures for CAD objects with their main identification, keeping interactive visualization rates in massive CAD models. The main challenges are discussed and solutions are proposed and tested.**

Keywords- autotag; CAD;texturing

I. INTRODUCTION

A major objective of the engineering departments of large companies, such as automotive and oil & gas, is the availability of engineering software systems easily adopted by their employees to manage their projects' life cycles. They seek computational systems that, in addition to simple access of databases with project information, provide resources for 3D visualization of their models with enough realism to be used in many interactive applications.

The engineering models are based on CAD (Computer-Aided Design) files that represent the objects of the building, vehicle, among others. When the CAD files are very complex, like an oil platform, these models are called massive models [1]. One of the challenges to build these systems in real-time 3D visualization is to develop applications using techniques that increase graphics quality and realism, while may be used with an acceptable performance in conventional computers, supplying all information needed by the engineers using the system.

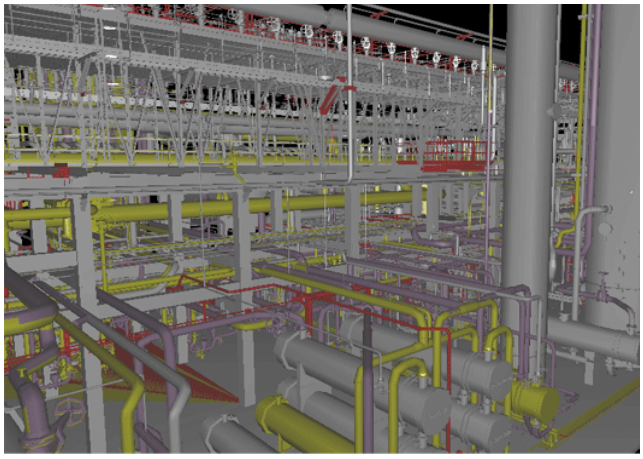

Figure 1. 3D model of oil refinery.

CAD models are usually not conceived to be visualized in real-time [2] In regions of high concentration of objects there is a high computational cost that can compromise the possibility to use in a virtual reality environment. Figure 1 shows a small portion of an oil refinery where it is possible to visualize the complexity of objects (there are thousands of cylinders, cubes, pyramids, and other primitives for the representation of real objects). Even to render a small part of a refinery, the computational cost is already high even for high-end machines.

Identification of objects is possibly achieved through an additional window that presents information of clicked objects, or even hovering the cursor, but this does not allow the user to visualize the identification of several objects at the same time. Other possibility was the use of billboards [3], but this solution can occludes areas of the virtual models that compromise the interaction, then this solution is only applied in annotations that need more space for a complete text as presented in the Figure 2 and not only few letters. Since there is a demand of the engineers responsible for projecting complex sites, on visualizing the name of the objects without need to interact with them since group of people can look at the scene at the same time, the autotag was proposed.

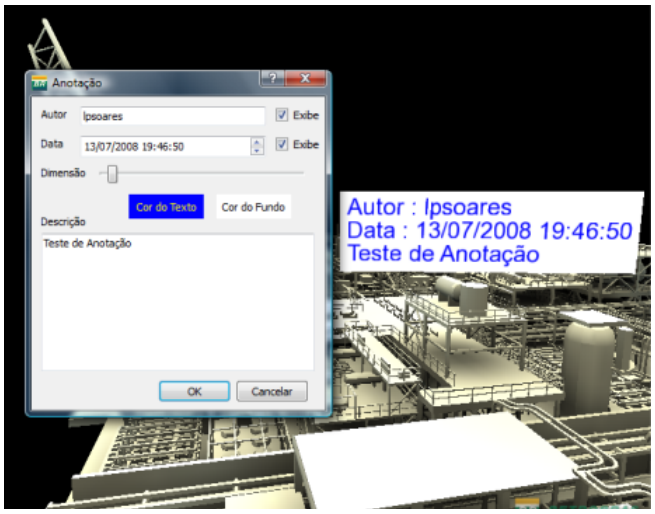

Figure 2. Anottation in CAD model.

Autotag is a technique that helps users of CAD software to display the additional information of objects in the scene with their references. This textual information is a texture applied in geometric shapes, which have the name or other relevant information written on it. Thus, their identification is facilitated and can be quickly noticed when a user look at the objects. Since the objects in the CAD models usually use colors only for identification, the tags are applied with maximum transparence in the region without the letters, not changing the original color of the object being analyzed.

