Virtainer: Graphical Simulation of a Container Storage Yard with Dynamic Portal Rendering

Miguel Escrivá, Marcos Martí, José Manuel Sánchez,Emilio Camahort, Javier Lluch, Roberto Vivó
Polytechnical University of Valencia
Computer Graphics Group
Camino de Vera s/n Valencia, Spain
{mescriva|mmarti|josanchez|camahort|jlluch|rvivo}@dsic.upv.es

Abstract

The popularity of 3d graphics has grown exponentially in the latest years, which has lead to an increase in the range of applications where this kind of representations are used. The monitoring of industrial processes is an area where this type of simulation is rarely used, being much more common to show the information in traditional 2d interfaces.

An application for data visualization in maritime container terminals is introduced here. We have developed a modular system adaptable to any stacked objects problem. This paper describes the architecture of our system, its features, and the graphics techniques applied to achieve a high frame rate and keep it independent of the data size.

1. Introduction

Stacked objects are very common in the industry, especially in logistic enterprises. Maritime container terminals are a particular case of these. The management of this process information is usually done by systems with poor alphabetical interfaces which lack the ability to adapt to changes and represent the information in an excessively abstract way. In contrast, advanced graphical simulations model the real process with high fidelity and show changes in a much more convenient manner.

This kind of applications require the access to the system information, usually stored in a database, and the ability to receive the changes occurred in that information through a communications layer. We will show the problems found in the non-graphical parts of these systems, and the approaches we have taken to solve them.

Large data-sets, characteristic of these industries, pose a challenge to the development of graphical simulation systems. The techniques we describe here solve two main problems: real-time visualization of very large data-sets and the modification of this data with an event oriented model. Although the processing power of graphics cards increases at a very fast pace, it is not enough to deal with the expanding complexity of new geometric models. In order to solve this problem, visibility culling techniques [2], such as geometric simplification, occlusion [5] or impostors, have been developed to reduce the amount of data employed to render the scene while maintaining the image quality.

Another necessary part in any Virtual Reality application is the tracking of input devices. Without a good interaction system, the most advanced graphical simulation is useless. We have implemented a tracking application, based on *OpenTracker* [18], which can be configured using standard XML files.

Existing commercial applications, such as those described in [8], offer text or 2d graphics interfaces, and are unable to respond to real time events. Our system offers a richer interface in a virtual representation of the maritime terminal. It is a multi-platform system with integrated data services, XML configuration files, standard database access, and a geometric layer based on an open source scene graph engine. It integrates several graphic accelerations which exploit the spatial coherence of stacked objects to speed up the rendering system. Virtual reality is completely supported, as our system works with any configuration of computers, screens and devices. Frame synchronization is done at swap buffers level, allowing the simulation to run synchronized in any number of computers.

The structure of this paper is described now. First, we begin explaining the antecedents and main goals of the project. Then we give some details about the graphic acceleration techniques we have used. A description of user interaction follows. The next section details *Virtainer's* architecture, its building blocks and how they work. Finally we present some conclusions and the main directions for future work.

2 Antecedents

This project was conceived as an extension of a previous 2d monitoring application, *Visual Gama* [12], adopted by the company Marítima Valenciana, one of the most important container terminals of the Mediterranean sea. The 2d interface of Visual Gama was not rich enough. So we add real-time 3d graphics to create a more realistic representation of the terminal with smooth animations to offer higher fidelity and more up-to-date information.

Virtainer is built on top of completely standard open source projects which deliver compatibility and great performance. *OpenSceneGraph* [1] is employed as our graphics engine, providing Virtainer with object management, standard device handling and other utilities. *OpenProducer* gives us a rendering context and camera control, and *OpenThreads* [10] provides thread management. The communication layer can employ any protocol, but we are currently using *SCore* [17, 16], because it is the system adopted in the container terminal to send and receive event messages. Finally, *OpenTracker* encapsulates the device events in messages which are received and parsed by *TrackerManipulator*. In *Gama* we have data access and communication layers, so *Virtainer* is on top of *Gama* and fully integrated with it.

