Distributed OpenSceneGraph

Master Thesis

Thomas Samuelsen

Supervisors:

Professor Otto J. Anshus

And

Associative Professor John Markus Bjørndalen.

Department of computer Science
Faculty of Science
University of Tromsø
9037 Tromsø

2007

Abstract

This thesis presents the design and implementation of Distributed OpenSceneGraph. Distributed OpenSceneGraph is a graphics visualization toolkit for distributed applications, in particular for *tiled display wall* systems. Distributed OpenSceneGraph allows for flexible and reasonably transparent development of distributed graphics applications by introducing the notion of distributed nodes into the well know OpenSceneGraph graphics toolkit.

By letting the Distributed OpenSceneGraph only concern itself with the state of individual scene graph nodes we achieve a great degree of flexibility. It is not in any way enforced that the local scene graph copies in any of the processes that make up the distributed system must be identical, nor is it necessary that all or any of the distributed nodes in the total distributed application be present in a processes scene graph copy. This enables an application developer to create applications with radically different scenes while still distributing what needs to be.

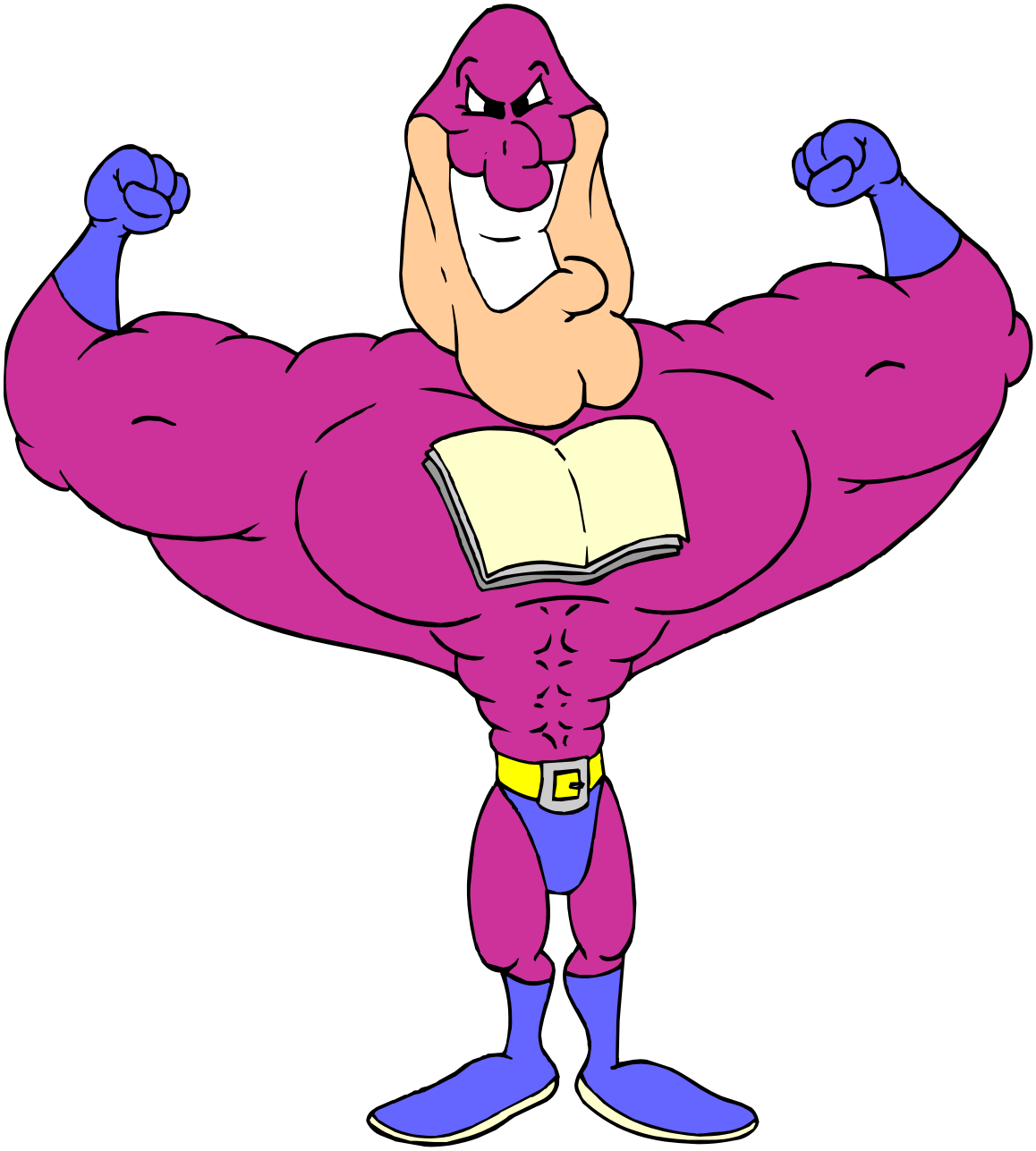
An important focus when implementing Distributed OpenSceneGraph was transparency. Therefore a lot of effort has been laid into enabling application developers to use Distributed OpenSceneGraph with only a few extra function calls beyond what they would have encountered when creating undistributed OpenSceneGraph applications.

The flexibility and transparency introduced to visualization of application data makes Distributed OpenSceneGraph suitable for use in the development of distributed graphics applications.

At time of writing the Distributed OpenSceneGraph library is unfortunately not quite finished. However a number of measurements and possible solutions are presented to show the path onwards.

Keywords: 3D graphics, graphics applications, distributed applications, scene graph, C++, OpenSceneGraph

"The only certain path to failure is to give up"



Thesis statement

This thesis is an attempt to lessen the complexity and potentially increasing the performance of parallel graphics applications running on tiled display walls.
This will be accomplished by introducing the notion of a distributed scene graph into a graphics toolkit in an as transparent as possible way.

Contents

List of figures

Listings

List of tables

Introduction

During the past decades the speed, memory and networking abilities of computers has evolved beyond all but the most optimistic predictions. The computing tasks that may be presented to personal computers today would have humbled all but the most powerful supercomputers a decade ago. However the size and resolution of commodity computer displays has in the same period only improved marginally in comparison.

When in need of visualizing large data sets with great detail common computer displays are often inadequate. Another option, video projectors alleviate the problem somewhat by producing a much larger image but the resolution still is very limited.

Another solution is using large scale LCD or plasma screens but the price tag associated with these when in sizes of 100" and beyond often make this a rather expensive option.

A solution which has come about the last decade is *tiled display wall* systems. These systems use a cluster of individual commodity computers each attached to its own monitor or video projector. Each computer in the cluster is responsible for producing a single tile of a composite image. By physically arranging the monitors or projectors appropriately they will generate a composite image with a screen size which is the sum of all displays involved and a resolution which is the sum of all resolutions of the displays involved. Thus a display wall is capable of generating large images with comparably very high resolutions.
The use of commodity components when designing display wall clusters also makes this a reasonably affordable alternative, at least when compared with other solutions with comparable screen sizes and resolutions.

A side benefit of using a display wall is that you will also have a reasonably powerful cluster of workstations (COW) at your disposal. This feature can be used to actually do the calculations that are to be visualized removing the need for a separate cluster or supercomputer.

To visualize datasets, results and other imagery some kind of graphics library is usually applied. This can be low level API's like Direct3D or OpenGL or they can be more sophisticated toolkits built on top of those API's. Examples of such toolkits can be OpenSceneGraph, COIN 3D, Open Inventor or OpenGL Performer. The heart of many such toolkits is the *scene graph*. A scene graph is a hierarchical data structure used to store, organize and manipulate graphics' scenes in *retained mode* graphics toolkits. Instead generating and drawing the graphics objects in the scene immediately scene graphs draws its graphics objects by traversing the graph in which they are stored. A simple example of such a tree and its resulting scene can be seen in figure 1.

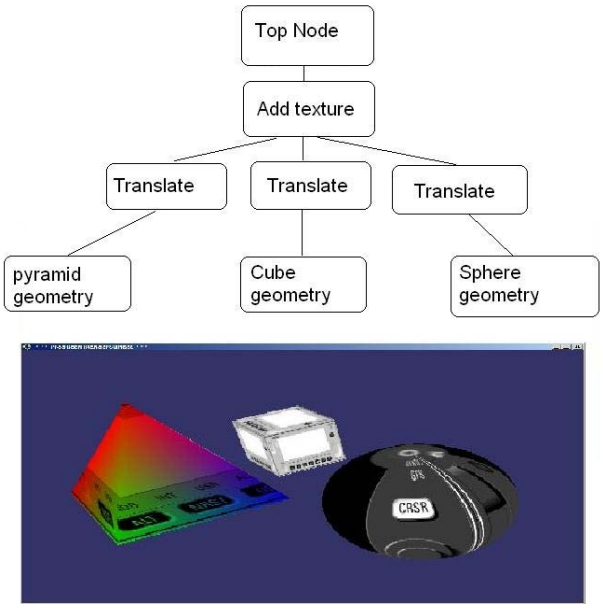


Figure 1 Simple scene graph tree and the resulting scene.

These graphics toolkits are usually tailored against single computers running the graphics applications built on top of them. Therefore they rarely contain any mechanisms for distribution and on a tiled display wall; they have to rely on the functionality of the split-screen mechanisms of the operating system to use all displays the wall consists of.

Instead of running one instance of the application and leaving it to the underlying display wall system to display the applications graphical output to the wall, an application programmer could run a single instance of the application on each display tile computer and generate the part of the total image to the tile itself. With good synchronization mechanisms between each process that then make up the total application this will often produce smoother and faster graphics on the display wall. The price for such a solution however is more complex applications and using more of the display wall's system resources since often the same calculations are done on each node.

In the case of using a scene graph toolkit each process of the application running on the individual computers making up the display wall, will have its own copy of the entire graph. Each process will have to traverse the graph and do the same manipulations on it for each frame.

Observing that each process of the application is actually running on an individual node of a computer cluster we see that it should be possible to lessen amount of necessary computations of each application process by splitting up and distributing the problem between them.

Two ways of splitting up tasks when distributing graphics toolkits can be to either capture streams of graphics commands and distributing them accordingly as is done in Chromium

[15]. The advantages of such a solution is that it both can be used to visualize the scene drawn by a single process onto all the display walls tiles or it can be adapted to draw the composite scene generated by a parallel application.
Another way, which is the approach this thesis will look at, is to use the notion of a distributed scene graph as done in Distributed Open Inventor [13] and the blue-c [14] scene graph. In a distributed scene graph each process in the distributed application is give ownership and made responsible for a subset of the scene graph nodes. The owner of a scene graph node is responsible for processing the updates of this node and propagating the resulting state onto the other processes in the application.

Why distribute?

The benefit of distributing the scene graph nodes would be to split up the computational load of recomputing the geometry data by giving each participating process the responsibility of only a subset of the objects to calculate.

Distributing the load of massive or large number of small computations has the potential of being advantageous if done correctly. This is in particular true where these computations take so much time that they hamper the frame rate of the graphics representation. Complex embarrassingly parallel computations [10] are very good examples of such problems.

As an example of such a problem we will use throughout this report will be using 3D visualization when calculating fractals like the Mandelbrot set shown represented in 2D in figure 2.

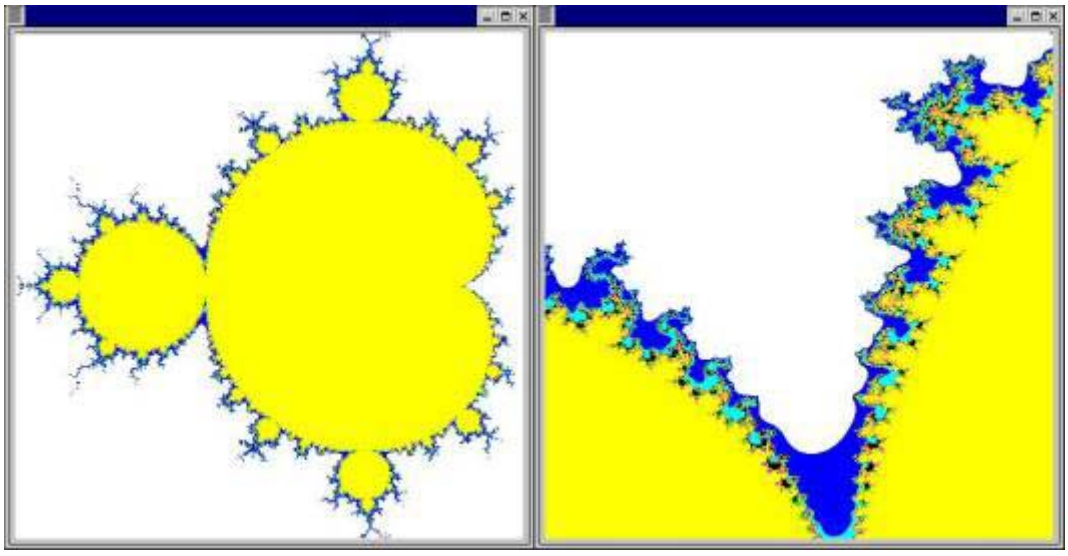


Figure 2: 2D representation of Mandelbrot set. Images from http://www.cs.uit.no/inf3201/2006h/assignments/DessertMap.html

Instead of just representing the calculation results merely as colours as in figure 2 we could represent them as coloured cubes where the result is used to represent the height of the box.

One way of doing this on a video wall is either to have one computer do the entire calculation and then passing the results over to a process on each computer that renders its view of the computation's visualized result.
Another way of doing this would be making each node run identical calculations, meaning that each node does all the work and then simply renders the result from its perspective. This might be a faster implementation but still the very same calculations are done in their entirety on all the processes in the system.

This report suggests another approach. By dividing all the cubes to be calculated evenly between all participating processes and propagating the results of each node to the other processes participating in the application we might gain better performance.

This thesis is an attempt of creating a visualization toolkit which does just that.

3D graphics and scene graphs

3D visualizations can programmatically be approached in two different ways: *Immediate-mode* [34] or *retained-mode* [35].

In *Immediate-mode* the representation of objects in the scene, their appearance, spatial relationships and position relative to the viewer are sent one at a time to the graphics library for rendering. As each frame of animation is drawn the data is retransmitted to the library for rendering. While this approach gives the maximum control and flexibility for the application programmer it can be complex and time consuming process to develop applications this way. This is usually accomplished using the underlying 3D API's like OpenGL or Direct3D directly.

To allow for application programmers to spend less time doing low-level work like loading, managing, culling, and rendering and thereby speeding up the development process r*etained-mode* API's is usually often used. By using *retained-mode* API'*s* application programmers can to a large extent leave managing and rendering peculiarities to the library functions and focus on the higher level functions of the application.

OpenSceneGraph [1], Performer, Open Inventor [21] and Java 3D are all *Retained-mode* toolkits. Unlike libraries like OpenGL, which many such libraries are built on; these libraries store 3D objects and their spatial information in a database and provide the functionality necessary to allow for interaction with these objects.