Section 2 will present a related work that was used as reference and motivation. Section 3 presents the technologies involved and section 4 presents how the autotag was developed, conclusions are discussed in the section 5.

II. Related Work

AvevaReview [4] is a CAD software used to produce complex 3D models. This software has a support for presenting information on some objects in a semiautomatic way. Figure 3 illustrates the use of these identifiers.

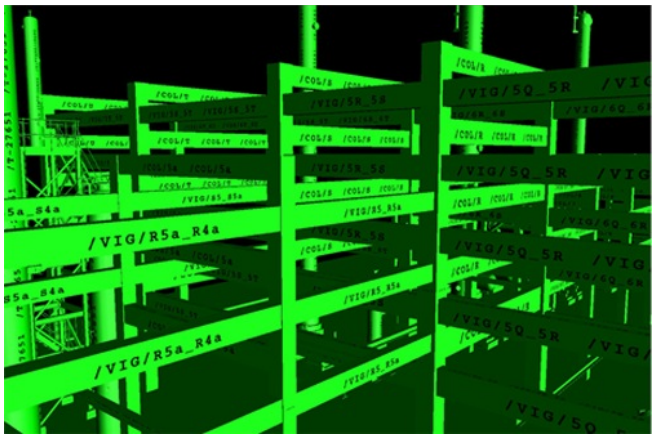

Figure 3. AvevaReview image of a model with autotags.

It is unclear how is the implementation of these tags in the AvevaReview, since this commercial software company does not provide information about implementations, but this solution was used as an idea to study ways to achieve the autotags and improve it. The main problem lies in how to display the tags in a large number of objects without extrapolating computer resources, especially the video card memory. The cost to display the tags compromises the performance and the proposed solution can present several tags, maintaining the interactive rate of the application.

III. Basic Concepts

In this section there will be an explanation of massive models, and also a discussion of some techniques for texturing.

A. Massive Models

Massive models in computer graphics, refers to high detailed geometric models [1]. Among their main characteristics, it is possible to highlight:

- They have a high level of detail that can not be seen by the human eye without magnification.
- They have data that consume hundreds of GB or TB for storage, billions of geometric primitives (cylinders, cubes, cones, etc.), and units of measurement ranging from angstroms to light years.
- They have data that exceeds the conventional capacity of processing and storage.

Massive models are getting bigger each day, models of 1 billion polygons are common, and these models have about 26GB of raw data, and user still want to render these models in real-time.

There are already several virtual reality software for 3D visualization and interaction with CAD models. Some examples are: WalkInside[5], Aveva Review [4], Navisworks [6], among others [2]

The AvevaReview can apply some tags with identifications over objects in CAD models, which greatly facilitates their identification, but this program only supports cubes (boxes) and cylinders.

TABLE I. Primitives in a Model.

pyramid	71462
box	32350
rectTorus	4696
circTorus	27388
slopedCone	69547
cylinder	118346
sphere	441
dish	10807
sphereSect	0
mesh	34839
Total	**369876**

The AVEVA "Review" files (RVM) were used as a reference for this research, mainly due to the fact that it is a format commonly used in the industry, and then larger files were available for evaluation. An example of an RVM model that is only part of an oil refinery, containing a total of 369,876 different objects, is presented in Table 1 showing the quantity of each primitive. The information of this model will be used during the text to show how complex is to apply the tags on it.

B. Textures

In computer graphics, texturing is a process that applies over a surface of an object an image, expression, or other data source. Figure 4 shows a brick wall example, this image can be applied several times on a surface creating the effect of larger wall geometry. Other solution could consist of a procedural texture, in this case a regular equations or functions can creates the colors as a real brick. It is a way to directly fill the colors of an object, programming in the video card.

Each texel of the texture is referenced by a pair of values (u, v) and u and v in general vary in the interval [0,1]. This coordinate system represents the texture space. Figure 4 shows two examples with their coordinate orientation.

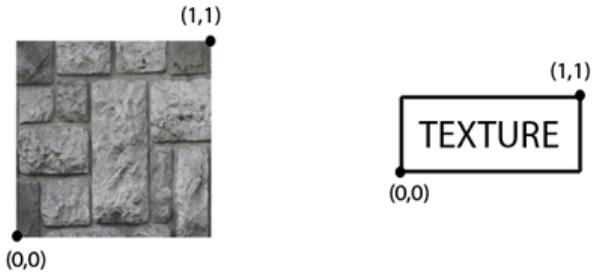

Figure 4. Image texture of bricks, texture image of letters.