The objectives of our system are real-time navigation with sustained frame-rate, realism and virtual reality support. An event oriented system is required to maintain the real state of the yard, and multi-view and multimode inspection (fly, walkthrough, cab views, etc) are also desired. The system should offer interaction options for selection and camera control.

3 Architecture

Usually container terminals store their information system in a centralized server, using different technologies like ODBC, OLE/DB or web services based on SOAP or XML-RPC. This database stores at any moment the state of the terminal, so the only information needed by our system resides here. Container, truck and crane information is stored in a certain relational scheme that is accessed to represent these elements graphically. It is generally possible to associate a trigger to each interesting event of the database. When any change in the information system occurs, this trigger can broadcast the change to any application or software agent.

Virtainer is a system based on standard C++ libraries and constructed over open source standard projects. This architecture simplifies the system, and allows the logical data to be independent from implementation details. *Virtainer* consists of a set of seven modules, implemented as dynamic libraries, each one handling a different aspect of the simulation. The modules are:

vrtUtil provides logging facilities and a connection to the Xerces-XML library used to parse configuration files.

vrtDB encapsulates the code required to connect to a database and load information from its relational schemes.

All the logical information of the terminal is stored in *vrtData*. Up-to-date logical positions, destination, vendor and other container data are maintained here. The same applies to trucks, cranes and other machinery.

vrtGeo manages the geometric aspect of the simulation. It loads the machine models, handles its animations, and is responsible of the creation, destruction and manipulation of the containers' geometry.

The communication layer resides in *vrtComm*, which is the module employed to receive terminal events and to interchange synchronization messages between the elements of a cluster in a mutiple machine configuration.

Keyboard, mouse events and related aspects are managed in *vrtGUIHandler*, including key bindings and object selection.

virtainer serves as an interface to the library set, providing methods to initiate and finish the execution of the library modules, as well as a Viewer class to control the visualization.

When the application begins, the data is loaded from the server database using the *vrtDB* module. This information flows to *vrtData*, and then to *vrtGeo*, where it is used to create the graphic simulation. The changes in this information are received via *vrtComm*, are passed to *vrtData*, and then to *vrtGeo*, where the simulation is updated according to the event occurred. Figure 1 shows this architecture.

Each module has one or more associated configuration files in standard XML format where the relevant parameters for that module can be specified. There is a large number of configurable elements, which are detailed now. There is one configuration file for database related parameters, such as the method to access the database, the protocols employed, user name and password. Other files define the communication mechanism employed: *Score*, UDP broadcast, *Bluetooth*. The parameters for *vrtGeo* specify the static topology of the terminal: elements, buildings, factories, wharf, sea, etc. Other simulation related parameters exist, such as types of container blocks, levels of piles and types of containers. It is also possible to change the geometric models associated to the different objects, including cranes, trucks and ships. Other constants allow the modification of the maximum number of cranes, the average speed of the cranes, etc. Finally, color coding policies, textures and other material information are also configurable.

4 Graphic Acceleration Methods

The large data-sets that are managed in container terminals require the use of graphic acceleration methods to

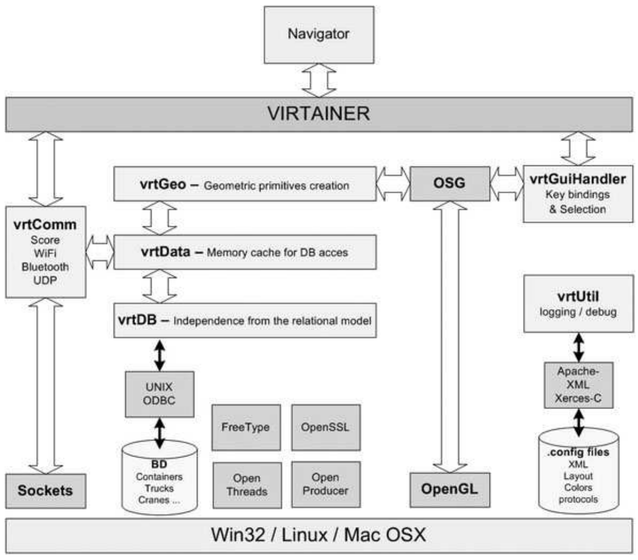


Figure 1. Virtainer modular structure

produce a real-time simulation. We describe now the techniques used in *Virtainer* to achieve high sustained frame-rates.

