

improving the visualization of

3D Simulations...



... using Computer Game technology

A masters thesis
By Jonatan Bijl - 1174347
For the Delft University of Technology and TBA

Table of Contents

1 Introduction.....	4
Motivation and Objectives.....	4
Research Approach.....	7
Overview.....	9
2 Selecting a Game Engine.....	10
Criteria for selection.....	10
Criteria for comparison.....	11
Ogre3D.....	13
OpenSceneGraph.....	15
Irrlicht.....	17
Crystal Space.....	18
Conclusions.....	21
3 Single- or multi-threading.....	23
Single loop.....	23
Multiple loops.....	24
Implementation.....	24
4 Interfacing C++ code to Java.....	26
Kinds of interfaces between Java and native code.....	26
Comparison of the interfaces for this thesis.....	27
5 Public interface.....	29
Configuration.....	29
The interface generated by SWIG.....	30
The interface for general use.....	31
6 Integrating Simulation and Visualization.....	32
DSOL implementation.....	32
Controls2 implementation.....	32
Static and Dynamic objects.....	35
7 Optimizations and container stacks.....	37
Material optimization.....	38
Object batching.....	40
Shadow techniques.....	42
8 Comparison of the visualizations.....	44
Usability.....	44
Appearance.....	45
Performance.....	47
9 Conclusions.....	49
Subgoal evaluation.....	49
Summary of achieved sub-goals.....	50
Answer to the research question.....	51
Future work.....	51
10 Appendices, Bibliography and Glossary.....	53



Abstract

The game industry is one of the quickly advancing areas of software industry. Other areas of industry have things in common with the game industry, but are way behind. This is especially the case for the rendering done for simulations. The graphical quality of what is shown in simulation packages is about 10 years behind compared to the game industry.

Because simulations in which the positions of objects play important roles, like logistics simulations, have a lot of things in common with computer games, they can use game technology. This research attempts to bridge the gap between the two industries, by creating a library that can be used to visualize the simulations, and that itself is based on a game rendering engine. In this way, the simulations can be visualized with a better quality, making use of the latest game technology.

In this research, a proof of concept of such a visualization library is created, which utilizes the Ogre3D rendering engine. The working of the library and its general usefulness are demonstrated by using it to visualize two simulations. One of them is a simple DEVS simulation, the other is a complex container terminal simulation based on the DEVS formalism.

It is found that the new visualization gives a huge increase in the realism of the graphics. By hiding implementation details and providing a configuration system, the solution is easier to use than the current visualizations of these simulations. Choices however need to be made in the trade-off between realism and rendering speed. As the solution is geared towards the future, it has been designed in a multi-threaded way. That has the drawback that it slows down the simulation on single-core processors.



Foreword

The past 13 months I have been working on my masters thesis, as an assignment from the simulation company TBA. Even though I did the opposite of the procedure that the university describes, by first choosing a company and an assignment and after that looking for a supervisor from the Delft university of Technology, the university agreed with my assignment. With the useful feedback from my supervisors Peter van Nieuwenhuizen (DUT) and Csaba Boer (TBA), I have done a literature study [Bijl, 2009], and now have also finished my masters thesis.

TBA is a company that uses its simulation software to optimize container terminal logistics, and test container terminal operating systems. I got to know TBA when I did my bachelors' project there, together with Jelle Fresen, in 2006. For that project we have also worked on improving 3D animation for the terminal simulation software. In both the projects I have enjoyed seeing how I could compensate my lack of knowledge in the area of simulation with my knowledge on the area of computer graphics, and how both TBA and I could learn from each other.

I want to thank my two supervisors, Peter van Nieuwenhuizen and Csaba Boer for their feedback on and interest in my work, and the rest of the committee for their time. Also I would like to thank my housemates for enduring that I spent less time with them as a result of my work on the thesis. Above all I want to thank God for giving me the brains that enabled me to go to university in the first place.

I hope that you will find this thesis interesting to read, and that the results of this research might help you to see new possibilities of using computer game technology for useful purposes.

Jonatan Bijl, September 30th, 2009



Chapter 1

Introduction

Computer Games have always been inspiring for people. Since the 50's the advances in computer technology have been accompanied by games that use the new possibilities. In the past two decades however, games have become such an important use for computers, that the choice for which computer should be bought is heavily influenced by which games need to be played. The game industry creates a demand for bigger and faster computers, and computer production provides these.

Motivation and Objectives

One of the areas in which the game industry is quickly advancing, is computer graphics. When comparing the graphics of a modern game to those of many simulation packages, one must conclude that the graphics of the simulation are unrealistic and look outdated. An important cause for this, is that the graphics for simulations have a different focus from those of games. As is elaborated on in [Bijl, 2009] simulation graphics have three main goals, namely validation, analysis and marketing. In games the goal of graphics is to draw the player into the game [Brightman, 2009], and this is often done by making it look realistic. These goals only partially overlap. However, the technological advancements in the gaming industry provide new ways of visualizing data, so that new opportunities also arise for areas of visualization that require clarity over realism.

By applying proven Object-Oriented design principles, various so-called *Game Engines* have been produced. These are software building blocks that can be re-used for several games which differ in storyline and appearance, but have many things in common in gameplay and game logic. Sometimes a game engine is assembled from smaller building blocks, that each handle a part of the game logic. For example there can be a physics engine and a rendering engine.

For logistic optimization simulations can be used that simulate at such a detailed level, that the positions of objects are an important factor in the behavior of the system. This could be an agent based simulation of crowds of people in an airport building, or a discrete event based simulation of a packaging system. This class of simulations has many things in common with games, which use discrete event simulation for doors, treasure chests and other objects, and use agent-based AI for computer-controlled characters. Because there are many things in common, this opens opportunities for simulation to use technology from games. One of these opportunities that is worth researching, is the use of a rendering engine to visualize the simulation in 3D. This is phrased in the following research question:

How can the use of a rendering engine improve the visualization of simulations?

In order to correctly answer this question, a solution will be programmed in the form of one or several libraries or executables. The solution does not need to be a fully functional program, but should fulfill as much of the requirements as needed to answer the question.

In the research question, some things need to be more clearly outlined. Firstly, there are several areas of improvement. This research will look at the usability, the appearance and the performance of the solution. To evaluate the improvement, a comparison will be made between old and new visualizations. Secondly, it needs to be defined for which types of simulations the solution should work. Thirdly, some guidelines need to be defined which help making the right decisions in the software design process. For each of these categories of restrictions, constraints and directions, several sub-goals will be defined.

Usability

Usability focuses on the process of using the visualization. The main task of a simulation engineer is to create, configure and validate a simulation, and possibly also analyze the results. None of these directly involve creating graphics, but graphics can be of help with all of them. Because creating graphics is not one of the main tasks of a simulation engineer, it should also not require much of his time or knowledge in that area. This gives us two sub-goals

- sub1) Using the solution should not require specialized knowledge of computer graphics***
- sub2) The solution should require only a minimal amount of work to visualize a simulation.***

Appearance

On the other hand, not only the process but also the end result is important. The appearance of the visualization should be more useful than current solutions. This can either be for the purposes of validation and analysis by showing more information or showing the information in a more insightful manner. Or it could be for the purpose of marketing, making it look realistic. As these uses of visualization are quite different, the solution should support both, but leave the decision to the simulation engineer. This gives us two extra sub-goals.

- sub3) The solution should allow more realistic rendering than current visualizations.***
- sub4) The solution should allow more insightful visualization than current visualizations.***

Performance

To make sure the visualization is able to show what the simulation does, it should at least achieve a frame rate that equals the update rate of the simulation. Because the frame rate is also important for user feedback when moving the camera, a minimal frame rate should be chosen. If the frame rate comes below 15 fps, the human brain will clearly notice the separate frames, so the animation can be considered sufficiently fluent if the frame rate is higher than that. The frame rate is heavily influenced by the render window size. For this thesis, the arbitrary choice is made for a resolution of 1024x768 pixels. The frame rate is also heavily influenced by the amount of objects that need to be shown. Therefore we state that this frame rate should be achieved for complex simulations.

- sub5) *The solution should have a frame rate of at least 15 fps at a window size of 1024x768 pixels, even with complex simulations.***
- sub6) *The solution should be able to achieve the same frame rate as the update rate of the simulation, even with complex simulations.***

Another important point to keep in mind is, that the visualization should be usable while running the simulation. Therefore the system resources that the simulation needs, like CPU time and memory usage, should still remain available.

- sub7) *The solution should leave enough CPU time and memory as is needed by the simulation.***

Simulation

Another part of the research question that needs clarification, is the type of simulation that is used. As mentioned earlier, basically any simulation in which the three dimensional positions of objects play an important role, is suitable for visualization using rasterization. This rules out fluid and air flow simulations, which rather require volume-based visualizations, and most of the continuous simulations that are more based on formulas. Some rather abstract discrete-event simulations are also no good candidates, because they are so abstract that the position of the objects does not relate enough to the objects' state. The lack of a general term for class of simulations that is targeted, results in an overly general research question.

- sub8) *The solution should be usable to visualize any simulation in which the change of the positions of objects over time plays an important role.***

When talking about simulation and improvement, this still is a huge area. To demonstrate the wide usability of the solution, it will be used for two different simulations, that both already have 3D visualization. The used simulations are a 3D animation example with the DSOL simulation library, and the Controls2 demo model. The latter will be used to evaluate the performance for complex simulations. Even though the solution will be used only with these two simulations, it should be programmed in such a generic way that it should be possible to adapt it for use with other simulation programs.

Software engineering

To achieve good software usability, not only the direct goals should be met. The solution should also follow general software engineering practices.

- sub9) *The code of the solution should be extensible, readable and well documented.***

As both simulations that will be used are platform-independent, so should the solution be. At least it should run on Windows and Linux, because these are the OSES used most in the industry, for desktops and servers respectively.

- sub10) *The solution should at least run on MS Windows and Linux.***

In the game industry, game engines have a relatively short life cycle. They are created, then used for three or four years, then they are succeeded by a new game engine that has been redesigned to fit the new trends and possibilities. In comparison to that, the life cycle of a simulation product is much longer. To make sure that the visualization can remain up to date and the simulation does not need to be edited every time the visualization is changed, a facade interface should be created, hiding implementation details but providing an interface that is easy to use for simulations.

sub11) The interface to the simulation must hide implementation details from the simulation, but provide a simple and generic interface.

Research Approach

To answer the research question and produce a piece of software that fulfills the subgoals, a certain approach must be chosen. The approach consists of three important steps, namely test case selection, product implementation and test case evaluation.

Test case selection

As mentioned earlier, the solution is meant to be widely applicable. On the other hand, it is practically impossible to test it with lots of simulations. Therefore some simulations should be chosen as test case. If the goal can be easily achieved using these simulations, it can be assumed that the solution will also work for similar simulations. To make a comparison possible, the two simulations need to already have some 3D visualization. For this research, DSOL and Controls2 are chosen. DSOL represents the broad range of simple 3D simulations, and Controls2 represents more complex, demanding and detailed simulations.

DSOL simulation library

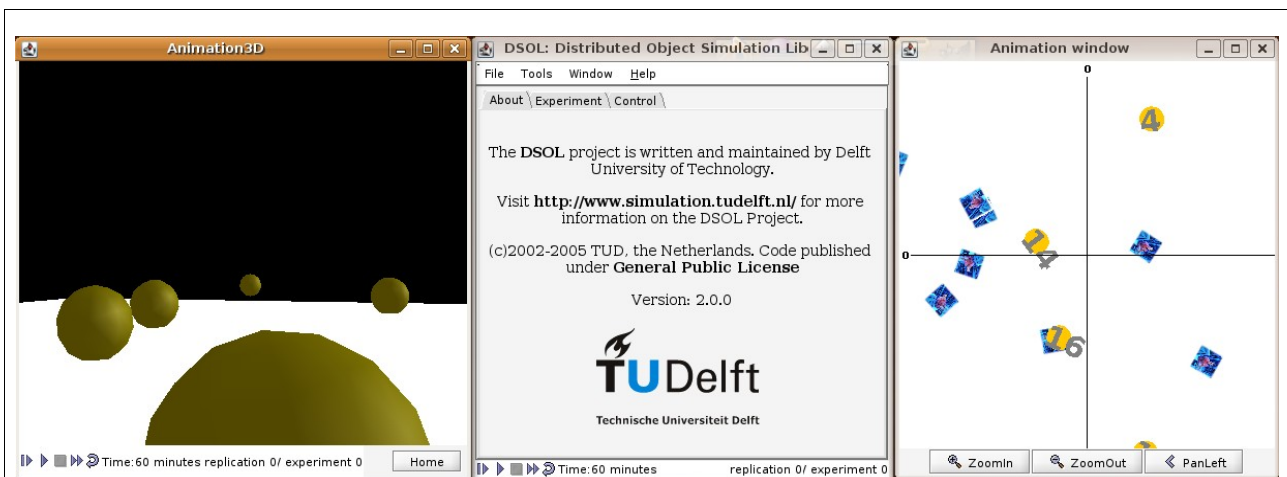


Figure 1: The DSOL application framework, with 3D and 2D animation

The DSOL simulation library is a library developed by Delft University of Technology. It contains the basic building blocks for discrete event simulation, in the form of a library of Java classes. It requires the simulation engineer to program the simulation objects himself, based on these building blocks. It also contains a framework with useful tools for running simulations, viewing them in 2D and 3D, and gathering statistics. Besides the current 3D visualization, the new solution should be provided as an alternative. As simulation model for this framework, one of the original DSOL example simulations will be used, which moves both continuous and discrete event controlled objects. The 2D and 3D view of this model is shown in Figure 1. The visualization for this extremely simple simulation should be seen as a proof-of-concept.

Controls2

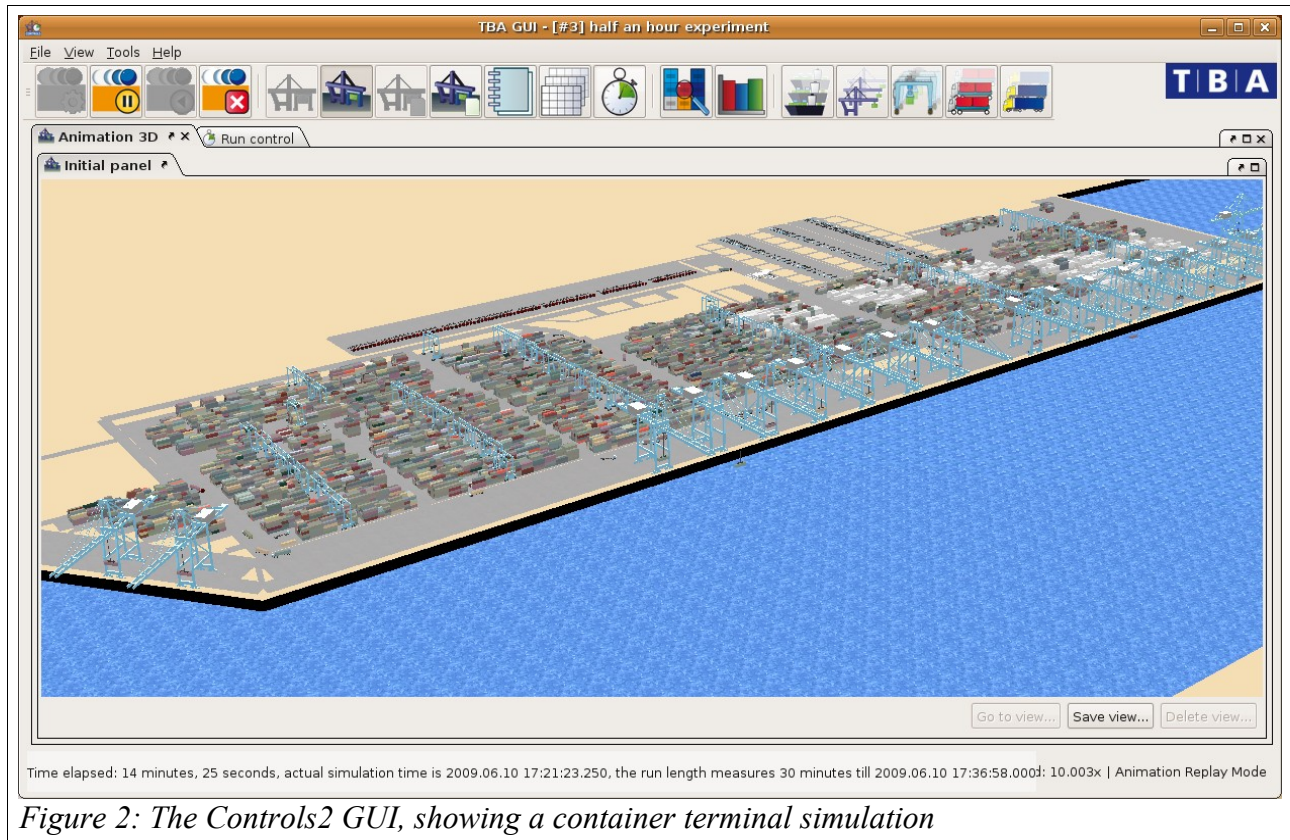


Figure 2: The Controls2 GUI, showing a container terminal simulation

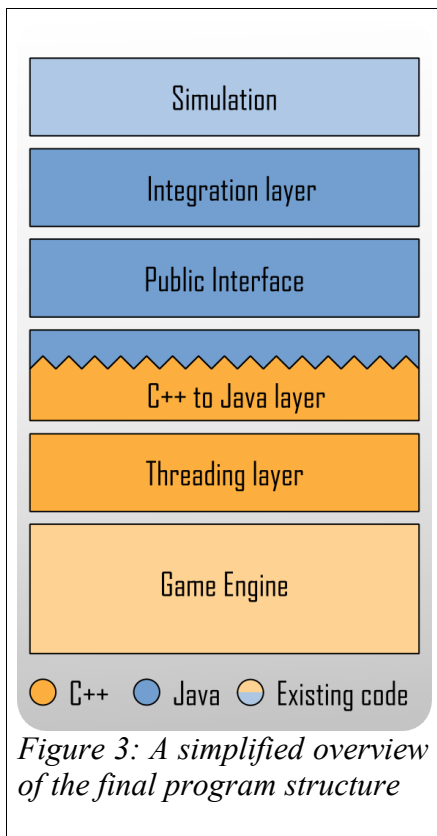
Controls2 is a product of the company TBA. It is a library that is specialized in the detailed simulation and emulation of container terminals. It can be run as separate program, or linked to a *Terminal Operating System*. Controls2 is based on DSOL, but due to the complexity of the simulations, a big part of the framework of DSOL has been replaced by custom-created code, and the 3D animation has been decoupled from the simulation quite well. The complexity of the simulation and the amount of objects that need to be shown make this the real test case for the solution.