It is necessary to define how to map the coordinates to apply a texture onto an object, indicating the correspondence between the points of the geometry and the texels of the texture image. Usually the vertices of the object are associated to the texture coordinates. One of the challenges of this project is how to correctly position the autotags on top of each geometry type. Depending on the way the texture is applied the text could appear up-side down or even mirrored, then the correct information of how the mapping is defined in the objects is fundamental.

The size of texture images used on the GPU (Graphics Processing Unit) is typically 2^m x 2^n texels. However, modern GPUs can handle textures of arbitrary size, which allows any image generated to be treated like texture [7]. The difference between the texture size and the surface where it is applied can be a problem. If the texture has a low resolution compared to the resolution of the geometry on the display, the texture will be magnified to cover the entire surface, so that the result might show an image without definition and aliasing.

As the texture size increases in order to achieve a better quality, more video memory will be spent to store that, and the memory usage grow quadratically. The texture coordinates for the mapping also uses video memory. For small and simple models it does not pose a significant problem, but for large models, the use of memory can be excessive.

The RVM model mentioned earlier (Table 1) has a total of 369,876 objects. Whereas an individual texture for geometry occupies 8KB of texture memory, the use of video memory only to store all the textures on the graphic card would be 369,876 x 8KB or approximately 3GB. Moreover, there is the cost of each object having to save the texture coordinates needed to map coordinates. The total number of triangles of this model is 11,288,337 and each vertex needs a pair of texture coordinates, totaling a use of hundreds of megabytes. The problem is that each object has a name and needs a specific texturization.

C. Procedural Texture

Another way of texturing objects, is the use of procedural texture with the implementation of algorithms in programmable shader [8]. This means that effects to simulate textures are created by the video card, which fills the colors of the pixels of the object to display a desired texture. Thus, the consumption of video memory is much lower, since there is no need to store previously created different textures on the graphic card memory and neither do the mapping coordinates. Each pixel is individually filled with the help of mathematical functions each time the shader program is executed. The processing cost to run the shaders is low since this is done with a high degree of parallelism in the graphics card.

IV. AUTOTAG

The project of autotag was developed on C++ with OpenGL and GLUT[9]. Using object orientation, the resources implemented during this research are going to be organized in the form of a library to be reused in several different projects.

Two main possibilities were considered to create the autotags: 2D textures and procedural textures. The first idea (two-dimensional textures applied to objects) was chosen as the initial test to see if indeed the bottleneck of the program would be the consumption of video memory or the processing resources that are not enough.

The option of developing a procedural texture was not implemented in the moment due to the fact that it is necessary a routine to draw each letter, and the implementation of such routine is very complex and the main idea right now is to discover the limitations of the GPU.

The first step to create the autotags, is the creation of a repository of characters where it is possible to store the letters, and after this letter will be assemble in an organized way in order to form a word of phrase to a specific textures when needed.

In order to evaluate the effect of the autotags in different situations, it was also created a simple scene graph algorithm to simulate real effects of a 3D visualization software in real-time and also allowing the creation of complex scenes for testing. The cubes and cylinders geometry produced for the

tests were done so that they were simple and did not have a number of vertices larger than necessary.

The cylinders constructor method has parameter values given by radius, height and how many times the texture should be repeated on it. This creates three arrays containing the position of its vertices, normals and texture coordinates of each vertex. These three arrays are passed to OpenGL to create a display list. The display list, use the function glDrawArrays receiving GL_TRIANGLE_STRIP. The caps of the cylinder, or discs, are designed using the parameter GL_TRIANGLE_FAN in the glDrawArrays.

In the case of the cube, its constructor method takes the dimensions of height, width and depth and how many times the texture should be repeated in each of their faces. Then the arrays are configured to return the vertices, normals and texture coordinates for each face. Display lists were used to improve application performance. The display lists are created by side and use the function glDrawArrays using GL_QUADS, to render the faces of the cube.

Colors are also applied in the geometries in order to evaluate the effect of the textures over the surfaces. It is desired that the text appears over the object not affecting the understanding of the shape of the geometry.