Frustum culling, explained in [9], is the most common method of culling used in computer graphics. Objects outside the camera frustum are removed from the rendering pipeline. The scene graph structure makes this technique very efficient, as complete object groups can be removed with a simple comparison between the frustum and the bounding volume of the group. *OpenSceneGraph* employs two object lists populated by the culling process. The first list stores the opaque objects ordered by state (material, texture, vertex programs, etc) and the second list stores the transparent objects, ordered by depth. Different bounding volumes are used to improve performance: boxes for geometry and spheres for object groups.

Multi-resolution objects are the next technique implemented. Machines in *Virtainer* have static pre-computed levels of detail, to reduce the number of polygons rendered without reducing noticeably the quality of the resulting images.

Impostors, as explained in [19], consist of rendering to a texture the objects which are far from the observer, and substituting the real objects with billboards containing that texture. As long as the camera does not move we can achieve very high frame-rates. This technique is used for the lowest level of detail of the machines.

We have also developed a specific method to create impostors for the container blocks, which uses the faces of the bounding box of each individual block as pre-rendered impostors. This method is used when the container block is far

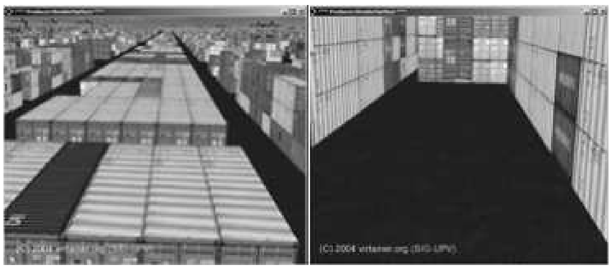


Figure 2. Non-visible geometry is never rendered

from the observer, and the geometry of a whole container block is reduced to only five quads.

Non-visible geometric culling is another method applied. Container faces which are not visible from any view are removed from the scene graph before visualization. This technique, shown in Figure 2 consists of rendering only the surface of each container block, which leads to a noteworthy reduction of the polygon number and an increase in the frame-rate.

Using *Back-face culling* [13, 11] we can avoid rendering the back parts of the containers. It is easy to see that of the six faces of a container we can only see three at the same time, so with this technique we can cut in half the number of polygons to be rendered.

Vertex array lists are another acceleration technique used in *Virtainer*. It consists of grouping the geometry of the container blocks in a certain way that is better handled by the graphics card. When a container has to be added or removed from a block, the affected block must be completely rebuilt. Due to the low speed at which cranes operate in the terminal, this is not a problem: the reconstruction of the block is not noticeable.

4.1. Dynamic Portal Rendering

We have implemented *Portal* support as another acceleration technique. A terminal of containers is very similar to a city. It is formed by streets and blocks of houses, but instead of buildings we have stacks of containers. Our system can use portal rendering [6, 7] to accelerate the visualization in walkthrough mode. When the application starts, the information about containers is loaded from the database, and it is used to create cells and portals automatically, following the *Breaking the walls* [14] algorithm.

Note, however, that containers are added and removed from the terminal regularly, making certain parts of the scene visible or occluding them. This means that whenever a change occurs in some of block of containers, the visibility of that block has to be recalculated to reflect those changes. Our method, based on [15], introduces portal reconstruction to adapt to the dynamic geometry of the container terminal.

5 User Interaction

OpenSceneGraph offers tools to manage standard devices (mouse and keyboard), but they have to be connected to the host where the final application runs. We have developed a utility to manage input devices which offers support for any controller supported by the operating system. Our tracking program sends incremental changes in position and orientation, and other interaction related data.