Controls2 will run a demo simulation of an average container terminal. It is called a demo simulation because it will not be linked to a TOS. The simulated terminal is an average-sized container terminal, featuring about 140 trucks, 65 RTG's and 19 Quay Cranes. The container stacks on the terminal often contain about 20 000 containers in total. Figure 2 shows the GUI with the terminal.

Product implementation

The solution will be implemented in a series of steps, each step focusing on one aspect of the problem. For every step, the considerations and choices will be discussed. In the design of the final solution most of these aspects can be recognized as layers, so that every layer handles one part of the problem. Figure 3 gives an overview of these layers.

In general the programming approach will be bottom-up, starting at the game engine, and ending with linking it to the simulation. After the simulations have been linked to the game engine, several layers might require improvements, either in software refactoring (to make the design more flexible or clear) or in optimization (to make the software run more optimally).



During development, the approach should make sure that the software engineering goals are met. Programming is done on a Linux computer, but the software is also regularly tested on a Windows XP computer to enforce platform independency. Automated *unit tests* are used for the parts of the system that can easily be tested in such a way. The parts that require manual testing, especially for the appearance of objects, are manually tested after every code change that could influence them.

Comparison and conclusion

Some of the subgoals suggest an improvement in comparison to the old visualization, and therefore should be compared. For the comparisons that can be objectively measured, the results will be shown. For the more subjective subgoals, like *sub3* and *sub4*, the differences will be described and evaluated.

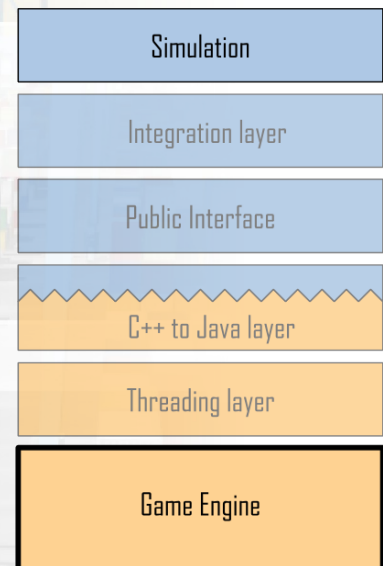
Finally, for all the subgoals it will be summarized to what extent they are met. After all these goals have been evaluated, it is possible to formulate a conclusion, which will answer the main question.

Overview

The rest of this thesis will follow the order of the approach described above. The test cases have been selected in this chapter. In the chapters 2 through 6 each chapter will handle one aspect of the problem, and thus also one layer of the design. Chapter 7 will describe several optimizations that have been done, and chapter 8 will compare the visualizations. Finally, chapter 9 will summarize the results and draw a conclusion.

Chapter 2

Selecting a Game Engine



The first part of this research is the game engine, which will be used to visualize the simulation. There are many commercial game engines available, but their prices range from \$1000 to \$300 000. For the purpose of this research it will suffice to use a freely available open source engine. The software will be designed in such a way that the game engine can easily be replaced by a better (commercial) one if the results are good enough. This is achieved by the layered design which we have seen in the previous chapter.

The goal of this chapter is to select a game engine that will be used for this research. This will be done by implementing a test application with several engines. First the criteria for selecting an engine will be discussed, and the engines will be selected. After that, the way of comparing them is described, followed by a description of each of the selected engines and the challenges that were met when implementing the test application with it. Finally, a comparison is made and the most useful engine is chosen.

Criteria for selection

For this research, a game engine needs to have the following requirements:

- It should be a native engine, not interpreted or running in a virtual machine. This is required to achieve a good rendering speed.
- It should be cross-platform. As DSOL simulations run on all platforms, so should the visualization.
- It does not need to be a full game engine, as the only feature we will use is the rendering.

Based on these selection criteria, four engines have been selected:

- Ogre3D
- OpenSceneGraph
- Irrlicht
- CrystalSpace

Criteria for comparison

From these engines, the most suitable engine needs to be chosen. The opinions of forum members on which engine is the best, are quite colored, and therefore hard to use. To make a good choice, a more substantiated opinion is needed.

Features

It is important that a game engine has useful features. These can be graphical features, like mentioned in [Bijl, 2009], resource management features, the used rendering system (OpenGL, DirectX, OpenGL ES, etc), the supported file formats. Although not all features of game engines are equally useful, the game engine should allow modifications on textures and 3D models without having to recompile. It should also be able to use the vertex and pixel shaders, which are quite necessary for efficient resource usage. If it is possible to render to multiple windows or viewports, or render to GUI widgets, that increases the possibilities of using it elsewhere.

Engine Design

The engine should not only have good features, but also have a good design. Using a badly designed engine will force workarounds and bad design in the abstraction layer. The design should be relatively easy to understand and use, and be consistent. It should also provide a way to prevent memory leaks. A possibility to only load the parts of the engine that are needed would be nice.

Coordinate System

One of the things that is important is the coordinate system that is used. In the world of 3D graphics, there is not just a single way to choose the coordinate system. There are three commonly used coordinate systems, as shown in Figure 4. The right-handed y-up system is used in OpenGL and in several 3D modeling programs. The right-handed Z-up is used for geographical applications, *3D studio max*, *Blender*, and in mathematics. The left-handed Y-up system is used by DirectX.

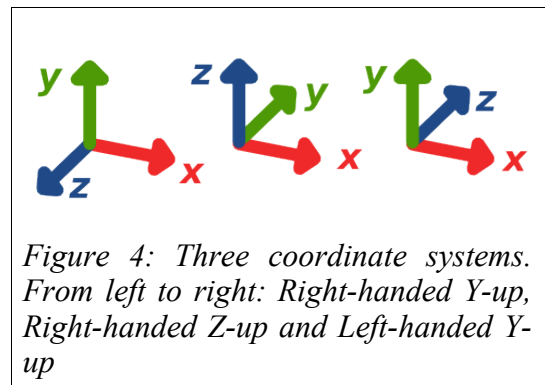


Figure 4: Three coordinate systems. From left to right: Right-handed Y-up, Right-handed Z-up and Left-handed Y-up

These differences make that in converting or loading files, often transformations need to be applied. When a model from blender must be converted to DirectX format, this means that it must be rotated by 90 degrees, the Z coordinate of each vertex should be inverted, and the order of the vertices per polygon need to be inverted.

The test application and the interface for this thesis use the right-handed y-up system, but as simulations probably also use different coordinate systems, a support for other coordinate systems might be added later. For now, each engine's coordinate system should be adapted to support right-handed y-up coordinates.

Development and support

For the use of any library it is important to have good support. Firstly, the library should be well documented, so that any question that arises when using it, can be answered in a few clicks. Secondly, a good set of tutorials helps to get to know the library. Often it is easier to get to know a system by using it in example programs with increasing complexity, than to just start implementing your own program. Good tutorials give a clear starting point for your programs, and explain why

certain things are done. They should not only be a how-to, but also a why-to. Thirdly, it helps if there are people to whom questions can be asked. In the open source community this often is a chatroom, a mailing list or a forum.

The library should not only be a good choice for now, but also for the future. The development should be continuing at a steady pace, with new releases adding new features and fixing bugs. The continuation of the development is ensured by a big user community, so that there is less chance that the project will be abandoned. If commercial companies use a library, they increase the chances of survival of a library because the abandonment of a library would also be a loss for them.

License

When using open source software, it is important to see which license it uses. A lot of open source libraries are licensed under the GPL, which enforces that all software that uses them, also publishes its code under the same license. One of the consequences is that everybody to whom the software is distributed, should also have access to the source code of the software. GPL-licensed libraries are hardly used for developing commercial software because the developing company would not benefit from availability of its own code to others.

Especially for this situation, the LGPL and a lot of other licenses have been created, which allow dynamic linking to the libraries. These only require changes to the library itself to be published, instead of all the code that makes use of the library. Because the solution for this research should also be useful for commercial use, these licenses are to be preferred.

Efficiency

It is very important that a rendering engine is efficient: it should not burden the CPU too much (because the simulation also needs CPU time), and it should achieve quite good frame rates.

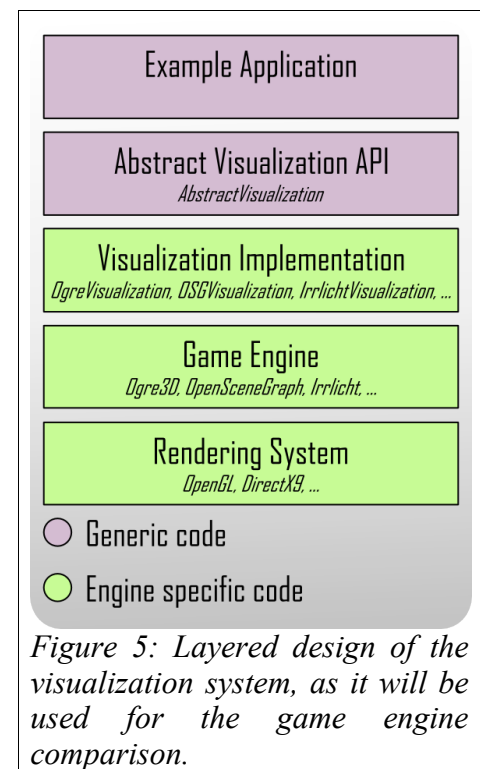
To achieve this efficiency, the engine should do as much as possible on the GPU, and have good scene management techniques. For none of the selected engines, this is done in an easy way. To also run on older machines, every engine works with the *fixed function pipeline*, and every computationally intensive task – like shadowing and animation – involves much CPU work. To use the GPU efficiently, shaders need to be written, and the engine should be told to do less on the CPU. Also in the scene management a lot of optimization can be achieved, by differentiating between static and dynamic objects, using *levels of detail*, and using *impostors* for the objects that are really far away.

The time for doing this comparison is limited to about one week per engine. Therefore, it is impossible to add all these optimizations. They will not be added to any of them, to keep the comparison fair.

Reference Application

To make a fair comparison, a reference application is created. To be able to really focus on the game engine, this application does not contain any threading or Java layers. The application accesses the game engine through an abstraction layer. The tasks of the abstraction layer are:

- To provide a simple interface to the game engine, using only primitive data types and strings as parameters.



- To hide the game engine-specific approach from the application.



Figure 6: The Ogre3D version of a simple test application.

This improves the chances of making the visualization reusable and the game engine replaceable. The reference application demonstrates this replaceability by only using this API. This is shown in Figure 5. The application code remains the same, even though it can be compiled together with the different engines.

This application consists of a huge amount of static objects, namely 100 000 cargo containers, a smaller amount of moving objects, namely 100 straddle carriers, a ground plane and a *sky box*. Each of the straddle carriers makes a random move after every frame. The Ogre3D version is shown in Figure 6.

Ogre3D

Ogre3D (Ogre) is a graphics engine that focuses more on design than on rendering speed. Its history goes back to 1999, but the first official release was in 2005. Yearly a new version has been released, adding features and improving design.

Features

Ogre is just a rendering engine. Collision detection, sound and networking features are not present. User input is done via the *Open Input System*, a separate library which is a spin-off of the Ogre project. Menus, toolbars and cursors can be drawn using the CEGUI library.

File formats

Ogre uses its own format for 3D models and animations. This means that 3D models made in some 3D modeling program like *Blender*, *Maya* or *3D studio Max* need to be converted to Ogre's own format. To make the writing of plugins for 3D software easier, there is an intermediate format: ogreXML. A small program converts the ogreXML file into Ogre's mesh format, which can be loaded into the game engine. The materials are specified in a separate text file, which is generated by the mesh exporter. This process is shown in Figure 7.

The exporter for blender is quite limited. It exports every part of a model as a separate mesh. To have the whole shape as a single mesh, the parts must be joined inside blender. Also, the more advanced material properties from blender are not exported, like normal maps and transparency maps. These can be added by adding extra lines in a material file, using a text editor. If Ogre is going to be the engine of choice, it might be a good idea to improve the exporter script to support these features.

The XML converter optimizes the 3D model for Ogre, slicing the model into triangle strips and triangle fans. It can even generate lower levels of detail, but for concave models these often look ugly.

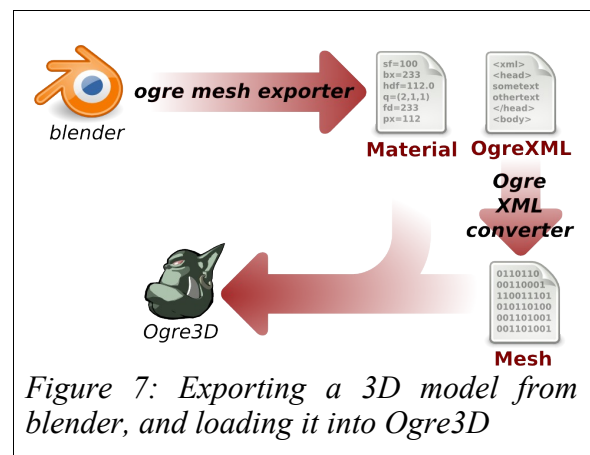


Figure 7: Exporting a 3D model from blender, and loading it into Ogre3D

Shaders

Ogre supports vertex and fragment shaders, which can be either written in HLSL or GLSL, or in nVidia's Cg. Using HLSL or GLSL however, limits the user also to using DirectX or OpenGL respectively. Cg is cross platform, as nVidia's compiler can compile it to both HLSL or GLSL shaders. Figure 8 shows a scene from a commercial game, that uses shaders for the reflection and waves in the water.

From the material file of a model, it can be defined which shaders should be used to render the material. Even multiple passes can be defined, when needed.

As the capabilities of shaders increase, the amount of work to be done on the CPU decreases. In Ogre, tasks like skeletal animation and shadowing are done by the CPU. If these should be done using shaders instead, there are special flags to tell Ogre “Don't do this, the shaders will take care of it”.

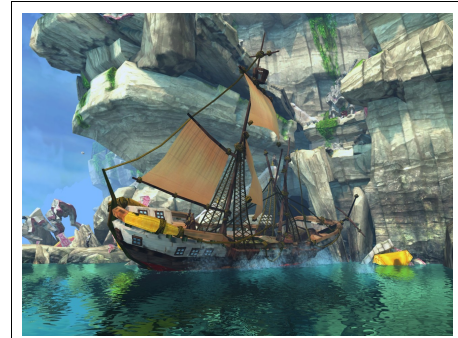


Figure 8: A screenshot from "Jack Keane", a commercial game by Deck13 interactive. The water effect is done using shaders

Plugin system

Ogre3D features a quite flexible plugin system. Libraries can be updated or added, without having to recompile the program. All that is necessary, is editing the configuration files to tell where the new shared library is.

Ogre has several kinds of plugins. There are several scene management plugins, providing optimizations for either indoor or outdoor scenes. There are also plugins for creating particle effects.

Even the rendering system is implemented as a plugin. There are two rendering plugins: OpenGL and DirectX 9. Work is being done on OpenGLES and iPhone support. Earlier, a port of Ogre to the Pocket PC platform has been made [Monte Lima, 2006]. Even though the performance on the Pocket PC was really low, it illustrates how the design of Ogre allows for a broad range of uses.

Engine Design

As mentioned, the developers of Ogre paid attention to the design. Many design patterns can be easily recognized in the system.

Partial use

Ogre is quite modular, in the sense that most of the more complex functionality is inside extensions. On the other hand, on class level it is quite tightly coupled. Ogre uses singletons quite much. A singleton is a class of which only one instance can be created, and which has a central point of access [Gamma, 1995]. For Ogre this means that in order to create any object, the Ogre `Root` object needs to be created first. This makes code using Ogre hard to unit-test. Also, the program is limited to having one Root and thus one render loop. This limitation however is not really a problem, because it is still

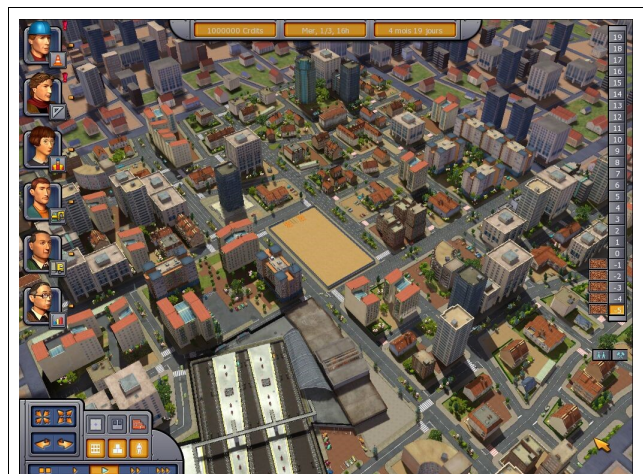


Figure 9: Building & Co., a game by Creative Patterns, uses Ogre3D as rendering engine.

possible to have multiple windows, and multiple 3D worlds.