The term *retained-mode* comes from the fact that they store the data needed to render the scene in advance in an internal, often hierarchical, data structure. Rendering of the scene is then accomplished by some algorithm used traversing this data structure.

This data structure is what is often referred to as a *scene graph*.

A retained-mode or Scene graph library usually exists as a middleware between a low-level API like OpenSceneGraph or Direct 3D and the application. Figure 3 shows this architecture.

The definition of what a scene graph is somewhat obscure due to the fact that developers that implement scene graphs tend to take the basic principles of a graph and adapt it to their own uses and applications.
For our purposes a scene graph is a collection of nodes organized in a tree structure. Any change applied to a parent node is automatically propagated to its children by accumulating state and translations during the traversal of the graph. The point of using scene graphs in computer graphics scene management is to simplify and abstract scene management so that application programmers can focus on the scene content.

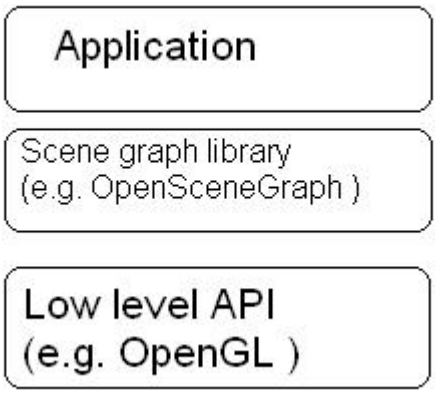


Figure 3 A 3D application stack using a Scene Graph library

A *scene graph* tree is headed by a root node which acts as the start point of the rendering traversal. Beneath this group nodes are used to organize the geometry and the rendering state that controls their appearance. At the bottom of the graph tree we find the leaf nodes which contain the actual geometry that make up the objects you actually see in the scene.

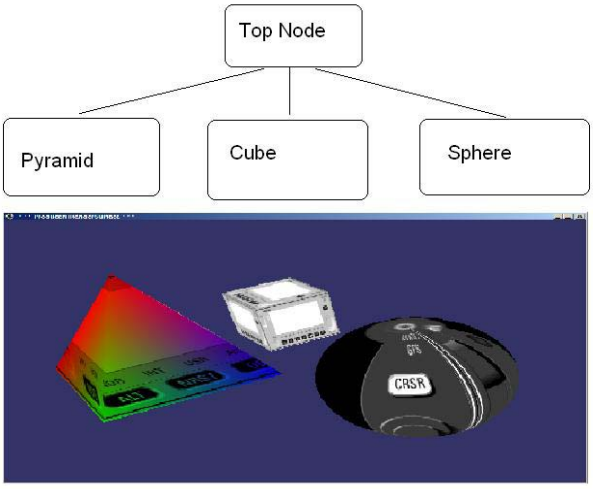


Figure 4 A very simple scene graph tree

Figure one shows the simplest view of a tree with three nodes and the resulting image, figure 2 shows the same tree but with translations and texturing added.

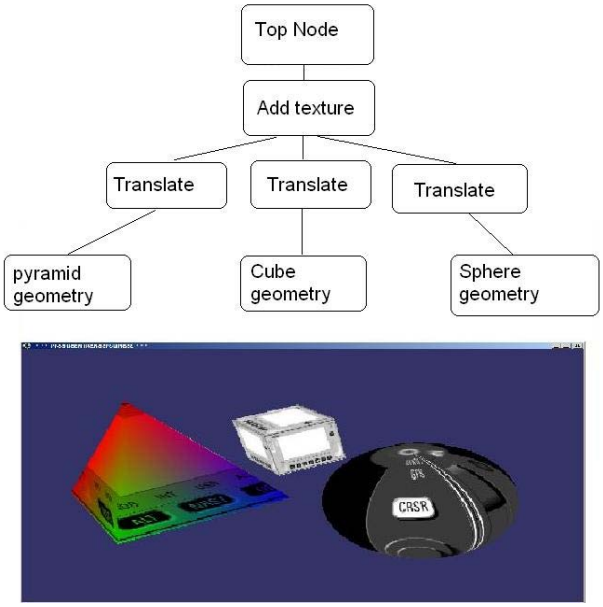


Figure 5 Simple scene graph tree with state and translation nodes added. Note, since the traversal down to the all leaf node must pass the "Add texture" node, all leaf nodes are rendered with the same texture and texture state.

To offer the level of functionality desired scene graphs usually offer a wide range of different node types like switches which allow for choosing between underlying branches of the tree, transformation nodes that modify the transformation state of their underlying geometry and level-of-detail nodes which select children based on distance from the viewer.

In addition to being an interface to the geometry and state functionality provided by the low-level API's like OpenGL and Direct3D they are built on, scene graphs provide a number features and capabilities of their own. These include, but are not limited to:

- Spatial organization: The scene graph structure lends itself naturally to intuitive spatial organization.
- Culling: Both view frustum and occlusion culling typically improves performance by reducing the rendering needed by excluding objects not in view from the processing.
- Level-of-detail: By allowing for selection between child nodes representing the same 3D object but at varying levels of detail based on the distance from the viewer the overall performance of the application can be improved.
- Blending: Blending operations are commonly supported by scene graphs. For blending to be performed correctly all non-opaque geometry must be rendered after all the opaque geometry in the scene. Furthermore the non-opaque geometry must be ordered correctly by depth coordinates in a back-to-front order and rendered in this order.
- State change minimization: To improve performance 3D applications will typically try to minimize state changes. This is usually done by grouping objects

that need the same state set together. Scene graphs usually have state management facilities that eliminate redundant state changes. This is usually done by allowing the state to be accumulated during the traversal which, se figure 6.

- File I/O: Scene graph libraries like OpenSceneGraph typically allow for reading and writing of a variety of file formats for both images and 3D models. Once loaded into memory the scene graph data structure allows for the easy manipulation of dynamic 3D data. Scene graphs can also be an effective intermediary for converting from one file format to another.
- Effects: Scene graph API's often have built in support for effects like shadows and particle systems.
- Full featured text support
- Cross platform support for input and output devices.

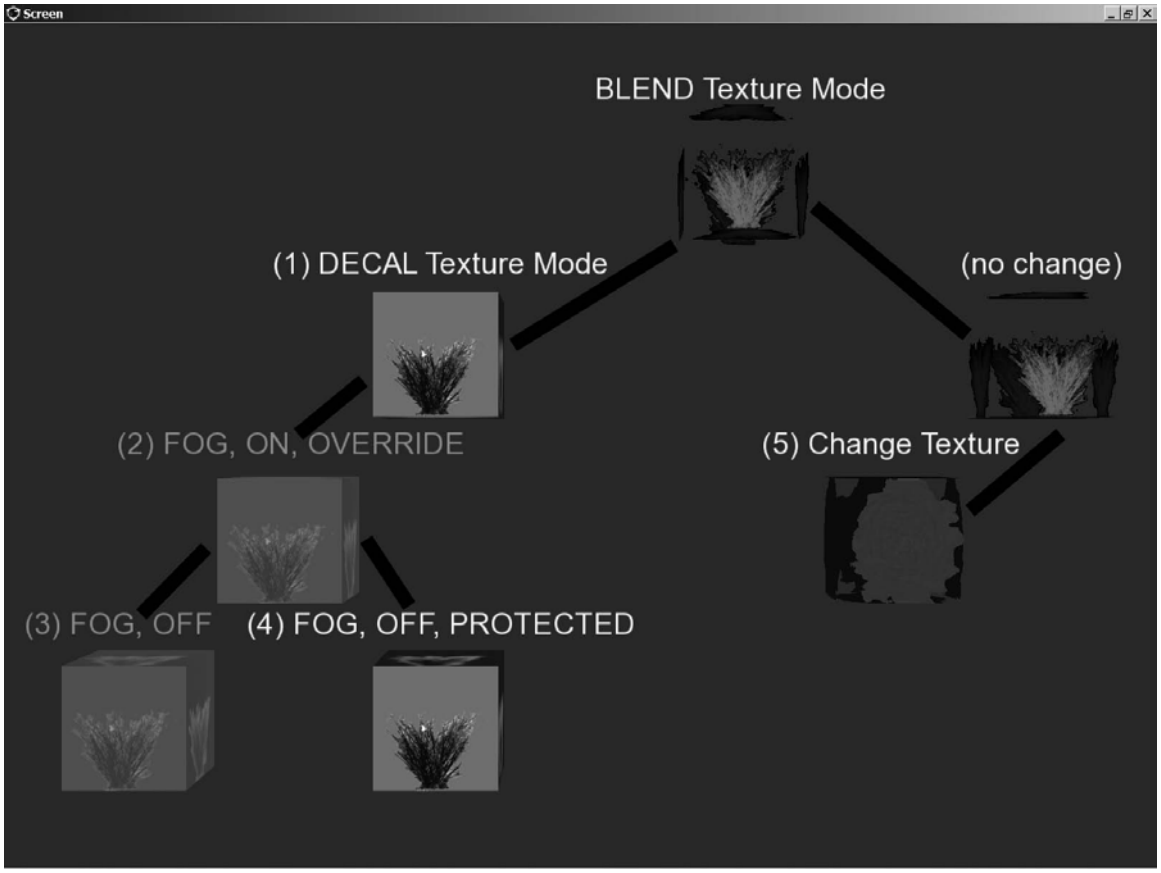


Figure 6 Accumulating state. The black lines represent the egdes of the graph. Also shows that you can override the accumulation of certain charatheristics. Image from (modified): Jospeh Sullivan NPS osgTutorial. http://www.nps.navy.mil/cs/sullivan/osgtutorials/

A typical 3D application will use at least some of this functionality, and having it readily available in the library itself versus the developers having to implement these themselves can make for much faster application development.

Rendering a trivial scene graph is done by traversing the graph and sending the resulting state and geometry data to the underlying hardware as OpenGL or Direct3D commands. This will usually be done once per frame. However, many of the features mentioned above, for example culling and sorting, require multiple traversals.

- Update traversal: This traversal is used to modify the state and geometry as specified by the traversal. These updates are done either directly by the application or by callback functions attached to the nodes that they are to operate on. An update can be something like changing the position or color of an object in the scene.
- Cull traversal: When the updates are complete a cull traversal can calculate what will actually be visible in the scene and pass a reference to the visible objects into the final rendering list. This traversal can also take care of the ordering of nodes for blending support. This traversal results in a data structure which is sometimes referred to as a render graph.
- Render/Draw traversal: This traversal traverses render graph generated by the cull traversal and passes this to the hardware as the underlying API calls to render that geometry.

Tiled Display Walls (TDW's)

> ***"Trying to view these datasets on a typical monitor can be analogous to looking through a straw into the haystack."***
> *David Lee*
> *BioWall Application Engineer, NCMIR*

Quote from:
http://searchenterpriselinux.techtarget.com/originalContent/0,289142,sid39_gci1011737,00.html

Tiled Video Walls

While memory and speed of computers and the speed of the networks connecting them has increased exponentially the past decades the size and resolution of their displays has improved only modestly.

When in need of visualizing large data sets with great detail common computer displays are often inadequate. Another option, video projectors alleviate the problem somewhat by producing a much larger image but still the resolution is very limited.

Another solution is using large scale LCD or plasma screens but the price tag associated with these when in sizes of 100" and beyond often make this a rather expensive option.

A solution which has come about the last decade is *tiled display wall* systems. These systems use a cluster of individual commodity computers each attached to its own monitor or video projector. Each computer in the cluster is responsible for producing a single tile of a composite image. By physically arranging the monitors or projectors appropriately they will generate a composite image with a screen size which is the sum of all displays involved and a resolution which is the sum of all resolutions of the displays involved. Thus a display wall is capable of generating large images with comparably very high resolutions.

The advantages of the TDW's include a relatively low price for a good screen size, since it typically uses commodity computers and display solutions, high resolution when compared to other solutions like video projectors and very large scale LCD or plasma screens.

A side benefit of using a display wall is that you will also have a reasonably powerful cluster of workstations (COW) at your disposal. This feature can be used to actually do the calculations that are to be visualized removing the need for a separate cluster or supercomputer.

Figure 7 This is a snapshot of vnview image show application shown by a 4x5 monitor tiled display wall.
Image from: http://goc.pragma-grid.net/pragma-doc/pragma11/CNIC%20Tiled%20display%20wall%20and%20astronomical%20Data%20Visualization.ppt

A core component to manage this visualization cluster is the underlying operating system of the computers that make up the wall. This is typically adapted distributions of operating systems like the Rocks cluster Distribution.

The Rocks cluster distribution is a Red Hat Enterprise Linux [41] or Cent OS [40] distribution. Which one, is chosen at install time.
After installing Rocks on the front-end, all compute-node installation and configuration is handled automatically by the Rocks system.

OpenSceneGraph

This section is by no means meant to be a thorough tutorial in the use of the OpenSceneGraph API but will shed some light on some of its functionality that was used when implementing the Distributed OpenScenegraph.

OpenSceneGraph basics

Please refer to www.openscenegraph.com site for reference guides for details of OpenSceneGraph. Also it's worth mentioning are the tutorials found at http://www.nps.navy.mil/cs/sullivan/osgtutorials/ . They are highly recommended as an entry point into understanding and using OpenSceneGraph.

osg::NodeVisitor class.

The NodeVisitor class is based on GOF's visitor pattern. The NodeVisitor is useful for developing type-safe operations to the nodes in the scene graph (as per visitor pattern), and adds to this support for optional scene graph traversal to allow for operations to be applied to whole scenes at once.
The visitor uses a technique of double dispatch as a mechanism to call the appropriate apply method of the NodeVisitor instance. To use this feature one must use the osg::Node::accept (nodevisitor) function found in each node subclass rather than the NodeVisitor's apply function directly. See listing 1 for details.

```
//instantiate visitor
DOSGVisitor myVisitor = new DOSGVisitor;
//start traversal at root.
root->accept ( myVisitor );
//Note the use of root->accept not
//the myVisitor->apply function.
//The latter will bypass the double dispatch
//mechanism and the myVisitor->apply function
//will not be called
```

Listing 1 Instantiating and attaching a derived node visitor to the root of the tree for traversal of the entire scene graph.

Updating nodes using callbacks.

OpenSceneGraph users can interact with the scene graph using callbacks. Callbacks can be thought of as user defined functions that is automatically executed depending on the type of traversal (update, cull or draw) being performed. Callbacks can be associated with individual nodes or they can be associated with specific types or subtypes of scene graph

nodes. For each traversal if a node containing a callback is encountered it, that callback is executed.
In OpenSceneGraph a callback class must always be derived from the osg::NodeCallBack super class. The derived callback class must override the operator() method that takes a pointer to the node it belongs to and a pointer to a node visitor instance as input. See listing 2 for an example of how this can be done.

```
class tankNodeCallback : public osg::NodeCallback
{
public:
virtual void operator()(osg::Node* node, osg::NodeVisitor* nv)
{
    osg::ref_ptr<tankDataType> tankData =
        dynamic_cast<tankDataType*> (node->getUserData() );
    if(tankData)
    {
        tankData->updateTurretRotation();
        tankData->updateGunElevation();
    }
    traverse(node, nv);
}
};
```

Listing 2 Example derived node callback class in OpenSceneGraph. The class is called once per update traversal. Code from: http://www.nps.navy.mil/cs/sullivan/osgtutorials/osgUpdate.htm

Another option for updating nodes is to put update code in the OpenSceneGraph main loop between the viewer.update ( ) and viewer.frame ( ) function calls. While the effect would be the same code using callbacks tend to be easier to update and maintain. Code that takes advantage of callbacks can also be more efficient when a multithreaded processing mode is used.