It is based on *OpenTracker*, benefiting of its advantages: configuration based on XML files, device abstraction, transparent network access, data flow of tracking data and an extensible C++ class library. Another feature is that it is independent of *Virtainer* libraries. Our tracking system can be launched in a different host that the main application, allowing the device management to run in a dedicated host. We use XML configuration files to map tracking data to movements and to select the interaction type.

Back-face culling delivers an increment of around 30% of our frame-rate by reducing the number of polygons sent to the graphics card. This differs from enabling backface culling in the graphics card, which brings no performance improvement, as the polygons have already been processed.

6 Results

We present now the results obtained when applying the acceleration methods described. Figure 3 shows the enor-

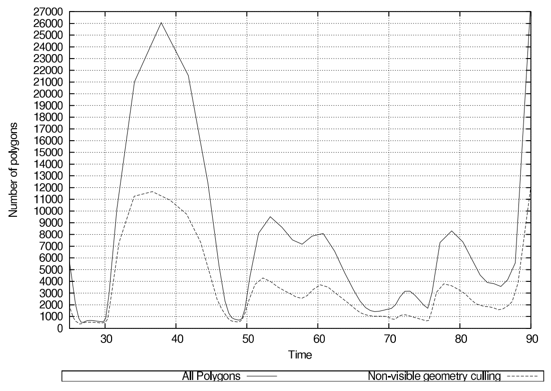


Figure 3. Number of polygons rendered with and without Non-visible Geometry Culling.

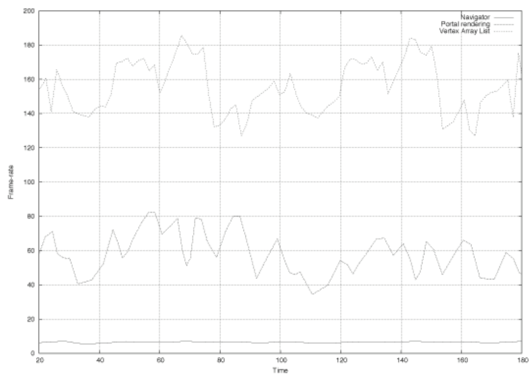


Figure 4. Frame-rate results using the fixed pipeline of the graphics card. In this case portal rendering is slower than just drawing every outside polygon of each container block.

mous reduction in the number of rendered polygons when removing non-visible container faces. Figure 4 shows the improvement in the frame rate achieved with array lists and our dynamic portal rendering method. Array lists deliver a great increase of the frame-rate, as modern graphic cards manage a very large number of polygons.

Our last technique, *Dynamic portal rendering*, delivers lower results in that figure, but if we use shaders to improve the graphics quality the situation is reversed. This is illustrated in figure 5.

Multiple machines using *Virtainer* can be interconnected to increase the computer power when rendering to more than one display device. We have chosen a master-slave architecture where the master computer controls the camera and sends the model view matrix to all the slaves. Then the camera can be modified to adapt the view to the physical display configuration. Our election of this architecture does

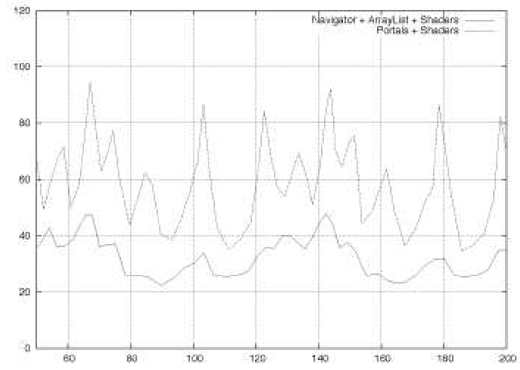


Figure 5. This plot shows that if shaders are applied to the containers, our portal rendering method delivers higher frame-rate than a simple array list approach.