A. Dinamic Texture

In order to build a different texture for each object, methods were developed organizing words vertically or horizontally. Depending on the geometry of the target object, it might be better to place the texture in the horizontal than in the vertical. These functions extract characters from the repository and rearrange them to form the desired new image, which can be used as texture. The repository is a PNG image that contains all ASCII characters, plus the LATIN-1 characters. In the future, new character, for different languages for instance, can be easily incorporated. Figure 5 presents the texture used.

Figure 5. Characters Table.

Specific methods take care of reading the image and store it in a texture that can be accessed globally. Each pixel is represented by the channels R, G, B and A (red, green, blue and alpha). Therefore, if the repository has a resolution of 256 x 256, the vector needs 256 x 256 x 4 bytes to store the texture. Since this is the original data, it is desired that it has a resolution good enough to be easily read by the users. Mipmap techniques although can be used in the textures used in the object to improve the performance of the application.

Once the repository of characters is in memory, the following methods may be called:

chImage* createStringImageV(const std::string & str)

chImage* createStringImageH(const std::string & str)

Both receive a string that is used to create the texture and returns a pointer to a struct chImage. This struct contains the width, height, and an texture array, which stores the new image created. The image stored in the vector has the same format in which the repository is stored. The functions load the file and organize the characters one at a time in a new vector, depending on the character optimizations were done in order to reduce the space between letters. For instance an "i" needs less horizontal space than a letter "m". Usually these textures are written vertically for the cylinders and horizontally for the cubes. Figure 6 illustrates this. The algorithm was also optimized to make letters clipped to decrease the space between the letters, thus optimizing memory.

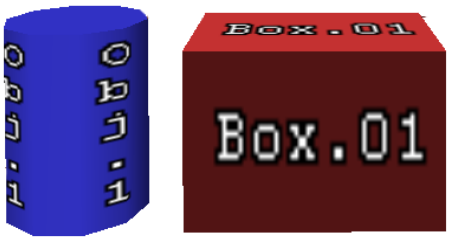

Figure 6. Tags in Cube and Cylinder.

All strings created are stored in a chImage struct in the CPU memory. So when it is decided they are no longer needed, the method freeStringImage(chImage* stringImage) should be called to release the vector from memory. An example of this case is when the texture has been submitted to the video card and is no longer needed in RAM.

The images created use the OpenGL function: glTexEnvi(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_DECAL).

This function defines the color of the primitive as the texture. As the illumination values are computed before the texture mapping, the texture color replaces the color calculated by lighting.

B. Tagging and scene construction

The user has control of the camera position of the evaluation software using the mouse. It is possible to perform translations and rotations in relation to the central point of the scene. This was very used to analyze the result and make sure the tags are not compromising the quality of the objects.

Two text files (Cylinder.txt and Boxes.txt) were created containing names of each cylinders and cubes. At the beginning of the program, these files are loaded and the names are stored within each object. Random numbers algorithms were developed to produce the primitives sparsely on the 3D world (more specifically cylinders and cubes). The geometries have different translations and rotations applied to them, creating a homogeneous scene for the tests.

The scene was set in order to see how many objects could be rendered without textures while maintaining an acceptable frame rate. In order to compare the results, all optimizations were turned off. With 60,000 objects, half of each type(cylinders and boxes), the frame rate was around four frames per second. Since the main idea of this project is the use in real-time, a frame rate of 30 frames per second was the target. For that the maximum number of objects was 5,000 cubes and 5,000 cylinders. Figure 7 shows these scenes, the cylinders are blue, and the cubes are red.

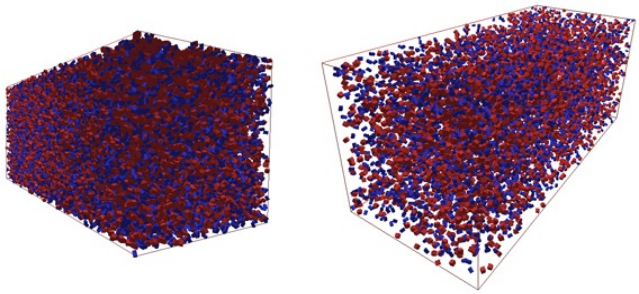

Figure 7. On the left a cube containing sixty thousand objects and the right side of the same cube, containing ten thousand objects.