Undocumented dependencies

Ogre is quite well documented. However, sometimes undocumented dependencies and side-effects are found. For example, adding a 3D model that casts shadows to a scene, can give exceptions if there is no `RenderWindow` yet.

Development and support

The code is quite well documented. There is a wiki with beginner and intermediate tutorials to help people get started. Some of the tutorials bother users with issues they should not be bothered with yet, like using macros to handle platform differences. There is also a forum where good questions get an answer in about half a day, and an IRC chatroom where questions are answered in minutes.

At the moment of writing, Ogre is being sponsored by two game development companies. Through the past years it has been used in several games, most of them role playing games and point-and-click adventure games. It has also been used in a building game (Figure 9).

Ogre's official development team has 5 members, but there is a big active community requesting and implementing features, adding plugins and answering each others questions. All in all, Ogre is an actively developed, well-supported engine.

OpenSceneGraph

OpenSceneGraph (OSG) started as a hobbyist's project to create a hangglider simulator in 1998. In the beginning the simulator used SGI's *Performer* scenegraph, but quite soon developed it's own scenegraph. In 2002 a community formed around OSG, and it was used for several training simulators and game projects through the years.

Features

OSG is purely a rendering engine with collision detection. For game development the Delta3D project was created, which combines OSG with sound, physics and animation libraries.



Figure 10: OpenSceneGraph was used in "Pirates of the XXI century" in 2005

File formats

OSG uses the opposite approach of what Ogre does: it has an interface for model loaders, and plugins are supplied for loading 3D models in all kinds of formats. For several file formats, OSG even has libraries to create the files in some of these formats. This means that automatically generated meshes can be stored into these file formats, or converted from one format to another. Because not every format is capable of storing all the information from an OSG scene, there is a special OSG file format which can be used to store and load scenes without data loss.

Shaders

OpenSceneGraph supports shaders, which can also be seen in the beautiful water effect of Figure 10. As a consequence of the choice for OpenGL, only GLSL is supported. OSG can switch between the fixed function and programmable pipelines.

Engine design

OSG heavily uses the benefits of OO programming. In contrast to Ogre however, it does not attempt to hide implementation details of OpenGL. A lot of the flags that could be set in OpenGL are also used in OSG.

Coordinate system

Implementation of the camera movements gave some problems with OSG. One of the reasons, is that despite of OSG's use of OpenGL, the world coordinates are in a right-handed z-up coordinate system. On the other hand, the camera itself uses OpenGL's coordinate system (right-handed y-up). If necessary, the imported 3D models were rotated for use in the z-up coordinate system.

Because the application interface for the test application uses a right-handed y-up system, the camera could be kept the way it was, but all the 3D models which were automatically rotated to z-up during loading, had to be turned back to y-up again.

In contrast to Ogre, where the camera could easily be attached to a node in the scene, the camera movement for OSG was much more complex. It was simple to select and use one of the standard camera controllers. But for the fair comparison, it was needed to implement a new camera movement. This happened to be a tedious job. Camera controllers were actually supposed to generate the viewing transformation, instead of just the camera position and orientation. A series of rotation and translation matrices had to be multiplied in the correct order. This in itself is not a difficult task, but because it was not clear from the documentation if these had to result in the camera-to-world transformation, or the world-to-camera transformation, it became hard to figure out. Overall, the approach was more primitive than could be expected from a scene graph library.

Visitors

In itself, the scene graph is just a tree data structure to store the transformations of objects. To handle many of the operations that need to be applied to the scene graph, the visitor pattern is used. By using this pattern, extra functionality can be added to nodes without editing the node code. A visitor can traverse the scene graph and apply the functionality for each of these nodes. This is used for the culling, for selection of levels of detail, for updating animations, for applying shadows and shaders.

Lighting issues

When creating the ground plane, the first attempt was to generate the shape in the program. This worked quite easily, but the plane would switch between lighted and

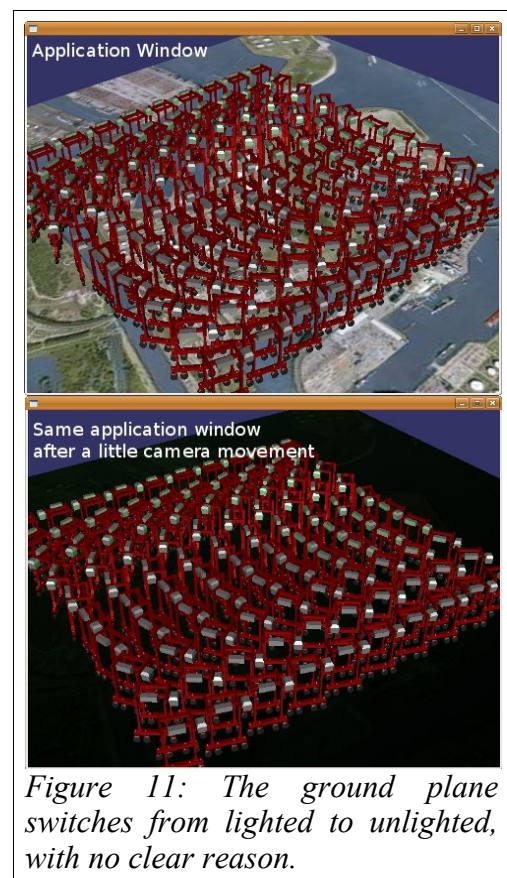


Figure 11: The ground plane switches from lighted to unlighted, with no clear reason.

not-lighted state. No solution to this could be found, not even with the help of the community. For the sake of time, a 3D model of the ground plane was loaded instead.

Development and support

The development of OSG is really active. In a timespan of one and a half year, five new versions were published. The tutorials are quite outdated. Especially the tutorials which are intended to be done as a series, which would be the most useful, are outdated. This is a pity. The documentation is quite ok, except that in several places the specific required information is missing, for example with the camera projection matrix.

The community around OSG is quite active. In 2008 a European user meeting was organized in Paris, and plans are to have another one in 2009. There is a mailing list/ forum which has active discussions and response in about half a day. OSG has been used in some flight- and driving simulators, virtual city guides, and several games.

Irrlicht

The development of Irrlicht started in 2002. While at first a programmer's solo project, the development team grew to 10 people through the years. It is meant to be a game library, but mainly focuses on graphics and collision detection. External libraries are needed to add sound, physics and AI.

Features

Irrlicht is a cross-platform engine, which can use either OpenGL or DirectX for displaying the graphics, or one of several software renderers. Because of this, it should be able to work on any PC, with any Operating System. The features Irrlicht provides are the ones that are common for most game engines: particle effects, shadows, character animation, and the loading of a lot of 3D file types. HLSL and GLSL shaders are also supported.

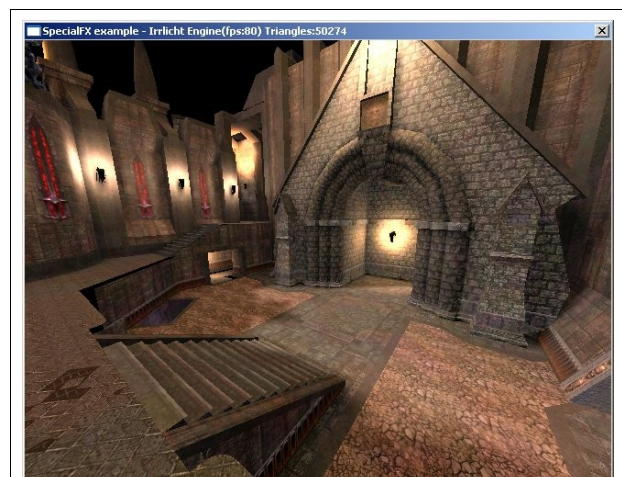


Figure 12: A screenshot of an Irrlicht scene.

Cross Platformness

The test application gave problems when running on different machines. Sometimes only the skybox would show up, sometimes the whole scene would show up. Because of this, the results in the comparison table at the end of this chapter are not measured on the machine which provided the other results.

Engine Design

The library is really geared to providing an easy interface. With a few lines of code, the programmer can already create a simple walk-around-the-neighborhood game. Only when more specific functionality needs to be added, the user needs to dive in to the “hard part” of the library. When correctly designed this is a really useful concept. However, with Irrlicht this resulted in the discovery that the underlying system was badly designed.

Rotations

One of the examples of bad design, was inconsistency in the type of rotations. The rotation of a scene graph node is set using *euler angles* in degrees. However if rotations need to be concatenated or interpolated, it is more useful to have *quaternions*. To create quaternions in Irrlicht, Euler angles in radians are needed, instead of in degrees. Apparently Irrlicht has internally not decided whether to use radians or degrees, and as a result neither can be consistently used.

The API of the quaternion was also quite inconsistent in the ways of creating a quaternion from a rotation in another format. The quaternion constructor could be used for some rotation formats, but some other formats could only be used to set an existing quaternion to a new value.

```
irr::core::quaternion answer;  
answer.fromAngleAxis(angle, irr::core::vector3df(x,y,-z));  
  
irr::core::quaternion answer(radiansRotation);
```

Listing 1: The two ways to construct quaternions. It is strange that there is no quaternion constructor with axis and angle parameters.

Result of these problems was that an extra series of conversion functions was needed to keep the code readable, providing easy creation of quaternions from parameters in axis-angle and euler-degrees formats.

Coordinate system

Irrlicht uses the coordinate system that Direct3D uses, namely the left-handed y-up coordinate system. As the interface uses the OpenGL coordinate system, all the coordinate information coming from the main program needs to be converted. When a 3D model is loaded that is stored using the OpenGL coordinate system, it is converted to the Direct3D coordinate system by inverting the X coordinate. This results in a 3D model that is facing backwards. To get the same behavior as the other rendering engines, the models need to be turned 180 degrees, so that they face forward.

Viewing windows

Irrlicht assumes there is only one window to draw on. Setting up the Irrlicht system immediately creates a window, and assumes that as the one and only rendering window.

Development and support

Irrlicht is actively developed by a team of 10 people. The latest release is from December 2008, the previous was from November 2007. Small updates to the source code are committed almost daily.

The user community is also quite active. There is a forum where questions of new users are answered, screenshots are shared and possible future features are discussed. The community has produced some tutorials to aid with the first steps in using Irrlicht. If you want to use more advanced features, there's the generated API documentation, which is quite good.

Irrlicht has been used for several racing games and a first person shooter.

Crystal Space

The first release of Crystal Space came out in 1997. In the years that followed a user community developed, and several university projects and small games were made. In the past two years

Crystal Space was one of the open source projects that joined the Google Summer of Code. In 2008 it was used for the Open Game project, which produced a computer game almost entirely with open source tools.

Features

Crystal Space takes care of rendering and sound. It comes with a library of standard shaders, like normal maps, specular maps, fog, etc. and uses OpenGL to render it. It uses an own file format for the 3D scenes and models, and has exporters for 3D studio max and blender. Crystal Space also has bindings for the programming languages Python and Java.

Cross platformness

The Crystal Space demo program behaves really different on different machines and Operating Systems. On one Windows machine it did not show anything, on the other one it did not show the textures, on the linux computer it rendered with textures.



Figure 13: Screenshot from the Open Game Project.

Engine Design

Crystal Space consists of two main parts: the Crystal Space library, and CEL (Crystal Space Entity Layer). CEL is a layer on top of Crystal Space, adding functionality that is required for games.

Macros

The code heavily uses *macros*, a construction in C++ to insert or remove code at compile time. In general practice the use of macros for programming tasks is discouraged. Crystal Space however uses them for a lot more tasks. This does not only have consequences for the code that directly uses the engine, but even for code that has nothing to do with it. The following example illustrates the problem. The utility class `HoverCamera` uses game-engine-independent constants for `PI` and `HALF_PI`. The code in which these are declared, is shown in Listing 2.

```
private:
    static const float PI = 3.141592654f;
    static const float HALF_PI = 1.570796327f;
```

Listing 2: The private constants `PI` and `HALF_PI` in `VisualizationUtils::HoverCamera.h`

This code worked fine for the other 3 game engines. However, when compiled against Crystal Space, it gives a compiler error, because Crystal Space has defined macros that replace `PI` and `HALF_PI` with their numerical values. The result is that the compiler receives the following code, and fails.

```
private:
    static const float 3.14159 = 3.141592654f;
    static const float 1.57080 = 1.570796327f;
```

Listing 3: the constant-definitions after the preprocessor has processed them, have become invalid C++ code

Eventually, to be able to use this general class for Crystal Space, it was necessary to add guarding macros to the general code. Although this in itself is quite bad, the macros could even do something worse. In some cases it might even occur that the macro makes header files of other third party libraries unusable, forcing a programmer to add macro guards to third party library headers, in order to protect them from ill-designed code.

Warnings

The Crystal Space development team clearly does not focus on code quality. When compiling the engine library, hundreds of C++ compiler warnings flooded the screen. Also when compiling the test application with Crystal Space, it gave loads of these warnings. When asked, one of the developers replied “Well, they are just warnings, so you can just ignore them”.

Namespaces

In C++, libraries are packaged in namespaces, which are quite similar to Java packages. They are supposed to prevent naming conflicts. If for example two libraries contain a class named `Visualization`, they can be distinguished by their namespace: `Namespace_A::Visualization` and `AnotherNameSpace::Visualization`.



Figure 14: A screenshot from PlaneShift, an MMORPG using Crystal Space.

Crystal Space however also has classes that use the general namespace, which might conflict with other libraries.

Plugin system

The plugin system for Crystal Space is designed to be flexible. Every class is defined in two parts: the interface, and its implementation. The plugin implementations are in separate DLL files which are loaded at runtime.

Scaling meshes

When implementing the abstraction layer, in order to be able to run the test application, a new limitation of Crystal Space came to the surface:

Objects in the scenegraph can not be scaled. This has to do with the collision detection and culling algorithms. If a scaled version of a 3D model is required, the factory must be scaled, and then used to create an instance. Scaling 3D models that are in the scene already, was not possible.

Development and support

Crystal Space is actively being developed by three people. Many features are added, but there does not seem to be a clear direction in the development. The website states that version 1.2 is the most recent stable version, but on the forum version 1.4 was recommended, and the documentation on the

site refers to version 1.9 which is under development.

The Crystal Space forum is not really active, but the developers are always ready to answer questions in the IRC chatroom. The tutorials are at quite a high level, so they are not useful for real beginners.

Crystal Space is used in several games: two point-and-click adventure games, and “Planeshift”, an *MMORPG* (Figure 14)..

Conclusions

Now it is time to compare the engines, and decide which one to use.

Features

When purely looking at the rendering features, the engines are quite similar. Each engine supports high level shaders, character animation and scene management.

There are also big differences. Crystal Space and Ogre only use their own file formats, where the other engines try to support many different file formats. Also in the file management these engines differ from the other two: through configuration files a hierarchy of directories and zip files can be specified, from which the resources should be loaded. In this way, there is no need for hard-coded file paths in the program itself.

Engine Design

In engine design Ogre and OpenSceneGraph are quite good. The sometimes too simplistic and often inconsistent design of Irrlicht makes it useful for quick game prototypes, but not for more advanced software engineering. The macros in Crystal Space, whose consequences reach further than the code that directly uses them, make it less suitable. Also the issues about the compiler warnings and the namespaces make Crystal Space a bad candidate.

The unexplainable glitches in the lighting and shadowing of OpenSceneGraph give Ogre a slight advantage.

Development and Support

All engines are actively being developed, in development teams ranging from 3 to 12 people, and user communities of 50 to 350 users¹. The most recently published commercial applications use Ogre and OpenSceneGraph, of which Ogre is used for more games. When looking at the community each of them is quite good.

Efficiency

As mentioned in the beginning of this chapter, the testing applications are not totally optimized. Scene graph optimizations and shaders are not used. Also, the Irrlicht application would not properly display on the measurement machine, therefore it was measured on another machine, and thus also has bad frame rates. The information can however still be used to give an idea of the influence of the view content on the frame rate.