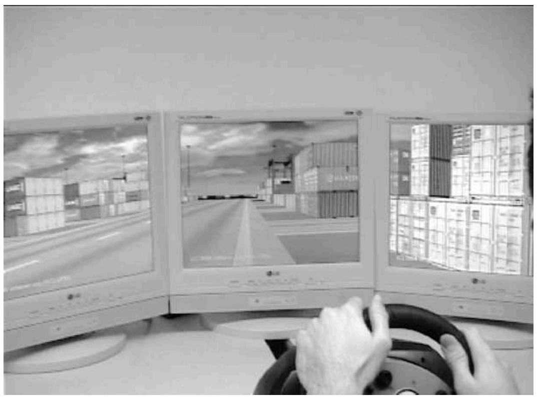
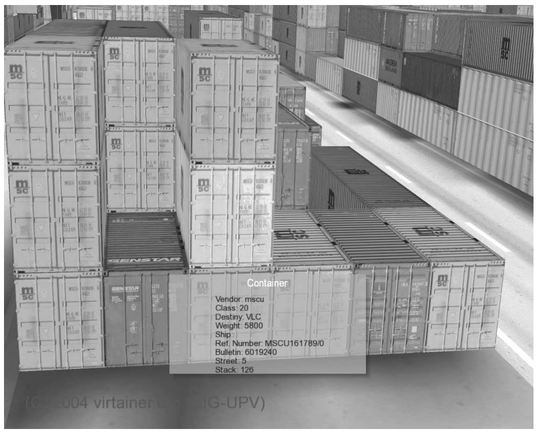


Figure 6. Virtainer running in a cockpit view configuration.



Container
Vendor: mscu
Class: 20
Destiny: VLC
Weight: 5800
Ship:
Ref. Number: MSCU161789/0
Bulletin: 6019240
Street: 5
Stack: 126

Figure 7. This image shows what happens when a container is selected. An information window appears with relevant information about the selection. The same applies to any crane.

not mean that we are limited to connecting the devices directly to the master: any machine can receive this kind of events and send them to the master, which will then notify the changes through the cluster.

We have used joysticks, game-pads and a wheel to control *Virtainer*, and we have also a tracker which follows the user head movements, enabling the user to look around the terminal. The screen configurations tested are jumbo-box, a cockpit view with three monitors, and a head mounted display. An example can be seen in Figure 6. Finally, containers and machines can be selected with the mouse to show information about them (see Figure 7).

Conclusions

We have presented here a multi-platform simulation system capable of representing a maritime container terminal in 3d with a sustained frame-rate. It offers support for virtual reality configurations, such as head mounted displays and caves, using swap-buffers synchronization to avoid disparity between the frames rendered in different screens. Any device can be used to control the navigation, and it is possible to select relevant elements of the terminal to show related information. The camera can be easily attached to any crane, as shown in Figure 8. We have created a web site where we publish project related information, located at http://www.virtainer.org.

We are currently developing an event generator to simulate the messages received through the communications layer. This application will allow us to represent the machinery animations outside the maritime terminal environment.

Other interesting areas are the use of *occluders* [3] for systems with lower specifications or even larger scenes. There is also some room for improvement in the synchronization method used; a peer-to-peer scheme may be better suited for our system than the current master-slave configuration. Finally, in the next months *Virtainer* will be ported to a *CAVE* system [4].

Acknowledgements

The *Virtainer* project is being supported by grant TIC2002-04166-C03-01 from the Spanish Ministry of Science and Technology and a STREP project, IST-2-004363 of the European Union. We thank *Virtainer*, *Gama* teams for their work and collaboration. We are also grateful to Marítima Valenciana for their support of our research ef-

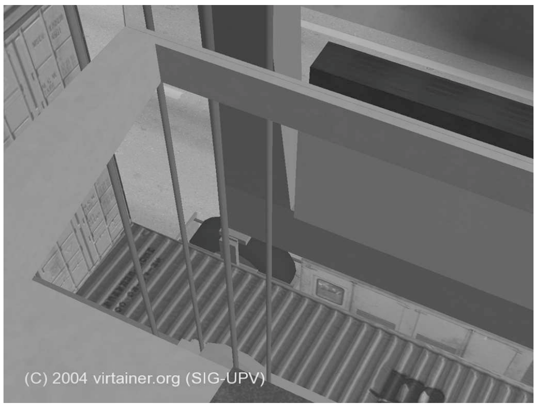


Figure 8. One of the interaction modes puts the camera in the cab of the selected crane.