Returning to our Mandelbrot set example, the callback class is where we typically would place the code for computing the values of each node that is used to visualize the result.

Creating the Distributed OpenSceneGraph

To create the distributed scene graph a number of issues had to be tackled.

Internal scene graph design The scene graph to be distributed would form the very heart of any application built on top of the toolkit. It should allow for efficient rendering, rapid development of applications and handle the distribution of scene graph nodes as efficiently and transparently as possible.

Communications subsystem To allow for distribution a communications subsystem is needed. Typical non-distributed scene graph API's will not have this functionality so this has to be added to the Distributed OpenSceneGraph.
The communication subsystem should also be designed in a way that interferes as little as possible with the rendering of the graph to preserve a good frame rate. Also, it should be designed modularly enough to allow for easy reconfiguration of communications by allowing for the possibility of using other communications and/or protocols if implemented.

Marshalling and demarshalling of node data members To be able to distribute the state of the scene graph nodes we have to be able to marshal the data stored within the nodes to a format that can be transmitted over the network and demarshalling and updating the corresponding nodes correctly on the receiving end.

Why OpenSceneGraph?

The OpenSceneGraph 3D API [1] [2] [16] was chosen as the basis for this Distributed scene graph after some consideration. The basis for this choice was twofold. For one part a decision was made to use C/C++ as the programming language used in this thesis which limited the choice of basic scene graphs to use as basis somewhat. Secondly the other considered options were reduced to the COIN/Open Inventor API (earlier known as IRIS Inventor) [21] or simply creating a distributed scene graph from scratch.
Creating a distributed scene graph from scratch was initially an interesting idea, however after some consideration it was found that this was a somewhat daunting task for the time given for the project. Using an existing non-distributed scene graph API as the basis for the project would speed up development significantly since its already existing functionality could be used to implement the distribution in a much more efficient manner. Also, using an existing scene graph would also lend its extra functionality to the resulting distributed scene graph, such as reading and writing of file formats, shadowing etc. which would make for a more interesting result.
It was decided against using Open Inventor or the COIN API since it has been used as the basis for other distributed scene graphs like the Distributed Open Inventor API [13] and the blue-c scene graph [14] [22]. Also, the OpenSceneGraph API was also tried and tested own the target display wall used for this project and was thereby considered a safe option.
A point worth noting about the choice between Open Inventor and OpenSceneGraph API's is that OpenSceneGraph started out as a Linux implementation of Open Inventor, or as it was known back then, IRIS Inventor 3D toolkit.

Development approach

Early on it was decided that to be able to test the functionality of each part of the system as it was developed it would be built around a simple test application.

The application consists of a complete scene graph of scene graph nodes that are automatically generated at start up. The actual number of nodes generated is configurable. To keep error checking and debugging simple the scene graph generation function used created nothing more spectacular than coloured and textured pyramid shapes. By doing this reading, and when necessary editing, the serialized state manually was made possible by keeping the amount of generated geometry data low. The function generating the pyramids would also colour the subset of scene graph nodes belonging to the generating hosts differently than the default colour scheme used on the other pyramids generated. This was done to in a simple way, be able to observe the actual distribution and ownership of the scene graph nodes.

To further ease the testing and debugging process the scene graph tree was extended with a HUD branch. The use of this HUD branch was twofold: Firstly it would be used to present the possibility of local variations in each process's copy of the distributed scene graph. Secondly, it was used to display statistical data like hostname, CPU and memory usage and frame rate. While the OpenSceneGraph library actually has functionality for some of this, it was decided to create a personalized solution since it would be able to be adapted to display whatever was felt needed as the needs arose. This HUD would also be coloured in the same colour as the pyramids belonging to the local process to make it easy to identify which node owned what pyramids displayed. Image 8 shows the resulting view of the scene together with a 100 pyramids.

This HUD proved quite valuable under development for among other things detecting memory leaks and immediately showing the impact of changes and approaches. Among others the problems seen when using strings for serialization as mentioned above.

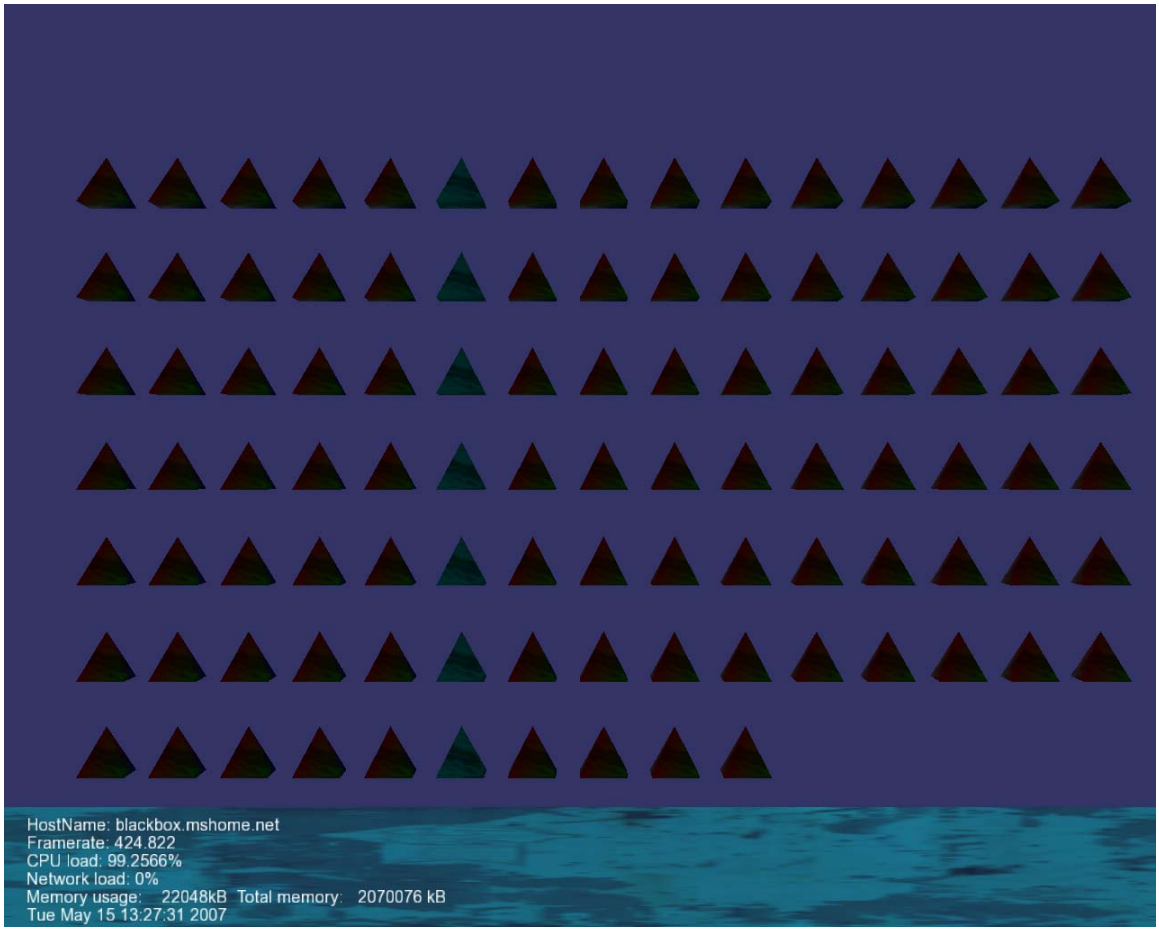


Figure 8 Test application with a generated scene graph of 100 textured and colored pyramids. Note that the pyramids belonging to the process is uniquely colored in the same color as the HUD. The HUD itself contains identity and usage statistics.

Basic operation

This section will attempt to give a high-level overview of how Distributed OpenSceneGraph works.

The very basic idea of this thesis is to create a multi process scenegraph rendering API where each process is responsible for a subset of the scene graph nodes. Each process will then only compute all common nodes (those that are not distributed), any nodes only found in the local scene graph copy and the subset of the distributed nodes in the tree whom the process has responsibility for. The other distributed nodes are updated by receiving their state from their respective owner processes in the system. Figure 9 shows a potential assignment of nodes over the 2D representation of out Mandelbrot set example.

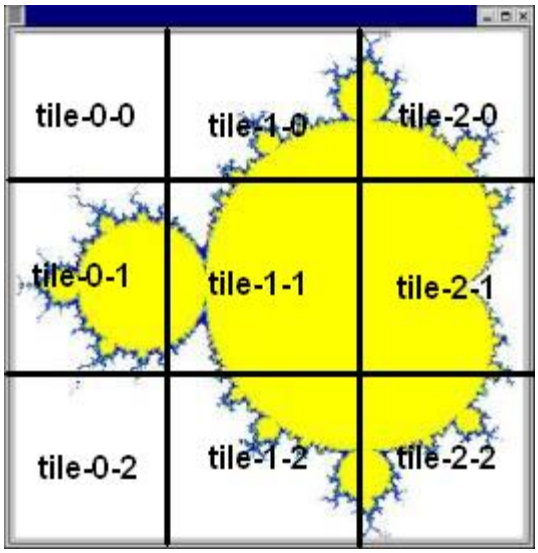


Figure 9 Dividing up the tasks of calculating each point of the Mandelbrot set. Each square is one process's domain. The state of the other domains will be transmitted to each node which subsequently updates its scene graph nodes with the state received. Original image from: http://www.cs.uit.no/inf3201/2006h/assignments/DessertMap.html

With this setup it is realized, that from a distribution viewpoint, we have three node types:

- **Common nodes.** These are nodes that all copies of the scene graph have. They can be static geometry that does not need to be recomputed and therefore are unnecessary to distribute. Nodes that are too lightweight for beneficial distribution, this means that the distribution overhead itself is higher than the potential gain of distributing the node. Finally nodes that behave in a predictable manner per frame.
- **Distributed nodes.** These are the nodes of the distributed classes. Their state will be distributed and updated by the underlying Distributed OpenSceneGraph system.
- **Private nodes.** The Distributed OpenSceneGraph system supports private nodes by not enforcing that the scene graph tree must be identical on each node. The only criterion for the system to work is that the distributed scene graph node with the same identifier is identical as a separate object in each copy of the graph. In effect a process's scene graph does not even have to contain all distributed scene graph nodes. It is completely legal in Distributed OpenSceneGraph to let a process's scene graph only contain a subset of the distributed scene graph nodes.

To achieve this each process of the application must be assigned its own domain and subset of distributed scene graph nodes, and it must know, at least, its neighbouring processes in the total system. I.e. The processes to whom it must transmit state.

When the application knows this it can start up the communications subsystem and commence to run.

The communications subsystem consists of two threads running separately from the scene graph threads. One thread runs a simple server that receives updates from the other processes and the other a client forwarding the updated state.

The true workhorse of the system is a node visitor which has two main tasks:

- To serialize and store the serialized state of the updated distributed scene graph nodes the local process it belongs to has ownership of.
- To update the distributed scene graph nodes whom it does not have ownership of with the stored received state.

The state of all distributed nodes in the scene graph is stored in serialized form within the visitor instance. Both the server and client thread along with the visitor itself works with this data. To prevent simultaneous updates to this data a mutual exclusion mechanism is used. To avoid contention of this data making threads block while waiting for the mutual exclusion to be released all three threads generally works with its own copy of this data and only updating as necessary.

The distributed scene graph visitor will traverse the entire scene graph once per frame looking for distributed nodes. When it finds one it will first check to see if it's a node the process owns.
If it is a node the process owns it will proceed to check if it is marked as DIRTY. A DIRTY marker will indicate that it has been updated and that the state must be collected and sent to the other processes in the system. The visitor will then collects and store the serialized state of the node and continues the traversal.
If it is a scene graph node owned by another process in the system the visitor checks if it has updated state of this node. If so it will pass this state to the node which in turn updates itself with the received state and continue the traversal of the graph.
At the end of each traversal the visitor will update its internal data structures with the updated state before beginning its next traversal.

Focuses of the basic design

The design of Distributed OpenSceneGraph focuses on three main issues:

- **Transparency** By transparency it is meant that as far as possible avoid having to burden the application programmer with the issues of distribution like setting up communication, load balancing and synchronization.
- **Flexibility** Allowing for the distribution of what needs to be distributed while not limiting the application programmer in any other way. This states that non-distributed nodes in the scene graph should be treated as normal. This also allows for local variations of the local copies of the scene graph in the processes in the

applications (for example the HUD containing statistics specific for the single process in the test application).

- **Extensibility** To allow for application developers to extend the Distributed OpenSceneGraph API in an efficient and consistent manner, the responsibilities of the tasks needed for the distribution must be well defined in the API.

Creating the Distributed OpenSceneGraph classes

OpenSceneGraph is an open source API and so the source code is accessible publicly. This makes it possible to modify OpenSceneGraph's internal code to extend its functionality. However in this project it was decided to instead use C++ inheritance mechanisms to add the extended functionality needed by the distribution as was done in *Distributed Open Inventor* [13]. By using inheritance it is possible to add extra functionality to the OpenSceneGraph API without necessarily knowing the details of how it works internally. Also by relying on inheritance of OpenSceneGraph classes extensions can be made without interfering with the functionality of the base classes, thus allowing for transparency and flexibility.

Object responsibilities in Distributed OpenSceneGraph

To achieve the goals mentioned above a number of decisions had to be made before extending the OpenSceneGraph classes for distribution could begin. Extension of the relevant OpenSceneGraph classes was done by putting common distribution functionality into abstract super classes and using C++'s multiple inheritance mechanisms to create distributed node classes. By doing this all that is needed to create Distributed OpenSceneGraph classes is to overload relevant functions and to add whatever extra functionality needed.
To create the Distributed OpenSceneGraph's needed super classes a clear definition of object type's responsibilities was needed. The responsibilities are as follows:

- **Collection of updated local node state.** The Distributed OpenSceneGraph visitor is responsible for checking each node the process owns and requesting the serialized state of any distributed scene graph node marked as DIRTY. The node itself is responsible for actually gathering of stored state and passing this on to the visitor as a single string.
- **Updating distributed nodes with serialized received state.** The visitor is responsible for detecting if updated state of a given node is received and passing this onto the distributed scene graph node. The node itself is responsible for the parsing the received serialized state and the correct updating of its internal values.
- **Serialization of node state** The serialization of the internal data of is the responsibility of the distributed node itself. By doing this the decision of what state to distribute is left to the distributed node thus encapsulating the what and how of serializing state into the distributed scene graph node. The distributed

scene graph nodes are also responsible for setting and clearing its own state flag that the visitor uses to check if it is to be updated.