1 Estimation based on forum users with more than 150 posts and chatroom activity

		<i>Ogre3D</i>	<i>OpenSceneGraph</i>	<i>Irrlicht</i>	<i>Crystal Space</i>
Graphics libraries		OpenGL, DirectX 9	OpenGL	OpenGL, DirectX 8, DirectX 9, Software	OpenGL
License		LGPL	OSGPL	zlib	LGPL
Coordinate system		Righthanded Y-up	Righthanded Z-up	Lefthanded Y-up	Lefthanded Y-up
3D model loading		Only own format	All kinds of formats	All kinds of formats	Only own format
Code quality		****	****	***	**
Shader support		Cg, GLSL, HLSL	GLSL	GLSL, HLSL	Cg
Wide usability		****	****	**	****
Documentation		***	**	**	**
Tutorials		***	**	**	*
Chatroom/Mailing list/forum		+ / - / +	- / + / +	- / - / +	+ / + / +
Commercially used		yes	yes	yes	yes
Total library DLL size		87 MB	65.9 MB	18.8 MB	64.3 MB
Frame rates ²	Best	23.25	48	9.8 ³	42
	Moving objects	22.8	45	8.4	33
	Static objects	3.76	2.14	1.0	3.0
	Whole scene	3.75	2.13	1.0	2.9

Table 1: Comparison of the four game engines

Conclusion

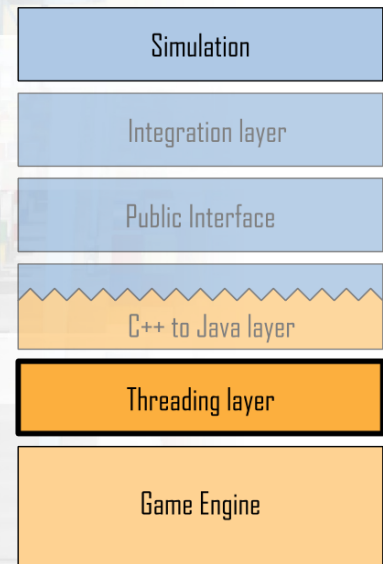
From this comparison we can conclude that Ogre and OpenSceneGraph are more useful than Irrlicht and Crystal Space for the purposes of the project. The slightly better tutorials and documentation, the broader shader support, the higher popularity for game development and the absence of lighting problems give Ogre a small advantage, and therefore Ogre3D is the Open Source engine of choice for this thesis.

² Framerates are measured on the Normal Windows testing system (page 56)

³ Measured on the Old Windows testing system (page 56), because it would not run properly on the measuring computer.

Chapter 3

Single- or multi-threading



For both simulations and games, the core of the program is a loop that makes things happen. In DEVS simulations this loop traverses the event list, in games it is responsible for consecutively updating the positions of players, doing some Artificial Intelligence, and rendering the new situation to the screen. These two loops can be seen in Figure 15.

When trying to combine the techniques of a game engine and a simulation, these loops provide a challenge. Either the two loops need to be combined into one big loop, or they should be created in separate threads. In this chapter, these two options will be shortly discussed, and then the details of the choice that was made for this thesis will be discussed.

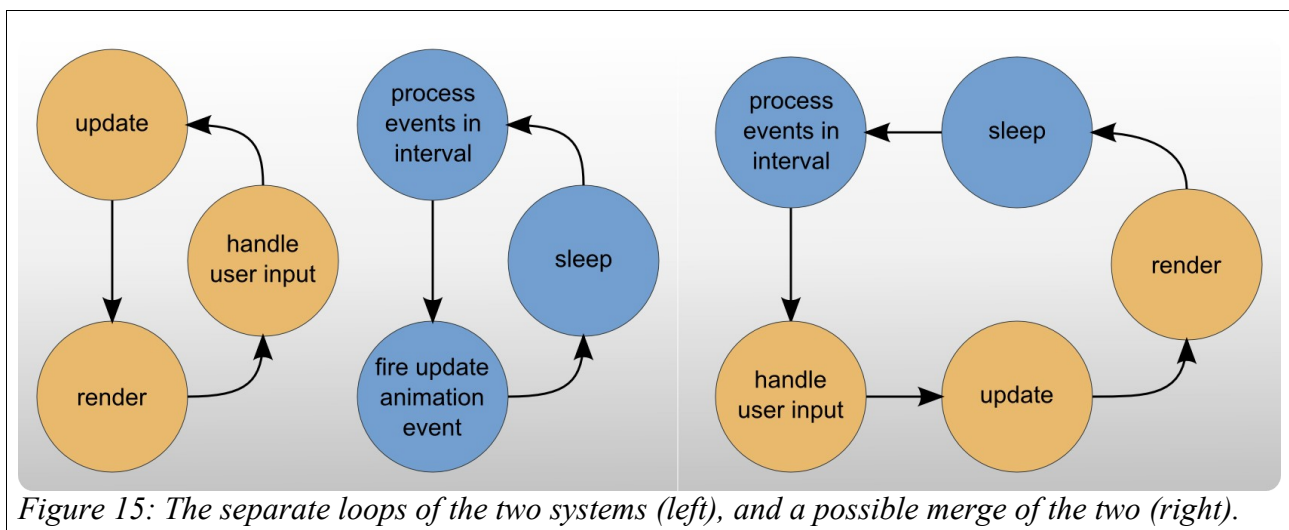


Figure 15: The separate loops of the two systems (left), and a possible merge of the two (right).

Single loop

It is possible to combine the two loops into one. This is a simple solution to implement. The control flow in the program remains understandable, and easily debuggable. However, there are also drawbacks. If the 3D animation is slow, it will inherently slow down simulation, because the rendering needs to be finished before the next simulation step can be executed. An other drawback is that it does not profit from multi-core CPU's, which by now have become common.

Multiple loops

The other option, is to have both loops run in separate threads. On the one hand, it makes implementation and debugging a lot more complex. *Mutexes* and *barriers* will be needed to synchronize the threads. Segmentation faults and other hard to track errors will occur during the development, as result of small mistakes in the synchronization, or in the memory allocation of the separate threads.

On the other hand, this solution will keep more of the original structure of both Ogre and the simulation intact. It will also enforce a design in which the data storage of the threads is separated, making it easier to adapt it for use on separate computers. It will also use multi-core CPU's more efficiently, and if the visualization appears to be a heavy task, that will hardly effect simulation speed, at least on a multi-core CPU. If it will be run on a single-core CPU, the code for thread synchronization will provide overhead, and most of the advantages will disappear.

Based on the current trends in hardware design and software engineering, it can be concluded that this is the way to go.

Implementation

The implementation process was tedious. For most of the projects where Ogre is used, Ogre's main loop is the main thread, and extra threads are used as helpers. But in this case, Ogre's main loop was to become a helper thread, which could be controlled by the *GUI* and the simulation thread. There were hardly any people in the Ogre community that could share any experience on the topic. During the implementation several challenges were met.

The simulation needed a single entry point to which the updates would be sent. Otherwise a multi-threaded simulation could be sending updates to Ogre in one thread, while it would have already finalized Ogre in the other, leading to segmentation faults. Because the root of the Ogre engine is a *Singleton*, and there can also only be a single entry point responsible for receiving updates from the simulation, it is decided that the single entry point will act as a *facade* and also be a *Singleton*. This entry point is called `Visualization`.

One of the toughest challenges was that the thread that initializes Ogre and the OpenGL render system, was also the only thread that could destruct it at the end. If any other thread would attempt to do that, this would give segmentation faults. To solve this, a renderthread-lifetime was defined, in which at the beginning an Ogre instance would be created, and at the end it would be destroyed. Through message passing the `Visualization` would communicate with the `RenderThread`, and signal it when to stop. If the renderwindow would be closed by pressing the X button, the `RenderThread` passes a message to the `Visualization` to notify it.

Another goal was that the updates from the simulation should only come during the update phase of the main loop. If it would be rendered while being updated, it would give segmentation faults. To solve this, the message passing system also involves a queue, which stores all the messages, and handles them when appropriate.

Some of Ogre's scene management features require a rendering context, generally meaning that a

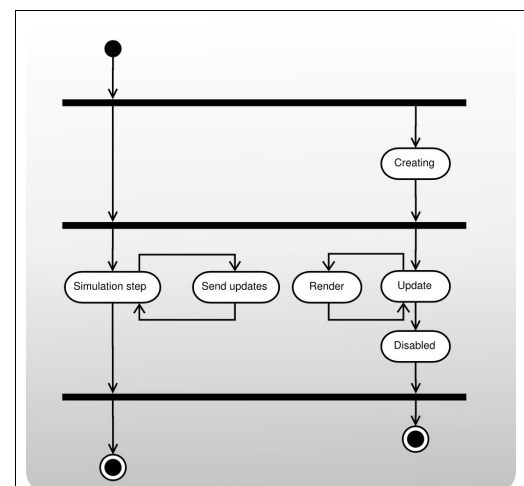
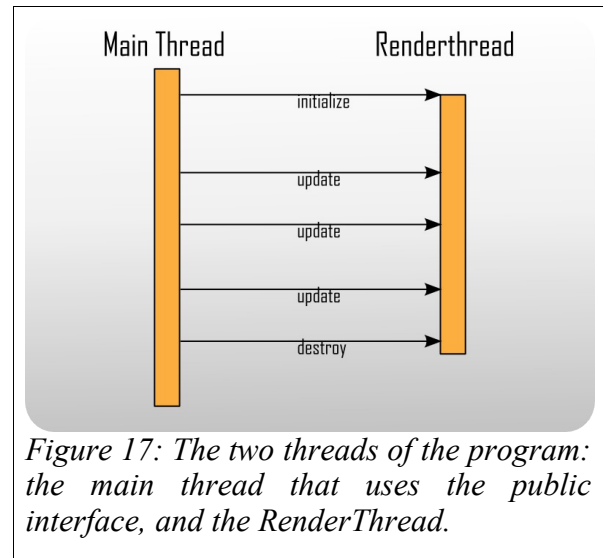


Figure 16: The thread synchronization between the main thread (left) and the RenderThread (right)

window should be open [Junker, 2006]. This is caused by materials and lighting effects using shaders: if no rendering context exists, these cannot be initialized. This requirement has some serious consequences for the usability. As long as no window is open, it is also impossible to add objects or set properties. The easiest solution to solve this was to create a window when the visualization is started, and to close it when it is finished. Maybe a nicer solution could be achieved, which creates a hidden rendering context and would allow one or more windows to be opened on demand. This is worth investigating, but for the scope of the thesis the simple solution will do.

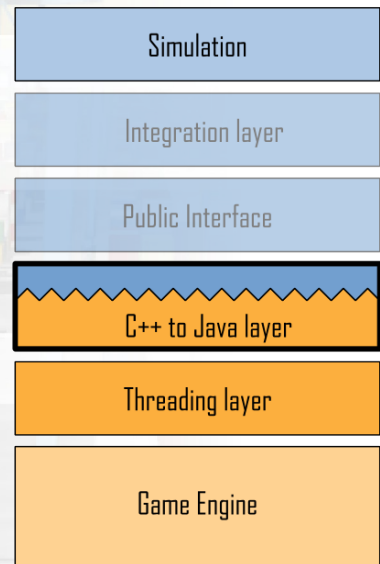
The program until now is like shown in Figure 17. There is a main thread, consisting of the main program and the `Visualization`, and a `RenderThread` which creates, uses and destroys Ogre. The main thread queues the update information, and when the `RenderThread` is in the update state, it processes all the updates. The `Visualization` can ask the `RenderThread` if it is still active. As soon as the `RenderThread` is not active any more because the X button of the render window has been pressed, the `Visualization` does not queue the updates any more.

This approach is relatively simple, but also a bit limiting. Because there is no window management, there are no options to re-open the render window after it has been closed, or to have multiple render windows. For the scope of the thesis however it is good enough.



Chapter 4

Interfacing C++ code to Java



The DSOL simulation library is written entirely in Java. Java programs are cross-platform, because they do not run directly on the operating system, but they run in a virtual machine instead. For the most common platforms, a virtual machine has been developed which provides the working environment that Java programs need.

Due to the required direct access to graphics cards, the main programming language used for games is C++. Almost all commercial games use C++ for the graphics rendering. C++ is compiled to *native code*, meaning machine code for a specific computer type and operating system. The native code is thus not cross-platform. Programs can be developed cross-platform using C++, meaning that the C++ code can be compiled to different platforms.

In this research, Ogre3D is used as game library. Because it is a native library, an interface needs to be established between the native code and Java code. There are several ways to do that.

Kinds of interfaces between Java and native code

In Sun's *The Java Native Interface*, an overview is given of three main ways how Java and native code can work together. After the writing of that book some more options arose, each with its own advantages and disadvantages.

TCP/IP connection

Two separate programs can be run, one in Java, one in native code. All the communication between the processes is done in TCP messages. This has the main advantage that the two do not necessarily need to run on the same computer. Also, this gives a really loose coupling between the two: as long as the communication interface is well-defined, either part of the application can be replaced.

Database

If the communication between two processes mainly consists of data, it can be stored in a database. Both Java and C++ have good libraries to communicate with databases. This method has the same advantages as the TCP/IP connection. An extra advantage is that if the database is the only form of data sharing, the database itself already takes care of some of the race hazards that can occur in

multi-threaded programs.

JNI

JNI stands for Java Native Interface, and is Sun's API to let Java and native code interact. This can be in either way: a Java program calling native methods, or a native program calling Java methods. A utility program is used to generate the required C header files from a Java class, so that the C++ programmer can implement that interface. The JNI generator however can only generate methods and variables. To make objects that consist of C++ and Java code, a lot of extra effort is involved, including calling C++ constructors from Java constructors, and holding a Java variable as pointer to the corresponding C++ class instance. This is a source of bugs and memory leaks.

JNI wrappers

Through the years, several JNI wrappers have been developed, which mainly are java libraries and utilities that generate Java code around native classes. These can prevent many of the memory leaks and other bugs that often come with JNI programs. One of them is SWIG (Simple Wrapper Interface Generator), which can be used to generate wrappers for C and C++ to all important scripting and programming languages, including Java. A configuration file is needed, which specifies the C++ functions that need to be accessed through the wrapper code. Then SWIG generates the C++ code to wrap around these functions, and the Java code to wrap around the generated C++ code. C++ classes are wrapped by Java classes, in such a way that when the Java class is created the corresponding C++ class is created, and when the Java class is garbage-collected, the corresponding C++ class is also deleted. If for some reason it is needed to delete the C++ object earlier, this can also be done explicitly from Java.

JNA

JNA is a Java library which instead of statically linking and compiling, dynamically locates the stubs of the native library and determines the right conversions. The programmer is required to program Java stubs, in order to compile the program. JNA will then automatically link these to the native library at runtime. JNA handles many of the difficult translations between Java and C++ structures, like strings, pointers and function pointers. For wrapping C++ classes, extra effort is required, just as with JNI. Also, JNA is a little slower than JNI, because the linking to the library is done dynamically.

Comparison of the interfaces for this thesis

For this thesis the purpose is to find out the usefulness of game graphics libraries for visualization of simulations. Even though the library that is developed in this research might not yet have all the functionality that is required, it should at least by its design allow a later addition of the extra functionality.

For commercial software, like simulation programs, it can be useful if the 3D view can be integrated into the existing GUI. This is one of those features that might not be directly necessary for the thesis, but is necessary for commercial usefulness of the library. To achieve this feature, JNI, JNA or one of the JNI wrappers will be required.

Writing the wrapping code yourself, be it for both the Java and C++ sides or just for Java, is a tedious job which requires expertise. Both JNI and JNA work easily with C, but to use the Object-oriented structures of C++, extra effort is required. There is no easy way to use classes. Also, every change in the interface to the C++ code requires quite complex programming, for JNA only on the

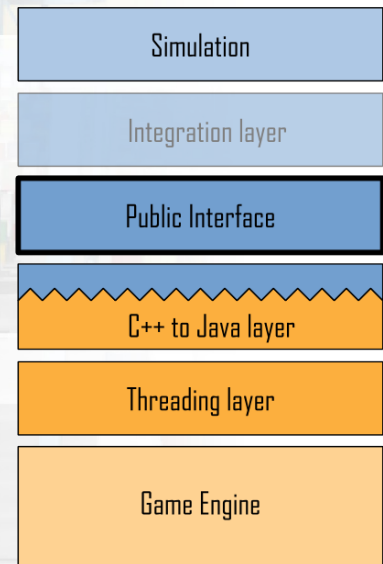
Java side, for JNI also on the C++ side.

Now the only option that remains, is using a wrapper generator, like SWIG. If the interface changes, automatically new Java files are generated. If the automated build system is configured correctly, it can even produce a new *jar file* after every interface-change.

SWIG has an extra advantage above other interface generating tools. It not only creates Java interfaces, but can also create interfaces to Python, Perl, Ruby and other languages. This increases the possibilities of using the visualization with other simulations.

Chapter 5

Public interface



Now it is time to describe the interface through which the visualization will be used. For the interface several requirements were stated. The interface should hide implementation details, and using it should not take much effort or require expertise in computer graphics⁴. The interface should also be usable for all kinds of simulations, as long as they concern objects that move. This chapter will first discuss different ways of configuration. Then the C++ interface is discussed, and its wrapper functions that are generated by SWIG, and finally a description is given of some Java classes that make the SWIG interface more friendly to Java developers.

Configuration

Basically, visualization is about mapping simulation data to a visual representation. Based on the different uses of simulation, a visualization is required to either look realistic and impressive, or show the data that is important.

To create this mapping, there are two main options: Doing the mapping through code, or in configuration files. To achieve subgoal 2 about ease of use, the mapping should be done in configuration files as much as possible. Subgoal 1 requires that this mapping should use understandable names, and have well-chosen default values.

This choice being made, there is a new choice to be made: implementing the configuration in Java, or in C++. Implementing it in Java is easier, but reduces the reusability of the visualization, because any non-Java simulation would not have the configuration. The other option is to implement it in C++. This makes it more difficult to implement, but provides this configuration functionality to any simulation. To make a good choice, first the requirements of the ideal configuration structure should be stated.