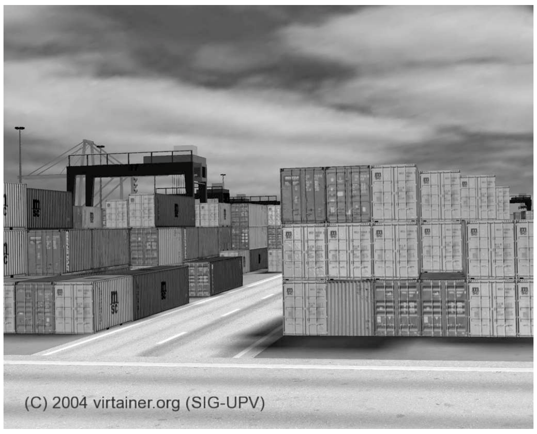


Figure 9. In this image a shader has been applied to the containers. It features per-pixel illumination, bump mapping and displacement mapping.

forts.

References

[1] D. Burns and R. Ostfield. Open scene graph - an open source solution for real time image generation. IMAGE Society, 2003.

[2] D. Cohen-Or, Y. Chrysanthou, and C. Silva. A survey of visibility for walkthrough applications. IEEE TVCG, 2001.

[3] S. Coorg and S. Teller. Real-time occlusion culling for models with large occluders. IEEE TVCG, 1997.

[4] C. Cruz-Neira, D. J. Sandin, and T. A. DeFanti. Surround-screen projection-based virtual reality: The design and implementation of the cave. In J. T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 135–142, Aug. 1993.

[5] J. El-Sana. Integrating occlusion culling with view-dependent rendering. IEEE TVCG, 1997.

[6] N. Elmqvist. Axis-aligned bounding box culling for portal rendering, 1999.

[7] N. Elmqvist. Introduction to portal rendering. 1999.

[8] L. M. Gambardella and A. E. Rizzoli. The role of simulation and optimisation in intermodal container terminals. European Simulation Symposium, 2000.

[9] M. Hadwiger and A. Varga. Visibility culling. The Institute of Computer Graphics and Algorithms, Vienna University of Technology, 1997.

[10] M. Haines and K. Lagendoen. Platform-independent runtime optimizations using openthreads. In *11th International Parallel Processing Symposium*, April 1997.

[11] J. Hultquist. Backface culling. In *Graphics Gems*, pages 346–347, 1990.

[12] P. Jorquera, T. Barella, D. Llobregat, and R. Vivó. Visual gama. EUROGRAPHICS, 2003.

[13] S. Kumar, D. Manocha, W. Garrett, and M. Lin. Hierarchical back-face computation. In *Eurographics Rendering Workshop*, pages 235–244, 1996.

[14] A. Lerner, Y. Chrysanthou, and D. Cohen-Or. Breaking the walls: Scene partitioning and portal creation. In *Pacific Graphics 2003*, 2003.

[15] D. P. Luebke and C. Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *Proceedings 1995 Symposium on Interactive Computer Graphics*, pages 105–106. Department of Computer Science, University of North Carolina, 1995.

[16] J. Posadas, P. Pérez, J. Simó, G. Benet, and F. Blanes. Communications structure for sensory data in mobile robots. *Engineering Applications of Artificial Intelligence*, 15:341–350, 2002.

[17] J. Poza, J. Posadas, J.E.Simó, and A.Crespo. Data and event management in a maritime terminal of containers. *New Technologies For Computer Control*, pages 269 – 274, 2002.

[18] G. Reitmayr and D. Schmalstieg. An open software architecture for virtual reality interaction. ACM, 2001.

[19] G. Schaufler. Dynamically generated impostors. In *D. W. Fellner, editor, Modeling Virtual Worlds - Distributed Graphics*, pages 129–136, 1995.