- **Updating node state with received serialized state.** The node itself is responsible for parsing the received serialized state and updating its internal attribute values. Again, this is done to encapsulate any node peculiarities within the object itself while presenting the visitor with a known interface against the object.
- **Communications server.** This is responsible for receiving the serialized updates disassemble them and passing them on to the visitor.
- **Communications client.** This is responsible for building messages from the total stored state and propagating this onto the other processes in the system.

Table 1 Distributed OpenSceneGraph classes and their spehere of responsibility

Class	Task
DOSGVisitor	Traversing the graph collecting serialized state of updated scene graph nodes this process owns and forwarding received serialized, updated state to scene graph nodes owned by other processes.
DOSGCommunicator	Multithreaded server. Receives distributed node state from neighbouring nodes and passes it on to the DOSGVisitor instance.
DOSGClient	Threaded client used for propagating collected local scene graph node state and received scene graph node state to other nodes.
DOSGDistributor	The distributed scene graph node abstract super class. Contains the distribution functionality needed along with pure virtual function for serializing an updating data that must be implemented in any subclass derived from this.
DOSGGeode	Distributed version of the osg::Geode class sub classed from the osg::Geode and DOSGDistributor classes. Used to group drawables containing geometry data. Responsible for detecting updates to geometry and setting state flag accordingly. Also responsible for serializing state of itself and its attached geometry and for demarshalling serialized state containing updates from other processes.

Serialization of node state

Some mechanism was needed to keep the copies of the scene graphs of all the processes synchronized. A decision was made to make an attempt to marshal the state and not serialize and manipulate the graphics commands such as it was done in Chromium [15]. The reason behind this was to try to avoid the situation where updates are lost because of

communications failures like packet loss. If a serialized operation of the type “add one to all fields in vector myvector” is lost, then this vector will be out of sync with the rest of the system forever afterwards. This can be alleviated by using election algorithms afterwards between the processes to decide on what is the “correct” value for myvector. This however has the drawback of adding extra complexity to the overall application. By sending the serialized state of the objects we avoid this since if an update of myvector is lost it will be eventually corrected by the next received update. By doing this we avoid much of the common synchronization issues and implicitly gain a fairly fault tolerant system.
The price of this approach however is bigger messages and more complex update and serialization traversals.

The task of collecting the serialized updates is left to the DOSGVisitor instance. This node visitor traverses the graph looking for DOSGGeode nodes. When one is found it is checked to se if it is marked as DIRTY. If so the node’s `getGeometryState` function is called. This retrieves all the data of the DOSGGeode and all of its associated geometry nodes and serializes it into a C++ string and returns it to the DOSGVisitor.

The serialization of all data, color, vertex, etc, is serialized by the use of adapted OpenSceneGraph Array- and Value visitors. By overloading the apply methods of these classes they are capable of generating fairly nice, bracketed string representations of the data they traverse. The adapted visitors, DOSGArrayVisitor and DOSGValueVisitor, are simply applied to the return values of the respective get – functions (getVertexArray ( ), getColorArray ( ) and so on). The visitors will then traverse the arrays and retrieve all the data they store. This approach supplies the serialized string not only with what kind of array it is (for example vertex array), but also the array type, the vector types it contains and the type of data stored within along with the values. By bracketing in with type information the demarshalling of the serialized data becomes much easier.

Something that was noted when implementing the marshal traversals of the library was the overhead of generating the serialized state of the nodes. After some investigation it was found that this was in large part due to the overhead of splicing C++ string and more importantly the functions used to convert basic data types, like floats, doubles and integers to strings. After some review of other options it was decided to use C++ ostringstreams instead where it was possible. C++ ostringstreams provided both automatic memory handling and much better performance than strings (see table 2). This even in face of having to convert these streams to strings, thus yielding exactly the same result. In addition ostringstreams also provide much easier conversion of data types into the stream (see table 3) thus avoiding the overhead of using the *stringify* function.

Table 2 Comparisons of strings and iostreams. Measurements made with 100000 splices of char arrays into strings. With and without float and integer conversion.

100000 splices of 649 bytes char arrays.	std::string (using the stringify function from [31] for conversions)	std::ostringstream
Without int and float conversion	0.26 seconds	0.32 seconds (with conversion to string)
Without int and float conversion	0.26 seconds	0.24 (without conversion to string)
With int and float conversion	1.03 seconds	0.43 seconds (with conversion to string)

Table 3 The syntactic difference between using strings and ostringstreams with varied data. The stringify function is a template conversion function from [31]. These two examples both provide an identical std::string as output.

Adding text, integers and floats into a C++ std::string	Adding text, integers and floats into a C++ std::ostringstream
`Stringdata += data + stringify ( 1 ) + stringify ( 1.0f);`	`streamdata << "data" << 1 << 1.0f; //convert to string: streamdata.str ( );`

The format produced by serializing a DOSGGeode with a single geometry node attached can be found in Appendix B.

Implementation of the DOSGVisitor class

DOSGVisitor is the name of the class that forms backbone of each process in the system. It traverses the entire tree once per frame and performs a number of tasks. The tasks include:
Traverse the graph to locate and, as necessary, serialize the state of DOSGGeodes this process owns. Keeping track of most currently updated serialized state of all distributed DOSGGeodes in the entire graph. Both the ones this process owns and others. Using a mutex, controlling the access by concurrent client and server threads to the internal data structure used to store the serialized state of all nodes.

Since a lot of the core application functionality is in the DOSGVisitor a lot of thought has been put into how to keep it as simple as possible. For example the code needed for logging is put into a DOSGVisitorData class which is set as the userData instance of the DOSGVisitor instances if logging is enabled. The DOSGVisitor is also responsible for

setting up communication. This is done simply by instantiating a DOSGCommunicator object that runs in a separate thread and only accesses the serialized node data through the DOSGVisitors get and set functions which uses a mutex to protect the data members from concurrent access.

The serialized data is put into a C++ std::map data structure for easy access. This map structure must be accessed by the server receiving updates as well as the client that forwards state onward to the other nodes in the system. Due to the multiple threads that may want to access this data concurrently, the set – and get – functions for accessing this data all use mutual exclusion. To limit the amount of contention between the DOSGVisitor and the server and client threads over this data the DOSGVisitor actually works on a copy of it. This copy is updated once for each traversal.

The DOSGVisitor also has functionality used for keeping track of frames. This is intended for use of functionality which should be used only once per frame (or n frames, for example logging) and for synchronization with other nodes. In the current implementation however, this counter needs to be updated manually per frame. Future implementations should be made able to do this automatically since keeping track of frames can be critical in many situations.

Communication

For the distribution of the scene graph to be accomplished the updated state of each scene graph node must be received and all local scene graph node state must be propagated to the other processes in the system. This is the task of the communication subsystem.

Basic function

The basic purpose of the communications subsystem is to receive updated state about scene graph nodes the process it belongs to does not own, and to propagate the updated serialized state (both for scene graph nodes it owns and state received about other nodes) to the other processes in the system.

The communication between the processes in the system uses a ring configuration. While this is not necessarily the most efficient solution to propagate state it does have some advantages. Synchronizing the processes within the system is implied since we know that when a process receives an update it sent itself it knows this state has been propagated to all other nodes in the ring. Also, a ring architecture for the communication is very easy to implement. The ring configuration was chosen for the first implementation because of ease of implementation and testing not efficiency. In more stable releases it should be changed to a more efficient architecture like perhaps a broadcast tree which should allow for significantly faster propagation of the state of scene graph nodes.

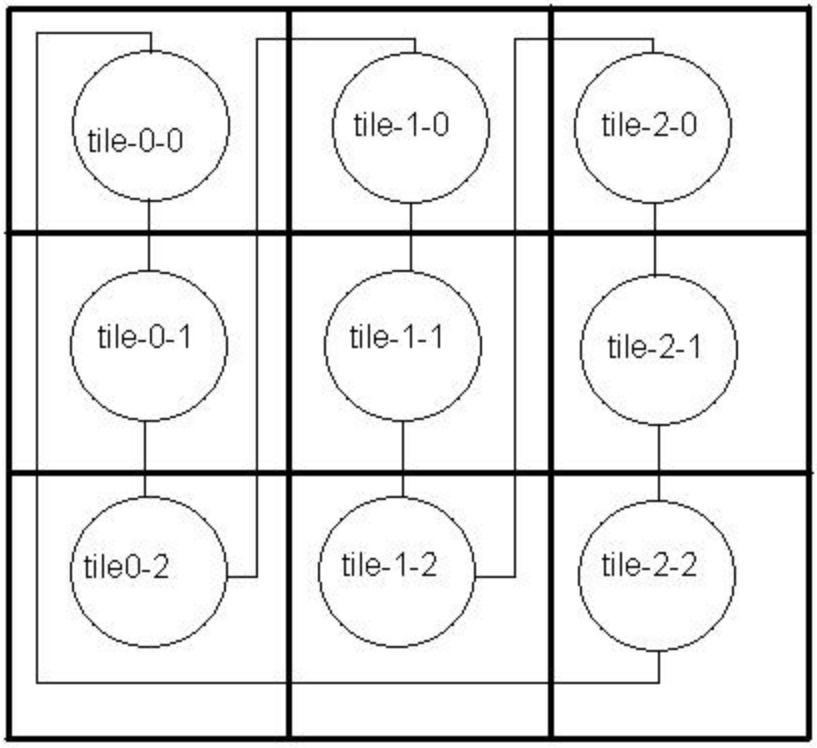


Figure 10 Example process configuration showing the lines of communication between processes in a ring configuration.

When an update message is received it is split up into single node updates and stored in a map structure within the DOSGVisitor instance. Updates in the package containing serialized state about nodes the receiving process owns itself is discarded. This can be safely done since, when a process receives ones own updates, it is implicitly know that those are correctly propagated onto the other processes in the system when using a ring network.

Why socketcc?

Socketcc [25] [26], and the library it is based on, pthreadcc, are both small libraries meant to be C++ wrappers around the basic Linux pthread and socket libraries. These libraries are open source and freely available.

Socketcc was chosen as the communications library because of its object oriented qualities made it well suited for integration with the objects needed for distribution. Furthermore, since it only provides the basic functionality of sockets, threads and locking mechanisms it was also seen to be very flexible and adaptable for the uses in this application.

Other communication libraries considered were MPI [9] [10], which was a fairly tempting option. Its abstraction of the communication basics could most definitely make

this an attractive option to use for the distributed library. However Socketcc was chosen on the basis of it being a more flexible solution to set up in the short run.
ReplicaNet [27] was another option reviewed. ReplicaNet is a game communications middleware. Its features include capabilities to share objects over the network, automatic load balancing and failure recovery, node ownership and ownership transferral and automatic load balancing. ReplicaNet is however a closed source library and a little tricky to get into using when compared to socketcc. Furthermore ReplicaNet is a library primarily meant for applications like networked games and massively multiplayer online games which often do not have quite as strict synchronization issues as a display wall where you can see all nodes simultaneously. Faced with ReplicaNets learning curve and being unsure if it would be capable of propagating updates fast enough socketcc was preferred.

The DOSGCommunicator class

The DOSGCommunicator is based on the pthreadcc and socketcc libraries [26]. It is a multithreaded TCP Client/Server which handles all communication to and from the process in which it belongs.

When the DOSGCommunicator starts it starts up a simple multithreaded TCP server and listens for connections. When an incoming connection is accepted a DOSGServiceClient class instance is instantiated to handle the request.
For efficiency connections are kept alive to avoid having to re-establish them for each new connection.
If for some reason the connection is lost or the socket is somehow made invalid a new connection is established at the next attempt to send data to the given process.
This version basically is only intended to have one single process sending messages to it, however the solution implemented has no problem with handling multiple connections without modification.

The data is marshalled into a XML – like format by the marshalling mechanisms of the system and sent as ASCII in the messages. This approach, together with the decision of using TCP as choice of protocol, was chosen by the need for ease and speed of implementation by reducing complexity of the first version of this system, as well as ease of testing and debugging. An attractive feature worth mentioning of the ASCII approach of the socketcc library TCPServer class was its semi-automatic memory handling where memory is allocated automatically by the receiving DOSGServiceClient instance. The only issue with this was to remember to delete the allocated memory when no longer needed to avoid memory leaks.

It should be possible to improve performance of the system by using a different communications scheme. Using UDP might improve performance somewhat. However considering that the computers running the display walls this system is meant to run on usually are interconnected by high-speed network systems (often Gigabit), this gain

should be marginal at best. And considering the added application complexity often associated with using UDP this option was discarded during development.

As shown in figure 11 both the client and the server runs in threads separate from the DOSGVisitor instance and operate one the data within the visitor through its public interface. The advantage of this is firstly that the sending and receiving of messages does in as little degree as possible hamper the speed of the visitor's traversals and thereby the frame rate of the application as little as possible. Secondly, by threading the server and client and only letting it update the DOSGVisitors data through a known interface, makes it relatively easy to add new or change the existing communications subsystem in the future. For example creating and using a UDP implementation simply involves creating new client and server classes and implementing the UDP specifics within the. To use the new class simply pass it a pointer to the DOSGVisitor and use its interface when receiving and updating data.

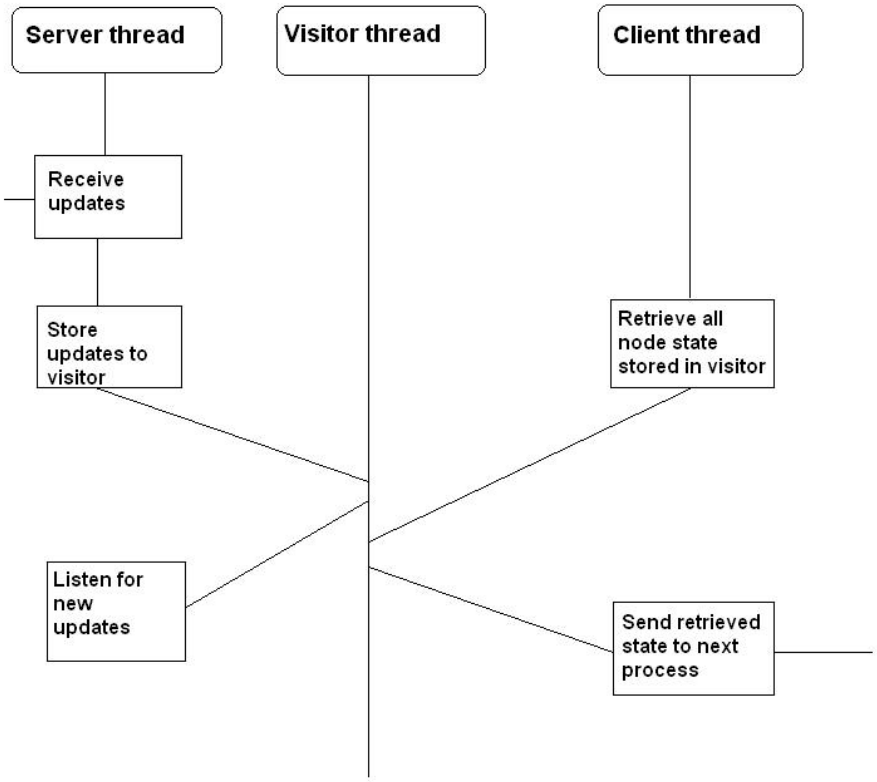


Figure 11 The threaded communication scheme of Distributed OpenScenegraph. The function calls presented to the client and server uses a mutual exclusion mechanism to protect the data from concurrent access.