Firstly, it would have a list of basic building blocks, from which the visualization can be constructed. Each building block would have its own type of appearance. For example a Model building block would be used to display a 3D model, and an Icon building block would display an Icon on a *Billboard*. The position, rotation, size and appearance properties of these objects could be configured, and changed during the simulation.

A configuration file would define the mapping of the simulation object's properties to one or more

⁴ These subgoals can be found on page 5 and following

of these building blocks. For example a truck could be shown using a 3D model of the truck, with a coffee break icon appearing above it when the truck driver has a break. Every time a property of a simulation object changes, this should be translated to visualization updates and then these updates should be processed.

In Java, it would be possible to use reflection to request the properties of the simulation object. This would be a clean solution, requiring just one class that reads the configuration file and creates, configures and updates the building blocks. Considering the simulation complexity of Controls2, the performance penalty of using reflection makes it a less feasible solution.

Because of this, and the limited time for this thesis, the decision is made to create an interface that would support a C++-based configuration method, but to only implement the basic building blocks. From Java these building blocks can be assembled and used. The C++-based configuration can still be added later on, if the functionality is required.

The interface generated by SWIG

The first interface is the one that is made in C++. The generated classes in Java and any other language for which SWIG works, closely resemble this interface. First this interface will be explained.

In C++, an inheritance structure for the building blocks is created. The base node is called `Entity`. This has child nodes that add more specific functionality. The `SceneNode` converts the position, rotation and scale properties to the corresponding properties in the Ogre scene graph. The `Model` and `Icon` are subclasses of `SceneNode`, so that they at least have the positioning properties, but also have extra functionality for selecting a 3d model or an image. `SceneNodeS` can also be added as child to other `SceneNodeS`, so that more complex shapes can be assembled from them.

Due to the multi threaded approach, no direct references from the simulation to the Entities are allowed. Instead, names are used as unique identifier of the building blocks. These are packed into a message by the `Visualization` object, and delivered during the update phase. The functions that are responsible for that are shown in Listing 4.

```
Visualization.createEntity(name : String, type : String)
Visualization.setProperty(name : String, property : String, value : Vector)
Visualization.setProperty(name : String, property : String, value : float)
Visualization.setProperty(name : String, property : String, value : boolean)
Visualization.setProperty(name : String, property : String, value : String)
Visualization.destroyEntity(name : String)
```

Listing 4: The public interface of the Visualization

The `setProperty` methods allow to set properties like the model name or the material. This structure also allows configurable objects, like proposed earlier. In that case, the type in `createEntity` would refer to a configuration file. The properties for `setProperty` would be the name of the simulation object's properties, and then in the C++ file these would be translated to setting properties of the building blocks.

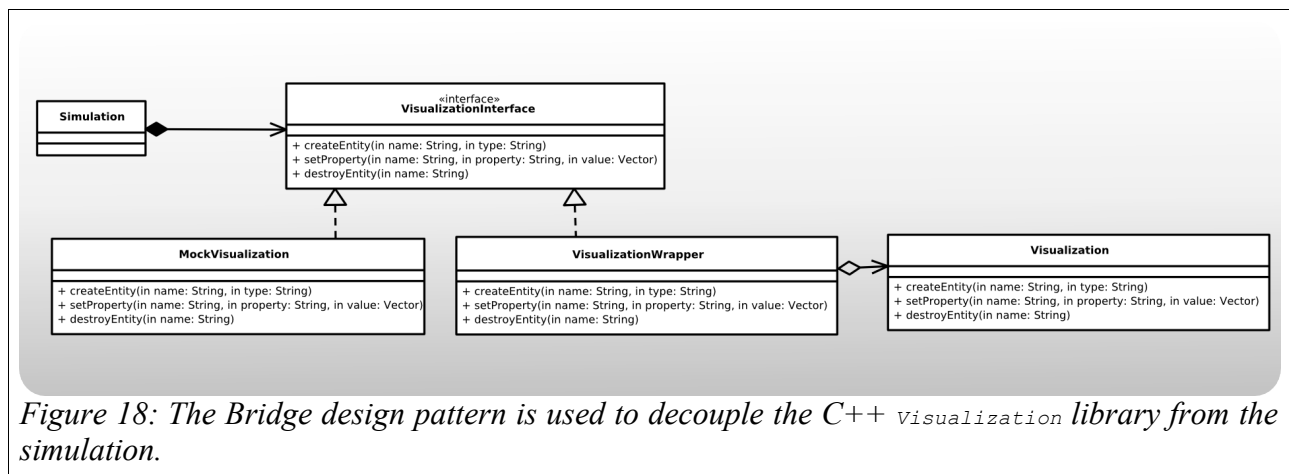
In this way, an interface is provided that is easy to understand. By not using direct references from the simulation to the visualization primitives, the design of the code remains simple, and the interface can also easily be wrapped by a TCP/IP messaging system, to also support remote visualization.

The interface for general use

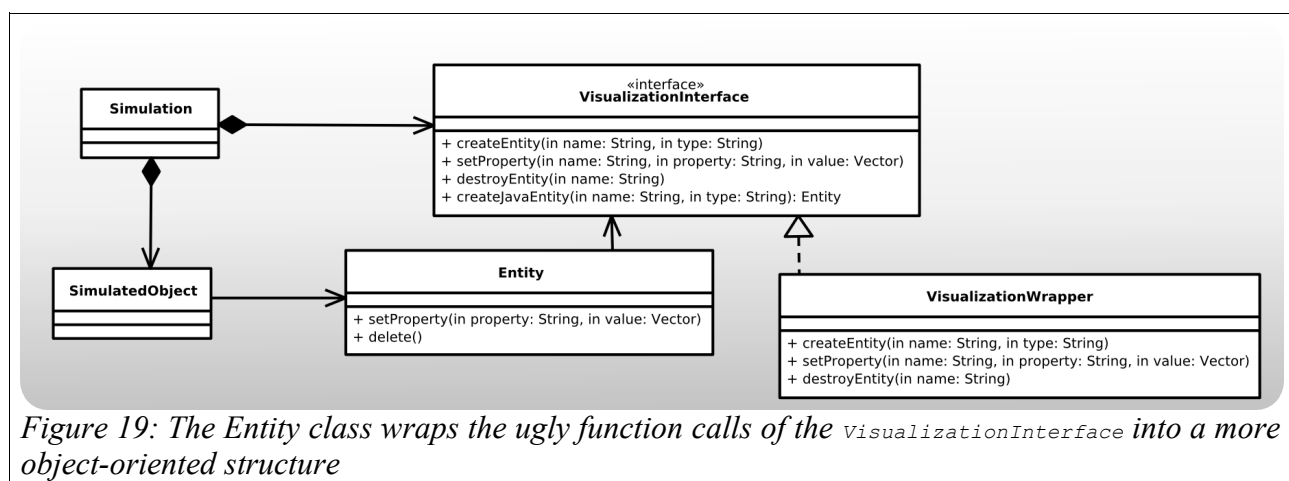
Although the interface is simple, it is quite ugly from a software engineering point of view. Calling the static methods makes the `Visualization` hard to replace by either a *Mock object* (for testing purposes) or a new implementation, because calls to the methods will be littered through all of the code. Also, it will become quite annoying to keep on repeating the name of the object when changing its properties.

The solution to the first problem, is to decouple the `Visualization` using the *Bridge* pattern. All the objects in the simulation will refer to a `VisualizationInterface` instead of to the `Visualization` itself. At only one place in the code it needs to be defined what kind of `Visualization` implementation actually is used. This is illustrated in Figure 18. If the real `Visualization` is not used, the DLL file will not even be loaded.

Even though this simplifies the design in one area, it makes it more complex in other areas. Now there is no general way to access the `Visualization` instance, because it is not static, nor singleton any more. Because of this, the `Visualization` object needs to be passed as a parameter to any object that creates, updates or destroys any shape in the visualization.

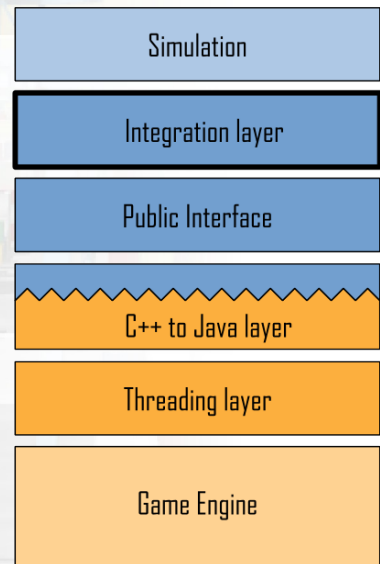


This problem, and the annoyance about repeating the name every time, can be solved by adding a wrapper class for the Entity. The Entity instances are created by a factory method of the `VisualizationInterface`, and hide almost all of the notion of `VisualizationInterface` from the user. The Entity itself can be told which properties need to be set. This solution is shown in Figure 19.



Chapter 6

Integrating Simulation and Visualization



Now an easy to use interface has been provided to Java. However, this still needs to be linked to the two simulations of choice. In this chapter, the design is described of how each simulation accesses and uses the features of the visualization.

DSOL implementation

The internal code of DSOL was quite badly documented. The DSOL library and GUI work as an event based system, but there was no clear documentation on which events the GUI should react upon. Therefore the implementation of the 3D animation is mainly based on the existing 2D and 3D animations.

For the Java3D visualization, the users were supposed to program their own `Renderable3D` classes, that would create and update the Java3D scene graph objects. The `Renderable3D` class could be used to visualize any data, as long as it would have a location and an orientation. Because that approach is against the subgoals 1 and 2, a different approach was used. A general `Renderable3D` class is provided, that can be used to visualize either `Model` or `Icon` types. If the C++ configuration is implemented, this `Renderable3D` will need little modification to also support these.

An encountered problem is that the GUI allows closing and re-opening the rendering window, while the C++ code opens a window at the creation of a `Visualization`, and closes it at destruction. Therefore a workaround had to be implemented, which manages a list of `Renderable3D`'s, and creates the according `Entities` as soon as the window is opened. This workaround is far from ideal, so if the visualization is to be used in real applications, the one-window-solution should be replaced by some more complex window- and resource management.

Controls2 implementation

In contrast to DSOL, Controls2 has been designed in such a way that different visualizations can be added easily. Each type of visualization has to implement the `AbstractAnimation` interface, and reads its configuration from an XML file. The simulation tells the `Animation` which objects it should display, and based on the configuration some `Renderables` are created for them. At regular intervals, the `Animation` tells all its `Renderables` to update themselves, after which each `Renderable` queries its simulation counterpart, and updates its appearance.

Configuration system

Because the use of *reflection* for Controls2 is discouraged, it was necessary to implement a *Renderable* for each type of simulation object that had different properties to visualize. Luckily, for most objects the only properties that needed to be updated were the location and orientation. For these a *LocatableRenderable3D* would suffice, which would request the location and orientation of the simulation objects, and send these updates to the visualization. For some objects, the *Renderable* itself is just a *LocatableRenderable3D*, but the initial configuration is a bit more complex. For example, for containers there are different colors, referring to different shipping lines. For every shipping line there is a texture that needs to be applied. However once the object is created, only the position and orientation of the container will change, so a *LocatableRenderable3D* suffices. To allow this initial configuration, the *Factory* pattern was used.

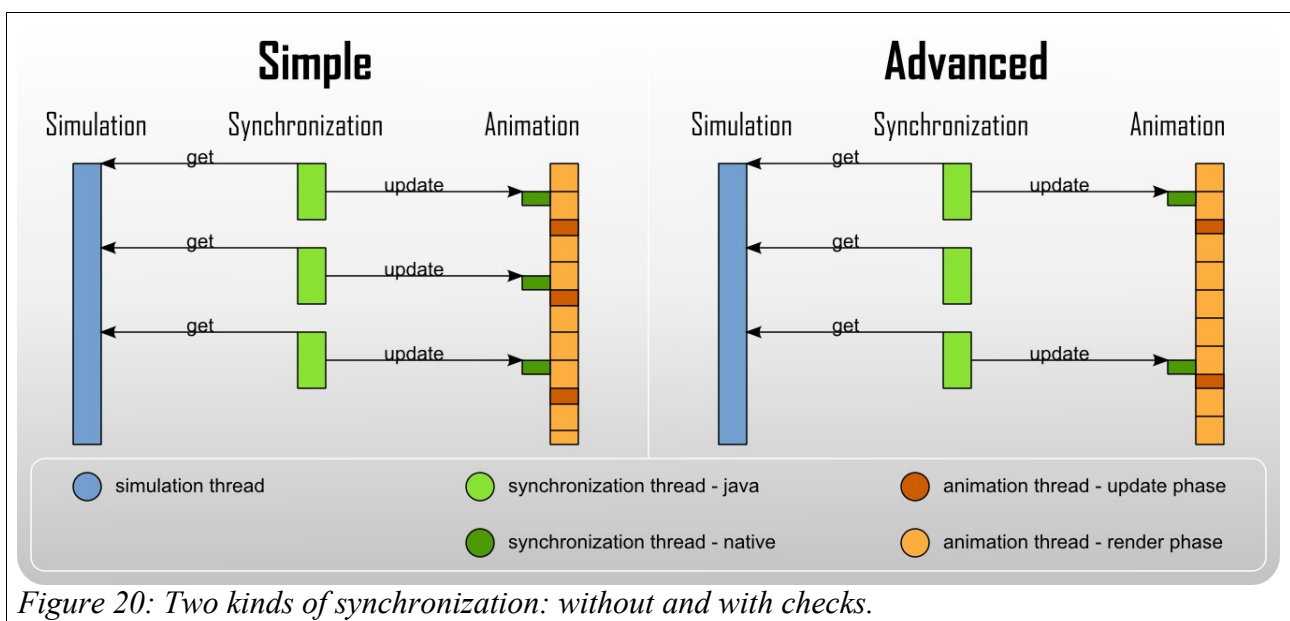
In Controls2 an XML file is used to configure the animations. In the configuration part for Ogre3d, a mapping can be defined between Java objects and the factory that should create the *Renderable3D*'s. In this part, extra parameters for the factory can be defined. For example, the factory for containers gets a list of the different shipping line textures that it can apply to the containers.

This results in a quite clear configuration structure, with a few general factory types, and even fewer *Renderable* types. Programming new classes is only required if an object has other properties that also need to be visualized, or when totally different options should be configured.

Simulation and Visualization synchronization

In Controls2, an interface already exists, which decouples the animation from the simulation. The simulation runs in one thread, and a separate thread is responsible for asking the simulation for data and updating the animation. These updates are done at a fixed interval. This pull-based solution is designed in this way, to prevent the visualization part from updating more frequently than is necessary. It also makes sure that the updates in visualization data (which were quite expensive for some structures like the containerstacks in Java3D), were not done by the simulation thread.

To make the Ogre visualization easy to use, it has a push-based solution: It does not ask for data updates, because that would require knowledge of the simulation. Instead, it waits till the simulation notifies it of new updates. They need to be pushed to the visualization. When combining this with the synchronization thread that Controls2 has, this results in a synchronization thread that asks for updates at fixed intervals, and pushes them to the visualization.



However, this simple solution might not be the best. It just asks for the data and sends that as an update, even if the data is not changed since the last update. A little more advanced solution is to check if the data is changed, and to only send an update to Ogre if that happens to be the case. This gives more complexity in the synchronization thread, but takes load from the Java-Native interface and the update phase in Ogre. These two options are shown in Figure 20. To make a good choice between these two options, the efficiency of each of them should be measured.

Experiment setup

The choice for a solution will influence both sides, the simulation and the animation. Therefore the influence on both sides needs to be measured. The frame rate is a good measure for the efficiency at the animation side, because the longer the update phase takes, the more time there is between two frames. It is important that the camera does not move during the measurements, because the frame rate is also heavily influenced by the amount of objects that are in sight. For the simulation, the execution speed is a good efficiency measure. The execution speed is the factor between the time that is simulated and the time that the simulation takes. An execution speed of 10 means that simulating 10 minutes takes only one real minute. The reason why we might want to keep the synchronization simple, is so that more CPU time is left for the simulation, so that the simulation can run faster. So if the achieved execution speed is higher, this means that the solution is a better one.

There are several factors that influence the efficiency. First there is the execution speed, specifying how fast the simulation runs, and thus also how many computation is being done. Second, there is the update frequency. This is the rate at which the synchronization will send the updates to the animation. The higher the update frequency, the more work the synchronization thread will do, and the clearer the effect of the different solutions will be. On the other hand, setting it too high might give too exaggerated results. Note that an update interval can not be used to calculate the update frequency, because the duration of the updates should first be added to the interval time.

For the measurements a Demo run will be used. The other options, namely an animation replay run or an emulation run would not be useful. The first does not use the simulation, and would therefore not tell anything about the CPU usage of the simulation, the second would limit the execution speed, because a Terminal Operating System would need to be connected, and these have limitations on the execution speed.

The measurements will be done on the Normal Linux testing system⁵.

Experiment results

The first measurements will be the frame rates. For the frame rates measurements were done with different execution speeds, and with an update interval of 10 ms. Although this is quite short, it does help to see the influence of the synchronization. Currently with the Java3D visualization an update interval of 100 ms is used, so there will also be some measurements with this update rate.

<i>solution</i>	<i>Simple</i>				<i>Advanced</i>			
<i>Exec. speed</i>	1x	4x	8x	Fast as possible	1x	4x	8x	Fast as possible
<i>Framerate</i>	33	28	25	22	43	40	35	35

Table 2: Frame rates at different execution speeds, and an update interval of 10 ms.