The next step was to apply textures to objects and measure the consumption of video memory to see how many objects at a time could have the tags. The program GPU-Z [10] was used to analyze the results. On a machine with Windows 7 whose graphics card is a GeForce 9600GT 512MB, the resource of video memory used was 110MB without any geometry. The first test was done with 40,000 objects very close to each other and without textures, the memory used was 148Mb, which means that the memory spent on the geometries, considering vertices and normals is only 38MB. The frame rate was ~8 fps. After applying the textures in the 40,000 objects at the same time, the use of memory was 508Mb and the frame rate went to zero fps.

If the textures have a resolution of 128 x 16, or 2048 texels, each texel represented by RGBA (four unsigned chars), the use of video memory for texture is 2048 * 4 bytes or about 8Kb. With 40,000 objects with texture, the theoretical use of memory would be 320Mb. Knowing that only the geometry uses 148Mb, 148 + 320 is equal to 468Mb, which is close to 508Mb that has been achieved in practice.

A second test was executed with 10,000 objects less to check the difference in fps. With 30,000 objects and texture off the frame rate was ~10 fps and memory use was 122Mb. With the application of textures on all objects the frame rate dropped to 2 fps and memory use was 500Mb.

With the previous result, we decided to examine the scene graph to see what was making the frame rate drop. It was seen that the function glBindTexture could be responsible for the drop in performance. Thus a test was done with 40,000 objects with textures but not making the "binds" before drawing the objects. The frame rate remained at ~8 fps, which was the same rate obtained when we did not apply the textures. This shows that glBindTexture was responsible for the fps drop, and not the high memory consumption. GlBindTexture function is not simply a void pointer, but it invalidates the cache and in the worst case, change all the state associated with that texture mode (clamp, mipmap generation, filters, etc.). The behavior of the function in reality will strongly depend on each hardware optimizations, but either way it is not efficient to call it 40,000 times per frame, as the loss of performance showed before. The number of "binds" that could be done by frame in very large models has become a major limitation.

An optimization for the autotag was then developed in order to reduce the number of binds. The solution chosen was to apply the tags only to objects closer to the screen. Indeed, if the objects are not too close to the screen the user can not read what is written in the tags. Moreover, when the user of the program is examining objects, he does not need to see the names of all objects in the scene at the same time, because the resolution of the display will limit that at certain moment.

To implement the culling due the distance from the viewer, adjustments were made to the scene graph algorithm. Before rendering a frame, it is calculated the camera position, the position of each object, and the distance between them. Now when the camera changes its position a method is called to evaluate whether the bind should be done or not. This method takes the current OpenGL modelview matrix and passes to each transform matrix that, in turn, calculates the distance of the entity from the camera. The distance is transmitted to the renderer, before calling the method to draw the cylinder or cube, checking whether the object is up to a maximum distance from the camera or not. If so the tag is applied, if beyond that distance it does not get the tag. The limit distance was calculated based on the resolution of the display and the visual acuity of a regular user. Figure 8 shows some results achieved in a complex scene.

As an additional resource to keep the frame rate high, there are also a limited number of objects that can receive textures at the same time. It is not expected that the user will visualize more than a certain number of objects without getting closer to the group of objects, the limit was configured to 500 for the tests, but in a regular use of the software this number can be smaller.

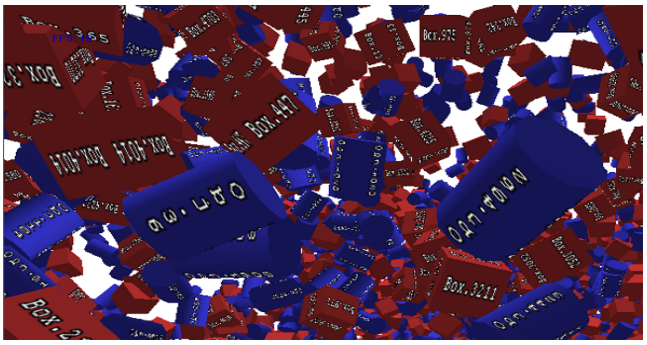

Figure 8. Closer objects with tags and farther without them.

The application performance has improved considerably, although there is still a drop in fps depending on the configuration parameters (maximum distance from the object to the camera and the maximum number of objects that may have tags). With a scene quite crowded with 10,000 objects that ran at 30 fps without the tags, there was a fall of up to 10 fps when it was used only as camera distance restriction. Also limiting the number of objects tagged in 500, the drop was only 3 fps, resulting in 27 fps for this simulation.