The DOSGArrayVisitor and DOSGValueVisitor classes

During the development of the marshalling and demarshalling of scene graph node state one of the issues expected to generate the most headache was the retrieval and serialization of state. This was mainly because of the fact that a lot of the state stored in, for example, the geometry nodes is stored in arrays of vectors and it was imperative that everything was tagged and stored in the correct order when serialized otherwise the demarshalling process would yield the wrong results.

After some consideration and reviewing of options it was realized that OpenSceneGraph actually must contain such functionality since it is capable of storing a scene as .osg files, which, in fact, are ASCII files.

After some searching it was found that it indeed had the classes ArrayVisitor and ValueVisitor which can be used to traverse the arrays they are applied to and retrieve the values stored there within.

By overriding these two classes and overloading their apply methods it was made possible to not only make these visitors retrieve the data stored but also correctly bracketing the data with the correct data types indicated.

These two utility classes greatly simplified the process of marshalling the stored state within the scene graph nodes. In particularly the geometry nodes since they potentially store a lot of array data.

Extending scene graph node classes for distribution

What OpenSceneGraph node classes to extend?

OpenSceneGraph has a vast number of different classes and deciding which of these to begin with to extend for distribution was no trivial task. The osg::Geode class was chosen as a starting point since this class is used to group Drawable instances which in turn store the actual geometry data such as vertices, texture coordinates and bindings.

The DOSGDistributor class

To limit amount of code needed to extend the classes in the OpenSceneGraph library C++'s double inheritance mechanism was used. Any extended class should also inherit from a pure virtual DOSGDistributor class. The DOSGDistributor class was implemented to include all typical functions needed for distribution. This class includes core functionality to manage identifiers, owner ids and maintaining the state of the object (CLEAN, DIRTY or UPDATE). This class also includes the pure virtual member functions getStateAsString and updateStateFromString which each class inheriting from this super class must implement. This function must be pure virtual since any extended

classes will usually not have the same members and attributes and therefore must implement their specific marshalling and demarshalling mechanisms.

Extending the osg::Geode class

The OpenSceneGraph library is by all standards well designed and fairly well documented so the process of extending the osg::Geode class to become the DOSGGeode class was fairly straight forward. Using the multiple inheritance mechanisms of C++ this class is based on both the osg::Geode and DOSGDistributor classes. It overloads all set, replace and removal functions. If any of these functions are called on a DOSGGeode instance the operation will perform as with an osg::Geode but in addition the instance will be marked as DIRTY. This will be detected by the next traversal of the DOSGVisitor which then subsequently will call this node's getStateAsString function and retrieve the serialized state of the instance.

At time of writing the state of a DOSGGeode is defined as its own state (i.e. Identifier, name, number of drawables etc.) and the state of its associated osg::Geometry instances. It may have other types of Drawables attached but these are currently ignored. It was felt that for testing purposes it would suffice with only retrieving the state of the osg::Geometry nodes since these are used to store the actual geometry data.

When the getStateAsString function completes with the returning of the serialized state the DOSGGeode sets its state to CLEAN.

A weakness of this setup is its inability to detect if the arrays containing the vertex data, color data and so on in its associated Drawables are changed directly. Only changes using the overloaded functions will be detected. For the purposes of testing this is acceptable since it can be controlled how the updates take place and thereby controlling the updates as necessary is possible. For a truly usable library however this is not good enough. There are however ways to solve this problem. First of, a distributed version of the osg::Geometry class can be implemented. This could first of all overload all set, replace and get functions using these to detect what arrays are being changed. Secondly mechanisms like a hash could be used to check the current content of an array with a stored reference from the last time the node was checked. These two approaches has the potential of greatly reducing the amount of data needed to be serialized and distributed thus, at least theoretically, giving a boost to the overall performance of the application.

Another current limitation is the fact that the DOSGGeode will retrieve the state of all of its Geometry nodes every time it detects that it is in the DIRTY state. This is in many cases unnecessary and should be avoided. The system can easily be modified to detect if a new Drawable is added, removed or replaced but to detect changes internally in the Geometry nodes we will need either a mechanism in the DOSGGeode that is capable of checking if there has been made changes to the data stored in the Geometry instance. Something like the hash approach mentioned above could be used. However this is likely to create a fairly large overhead since a lot of state must be serialized to have a string to

make the hash of. A better solution here, again, seems to be to create a distributed version of the Geometry class as well. This would in a greater degree be capable of checking and detecting if it has been altered and set the DIRTY flag on itself. Then the DOSGGeode instance would only need to check the flag to see if it needs to serialize the stored geometry.

Synchronization

The current version does not possess any synchronization mechanisms. However during development a number of options were reviewed.

A simple synchronization mechanism could be implemented by tagging the node updates with a frame stamp. This could be a simple epoch – frame count tag. An epoch should be used to guarantee that the frame count does not run out of bounds. This would enable each process to check what frame each node update belongs to. By using these frame stamps a process can find out if it is running ahead of the other nodes. When this is detected the fastest nodes can gradually slow down until the slower processes catch up. Using this approach and making the faster processes slow down instead of stopping and waiting would make for a more smooth synchronization.
Also using this approach it could be made configurable by passing a variable describing how unsynchronized we can let the processes become. In an application like the Mandelbrot example and when displaying results of computations we probably would want each single frame to be synchronized. But in other situations the processes could be allowed to run somewhat out of sync to allow for better frame rates.

Creating and removing nodes at runtime

The process of removal and creating of nodes at runtime, perhaps as a result of user actions, has not been implemented in this version. However a lot of thought has been put into the idea and some potential solutions has been reviewed.

First, the removal of a scene graph node could be implemented by creating new classes by inheriting from the osg::Group and its derived classes and overloading the removeChild, removeChildren and replaceChild functions. What the overloaded functions need to do in addition to the work of the original functions is to replace the data value in the DOSGVisitors map of serialized DOSGNodes with an identifier that indicates that this node is deleted. It should then be possible, with little effort, to add the necessary functionality to the demarshal functions to simply make sure the copy of the deleted node is deleted as part of the updating of the scene graph on the other processes. In this manner the deletion will be propagated to all processes will delete the node within reasonable time.
To allow for the situation where the same node is deleted simultaneously on all processes, the indicator of the node being deleted should be kept stored in the DOSGVisitor data structure. The price of this would be an insignificant amount of

memory. By keeping this deletion indicator we will avoid the situation where an update that was in transit when the call to delete the scene graph node arrived is received *after* the deletion has taken place brings the deleted node back to life. By keeping the deletion identification of the node in place an attempt to make a new update of the node can be detected, discarded and stopped from being propagated to the next node in the ring.

The runtime creation of new, distributed, scene graph objects is possible but not directly supported in this version. This could be better supported by implementing a class factory design pattern [11]. Since all OpenSceneGraph classes have the className function that identifies the class used, it should be reasonably simple to implement a class factory into this solution. This would enable a developer to create a new object on a single process and simply let it be propagated onto the rest of the processes as a regular update. When such an update is received it could be detected that a node with this identifier does not exist (or has previously existed) and the class factory can be called with the appropriated values.

Naming distributed scene graph nodes

For the distribution to work at least each node in the tree that is to be distributed must have a unique identifier associated with it. This is necessary for each node in the system to be able to update the correct scene graph nodes with the correct data received. An identifier for this system need to have the following two properties:

1. **The identifier of a scene graph node must be unique in the sense that no other node in the scene graph tree can have the same identifier.**
2. **The identifiers given to a scene graph node must be identical for the same node in each node the application runs on in the cluster.**

The first point is obvious. The whole point of the identifier is to be able to locate one and only one scene graph node in the scene graph. Without this there is no way of knowing which serialized data belongs to which node when it is to be updated.
The second point is a little trickier. When the serialized updates are received the data must be demarshalled and the correct node updated with the received data. For this to work the data must include some identifier pointing the system to the correct node. This implies that there must be either an identifier for that scene graph node that is common for all the cluster nodes copies of the scene graph or there must exist a mapping scheme between the identifier given to the scene graph node on one computer to the identifier given to the same scene graph node on another cluster node's scene graph copy.
For this revision the first approach was chosen since it promised the least amount of complexity to the application. However this approach has a few challenges of its own. Generating a unique ID for each node is quite possible with little effort. However making sure the same unique ID is generated for each copy of the distributed tree in the application is however not. Using randomizer or time functions will surely give different

ID's for the scene graph nodes in each process so this common implementation approach cannot be used.
Instead a suggested approach can use an adapted node visitor to generate an identifier for each DOSG node. This visitor could, for example, generate the identifier as a string containing a string representation of the path to the node, plus the node's child position in the immediately above group node plus the frame number when it is created. This frame number will be zero on all processes when the nodes are generated at start up of the application.

To limit the size of the node identifiers when creating them this way we could generate a hash of the identifier and use it as the node's ID.

This should work since these parameters should be identical for the trees of all the processes. It is neither necessary to update this identifier if the tree structure changes at runtime since involving the frame count will prevent a new node, even in the event of a new node replacing an existing in the exact same position in the tree, from getting the same node identifier.
One issue with this approach is however the situation where we want to create a new node at runtime. If this is created at all nodes at the same point in the application there is no guarantee that this will happen at the same frame count in the face of a relaxed synchronization between the nodes. This implies that the nodes will receive different identifiers which in turn will thwart the updating of the nodes.
There are at least two different approaches to prevent this situation. One is to prevent the creation of the new node until a given frame. If the system is reasonably synchronized this should work since it will make sure the frame count is identical on all nodes.
Another solution is to give the task of creating the new scene graph object to a single node. What should happen then, if the system allows creating new nodes using, for example, the class factory approach, is that the usual update sequence will take care of the creating of the node on all processes in turn. This should work since then the copy of the node created on the other will have the same identifier as the original.

All classes inherited from DOSG_Distributor class will all have a _nodeID data member. This data member is a simple string used for storing a unique identifier for the DOSG nodes created.
In future implementations it could be generated automatically using one of the approaches mentioned above but at present the creating a suitable identifier for each DOSG node is left to the developer. Applying an identifier to each node is nonetheless required for any distribution to work. The reason any identifier generation was not implemented for this version is simply that it was not immediately necessary since testing could do without. The testing applications generated the DOSGGeode nodes using for – loops and the counters for these loops were used as identifiers.

Distributed OpenSceneGraph programming guide

Since a lot of work on this project has gone into make the distribution as transparent as possible programming with Distributed OpenSceneGraph isn't all to different than programming with the standard OpenSceneGraph API. However for distribution there are still some issues a Distributed OpenSceneGraph developer must deal with:

- Splitting up the tasks of the problem at hand between the Distributed DOSGGeodes by creating node callbacks and passing them to the DOSGGeodes.
- Instantiating and running the DOSGVisitor.
- Instantiating the DOSGCommunicator and passing it a pointer to the DOSGVisitor instance.
- Instantiating the DOSGClient instance and passing it a pointer to the DOSGVisitor instance.
- Generating and passing frame counts to the DOSGVisitor once per frame.
- Generating and setting ID's for the distributed scene graph nodes. These must be unique for the graph but with the added restriction that they must correctly identify the same node on each process's graph. Beyond that the id can be anything and can be chosen as the developer sees fit.
- Choosing and setting ownership on each distributed node. This is what actually decides which process that actually updates the node.

Distributing the Mandelbrot set

The Mandelbrot set [33] is what is known as an embarrassingly parallel computation [10]. It can easily be split up between processes and calculated without passing data between them. A sequential C code example for solving this problem can be found in appendix C.
For our example we want to visualize the result with cubes where the Z coordinates represent the value of the calculation. The result could, for example, look like something as shown in figure 12.

The tasks needed to distribute this problem are as follows:

1. Split the total problem domain into pieces for each DOSGGeode to work on.
2. Create callback functions that will recalculate the geometry used to visualize the result per frame. The callback functions are where the true calculations actually take place.
3. Instantiate the nodes of the scene graph passing the relevant parameterized callback functions to each scene graph node.
4. Generate the ID's for the distributed nodes.
5. Set the ownership of each distributed node.
6. Instantiate the DOSGVisitor instance attach it to the root node of the graph.
7. Instantiate the DOSGCommunicator instance passing it a pointer to the visitor..

8. Instantiate the DOSGClient instance passing it a pointer to the visitor.
9. Run the main loop of the system.
10. For each frame update a counter indicating the frame number and pass this to the visitor.

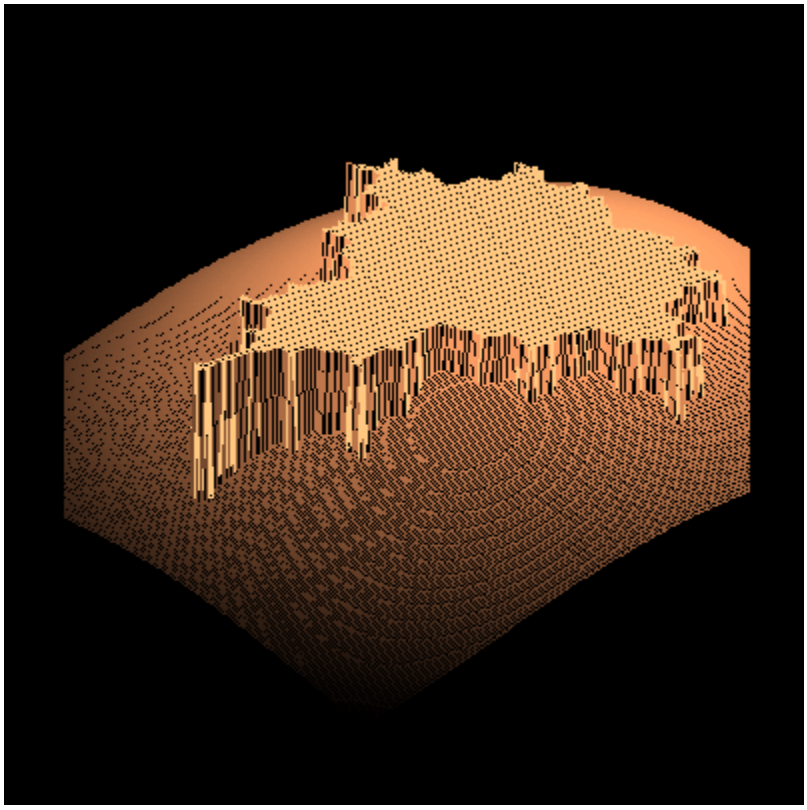


Figure 12 3D view of the mandelbrot set generated in Matlab.
http://www.ciram.unibo.it/~strumia/Fractals/FractalMatlab/Trid.html

The first task for the developer is to split up the problem into domains for each scene graph node. Using the example of the Mandebrot set this can be done in a loop generating start x and y values for each instance to use. This loop is typically also where you would want to instantiate both the callback and the distributed scene graph nodes they belong to.

```
for ( int x = 0; x < max_x; x++ )
{
    for ( int y = 0; y < max_y; y++ )
    {
        mycallbackclass nodecallbacks [ x*y ] = new mycallbackClass ( x, y );
        DOSGGeode theDistributedNodes [ x*y ]= new DOSGGeode;
        theDistributedNodes[ x * y ]->setCallBack ( nodeCallBacks []);
    }
}
```

Listing 3 Example loop instantiating callback and distributed scene graph nodes

When it has been decided how to split up the problem domains the callback classes must be implemented. There is nothing special about these; this is done exactly as you would in any OpenSceneGraph based applications. The only difference is that the callback must be instantiated with the correct startup parameters for its domain. An example of how to do this is shown in listing 3. The callback class should contain the actual code of the application. In the case of the Mandelbrot set the code to calculate the Z parameters of the cube used to visualize the result.