⁵ See appendix C

<i>solution</i>	<i>Simple</i>	<i>Advanced</i>
Execution speed	Fast as possible	Fast as possible
Framerate	33	35

Table 3: Frame rates at maximum execution speed, with an update interval of 100 ms.

From these measurements it can be seen that for the advanced solution, the amount of updates of the synchronization does not really matter at maximum execution speed. If the execution speed is lower, the frame rate increases quite much. This means that with this solution the bottleneck is still the update rate of the simulation. For the simple solution, it can be seen that both the execution speed and the synchronization update rate heavily influence the frame rate. This means that the simple solution slows down the rendering a lot, and probably also takes more CPU time. To more accurately investigate that, we need to measure the execution speed.

The maximum achievable execution speed is directly dependent on the CPU usage of other threads, like the synchronization and rendering threads. When a simulation runs as fast as possible, the execution speed often has a peak in the first 10 seconds of execution, after which it slowly declines. Three points for measurement were chosen: the peak, the speed after 1 minute, and the speed at the end. Runs of one and of three hours were done. Here also there are runs with an interval of 10ms and of 100ms.

<i>solution</i>	<i>Simple</i>			<i>Advanced</i>		
Simulated time / intervaltime	1 hr/10ms	3 hrs/10ms	3 hrs/100ms	1 hr/10ms	3 hrs/10ms	3 hrs/100ms
Peak speed	19.3	20	22.2	23	21	24.1
After 1 min	17.34	18	20.3	20.5	20	21
At end	16.5	16.5	19.4	20	19.5	21

Table 4: Execution speeds with different setup

In these results we can see that the execution speed has a small decline with the simple solution, and an even smaller decline with the advanced solution. This decline is probably caused by the containers that are spawned during the simulation. Because the containers on the stack do not move any more, they are a source of unneeded updates that make the advanced solution better than the simple one.

Conclusion

From the measurements it can be concluded the advanced synchronization thread is a better solution. It is better to add some logic on the Java side of the program, be it in a separate thread, if that can reduce the amount of calls to native code that need to be made.

Static and Dynamic objects

Even though the CPU usage can be reduced by a more advanced synchronization system, a more basic optimization can be done. This is by discerning between static and dynamic objects. Static objects are objects that will not move. For these objects, the animation also does not need to be updated. This can save CPU time for the synchronization thread, but also save memory usage. If the simulated object does not change, the Renderable itself becomes useless, because its sole task is to update the changed properties.

In the Ogre implementation, the Animation object holds registry of static objects, which are pairs of a simulation object and its unique name in the Visualization. It also holds a registry of dynamic

objects, which are pairs of a simulation object and its corresponding Renderable. Only the dynamic objects are updated. Dynamic and static objects can be converted by the factory that created them. In this way, the settings from the configuration file can still be respected.



Chapter 7

Optimizations and container stacks

For game engines like Ogre there are a lot of different ways in which the rendering can be optimized. Each way has its own pro's and con's, and its own area of use. Until now, the scenes that have been visualized have been quite simple, and therefore not much attention has been paid to these optimizations. The defaults were used instead.

But since the visualization has been connected to the Controls2 simulation, the visualization does not run fluently any more. During implementation it was discovered that this is mainly caused by the container stacks of Controls2. A container stack can contain hundreds of containers, and tens of stacks are placed on a terminal. The terminal that is used for the demo contains 60 stacks, with in total about 20 000 containers. Some other terminals are even bigger, with 100 stacks of about 700 containers each. The sheer amount of containers itself makes it a challenge to render it efficiently. Having to draw all these objects individually is a heavy task for a computer. Now the stacks have been added, the frame rates for the demo model have dropped below 1 frame per second. Now the scene is complex enough to be able to evaluate the alternative options for optimization.

There are two kinds of optimizations, those that change the appearance, but in a hardly noticeable way, and those that change the way the rendering is approached. Examples of the first kind of optimization are *LOD* techniques and *impostors*. To display the container stacks in Java3D, a special model was generated that would take the shape of the whole stack. This is also an example of the first kind. The second kind consists of grouping objects in scene graph nodes, grouping them by material, implementing the scene graph traversal on the GPU. Especially techniques that use the GPU efficiently provide good speedup, but in the same time limit the usability to computers with modern graphics cards.

In this chapter, techniques from both categories will be evaluated, using the Controls2 demo model. The goal is to achieve an acceptable frame rate on the testing computer. If techniques are chosen that will not run on older computers, it is needed to select alternatives, or disable them. Techniques that are not usable as generic solution, like the generating of a special stack shape as was done in [Bijl, Fresen, 2006], will be avoided as much as possible, because that would not facilitate achieving the goal of re-usability. First it will be measured how much improvements there can be made by different material settings. Then some different techniques in Ogre itself will be measured.



Figure 21: The container stacks in Java3D and Ogre3D

Material optimization

At the time of writing, the implementation of the container textures uses two textures for a container. One texture is a black-and-white texture that shows the details of the container shape, the other is a colored texture that colors the container and shows the shipping line logo. These two are blended at runtime. This design was chosen because it makes it extremely easy to add new types of shipping line logos. It only involves drawing a new colored texture (based on a template), and adding 3 lines to the material file, and adding the shipping line to the configuration of the Controls2 container.

There however are also other ways to render the containers. A material with only one texture would save the texture-combining, and a material without any texture would save texturing at all. The first one could be used instead of the current technique, but the second one could not because it would sacrifice the realism. The second one however could be used as a lower-level material for a *material LOD*.

Something else that can be varied, is the type of texture filtering. Of a texture, several *mipmaps* are made, which are used as texture for objects that are further away. There are several ways in which these textures can be rendered. In Ogre the choice is between no filtering, bilinear, trilinear and anisotropic.



Figure 22: Artifacts with filtering techniques. The difference between bilinear and trilinear filtering (left) and the moire patterns when no filtering is done (right)

The difference between these filtering techniques can be seen, if you know where to look. In the bilinear approach, a seam can be seen between two levels of mipmaps. With the trilinear and

anisotropic techniques this is not visible because the two levels of mipmaps are blended. When no filtering is used at all, *moiré patterns* can be seen. These are shown in Figure 22.

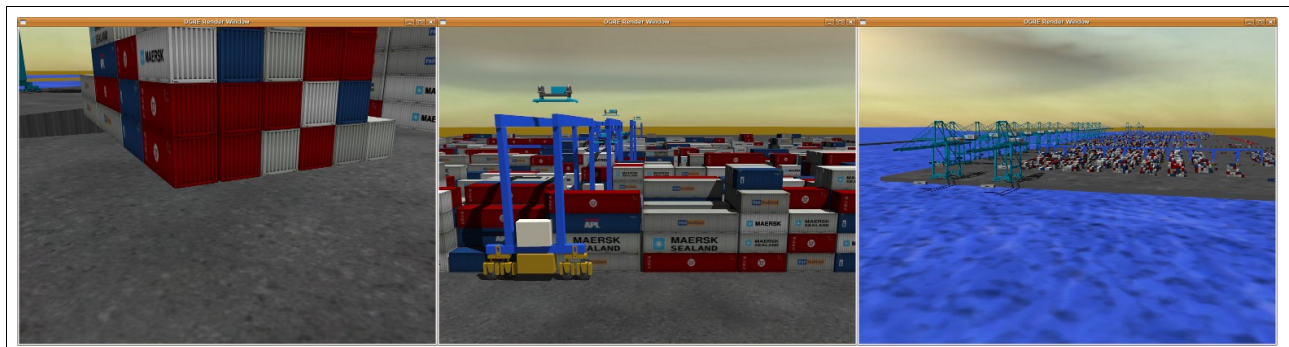


Figure 23: Three camera positions, ranging from Close by to Far off

Test setup

To investigate the usefulness of these alternatives, several tests with the demo model will be run. For these runs, the frame rate (the rate at which the animation is rendered) will be measured. Because the frame rate is influenced by the update rate (the rate at which the updates are sent to the animation), this will also be measured. However, no big differences in the update rate are expected.

A replay of an old simulation will be run, so that the simulation work will interfere as little as possible with the rendering speed. The Java3D animation will be turned off. The rendering will be done on a fullscreen window (1920x1200), to maximize the effect of the selected technique in the scene. The measurements will be done at several different camera positions (Figure 23), which vary in scene complexity. Because the frame rate fluctuates over time, it will be averaged over a minute.

Test results

<i>Material technique</i>	<i>Close by</i>	<i>Mid distance</i>	<i>Far off</i>
No texture	9.1	4.2	1.2
Single texture, anisotropic	9.1	3.9	1.3
Two textures, no filtering	8.7	3.5	1.4
Two textures, bilinear	8.4	3.2	1.3
Two textures, trilinear	8.1	3.6	1.3

Table 5: Measured framerate (in frames per second) for different material settings for containers

For comparison, some of these tests were also done on the slow test system. Because it is just a single core, the animation is not even being replayed, to make sure the influence of the technique can be significantly measured. Only the mid distance measurements are done, because the frame rates were so slow that it was hard to position the camera.

<i>Material technique</i>	<i>Mid distance</i>
No texture	0.46
Single texture, anisotropic	0.35
Two textures, no filtering	0.48
Two textures, anisotropic	0.35

Table 6: Frame rates with different filtering settings on the slow computer

Conclusion

Both using no texture and using no filtering achieve some frame rate increase, but not enough to make up for the loss in realism. Therefore the most realistic material settings should be chosen, either with a single, or multiple textures.

Object batching

A technique that requires some changes in the code, is Object Batching. The work it requires is mainly to allow scene nodes to be “batched” and “unbatched”. First the technique itself and its background will be explained in more detail.

When objects are rendered using a scene graph, often a fixed series of steps is done. First the transformation is applied, then the material is selected. After that the polygons using the material are drawn. If the object contains multiple materials, the next material is selected, and the polygons for that material are drawn.

There are several big factors in this process, that slow down rendering. The first is that all the objects need to be transformed separately. A simple optimization to that is done by the tree structure of a scene graph. A better optimization can be done if objects don't move. In that case, it is possible to transform all the vertices at the loading of the model, and then never touch them again. This saves one transformation for each of these static objects.

The second great slowdown factor is the material changes, which is often more generically referred to as *render state changes*. After every object that is rendered, or sometimes multiple times during the rendering of an object, different surface rendering information needs to be sent to the GPU. Because GPUs are SIMD units, they are especially built for applying a single technique to loads of data. So if the data can be sorted by material, and then sent to the GPU, this can save thousands of render state changes, and will use the GPU more in the way it is intended to be used.

The third slowdown factor is the bandwidth between CPU and GPU. The more data needs to be transmitted, the slower the application might get. However if a model can be sent to the GPU once and then be reused several times, that saves bandwidth.

The amount of improvement that can be achieved using these optimizations heavily depends on the scene that is to be rendered and the computer on which it is done. If the bottleneck is the CPU, optimization can be achieved by sending more work to the GPU. If the scene contains hundreds of different materials, optimizing the bandwidth usage might not help, but restructuring the materials will. In [Griffo, 2006] some tests with these differences are already done.

Ogre has two implementations of these optimizations. **Static Geometry** is done on the CPU. All it does, is sorting all the objects based on their material, and applying their transformations. These objects are then grouped into batches, based on position and material. The culling of objects that are out of view is done by checking the visibility of these regions. If a small part of the region is visible, the whole batch will be rendered. The batches that are visible, are sent to the GPU and

rendered. This technique uses more memory than having a normal scene graph, and still has to send all the polygon to the GPU each frame. The optimization is in that each render state only needs to be sent to the GPU once, and that all polygons using the same material can be rendered at once.

In contrast to static geometry, **Instanced Geometry** is done on the GPU. A template of the model is loaded to the GPU, and duplicated on many different positions. The main differences with static geometry are that it saves a lot of CPU work, bandwidth and memory usage. In theory this technique can also be used with skeletal animation and moving objects, making it the ideal solution for rendering big armies in strategy games. At the time of writing however, the implementation in Ogre contains bugs and can not be used.

Another issue concerning batching, is the question what to batch. If a container is added to the stack, the batches of the stack need to be rebuilt. This is a CPU-intensive task. It is possible to batch all the containers of a single stack into a group. This limits the amount of work needed when the batch is modified, but also reduces the amount of optimization because containers that are in different stacks are not batched together. The alternative is to let all the stacks be batched together in one big batch. This will mean that the modification time will be much longer, but the rendering will probably be faster.

Test setup

The two options are tested by running them in replay mode. Stencil shadows are used, and measurements are done from the same three camera viewpoints as the previous tests. Frame rate measurements are done when the simulation is not running, because the batch time has so much influence on the frame rate that the frame rates will not tell anything about the rendering speed any more.

Test results

<i>Batching technique</i>	<i>Close by</i>	<i>Mid distance</i>	<i>Far off</i>	<i>Update time</i>	<i>Memory usage</i>
Separate containers	7.0	4.3	1.4	0 s	1.0 GB
Static geometry – per stack	44	14	9	0.01 s	980 MB
Static geometry – one group	40	14	11	0.6 s	1.0 GB

Table 7: The consequences of the different batching setups for the rendering speed, the update time and the memory usage

Conclusions

The difference in rendering speed is between the two options with static geometry is hardly noticeable. Therefore the one with the best update time should be used. The static geometry clearly gives better frame rates than the normal scene graph, at a cost of a little longer update times. For normal simulation runs these update times are acceptable, but when a simulation replay is fast-forwarded they will severely influence the frame rate. For these cases it might be necessary to cache stack changes, so that a stack will only be updated after multiple containers have been added. The increase in memory usage is lost in the memory usage of the whole application, and can therefore be ignored.

It is too bad that the instancing could not be tested. It might be worth the effort to try and fix the bug.

Shadow techniques

Shadows have always been something difficult for *rasterization*-based rendering. The reason for that is that rasterization mainly works with the positions and orientations of polygons and lights, but for shadows also the positions of polygons in relation to other polygons need to be checked. Some workarounds have been found to create shadows. The first shadows in games were just a gray picture that would move along with the characters. Several of this kind of tricks have been used, but they were always really application-specific. There are two techniques that can be used for more general cases. These are called *stencil shadows* and *texture shadows*, and are both available in Ogre.

Stencil shadows are rendered in three steps. The first is that shadow volumes are calculated: 3D shapes that define the area that is in the shadow. Then the scene is rendered with the stencil objects, to see which parts are in the shadow. Finally the normal scene is rendered, and combined with the shadow render, to get the final image. The main disadvantages are that the CPU is needed to generate the volumes, leaving less CPU time for the rest of the software, and that the shape deformations done by vertex shaders are not known to the CPU, so the shadows will not follow them. It also does not support semi-transparent objects, or objects with a transparency texture applied to them.

Texture shadows are a newer technique, which needs less CPU power. First a depth render is done from the light position. This contains all the distances of objects to the light. Then a normal render is done, using an extra shader script. For each pixel the script renders, it calculates the distance from that pixel to the light. If that is more than the distance stored in the depth render, apparently something is closer to the light than this pixel, so this pixel is in the shadow. There are variations of this technique that produce shadows with soft edges. The two main

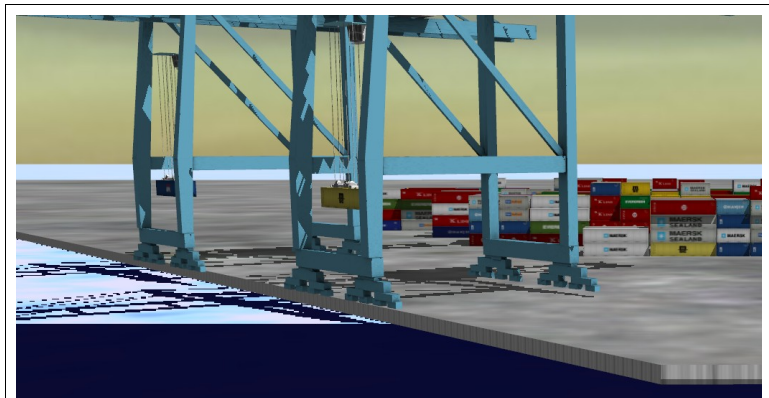


Figure 24: Texture shadow artifacts: jaggy pixel edges can be seen on the quay cranes, and the edge of the shadow texture can be seen on the water

disadvantages are that they can not be used on computers with older graphics cards, and that if the resolution of the depth map is configured incorrectly, pixels can be seen on the shadow (Figure 24). Texture shadows have an extra issue when using directional light (like sunlight). Where spotlights clearly define which area is to be lit, and thus also for which area the depth render should be done, directional lights do not. Therefore the most appropriate area needs to be chosen based on the light direction and the camera position. In contrast to stencil shadows texture shadows do allow semi transparency and transparency textures, albeit with a lower performance.

There is another distinction in the shadow techniques, mainly on how they are applied to the objects. There are *additive* and *modulative* shadows.

Modulative shadows assume that there is only one light, although they also work for multiple lights. The areas that are in the shadow are multiplied (modulated) with a shadow value, and thus get darkened.

Additive shadows work in the opposite way: An unlit material is assumed, and for every light the lighted areas are calculated. For the lighted areas, the light is added to the material. If extra lights are added, this makes the material brighter, just as in reality. With modulative shadows, adding extra lights does not actually add lights, but adds shadows and makes the shadows darker.

Test setup

The test setup uses three different camera positions, just like the material test. However this time the simulation is run as fast as possible, in order to detect the difference between the GPU-based and CPU-based shadow techniques.