Since the problem of 'bind' was reduced, the video memory becomes a point of focus again. Even though only some objects receive tags, it is still mandatory to keep on board a different texture per each object. Optimizations were developed trying to compress the images on the graphic card, but this kind of operation did not really work as expected, the better way to use less memory is the use of programmable shaders that is the focus of the future work.

V. CONCLUSIONS

The display of the tags on the 3D objects is a resource desired by the people who work with CAD models, since the efficiency to identify objects that are being examined is very high. The user can already see the name of the structure in the object tree, but with tags applied directly in the geometric objects, the identification is much faster and precise. Of course this technique can visual overload the scene, but tuning the algorithm can produce adequate results. During the integration of the algorithm in the main software developed by the group of developers an additional option was implemented allowing the user to select which information is to be displayed, like the name of the object, the size, weight, manufacturer, among others.

The solution presented here was implemented and tested with a massive model, but for the moment, only certain shapes that can receive tags. In the future the algorithm will support more irregular shapes, such as cones and torus.

Application performance is not completely compromised with the tags. In regular models the frame rate is higher than 30fps. However, some configuration settings in the algorithms still need to be made so that the solution can be more adapted for different situations and styles of working. The textures can be created as objects need and a smart algorithm can manage when they are removed or stored on the memory of the graphic card, known as streaming texture data from disc.

The current solution limits the number of objects with tags in order to achieve better performance of the application. The choice of which objects receive tags each frame depends only on their distance from the camera and the limit of objects that can receive these tags. In order to set these controls of the amount of tags to exhibit, it is important to allow the users of the program to configure and evaluate and see what they actually prefer. This idea is very similar of maps software that as you get closer to the streets you can see that name of them.

As a future work the orientation of the text should be addressed. Currently, the texture is placed under the guidance of the object position. As in the case of randomly generated scene, objects can be upside down in relation to the camera inverting the text. It would be interesting to align the text according to the camera UP vector.

Another optimization that is important regards the size of each geometry. The size of each object could be taken into account, so objects away from the camera, but that occupies a larger amount of pixels on the screen should also receive tags. The same would apply in reverse: very small objects near the camera would not receive tags.

Procedural texture with programmable shaders is another area being developed right now. One single shader could be used for all the textures, but the current problem is to find a way to store each letter in the graphic card in the form that it can be reused. A problem here would be how to map the characters to objects of different types and sizes and not compromise the quality of the text being displayed.

ACKNOWLEDGMENT

This work has been partially supported by Petrobras. Alberto Raposo receives a grant from FAPERJ E-26/102273/2009.

REFERENCES

[1] YOON, S., GOBBETTI, E., KASIK, D., MANOCHA, D. Real-Time Massive Model Rendering. Synthesis Lectures on Computer Graphics and Animation, Morgan & Claypool Pubs., 2008.

[2] RAPOSO, A., SANTOS, I., SOARES, L., WAGNER, G., CORSEUIL, E. and GATTASS, M. 2009 . Environ: Integrating VR and CAD in Engineering Projects. In IEEE Computer Graphics and Applications

[3] SOARES, L., CARVALHO, F., RAPOSO, A., SANTOS, I. 2009. Managing Information of CAD Projects in Virtual Environments . In Proceedings of the XI Symposium on Virtual and Augmented Reality

[4] AVEVA REVIEW. Aveva Home Page. http://www.aveva.com/ Accessed in: oct 2011.

[5] WALKINSIDE. VRcontext Home Page. http://www.vrcontext.com/ Accessed in: oct 2011.

[6] NAVISWORKS, http://usa.autodesk.com/navisworks/

[7] EBERT, D. S., MUSGRAVE, F. K., PEACHEY, D., PERLIN, K., WORLEY, S. Texture & Modeling: A Procedural Approach. 3rd Ed., Morgan Kaufmann, 2003.

[8] AKENINE-MÖLLER, T., HAINES, E., HOFFMAN, N. Real-Time Rendering. 2008. 1027p.

[9] THE OPENGL UTILITY TOOLKIT. GLUT 3.7 Home Page. www.opengl.org/resources/libraries/glut Accessed in: oct 2011.

[10] TECHPOWERUP GPU-Z. GPU-Z Home Page. www.techpowerup.com/gp Accessed in: oct 2011.