The next task is to generate and set the ID of each distributed node. This is usually something you would want to do in the same loop as above. The API does not define any restrictions about what the ID should be. As long as it is unique for the scene graph and identifies the same node on each processes copy of the graph you can use whatever ID you like. Listing 4 shows an example of how to generate ID for the nodes in the same loop as the callbacks and DOSGGeode nodes are instantiated.

```
for ( int x = 0; x < max_x; x++ )
{
    for ( int y = 0; y < max_y; y++ )
    {
        mycallbackclass nodecallbacks [ x + y ] = new mycallbackClass ( x, y );
        DOSGGeode theDistributedNodes [ x + y ]= new DOSGGeode;
        theDistributedNodes[ x + y ]->setCallBack ( nodeCallBacks []);
        theDistributedNodes [ x + y ]->setID ( hostname + "." + stringify ( x ) + "." + stringify ( y ) );
    }
}
```

Listing 4 Example loop showing how to generate node ID's for the distributed nodes in the Distributed OpenSceneGraph application

The ownership of the nodes must also be set. The DOSGGeodes ownerID will be compared to the hostname of the computer the hostname runs so it should be set to the value found in the `/proc/sys/kernel/hostname` file in on Linux systems. To allow for this to be set in a setup loop as shown above the DOSGGeodes can also be set with a numerical ID. This ID can in turn be used to set the ID on a selection of nodes as shown in figure ###. Note that the owner ID strictly only needs to be set for the distributed nodes that the process itself owns. The system does not need to know the actual

ownership of the other nodes in the system. Example code of how to do this is shown in listing 5

```
for ( int x = 0; x < max_x; x++ )
{
    for ( int y = 0; y < max_y; y++ )
    {
        mycallbackclass nodecallbacks [ x + y ] = new mycallbackClass ( x, y );
        DOSGGeode theDistributedNodes [ x + y ]= new DOSGGeode;
        theDistributedNodes[ x + y ]->setCallBack ( nodeCallBacks []);
        theDistributedNodes [ x + y ]->setID
                ( hostname + "." + stringify ( x ) + "." + stringify ( y ) );

        if ( x == hostData->getHostUIntID ( ) )
            theDistributedNodes [ x + y ]->setOwnerID ( hostname );
    }
}
```

Listing 5 Example configuration loop showing how to set the owner ID with the help of the distributed node's getHostUIntID function.

When the distributed scene graph nodes has been properly instantiated and configured the next task is now to create a DOSGVisitor instance. As mentioned, this can be done either by having it create a default DOSGCommunicator instance or passing it a pointer to an existing instance. Se listing 6 for details

```
//Create the DOSGVisitor with logging enabled
DOSGVisitor *_myVisitor = new DOSGVisitor ( true );
```

Listing 6 Instantiating a DOSGVisitor

The next step is to create the DOSGCommunicator instance. You can also configure it to use another port number if another port than the default TCP port 25000 needs to be used.

```
DOSGClient *_client = new DOSGClient ( nexthost, _myVisitor );
DOSGCommunicator *_coms = new DOSGCommunicator ( nexthost, _myVisitor );
_coms->setServerPort ( 26000 );
```

Listing 7 Instantiating and changing port numbers to a DOSGCommunicator instance.

```
DOSGVisitor *_myVisitor = new DOSGVisitor ( false );
DOSGClient *_client = new DOSGClient ( nexthost, _myVisitor );
DOSGCommunicator *_coms = new DOSGCommunicator ( nexthost, _myVisitor );
```

Listing 8 Passing the DOSGCommunicators the DOSGVisitor pointers.

The final step is to run the actual OpenSceneGraph main loop. In this version this involves also updating the DOSGVisitors traversal number for synchronization. The visitor must be attached to the root node of the scene graph and run once per frame to function correctly. This is shown in figure ###

```
int count = 0;  //frame counter
while ( !viewer.done ( ) ) {

    viewer.sync ( );
    viewer.update ( );
    viewer.frame ( );
    //Set start point for traversal
    root->accept ( *_myVisitor );
    //update _myvisitor with counter
    _myVisitor->setTraversalNumber ( count );
    count++;    //increment counter
}

viewer.sync ( );
```

Listing 9 Example main loop that runs the DOSGVisitors traversal each frame and updates the traversal number

Reducing distribution overhead by using OpenSceneGraph LOD nodes

One way of reducing the computational overhead of a scene using distributed nodes could be using OpenSceneGraph Level Of Detail (LOD) nodes. These nodes choose one of its child nodes based on its distance from the eye point in the scene. Since the DOSGGeode nodes are inherited from the osg::Geode class it can be treated as such when used with LOD nodes. A LOD node stores its child nodes with a reference of their max and min distance which is the distances from the eye point where it is to be seen. This list is also used to limit computations at times of high loads. Se table 4 for an example of a LOD node child list

Table 4 Example LOD node child list and the effect on rendering. Note that this list is also used to limit load in high load situations.

LOD child no.	Node	Min distance	Max distance	Result
3	Simple untextured cube	10000	100000	Simple cube no distribution of state
2	Simple textured cube	3000	10000	Fair detail no distribution of state
1	Textured DOSGGeode	1000	3000	Good detail and distribution of state
0	Multi textured DOSGGeode	0	1000	Highest detail and distribution of state

For our Mandelbrot set example where Z values must be evaluated per frame it is worth noting that the computations of Z coordinates must continue even in the situations where the non-distributed children of the LOD node are chosen. To achieve this, the same instance of the callback class must be passed to all the child nodes of the LOD node. Even to the non-distributed ones. The result of this will be that at the larger distances from the eye point the cubes that present the result of the scene graph node's calculations will not be updated. At ranges where this can barely, or not at all seen this would be acceptable. When the eye point moves towards the scene graph nodes they will again be updated with the correct state.
It should however be noted that this is however untested at time of writing.

Extending classes for Distributed OpenSceneGraph

The types of nodes which have been extended for distribution is at present the bare minimum needed for testing purposes. However Distributed OpenSceneGraph has been created to be as adaptable as possible when it comes to adding new classes. By using the C++ multiple inheritance mechanism an attempt is made to lessen the work of distributing nodes to the minimum of handling scene graph node peculiarities.

When extending an OpenSceneGraph node class for distribution, it is important to be aware of the responsibilities the created distributed node has in the Distributed OpenSceneGraph system. These responsibilities are summed up as:

- A distributed node must be capable of detecting changes to its own state and to set its DIRTY flag accordingly.
- A distributed node is responsible for the correct marshalling of its internal state into a string.
- The distributed node is responsible for correctly parsing the type of string it generates through the marshalling process and to demarshal it into the correct values and update its state accordingly.

The actual format of the string generated by the marshalling process is completely left to the developer extending the distributed class. The DOSGVisitor traversing the scene graph tree handles the outer bracketing of the serialized node data and makes sure its identifier is correctly bracketed at the start of the finished serialized state string. The data between the brackets added by the DOSGVisitor only needs to make sense for the copies of that scene graph node in the other processes in the systems scene graphs. For the rest of the system it has no relevance how this data is formatted. It does not care nor does it at any point need to look at it.
Further what state the extended class wants to distribute is also left to the developer. Creating a distributed osg::Geode that only distributes its texture coordinates is entirely legal by this system.

So the process of extending new classes for distribution boils down to the following:

- Using multiple inheritance inherit from both the DOSGDistributor and the relevant OpenSceneGraph class.
- Use the OpenScenegraph Meta_Node macro to generate the standard isSameKindAs, className, libraryName, clone and accept methods. See listing 10 for how to do this.
- Decide what state that should be distributed.
- Decide for a format of the serialized data. Using the bracketed approach that the rest of the system uses is recommended but not necessary.
- Implement the ###getStateAsString and setStateFromString functions.

```
/**
 * META_Node macro define the standard clone, isSameKindAs, className and accept methods.
 */
META_Node ( DOSG, DOSGGeode );
```

Listing 10 Using OpenSceneGraph Meta_Node macro to define standard functions.

A tip when deciding how to serialize the data of a class: The DOSGArrayVisitor and DOSGValueVisitor are classes implemented just for this purpose. They can be used to create bracketed strings from the data values arrays that commonly make up the state of the nodes. Using these will serialize the state of the array they traverse returning nicely bracketed state of the arrays and values therein complete with data types. Figure ### shows the usage of the DOSGArrayVisitor class and figure ### shows example output from the DOSGArrayVisitor. Internally the DOSGArrayVisitor uses an instance of the DOSGValueVisitor to retrieve the actual values stored.

```
osg::Array *vertices = geom->getVertexArray();
if ( vertices != NULL )
{
    DOSGArrayVisitor nv;
    vertices->accept ( nv );
    data << "<vertices>" << std::endl
    << nv.getData ( )
    << "</vertices>" << std::endl;
}
```

```
<vertices>
<Vec3Array>
<osg::Vec3>
<x>0</x>
<y>0</y>
<z>0</z>
</osg::Vec3>
<osg::Vec3>
<x>10</x>
<y>0</y>
<z>0</z>
</osg::Vec3>
<osg::Vec3>
<x>10</x>
<y>10</y>
<z>0</z>
</osg::Vec3>
<osg::Vec3>
<x>0</x>
<y>10</y>
<z>0</z>
</osg::Vec3>
<osg::Vec3>
<x>5</x>
<y>5</y>
<z>10</z>
</osg::Vec3>
</Vec3Array>
</vertices>
```

Listing 11 Using the DOSGArrayVisitor class to serialize data. The text on the left is the bracketed result generated by this code and the DOSGArrayVisitor's traversal.

Currently, to detect distributed scene graph nodes the DOSGVisitor traverses all nodes an checks for the correct class names. For example: DOSGGeode. If it finds a class with the correct class name it checks to see if this process is the owner of the node or not. If it owns the node it commences to check if it is marked as dirty and if so retrieves the serialized state. If it is not the owner of the node found it checks to see if it has update state for it. And if so passes this to the scene graph node.

```
//Check if this is a node of the type we're looking for
if ( searchNode.className ( ) == "DOSGGeode" )
{

    //We now know that this is a DOSGGeode instance. Downcast the
    //pointer to the correct type.
    osg::ref_ptr < DOSGGeode > thisDOSGGeode =  dynamic_cast < DOSGGeode* > ( &searchNode );

    /**
     * Check to se if this is a node we are to read or update.
     * ( If I'm the owner. Read state, otherwise see if we have to update state )
     */
    if ( _hostName == thisDOSGGeode->getOwnerID ( ) )
    {
        /**
         * I own this node so we have to check if it is in
         * need of serializing ( i.e. check to see if dirty )
         */
        if ( thisDOSGGeode->getState() == DIRTY )
```

Listing 12 The current code checking for nodes marked DIRTY in the DOSGVisitor class. Note that it checks for a specific class name.

A weakness in the current code, shown in listing 12, is that it checks directly for the class name of the nodes it inspects. Therefore to introduce new distributed class types this code needs to be updated.

In future releases this will be changed to instead only checking for the correct library name of the class and downcasting it to its super class. This should be possible since all the function calls the DOSGVisitor uses are declared in the super class all distributed scene graph nodes must be inherited from. Such an approach would eliminate any need for developers who wish toe extend classes for distribution to change the source code of the DOSGVisitor.
This change would imply that a given library name (in our case, “DOSG”) must be enforced for all classes who intend to be discovered by the DOSGVisitor.
For now however, developers who wishes to extend their own distributed node classes must add their own check in the DOSGVisitor’s source code. An example of this code is shown in Listing 13.

```
//Check if this is a node of the type we're looking for
if ( searchNode.className ( ) == "foo" )
{

    //We now know that this is a DOSGGeode instance. Downcast the
    //pointer to the correct type.
    osg::ref_ptr < foo > thisfoo =  dynamic_cast < foo* > ( &searchNode );

    /**
     * Check to se if this is a node we are to read or update.
     * ( If I'm the owner. Read state, otherwise see if we have to update state )
     */
    if ( _hostName == thisfoo->getOwnerID ( ) )
    {
        /**
         * I own this node so we have to check if it is in
         * need of serializing ( i.e. check to see if dirty )
         */
        if ( thisfoo->getState() == DIRTY )
        {
            /**
             * Is dirty. Get state and store as string
             * in the _serializedNodes map.
             */
            data << thisfoo->getStateAsString ( );
            _mapMutex.Lock ( ); //Lock here to access _serializedNodes
            _serializedNodes [ thisfoo->getNodeID() ] = data.str();
            _mapMutex.Unlock ( );

        }
    }
}
```

Listing 13 Code that needs to be added to the DOSGVisitor's apply method to detect and retrieve updated serialized state.

Related work

Common graphics and scene graph API's

As mentioned earlier graphics toolkits can be described as either being *immediate-mode* or *retained-mode*. By immediate mode it is meant that the graphics objects, their translation and other state are created and sent to the graphics pipeline as the main thread of execution passes their point in the code.
With retained mode it is meant that the graphics objects and their state is stored in advance in an internal, often hierarchical data structure and rendering is done by letting an algorithm traverse this structure.