Materials are rendered with the double trilinear texture filtering. For texture-additive shadows, a lot of fine-tuning has to be done, for choosing the area which is to be textured. This will probably influence the appearance a lot, but hardly improve the frame rate. The settings that are used for these measurements are mostly default settings, with an algorithm that is optimized for flat landscapes. If texture-additive will eventually be the technique of choice, the shadow technique should be configured better.

Test results

<i>Shadow technique</i>	<i>Close by</i>			<i>Mid distance</i>			<i>Far off</i>		
	<i>fps</i>	<i>upd</i>	<i>exec</i>	<i>fps</i>	<i>upd</i>	<i>exec</i>	<i>fps</i>	<i>upd</i>	<i>exec</i>
none	189	17	14	94	16	14	37.5	16	14
Stencil-modulative	9	16	11	5	16	12	3.8	15	11
Stencil-additive	9	16	9	5	13	8	3.5	13	9
Texture-additive	96	17	14	19	17	14	7.5	17	14

Table 8: measured speeds of the shadow techniques (fps, update/s, simulation speed)

Also for the slow computer an attempt was made to do some measurements, but with each of the shadow options the computer became extremely slow, and crashed. Without shadows there were measurements, but because the computer is a single-core, the influence of the simulation on the rendering speed was so big that the simulation had to be turned off to get results relevant for the rendering.

<i>Shadow technique</i>	<i>Close by</i>			<i>Mid distance</i>			<i>Far off</i>		
	<i>fps</i>	<i>upd</i>	<i>exec</i>	<i>fps</i>	<i>upd</i>	<i>exec</i>	<i>fps</i>	<i>upd</i>	<i>exec</i>
none	44			14			11		

Table 9: Frame rates of shadowing techniques on the slow testing computer

Conclusion

Using shadows significantly slows down the frame rate. On the other hand, they also significantly increase the realism. For non-fullscreen renders the stencil- and texture shadows are usable right away, if the computer is a recent one. For full-screen rendering or older computers an option should be provided to turn off shadows, in order to keep the frame rate acceptable.

It can clearly be seen from Table 8 that texture shadows hardly use the CPU, where the stencil shadows do, and that they also achieve better frame rates. This makes them the better shadow technique for this research. On the other hand, as long it still gives quite serious rendering artefacts, which makes the shadows look unrealistic and in some cases even disturbing, they are not considered useful for the simulation.

For now, the stencil-modulative shadows will be used, and the option of using no shadows at all will be provided as backup for old computers. If the solution is to be really used, the stencil shadows should be replaced by the configured texture shadows.



Chapter 8

Comparison of the visualizations

Through a long process of implementation and optimization, finally a solution has been created that might serve as a replacement for the Java3D visualization. In the first chapter, several subgoals were defined that the solution should meet in order to be a useful improvement of the current Java3D implementation. In this chapter these subgoals will be evaluated by comparing the Ogre visualization with the original Java3D implementation. Some of the subgoals are not specifically formulated as a comparison, but still a comparison would help evaluate if the subgoal is met or not.

The subgoals can be roughly divided into three categories: usability, appearance and performance. For each of these categories, the subgoals will be evaluated.

Usability

The following subgoals are about usability:

- sub1) Using the solution should not require specialized knowledge of computer graphics*
- sub2) The solution should require only a minimal amount of work to visualize a simulation.*

DSOL

In the Java3D implementation, the simulation engineer was forced to program his own `Renderable3D` class in order to visualize an object. This required knowledge of the Java3D API, and of the graphics related vocabulary. It also involved quite a lot of work for the programmer. On the other hand, the solution allowed for many different ways to visualize objects. Any property of the simulation object could be visualized in any way, because the programmer had to program it himself. For example, the speed or temperature of an object could also be shown by its color.

The Ogre implementation provides more or less the opposite: 3D models and 2D icons are currently the only way to visualize DSOL simulation objects. This restricts the ways in which objects can be displayed, but the simulation engineer only needs to add 1 line of code to each simulation object, and does not need to have any knowledge of computer graphics.

Controls2

In the Java3D visualization for Controls2 an extremely big library of Java classes was used, with at least one for each simulation object that needed to be visualized. The Java classes contained lots of Java3D code that builds scene graphs and transformation nodes, and was hard to read. If a new simulation object was created, a corresponding Renderable3D class was created to display it, often containing lots of copy-pasted code. This made the whole structure hard to maintain, and required specific 3D knowledge.

The Ogre version uses just a few factory classes, even less Renderable3D classes, and has most of the choices configurable in an XML file. For the simulation engineer, the main task is creating an extra entry in the XML file, and defining the right factory and parameters for it. In some cases a whole new factory class should be created, but both the amount of work and the amount of required specialized graphics knowledge are far less than with Java3D.

The Java3D visualization was incorporated into the GUI, as part of the main window. Ogre creates its own render window. This makes it possible to use 2 monitors efficiently. On the other hand, it can be annoying for use on a single monitor, and does appear less professional.

3D Models

There is however one area in which Ogre is much more narrow-minded than Java3D: the 3D models. Ogre only supports its own file format, so every 3D model should be converted to an Ogre mesh. In itself that is not much of a problem, except that it requires someone with more knowledge of Ogre and 3D modeling software. For Java3D there are all kinds of 3D model loaders, so that almost every widely-used file format can be loaded. This means that for the Java3D implementation, the models could just be bought on the internet or produced by a freelance artist, while for Ogre some artist is required that can convert the models, and maybe adjust them where needed.

Appearance

The subgoals about the appearance of the visualization are the following:

- sub3) The solution should allow more realistic rendering than current visualizations.***
- sub4) The solution should allow more insightful visualization than current visualizations.***

These can be evaluated in several areas.

Engine features

When it comes to realistic rendering, Ogre has more useful features. Java3D supports textures and fog, and even GLSL shaders. But Ogre also supports shadows, which add a sense of depth to the scene, and also has other useful features. For example, in Ogre it is easily possible to combine moving textures to create the illusion of waves in the water. It also has the possibility to add post-processing effects, like bloom and ambient occlusion. In short, Ogre just can do much more, especially for the purpose of adding realism.

3D Models

For both Ogre and Java3D, a lot of the realism depends on the skills of the artist. That Ogre is more efficient makes it possible to add more textures, and thus improve the realism. Ogre also easily allows adding shaders for bump maps and specular maps.

For this research, most of the VRML files that were used for Java3D, were converted for Ogre. To show how textures can improve realism, ambient occlusion textures were generated. In Figure 25a and Figure 25d it can be seen how this adds small light and shadow nuances to the RTG and the building respectively. For some VRML files, better ones were modeled and textured because the original ones were not considered realistic.

No *LOD* is used for Ogre, even though the Ogre file format supports it. The reason is that either every VRML file had to be *texture mapped* separately, or new lower level models had to be created from the detailed one, both being quite time consuming tasks.

Container stacks

The Java3D solution uses a trick to render the container stacks. The space between containers on a stack is considered to be zero, and the stack is shown by a single shape that looks like a lot of containers. As soon as a container is added, the container shape is deleted and the stack shape is updated. This results in a small glitch in the animation when a container is added to the stack, and a that there is no space between containers on the stack. For Ogre, such a trick was not needed. The static geometry makes it possible to just keep the appearance of the containers the same, and still achieve a quite good rendering speed. For Ogre it is also possible to have containers with different sizes or shapes, because it does not assume anything about the shapes it batches.

Insightful visualization

As mentioned earlier, the ways to visualize an object are narrowed down a lot with Ogre. This is caused by the design choices that were made to hide the graphics details, but also by the lack of time to implement a whole library of visualization building blocks. As a result of this, it currently is impossible to draw a path where an object has been, or display an attribute by changing the color of

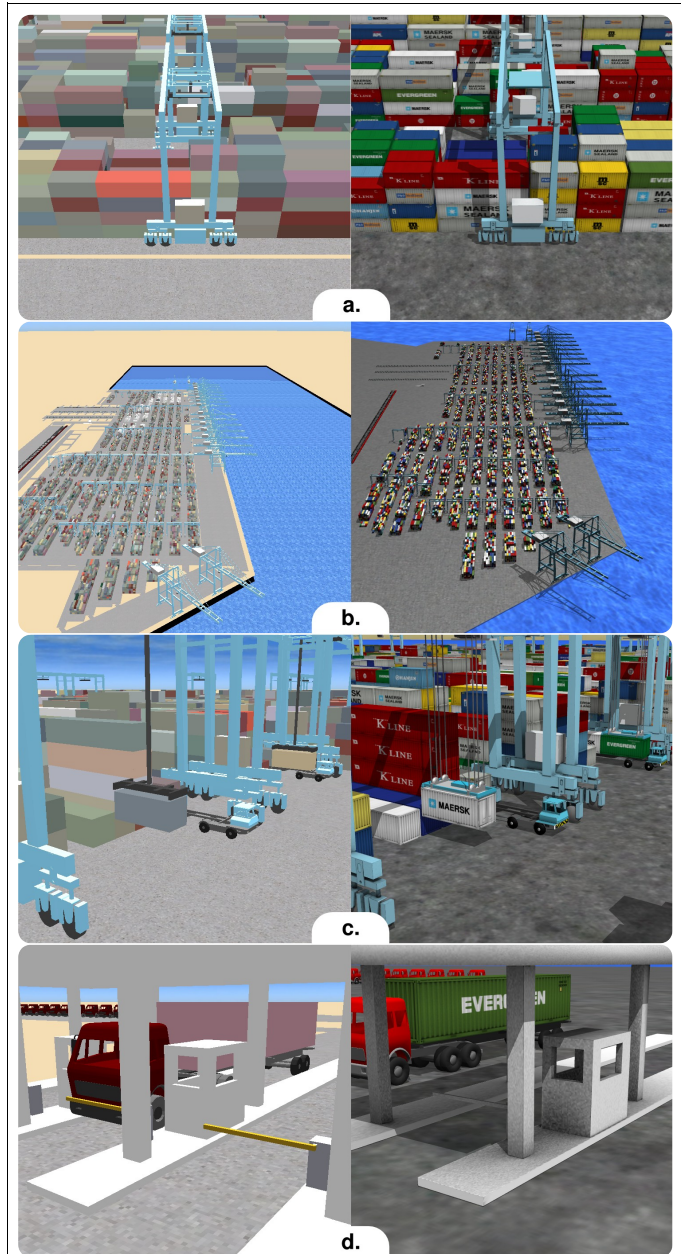


Figure 25: Comparison of the appearance of the Controls2 model with Java3D (left) and Ogre (right)

the object. If the C++-based XML configuration system as described in Chapter 5 would have been implemented together with some building blocks, the choice of ways to visualize would be less narrowed-down, but still hide the graphics details.

Performance

The third important area to measure, is the performance. There are subgoals concerning the frame rate of the visualization, and a subgoal concerning the CPU usage. These will be separately evaluated.

Achieved frame rate

The subgoals concerning the frame rate are formulated as follows:

- sub5) The solution should have a frame rate of at least 15 fps at a window size of 1024x768 pixels, even with complex simulations.*
- sub6) The solution should be able to achieve the same frame rate as the update rate of the simulation, even with complex simulations.*

Because the choice for the shadow technique was not totally definitive, the Java3D implementation will be compared with the implementation without shadows and the one with stencil-modulative shadows.

The comparison made here is quite unfair, as the Ogre visualizations use more textures, and even shadows. Also the Ogre models do not have several levels of detail, while the Java3D ones have them. When analyzing the results, this should also be taken into consideration. If the frame rates are the same, that is considered an improvement. If they are less, this means there is a trade-off between appearance and frame rate.

Test setup

Tests are done on the Normal Linux testing system, using a simulation replay. The simulation replay will run at 10x realtime, which should show quite fluent animation. This will cause the visualization to be updated frequently, but the simulation itself will hardly influence the frame rate. To run the tests, the application is run from an executable jar. This is similar to how the application would be packaged when provided to a customer.

Frame rates are measured at the same three positions as in Chapter 7. For Java3D a camera position is chosen that is as close to these positions as possible. Also the size of the rendering area for Java3D will be as close to 1024x768 as possible, but it might not be exactly the same, due to how it is integrated in the GUI.

Test results

<i>Visualization</i>	<i>Close by</i>		<i>Mid distance</i>		<i>Far off</i>	
	<i>fps</i>	<i>upd</i>	<i>fps</i>	<i>upd</i>	<i>fps</i>	<i>upd</i>
Java3D	280		24		12	
Ogre – no shadows	403	17	118	17	33	17
Ogre – stencil shadows	43	15	15	15	6	15

Table 10: Frame rate measurements for the different visualizations

From these measurements, it can be seen that the visualization without shadows performs extremely

good. The visualization with shadows does not achieve the subgoals, as for the far off views the frame rate drops below both the update rate and 15 fps.

Conclusion

The solution without shadows performs much better than Java3D, and therefore can be considered an improvement. The one with shadows however performs less. There a trade off decision should be made: beauty or speed. In this decision the kind of view that is used most, far off or close by, can play an important role.

Resource usage

The other area of performance, is the usage of resources. That is formulated in a sub-goal like this:

sub7) The solution should leave enough CPU time and memory as is needed by the simulation.

Test setup

The test setup is similar to the previous one, except that the simulation is run as fast as possible. The simulation speed gives an idea of the CPU management. The memory usage is taken from the system monitor window, because Java's internal memory representation excludes dynamic libraries. The demo simulation is run at maximum speed. This execution speed shows how much the simulation is influenced by the visualization. If the visualization consumes more resources, less is left for the simulation, and therefore the simulation will run slower.

Test results

<i>Visualization</i>	<i>Close by</i>	
	<i>Execution speed</i>	<i>Memory usage</i>
None	16	0.9GB
Java3D	11	1.1GB
Ogre – no shadows	10	1.1GB
Ogre – stencil shadows	10	1.1GB

Table 11: The execution speed (higher is better) and memory usage (lower is better) of the different visualizations

Conclusion

The resource usage on a multi-core computer is the same for both the visualizations. But the test on the single-core computer in the previous chapter, showed that the simulation and its animation do not execute concurrently. Every time the simulation thread was working, the frame rate was zero, and when the rendering thread was working, the simulation was not being updated. It can be concluded that on multi-core computers, the resource usage of the Java3D and Ogre solutions is about the same. For single core computers however, the original Java3D solution is better.



Chapter 9

Conclusions

The new solution has been made, and it has been compared with the old one. The subgoals that involved this comparison have already been evaluated in the previous chapter. Still a few subgoals remain, concerning the types of simulations and the code quality. These will be evaluated in this chapter. After that an overview of all the subgoals is given, summarizing in what extent they are met. Then the main research question is answered, and some tasks are described that were not done for this thesis, but can still be done to make the software achieve all the sub-goals, and thus provide the improvement this research is about.

Subgoal evaluation

The subgoals considering usability, appearance and performance were evaluated in the previous chapter. The remaining subgoals will be evaluated here.

sub8) *The solution should be usable to visualize any simulation in which the change of the positions of objects over time plays an important role.*

Both the DSOL simulation and the Controls2 simulation are visualized. The design is done in such a way that anything that has a position and an orientation can be visualized. It is a push-based system, meaning that the simulation itself is responsible of sending the new positions and orientations to the visualization. For every simulation some code needs to be programmed that does that. If the simulation uses C++ or Java, or one of the languages supported by SWIG, that task is fairly easy. If not, it will take some extra effort to program it.

sub9) *The code of the solution should be extensible, readable and well documented.*

The code is mostly readable, and about 50% is documented. New building blocks can be programmed and used easily, without requiring any changes to the public interface of the visualization library. It will however require a recompile of the whole visualization. With some extra effort a DLL plugin system could be used, but this probably is hardly worth the effort.

sub10) *The solution should at least run on MS Windows and Linux.*

It runs on Linux. Due to some restrictive DLL that Microsoft uses, it initially only worked on Windows computers on which Microsoft Visual Studio C++ was installed. After some hassle with configuring and compiling all the dependencies of Ogre, and Ogre itself, it was possible to create a

distributable DLL.

sub11) The interface to the simulation must hide implementation details from the simulation, but provide a simple and generic interface.

The interface indeed provides a simple and generic interface, which requires no knowledge of computer graphics or the game engine. The Java code has no reference to which game engine is used, and all the information that it sends to the game engine is in a format that can be used by any game engine. If the Ogre engine should be replaced by an other engine, in C++ a lot needs to be re-programmed, but in Java only the generated jar files and the name of the DLL need to be changed. The changes in Java are limited to the wrapper classes around the visualization. In the code using these wrapper classes, no changes need to be made.

Summary of achieved sub-goals

<i>Sub-goal</i>	<i>Description</i>	<i>Achieved</i>	<i>Comments</i>
1	No specialized knowledge		As long as no special features needed
2	Minimal work		If special models needed, they need to be converted to Ogre format
3	More realistic		Even more realism can be added by bump maps
4	More insightful		More insightful visualization building blocks still need to be programmed
5	Fps at least 15		With shadows is too slow. Without shadows is fast enough; it is faster than Java3D
6	Fps at least update rate of simulation		
7	Enough resources left		About the same on multicore systems, problematic on singlecores.
8	Usable for any 3D simulation		
9	Readable, extensible documented code		Code about 50% documented, but well extensible.
10	Windows and Linux		Compiling under windows is cumbersome
11	Interface hides implementation details		

Table 12: Overview of the extent to which each subgoal is achieved

Answer to the research question

Finally we can answer the research question:

How can the use of a rendering engine improve the visualization of simulations?

The use of a rendering engine can improve the visualization in performance or realism, or both, depending on the amount of effects that are used. This is a trade-off in which the final choice has to be made on the basis of for what purpose the simulation is to be used. By using modern GPU techniques as much as possible, most of the CPU time can be left to the simulation, so that the visualization hardly interferes with the simulation.

Because design decisions were made in favor of usability, the visualization can easily be configured by the simulation engineer without requiring much graphics-specific knowledge. In this way, not only the performance or appearance of the visualization is improved, but also the productivity of the user. However, some extra building blocks would be useful, to allow more insightful ways to show the simulation data. Also, because Ogre uses its own file format, it is necessary that either a 3D file format converter is supplied, or a huge library of 3D models in Ogre format.

Future work

The solution that was programmed for this research is not yet a finished product. In this thesis some recommendations have been made to further improve it, to turn it into a fully configurable re-usable visualization tool for all kinds of 3D visualizations. The improvements are also categorized.

Usability

Window closing support

Currently the assumption is made that the visualization always has exactly one window. Even though this is nice for the most general use, it could occur that a user of the software wants to close the window for now, and re-open it later on. Also it could be useful to have several views of the same world. This requires some research on how Ogre handles multiple windows, and which techniques require to have at least one render window open.

User interaction

Currently the visualization only supports viewing. However for better analysis, other ways of interaction might be needed. Possibilities for selecting objects in the 3D view, or having the camera follow an object could be added.

More building blocks

Extra building blocks should be added which allow assembling insightful visualizations. For example a building block that draws a line where an object has been, or a building block that can display several keyframe animations.

XML configuration system

As described in Chapter 5, it would be very useful to implement an xml configuration system in the C++ part of the visualization, so that objects can be visualized with combinations from the building blocks, without needing extra programming on the simulation side.

Feature configuration system

It would also be useful to be able to configure the visualization using a configuration file. Settings like the default window size, the use of antialiasing and the use of shadows should be easily configurable.

Appearance

Texture shadows

The measurements showed that texture shadows are way better than stencil shadows. The only reason why they are not used in the final solution, is that they could not yet be configured correctly. If they are configured correctly, they can improve the frame rate to acceptable heights, and also increase realism.

Per-fragment lighting

The solution currently uses per-vertex lighting. Per-fragment lighting however looks much better, especially for low-poly models. This shows the difference between the two. This also makes way for using models with an even lower polygon count, without people noticing it.

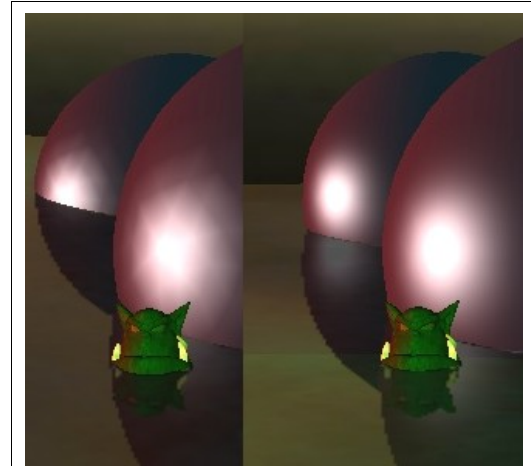


Figure 26: Per-vertex (left) and Per-fragment (right) lighting.

Examples provided by Nir Hasson of the Ogre community

Deferred shading

The realism can be even further improved by using deferred shading techniques. These allow real-time ambient occlusion, glow, and easy lighting and contrast enhancements.

Performance

Instanced geometry

Using instanced geometry instead of static geometry might improve the performance of the rendering of the container stacks, by moving more of the calculation to the GPU.

By doing all these extra tasks, a solution can be created that achieves all the subgoals, and that can really be called an improvement of the visualization. Apart from these subgoals there are also loads of opportunities to improve the usability of the visualization for Controls2.



Supplement

Appendices, Bibliography and Glossary

Appendix A: 3D File format support of Game Engines.....	54
Appendix B: 2D File format support of Game Engines.....	55
Appendix C: Test Computer Setup.....	56
References.....	57
Glossary.....	58

Appendix A: 3D File format support of Game Engines

<i>Feature</i>		<i>Ogre3D</i>	<i>OpenSceneGraph</i>	<i>Irrlicht</i>	<i>CrystalSpace</i>
3D file formats	Collada .dae		+	+	
	Lightwave .lwo		+	+	
	Wavefront .obj		+	+	
	OpenFlight .flt		+		
	TerraPage .txp		+		
	Carbon Graphics .geo		+		
	3Dstudio .3ds		+	+	+
	Performer .pfb		+		
	AutoCad .dxf		+		
	Quake2 character .md2		+	+	
	Direct X .x		+	+	
	Inventor ascii .iv		+		
	Vrml 1.0 .vrml/.wrl		+		
	Vrml 2.0 .wrl		-		
	X3D .x3d		-		
	Designer workshop .dw		+		
	AC3d .ac		+		
	Ogre3D .mesh	+		+	
	OpenSceneGrpah .osg		+		
	Irrlicht scene .irr			+	
	Irrlicht mesh .irmesh			+	
	B3D .b3d			+	
	Quake3 models .md3			+	
	Quake3 levels .bsp			+	
	Milkshape .ms3d			+	
	My3DTools .my3d			+	
	Pulsar Lmtools .lmts			+	
	FSRad oct .oct			+	
	DeleD .dmf			+	
	Cartography shop .csm			+	
	STL 3D .stl			+	
	Cal3D .cal				+
	Ogre scene .scene	+			
	CrystalSpace world				+

Appendix B: 2D File format support of Game Engines

<i>Feature</i>		<i>Ogre3D</i>	<i>OpenSceneGraph</i>	<i>Irrlicht</i>	<i>CrystalSpace</i>
2D file formats	JPEG .jpg	+	+	+	+
	PNG .png	+	+	+	+
	Targa bitmap .tga	+	+	+	+
	.rgb		+		
	.gif	+	+		+
	.tiff		+		
	.pic		+		
	DirectDraw surface .dds	+	+		+
	Windows bitmap .bmp	+	+	+	+
	Adobe photoshop .psd			+	
	Zsoft paintbrush .pcx			+	
	Portable pixmap .ppm			+	
	Quake2 texture .wal			+	

Appendix C: Test Computer Setup

Tests were done on two systems. Mainly they were done on a new computer, which embodies the normal computers that the average TBA employee will use the coming two years. The system runs Linux, in order to test the cross-platformness of the code.

The second test system is used less often, because it is very slow and has long compile times. It represents old computers that would probably be replaced by newer ones quite soon. It gives an indication of how bad the software will run if the CPU is single core, and the graphics card is quite limited.

Normal Linux testing system

Processor:	Intel Core 3 DUO E8500, 3.16GHz
Ram:	4 GB
Graphics card:	nVidia GeForce 9800 GT 512MB (2008)
OS:	Ubuntu 9.04 64 bits

Normal Windows testing system

Processor:	Intel Core 3 DUO E8500, 3.16GHz
Ram:	4 GB
Graphics card:	nVidia GeForce 9800 GT 512MB (2008)
OS:	Windows XP 32 bits

Old Windows testing system

Processor:	Intel Pentium 4 3.2 GHz
Ram:	1 GB
Graphics card:	ATI mobility radeon 9700 128MB (2004)
OS:	Windows XP 32 bits

References

- Bijl, J.L., **How Computer Game Technology can be used to improve Simulations**, Delft University of Technology and TBA, January 2009.
- Bijl, J.L., Fresen, J., **Improving 3D animation for Controls2**, Delft University of Technology and TBA, July 2006
- Brightman, J., **Should Ultra Realism Be The Ultimate Goal?**, IndustryGamers.com, <http://www.industrygamers.com/galleries/opinion-should-ultra-realism-be-the-ultimate-goal/4/> , July 2009
- Collins, W., **Crystal Space or Ogre, which engine should I choose for my game project?**, <http://www.arcanoria.com/CS-Ogre.php>, viewed on march 19th, 2009.
- Friesen, J. , **Open Source Java projects: Java Native Access**, JavaWorld, <http://www.javaworld.com/javaworld/jw-02-2008/jw-02-opensourcejava-jna.html> , February 2008
- Gamma, E., Helm, R., Johnson, R., Vlissides, J., **Design Patterns, Elements of Reusable Object-Oriented Software**, Addison-Wesley, 1995.
- Griffo, J. B., **Shader Instancing**, Google Summer of Code 2006, August 2006
- Junker, G., **Pro OGRE 3D programming**, Apress, 2006, pp. 152
- Liang, S. **The Java Native Interface**, Addison-Wesley, June 1999
- Monte Lima, J. P. S., Farias, T.S., Teichrieb, V., Kelner, J., **Port of the Ogre 3D Engine to the Pocket PC Platform**, Universidade Federal de Pernambuco, May 2006.

Glossary

3D Studio Max

One of the best-known commercial 3D modeling programs. It is used by many professionals in the game and movie industries. At the time of writing, 3D studio max costs more than \$3000. This is an often quoted drawback of the program.

Anisotropic filtering

A more realistic filtering technique than Trilinear filtering. The decision on which mipmap to use is not only based on the distance, but also on the angle of the surface.

Barrier

A synchronization technique in concurrent computing. Barriers define places in the code where a process will wait till all other processes have also reached their corresponding barrier, after which all processes will continue.

Bilinear filtering

A texture rendering technique. The four pixels closest to the texture position that should be drawn, are averaged to find the color that is needed.

Billboard

A technique to place an image in a 3D scene. The image always faces the camera, but is placed in the 3D scene. In games these are used to display health information above the heads of characters, to show clouds of smoke and dust from explosions, and to show trees that are far away.

Blender

The biggest free open source 3D modeling and animation program. Blender can also be used for movie editing, rendering, and even contains a game engine. Blender originally was a commercial package, but after the company bankrupted it was licensed under the GPL. It is seen by many as “the free alternative to 3d studio max”.

Bridge pattern

A design pattern that hides the implementation of a class to allow multiple implementations to be used interchangeably.

DEVS (Discrete Event System)

A formalism for simulation, which uses state machines and events to describe the system. Events trigger state changes, and state changes can trigger even more events. The simulator simply traverses and executes the event queue.

Euler angle

A way to describe a rotation. It is described as a vector, containing the rotation around each axis. These rotations are applied in the order [X, Y, Z]. angles can be either in radians or degrees.

Factory Pattern

A design pattern where a class is not supposed to be created directly using its constructor, but is created using a Factory class instead. This allows to define different ways of creating the instances of the class or one of its subclasses.

Fixed Function Pipeline

Predecessor of the *Programmable Pipeline*. The pipeline is the series of algorithms that need to be applied to the 3D data to draw it on the screen. In the fixed function pipeline, the user can only influence the outcome by selecting which algorithms should be used.

Game Engine

Software or part of software that supplies an important part of a game. It often contains an AI engine, a rendering engine and a physics engine. Game engines are often licensed to game companies, allowing them to develop a high quality game in a relatively short time, because the hardest part in game development is done in the game engine. This leaves the company with the game-specific tasks of storywriting and art.

GUI

Graphical User Interface – The whole of buttons, windows, text areas and pictures that provide the visual representation of the program and the ways for the user to interact with it.

Header file (C / C++)

In C and C++, the implementation of a class or module is separated from its specification. The interface is specified in a header file, which often has an extension *.h*, *.hh* or *.hpp*. When a program uses a library or class, it uses the interface, and only at linking time the implementation is connected to the used stubs.

Impostors

Name for a technique where a 3d shape itself is not drawn, but a flat surface with a texture is shown instead. This can either be done with pre-rendered images which are loaded from a file, or with images which are rendered when needed.

Jar file (Java)

A Java archive file is the most common way of deployment for java libraries and files. It essentially is a file containing the compiled java classes. It can also contain extra resources that are needed by the program, and can even be configured so that when doubleclicked, it starts the program contained in the jar.

JNA – Java Native Access

A technique which allows a programmer to access C/C++ code from Java. In contrast to JNI, the links of the Java to the native code are established at runtime, and only the Java code needs to be developed. Therefore JNA can even be used for shared libraries for which no C++ code is available, like Windows system libraries.

JNI – Java Native Interface

A technique which allows a programmer to program C++ or C code for Java applications. From a Java class file, Sun's *javah* tool generates a C/C++ interface, which can be used by the C++ programmer to interface to his C/C++ code.

LOD (Level Of Detail)

Technique where the amount of detail to be drawn depends on a certain parameter. Most frequently the distance of the object is used as parameter. Of each object, a complex and several simpler versions exist, and the further away the object is, the simpler version is used. Here the complexity is defined by the amount of polygons that need to be drawn.

Material LOD

A LOD technique that does not reduce the amount of polygons, but the amount of details in the material. For example, objects that are close by can have textures, normal maps and reflections, but objects that are farther away just have textures.

Maya

A commercial 3D modeling and animation program that is used a lot for movie creation and game design. It costs about \$4000.

Mipmap

A series of smaller copies of a texture. These are used to texture the objects that are further away. This is not exactly done for optimizing the rendering time, like LOD is, but to make sure that far away textures won't look jaggy.

MMORPG

Massively Multiplayer Online Role Playing Game – a game in which the player controls a single character, often with its unique strengths and weaknesses developed during the game, which interacts with hundreds or thousands of other characters in a virtual world. The game has a client-server structure, in which the clients can either be browser based, like applets or flash games, or installed software.

Mock object

An object type used in automated testing. It acts like a real object, but replaces its functionality with simple code or with checks. It can be used to make the test run faster, or to check if the module under test interacts correctly with the other objects.

Moiré patterns

Artifacts that occur when fine patterns are shown using pixels. The sampling of the pattern to the pixels creates a new, and often unwanted, pattern.

Mutex (Mutual Exclusion)

A technique in concurrent computing, where a variable is used to make sure that a variable or part of code can only be accessed by one thread of execution at a time. The most common implementation is a boolean variable *locked*, which is checked by every process using it, before using the critical section of the code. Many threading libraries contain more advanced implementations of this technique.

Programmable Pipeline

Successor of the *Fixed Function Pipeline*. Programs using the 3D hardware, can also contain algorithms that should be done as part of the pipeline. Either *vertex shaders* or *fragment shaders* can be specified, being algorithms that modify positions or other attributes of vertices, or colors of pixels, respectively. Early shaders were written in an assembler-type language, but now also higher-level languages are defined. OpenGL shaders can be written in GLSL, DirectX shaders in HLSL. The graphics card vendor nVidia the language Cg, which can be compiled into either GLSL or HLSL.

Quaternion

A way to describe a 3D rotation, based on complex numbers. It is especially useful for interpolating rotations.

Rasterization

The way of rendering that is used most for real-time graphics. 3D transformations are applied to scene objects, after which they are projected to the screen. It is quite fast, but not physically accurate. A lot of tricks and workarounds are done to make it look realistic.

Singleton

A Design Pattern [Gamma, 1995] which is used in Object-Oriented software to insure that at most one instance of a class is created and used.

Skin

An outward appearance for an object. 3D first-person shooter games often contain several different textures for the same character. These can be used to give the character the colors of the team he is in.

Sky box

A trick to create a sky background for a 3D scene easily. Pictures, possibly even photos are applied as texture on a box. This box is placed around the whole scene.

SWIG – Simple Wrapper Interface Generator

A tool which uses a configuration file to wrap C++ code to different languages, including C#, Java, Python, PHP and Ruby. For a connection to Java it uses JNI.

Terminal Operating System

The software that controls a container terminal. It is often responsible for sending tasks to the drivers that work in the terminal and planning where containers should be stacked.

Texture mapping

The process of defining how the surface of a texture should be applied to a 3D model. It involves laying out the polygons of a model on a texture. To efficiently use the whole area of the texture, texture mapping often resembles a jigsaw puzzle, attempting to fit as many polygons on the model as possible. An example is shown in Figure 27.

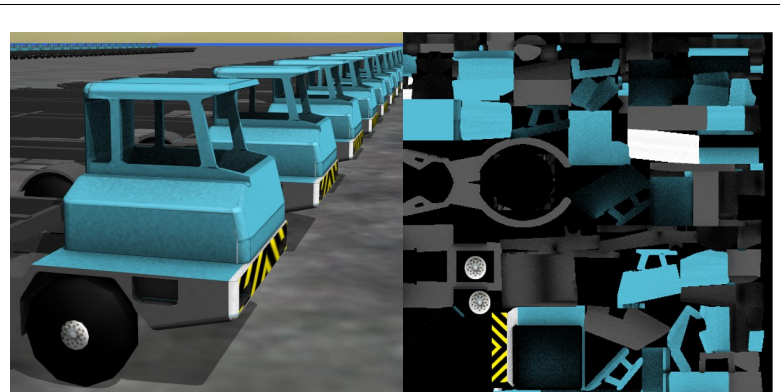


Figure 27: The terminal trucks, and the texture that is applied to them. Note how little of the texture area is left unused.

Trilinear filtering

A mipmap rendering technique. It does the same as bilinear, except that it also averages between two mipmap levels. This technique makes sure that no clear seam is visible between the close-by high level mipmaps and the further away low level mipmaps.

Unit test

A program that automatically tests the functionality of a separate module. The unit tests are created along with the module that is being tested, in order to ensure the module behaves as required. Unit tests help to locate behavior differences when the code, compiler or platform is changed. The tests are to be run often, so that a bug can be detected as soon as it has been introduced.