- **Direct3D.** Being a part of the Microsoft DirectX API this is only available for Microsoft Windows systems. It is also the base API on Xbox and Xbox 360 consoles. The current version of this API is Direct3D 10.
  Direct3D is built over two big APIs. An *immediate mode* and a *retained mode* API.
  The *immediate mode* API provides access to the graphics hardware's 3D functions like lighting, clipping, textures and materials.
  The *retained mode* API of Direct3D is built over the *immediate mode* API and provides higher level functionality like hierarchies and animation.
  http://www.microsoft.com/windows/directx/default.mspx
- **OpenGL.** Originally developed by SGI in 1992 OpenGL is now a standard specification defining a cross-platform 3D graphics development API.
  OpenGL is cross-platform and implementations exist for many platforms including Microsoft Windows, Mac OS, Linux, many UNIX distributions and even Playstation 3.
  - OpenGL is aimed at serving two purposes:
    To hide the complexities of interfacing with different hardware accelerators by presenting the developer with a single, uniform API.
  - To hide the differing capabilities of different hardware platforms. This is done by requiring that all implementations must support the full OpenGL feature set using software emulation if necessary.

  http://www.opengl.org/
- **SGI Open Inventor.** Formerly known as IRIS Inventor and based on the OpenGL API. Developed by SGI around 1988-89 as a toolkit to reduce the effort needed when developing 3D applications and becoming open source in 2000. The main focus of Open Inventor was ease of use before performance and does this by hiding a lot of the complexity of writing OpenGL applications. Open Inventor uses a scene graph to structure the objects in the scene and apply culling.
  http://oss.sgi.com/projects/inventor/
- **SGI OpenGL Performer.** Formerly known as IRIS Performer it came about around 1991. Performer is a commercial utility library built on top of OpenGL for the purpose of enabling hard real time applications. OpenGL Performer was

originally developed by many of the same people who developed Open Inventor who wanted a library with a greater focus on performance than it had been in Open Inventor.
The API is focused around a scene graph that can be rearranged on the fly for performance reasons. This allows various necessary rendering passes to be executed in parallel in multiple threads. Performer also lets the developer to rank objects in the scene by importance. This enables Performer to cull less important objects to maintain a good frame rate in high load situations.
Open Inventor is primarily built around two libraries. One lower level that provides an object oriented interface to high-speed rendering functions and a higher level library that allows for scene graph management, scene processing, level of detail management and so on.
http://www.sgi.com/products/software/performer/

- **Coin3D.** This is a clone API based on the popular SGI Open Inventor library. It is a complete rewrite of Open Inventor and as such shares none of its code with Open Inventor. However it implements the same API for compability reasons and the current version is fully backwards compatible with Open Inventor 2.1. Since the Open Inventor API has reportedly fallen into obscurity by SGI showing little commitment to it, Coin3D is currently the choice for those familiar with the Open Inventor API.
  http://www.coin3d.org/
- **OpenSceneGraph.** This is the toolkit on which the Distributed OpenSceneGraph was built. It is a high performance 3D graphics toolkit that started off around 1998 as a hobby project trying to port a hang gliding simulator written on top of Performer to Linux. The OpenSceneGraph library is written entirely in C++ and OpenGL and runs on all Microsoft Windows platforms as well as GNU/Linux, Mac OS, IRIX, Solaris and FreeBSD. The current, stable version of OpenSceneGraph is 1.2 with version 2.0 in the works.
  www.openscenegraph.com
- **Java3D.** Starting out as a collaboration between Intel, SGI, Apple and Sun this is the scene graph oriented API used for 3D development on the Java platform. The first stable version was released in 1998. After being discontinued for a year it was further development of this API was started up again in 2004 as a community source project.
  It runs on top of either Direct3D or OpenGL. When compared to other solutions Java3D is not only a wrapper around the functionality of its underlying API (Direct3D or OpenGL), but an interface that encapsulates the 3D programming in a real object oriented concept. Being a Java library this solution is as platform independent as Java itself is.
  http://java.sun.com/products/java-media/3D/
- **OGRE.** The name is an abbreviation for Object-oriented Graphics Rendering engine. This is a scene oriented and flexible 3D game engine written in C++. It is designed to make it easier and more intuitive for developers to produce games and demos utilizing 3D hardware. The class library abstracts all details of using the underlying system libraries like Direct3D or OpenGL and provides an interface

based on world objects and other intuitive classes.
http://www.ogre3d.org/

- **The Visualization Toolkit.** This is an open source, freely available cross platform software system for 3D graphics, imagery and visualization. VTK consists of a C++ library and several interpreted interface layers including Tcl/Tk, Java and Python. It also features a vide variety of visualization algorithms. In contrast to the Scene Graph oriented API's VTK uses a pipeline approach.
www.vtk.org

Existing distributed rendering and scene graph solutions

A lot of work has gone into solutions for distributed visualization and distribution of scene graphs. This section lists a number of these.

- **Chromium.** [15] this is a rendering system for clusters of graphics workstations. It uses a system for manipulating streams of graphics API commands and has filters which can be arranged to create sort-first and sort-last parallel graphics architectures that, in many cases, support the same applications while only using commodity graphics accelerators. Chromium uses a completely general stream processing mechanism that allows for any cluster-parallel algorithm to be implemented on top of or embedded into Chromium.
http://sourceforge.net/projects/chromium/
- **Panda3D.** [18] This was developed by Disney VR for its massively online multiplayer game Toontown online. In also shows a strong networking architecture which allows for the quick and easy creating of shared experiences. The studio has released the library as open source and coordinated with universities in order to prepare it for the open source community. In 2003 Panda3D came about through Carnegie Mellon University. Panda3D is an application driven library which makes it well suited for games and simulations.
http://www.panda3d.org/
- **The blue-c distributed scene graph.** [22][14] This was developed as the basis for a novel collaborative immersive environment system. It extends the OpenGL Performer toolkit to provide a distributed scene graph maintaining full synchronization down to vertex and texel level. It provides a synchronization scheme including customizable, relaxed locking mechanisms. The scene graph itself features abilities like locally stored scene graph nodes for static scene data as well as nodes shared across multiple sites.
The blue-c scene graph is a part of the blue-c API which is a collaborative immersive environment system that also supports sound and various input devices. Distributed OpenSceneGraph uses a number of the ideas developed for the blue-c scene graph project.
http://csdl2.computer.org/persagen/DLAbsToc.jsp?resourcePath=/dl/proceedings/&toc=comp/proceedings/vr/2003/1882/00/1882toc.xml&DOI=10.1109/VR.2003.1191157

- **Distributed Open Inventor.** [13] The distributed Open Inventor is an extension to the popular Open Inventor toolkit. The toolkit itself is extended with the concept of a distributed shared scene graph. From the application programmers perspective multiple workstations share a common scene graph.
  The system provides a convenient way for application programmers to write distributed graphics applications in an almost transparent manner. By also allowing for local variations in the scene graph it provides the means necessary for a wide range of applications. It also uses low latency mechanisms called input streams to enable high performance while saving the programmer from network peculiarities.A lot of the lessons learned and solutions found by the Distributed Open Inventor team served went into the Distributed OpenSceneGraph project.
  http://portal.acm.org/citation.cfm?doid=323663.323675

Distribution tools

Following is a short list of some libraries that can be used when developing distributed applications. All the ones found in this list was reviewed at some time in the project.

- **ReplicaNet** [27] This is a cross platform object oriented networking middleware used for games and other networked applications. It supports automatic load balancing and failure recovery and has the ability to publish shared objects. ReplicaNet is a close source library but free for use for non-profit and educational purposes.
  http://www.replicanet.com/default.html
- **RakNet** [42] Raknet is a cross platform C++ UDP game network library designed to allow programmers to add response time-critical network capabilities to their applications. It is mostly used for games, but is application independent.
  http://www.rakkarsoft.com/
- **MPI** [9][10] The Message Passing Interface. This is really a standard definition much like OpenGL and comes in many flavors. MPI is a message passing library that provides library routines for message passing and associated operations.
  http://www.lam-mpi.org/
- **Socketcc** [26] Small and lightwight C++ library wrapping the socket functionality of Linux systems.
  http://www.ctie.monash.edu.au/SocketCC/

Results

This section will present the measured results and the techniques used to obtain them.

Test application

To properly perform initial testing and debugging of the Distributed OpenSceneGraph API it was decided not to use the display wall available at the University of Tromsø. This was partly because of time constraints but also since it was felt that a more controlled scenario for testing would be advantageous.

To test the functionality of the Distributed OpenSceneGraph API a configurable test application was developed. A screenshot of this application can be seen in figure ###

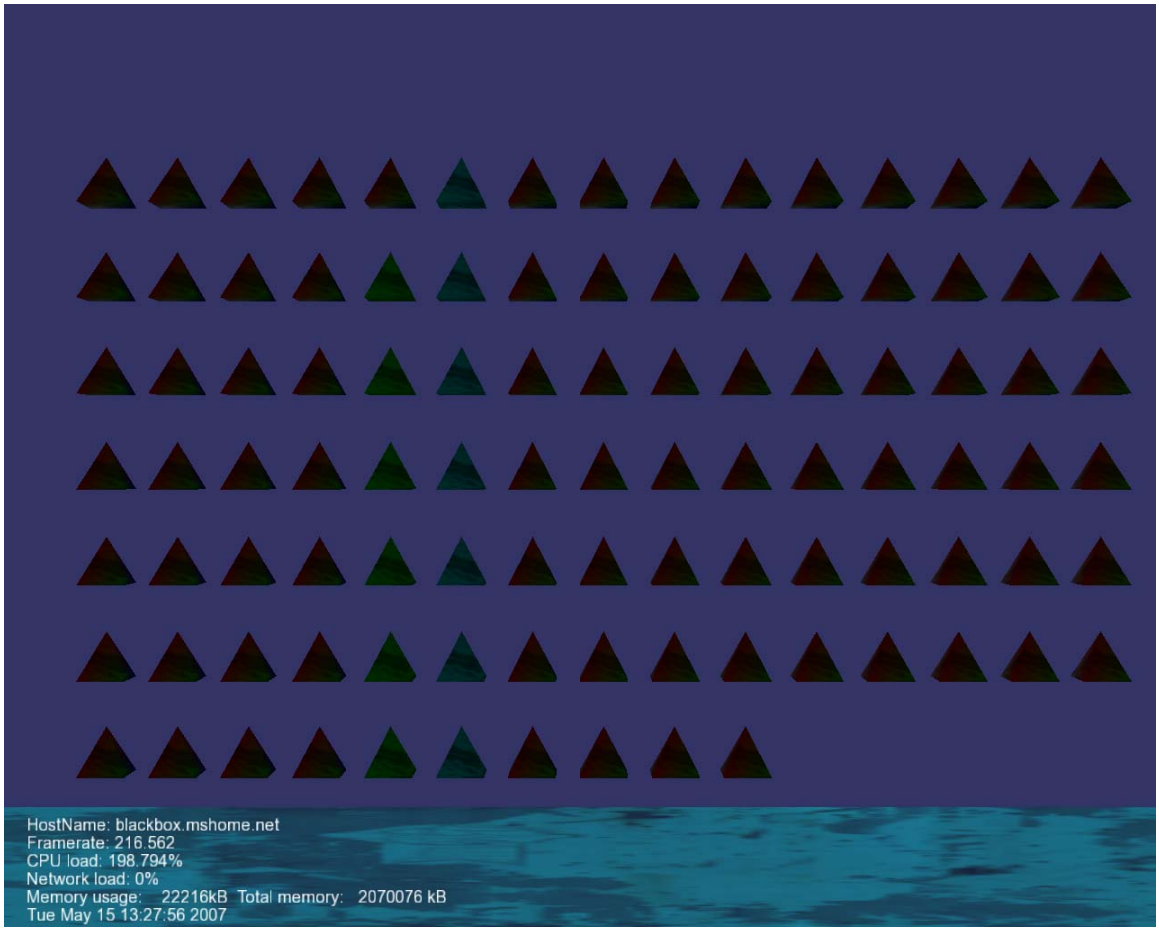


Figure 13 Screenshot of the test application.

The test application was developed to be easily adaptable to different test scenarios (more nodes, more geometry), to present the tester with current statistics and identification info

for performance testing and to in an orderly fashion present the testing staff with an easy way of confirming the success of distribution.

The DOSGVisitor class can also be configured to generate log statistics and averages. This is done simply by passing the true to the Boolean logg parameter when instantiating the class. The log the DOSGVisitor instance generates is not viewable on screen but is appended to a log file once per second. In addition to logging the same parameters as shown in the HUD in the screenshot in figure ### it also logs frame rate and CPU usage averages.

Test client and server

Instead of using multiple instances of the test application for testing a client and a server was developed specifically for this purpose. The reason for this was that it was felt that by using these much less complex programs it would be easier and faster to reconfigure these for different tests, as well as control and read their input and output for verifications of things working as excepted. These were used with success in different configurations. To make the client implemented provoke scene graph node updates on the Distributed OpenSceneGraph test application a group of serialized node strings were written to file from the test applications marshal mechanism and hand edited. This edited file is read by the client and continuously sent to the computer running the Distributed OpenSceneGraph test application.
The implemented test server writes the received serialized state picked up from the test application to file. This file was parsed manually for confirmation of the correctness of the marshalling and propagation mechanisms.

Test systems

Table 5 shows the configuration of the test systems involved.

Table 5 Test hardware. These computers were connected through a 100Mbit LAN.

Parameter	**Computer running test application**	**Computer running the test client and server**
CPU type	3 GHz Intel Pentium 3 D CPU Dual Core	Intel Celeron 2.5GHz
Memory	2 GB	512 MB
Network card	GigaBit NIC	100 Mb NIC
Graphics card	Nvidia NX 7600 GT	Nvidia TI 4400
OS	Fedora Core 6	Ubuntu 6.10
OpenSceneGraph Version	Version 1.2	Version 1.2

The two computers were connected through a 100 Mbit local network.

Tests performed

The tests were performed with the same base application. The application was edited for each run to test the different parts of the systems impact on performance. The editing done for each run was simply commenting out the relevant code in the application. Where nothing else is mentioned in the test results logging was turned on. The logging mechanism used, in addition to simply observing the measurements presented by the HUD, are the file logging implemented in the DOSGVisitorData class. This introduces a small performance hit since it involves file accesses, but since this is done only once per second the resulting performance reduction is viewed as insignificant.

All tests were performed by generating a number of DOSGGeodes. Since these should behave exactly as their osg::Geode super class (beyond using an insignificantly larger amount of memory due to their extra data members) when no distribution is done it was viewed as unnecessary to perform any comparison tests using osg::Geode nodes instead of their distributed counterparts.

The tests performed were:

- **No distribution.** This is to measure the true OpenScenegraph performance on the test computer.
  A should be noted that since the DOSGVisitor is not instantiated in this case, this configuration will not suffer the performance hit of its log systems file accesses. Since these file accesses are quite infrequent the impact of this when compared to the other configurations should be insignificant.
  To configure for this the following criteria were met:
  - The DOSGVisitor was not instantiated. The result of this was that no traversals by it were be made.
  - The DOSGCommunicator and DOSGClient instances were not instantiated or run. The result of this is no server or client threads running and no communication taking place.
- **Marshalling of node state but no communication.** This configuration involved instantiating the DOSGVisitor and make it perform its traversals. The resulting function of this was a situation where only the marshalling performance hit could be measured by comparing it to the results of the "no distribution" results. To ensure that the demarshalling was doe each frame it was made sure that each node's DIRTY flag was set back to DIRTY immediately after its state was collected (which sets the flag to clean).
- **Marshalling and propagating.** This configuration is the same as the above one but with the DOSGClient instantiated. The result of this is that the system traverses all nodes and collects the state of the nodes it owns. The difference is that now this state will be propagated to the test server running on the other computer in the test system. This test measures the performance hit of the total process of demarshalling and propagating onward local state.

- **Demarshalling. No propagation of state.** This setup involved instantiating the DOSGCommunicator instance but not the DOSGClient instance. For this test the marshalling function calls also were commented out. To have something to demarshal and update from a client was run from the other computer in the system. This client continuously sends an update for all the nodes in a branch of the scene graph. This test measures the performance impact of the receiving of and demarshalling updates.
- **Full fledged.** This is a test with all functionality enabled. The test client and server run sending and receiving updates from the test application running on the Pentium computer. This test measures the true performance in a controlled environment. It also displays the performance hit taken when the DOSGVisitor, DOSGClient and DOSGCommunicator instances contend over the mutual exclusion protected data structures stored in the DOSGVisitor.

Results

The test results of the measurements are shown in table ###. It should be noted that the reason the measurements of CPU load can arise to 200% is that the measurement is done calculating CPU time and the test computer uses a dual core CPU.

Table 6 Test results of different configurations of the Distributed OpenSceneGraph system.

Test performed	Frame rate Average	CPU average (dual core)	Memory usage	Notes
No distribution	1100 fps	99%	21700kB	HUD observed state. Not average.
Marshalling of node state but no communication	435 fps	99%	21800kB	
Marshalling and propagating	431 fps	100,6%	21900kB	
Demarshalling. No propagation of state	403 fps	195 %	22200kB	
Full fledged	223 fps	189 %	22300kB	

Some notes about the measurement results.

As expected, a performance loss is incurred when distributing compared to running the scene graph application as a normal single process application.

Also the main performance hits seems to be related to the tasks done when marshalling and demarshalling data. However the amounts of such work in the test scenario is

relatively light since the pyramids the scene is made up of are relatively simple geometry shapes. This is also the reason why the frame rate can be so high when running undistributed. This can in particular be observed when comparing the scenarios where the system just is marshalling and where it marshals data and transmits it. There is no notable difference in performance between these scenarios. The difference observed in the scenarios of only marshalling or demarshalling when compared to running undistributed however is huge. One reason for the small impact of the communication is probably the fact that the client and server runs in separate threads. And when there is little contention for the data stored in the visitor they have an almost insignificant impact on the frame rate.

However when faced with continuous communication, such as induced by the test client the client and server threads increase the CPU load to very high levels something that will hurt the overall efficiency of the application.

When running the full fledged tests it was also noted that there would be periodical frame rate drops. The frame rate was generally observed to be in the range 250-280 fps but occasionally dropped to about 100-150 fps. A probable reason for this is that the threads that make up the application contend about CPU resources and when the communication client and server threads use a lot of CPU power it hurts the thread used for updating, rendering etc.

Conclusions

The version of this system that is in existence on the time of writing is somewhat unfinished. However it serves as a good starting point for future work. It does indeed distribute node state. And it is flexible when used in regular OpenSceneGraph applications to distribute specific nodes in the graph. It's fairly transparent beyond making a developer using the system having to split up the problem and make a few extra function calls in main.

Because of the clearly defined spheres of responsibility Distributed OpenSceneGraph has proven to be easy to extend with new classes. As long as a developer of a new distributed class inherits from the DOSGDistributor class and implements its abstract functions, it does not in any way enforce what needs to be distributed in a new distributed class. How and what to distributed in the inherited class is up to its developer. It does neither enforce any limitations on what the resulting serialized state looks like. As long as the inherited class knows how to parse it and set its state from it the Distributed OpenSceneGraph visitor will handle the rest.

Also by only concerning itself with individual node states, the Distributed OpenSceneGraph further enhances its flexibility by not restricting an application programmer to in any way need to have identical copies of the scene graph tree. It is perfectly legal within this system to have radically different scene graph trees in each process in the distributed application. Even with the distributed nodes at different places in the tree in each graph. it is not even required that all scene graph copies in a distributed application include all the nodes. Or any distributed nodes for that matter. The state will still be distributed onward to the other nodes in the system that needs it.
This is because of the way the DOSGVisitor traverses the scene graph tree and updates the nodes. It will only update nodes found. Any node it has stored state for but that does not exist in the scene graph the visitor traverses will simply be ignored. The stored state for the node will however be propagated onward as usual since the DOSGClient will send all state stored.

However as the measurements of the current version of the system indicate there is still a lot of work to do to make this a truly usable library. Its efficiency must be improved, especially in the demarshalling and marshalling functions, and a synchronization mechanism must be implemented. The communications threads should also be reimplemented to reduce their CPU usage and the contention over shared data.

While not quite a full fledged API at the time of writing it is nonetheless felt that Distributed OpenSceneGraph shows promise and can become a very good visualization tool for programmers of distributed applications.

Future work

The system as-is is rather basic and only complete enough for proof-of-concept and testing purposes. To be a truly worthwhile distribution system for OpenScenegraph a number of issues must be handled.

First of all a synchronization mechanism has to be implemented. The current ring organization of the communications subsystem can easily be used to implement a simple synchronization mechanism by letting each node detect when it has received its own updates when they have propagated through the ring. Further this approach can also be implemented to allow for user configuration of the tightness of the synchronization. Since the system counts frames a synchronization mechanism can be implemented to make the systems processes stay within given boundaries of frames. By using a smaller boundary scope the synchronization will be tighter but the actual frame rate will be lower. This can be acceptable for presentations of scientific data where the correctness of what is seen onscreen is more important than the total frame rate. With higher boundaries of the synchronization limits a more visually smooth approach can be used. If a process detects it is closing to **max** frames a head of the slowest process in the system then the DOSGVisitor can be adapted to wait a little before it starts up its next traversal. By waiting for comparably small periods before continuing it is possible to spread the 'tightening in' of the synchronization of the processes over multiple frames thereby avoiding jerky stops of the scene rendered on the fastest nodes.

While the synchronization approach above might work it would probably also be well worth the time looking into other solutions for synchronization mechanisms.

The testing to date has shown that the one of the most pressing issue to assess is the performance of the communications, marshalling and demarshalling of nodes. The way this is currently done is easy to test and debug but a more performance oriented solution should be implemented.

Reducing the amount of data to distribute is also an important measure for improving performance. Work was started, but unfortunately not finished in time for testing on a distributed version of the osg::Geometry class. By extending a distributed version of this the possibility of a much finer granularity of the marshalling of the geometry data is possible. For example the current version if a Geode is marked as DIRTY it will retrieve all the data from all its drawables. By implementing a distributed geometry drawable we can firstly limit the marshalling to only the drawables that has been updated. Secondly the drawables themselves can return only the state of its updated fields.

Furthermore, implementing a different communications architecture than the ring architecture Distributed OpenSceneGraph uses currently could prove beneficial. An interesting approach could be using broadcast trees to propagate the state of the nodes.

This should provide the system with much lower communication times by not making the updates pass every single process in turn.

Different protocol DOSGCommunicator and DOSGClient classes should also be implemented to enhance efficiency in the distribution of scene graph objects. Using other protocols like UDP might improve overall performance by reducing the communication time.

To make the system truly usable as a distribution system for OpenSceneGraph it also has to implement distributed versions of a lot more of the scene graph node classes that make up the OpenSceneGraph API. The classes extended for this thesis are strictly the ones needed for testing the system. To be a usable system more distributed functionality is needed. An obvious example would be the translation classes. These should be distributed to allow distributed controlled animation of the scene. Also group nodes must be made distributed to allow for distribution of objects read from various file formats.

The current version of the DOSGVisitor finds the distributed scene graph nodes by checking for the correct class name using the nodes className function and downcasting the nodes to the correct type. This means that for actually finding and processing new extended OpenSceneGraph scene graph classes, the source code of the DOSGVisitor class actually has to be modified to add the new class. This should be avoided.

A solution to this could be to enforce that all distributed classes to use a common library name. This name is currently “DOSG” and can be retrieved by using the libraryName function inherited from the abstract osg::Object class. Further, by rewriting the DOSGVisitor class to instead of checking directly for the class name to check for the scene graph node’s library name it is possible to in a simple way to find a much broader range of distributed classes. When a node belonging to the correct library is found it can be downcasted to the Distributed OpenSceneGraph node super class, DOSGDistributor. By only downcasting it to this we do not restrict ourselves to only the classes which the DOSGVisitor class explicitly has been implemented for. This should work since all function calls that the DOSGVisitor uses to detect state, retrieve serialized state and to push updated state onto scene graph nodes are declared in the DOSGDistributor superclass.

By doing this the process of extending OpenSceneGraph node classes should be made much easier.

Appendix A - Glossary / Definitions

- **Node:** To avoid confusion with cluster nodes this report defines a node as a node in a scene graph.
- **Process:** A separate branch of the application. For our purposes it can be assumed that it can only be one instance of the Distributed OpenSceneGraph running on any one computer. Therefore if nothing else is specified a process and a cluster node can be used interchangeably.

Appendix B – An example serialized node

The following is an example of the serialized state of a single pyramid. The pyramid contains both colour and texture coordinates in addition to vertexes:

```
<node>
<nodeid>localhost.localdomain:id:5:node:0</nodeid>
<owner>blackbox.mshome.net</owner>
<name>localhost.localdomain:id:5:node:0</name>
<classname>Geometry</classname>
<drawable>0</drawable>
<vertices>
<Vec3Array>
<osg::Vec3>
<x>0</x>
<y>0</y>
<z>0</z>
</osg::Vec3>
<osg::Vec3>
<x>10</x>
<y>0</y>
<z>0</z>
</osg::Vec3>
<osg::Vec3>
<x>10</x>
<y>10</y>
<z>0</z>
</osg::Vec3>
<osg::Vec3>
<x>0</x>
<y>10</y>
<z>0</z>
</osg::Vec3>
<osg::Vec3>
<x>5</x>
<y>5</y>
<z>10</z>
</osg::Vec3>
</Vec3Array>
</vertices>
<colors>
<Vec4Array>
<osg::Vec4>
<x>0</x>
<y>1</y>
```

<z>1</z>
<w>0.5</w>
</osg::Vec4>
<osg::Vec4>
<x>0</x>
<y>1</y>
<z>1</z>
<w>0.5</w>
</osg::Vec4>
<osg::Vec4>
<x>0</x>
<y>1</y>
<z>1</z>
<w>0.5</w>
</osg::Vec4>
<osg::Vec4>
<x>0</x>
<y>1</y>
<z>1</z>
<w>0.5</w>
</osg::Vec4>
</Vec4Array>
</colors>
<Indexarray>
<osg::Array>
<GLuint>0</GLuint>
<GLuint>1</GLuint>
<GLuint>2</GLuint>
<GLuint>3</GLuint>
<GLuint>0</GLuint>
</osg::Array>
</indexarray>
<texcoords>
<Vec2Array>
<osg::Vec2>
<x>0</x>
<y>0</y>
</osg::Vec2>
<osg::Vec2>
<x>0.25</x>
<y>0</y>
</osg::Vec2>
<osg::Vec2>
<x>0.5</x>
<y>0</y>
</osg::Vec2>

```
<osg::Vec2>
<x>0.75</x>
<y>0</y>
</osg::Vec2>
<osg::Vec2>
<x>0.5</x>
<y>1</y>
</osg::Vec2>
</Vec2Array>
</texcoords>
</node>
```

Appendix C – Mandelbrot set calculations. C code.

This is an example of sequential C code that can be used for solving the Mandelbrot set for. Example from: http://www.cs.uit.no/inf3201/2006h/assignments/DessertMap.html

```
#include "graphicsScreen.h"
#include "StopWatch.h"
#include <stdio.h>

#define WIDTH 500
#define HEIGHT 500
#define SIZE WIDTH*HEIGHT

int zooms=10;

#ifndef GRAPHICS
int crc;
#endif

int colortable[] = {
  WHITE,
  WHITE,
  BLUE,
  CYAN,
  BLACK,
  GREEN,
  MAGENTA,
  ORANGE,
  PINK,
  RED,
  YELLOW
};

int colorcount=10;

double box_x_min, box_x_max, box_y_min, box_y_max;

inline double translate_x(int x) {
  return (((box_x_max-box_x_min)/WIDTH)*x)+box_x_min;
}

inline double translate_y(int y) {
  return (((box_y_max-box_y_min)/HEIGHT)*y)+box_y_min;
}

inline int solve(double x, double y)              //Simple Mandelbrot
{                                              //divergation test
  double r=0.0,s=0.0;
  double next_r,next_s;
  int itt=0;

  while((r*r+s*s)<=4.0) {
    next_r=r*r-s*s+x;
    next_s=2*r*s+y;
```

```
    r=next_r; s=next_s;
    if(++itt==100)break;
  }

  return itt;
}

void CreateMap() {        //Our 'main' function

  int x,y,color;                                  //Holds the result of
slove


  for(y=0;y<HEIGHT;y++)              //Main loop for map generation
    for(x=0;x<WIDTH;x++){
      color=solve(translate_x(x),translate_y(y))*colorcount/100;
#ifdef GRAPHICS
      gs_plot(x,y,colortable[color]); //Plot the coordinate to map
#else
      crc+=colortable[color];
#endif
    }
#ifdef GRAPHICS
  gs_update();
#endif
}


int
RoadMap ()
{
  int i;
  double deltaxmin, deltaymin, deltaxmax,deltaymax;

  box_x_min=-1.5; box_x_max=0.5;           //Set the map bounding box
for total map
    box_y_min=-1.0; box_y_max=1.0;

    deltaxmin=(-0.9-box_x_min)/zooms;
    deltaxmax=(-0.65-box_x_max)/zooms;
    deltaymin=(-0.4-box_y_min)/zooms;
    deltaymax=(-0.1-box_y_max)/zooms;

    CreateMap();                       //Call our main
    for(i=0;i<zooms;i++){
      box_x_min+=deltaxmin;
      box_x_max+=deltaxmax;
      box_y_min+=deltaymin;
      box_y_max+=deltaymax;
      CreateMap();                       //Call our main
    }

    return 0;
}

int main(void){
  char buf[256];
```

```
#ifdef GRAPHICS
  gs_init(WIDTH, HEIGHT);
#endif

  sw_init();
  sw_start();
  RoadMap();
  sw_stop();

  sw_timeString(buf);

  printf("Time taken: %s\n",buf);

#ifdef GRAPHICS
  gs_exit();
#else
  printf("CRC is %x\n",crc);
#endif

  return 0;
}
```

References

[1] http://www.openscenegraph.com/

[2] Wikipedia ”OpenScenegraph”:
http://en.wikipedia.org/wiki/Openscenegraph

[3] Richard S. Wright jr. and Benjamin Lipchak:
“OpenGL SuperBible Third Edition”
ISBN: 0-672-32601-9. SAMS

[4] Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis:
“OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2”
ISBN: 0-321-33573-2. Addison Wesley

[5] Else Lervik and Mildrid Ljosland:
Grunnleggende programmering i C++
ISBN: 82-417-0278-7. ad Notam Gyldendal

[6] Else Lervik and Mildrid Ljosland:
”Objektorientert programmering i C++”
ISBN: 82-417-0277-9. ad Notam Gyldendal

[7] Bruce Eckel:
”*Thinking in C++”*
ISBN: 0-13-917709-4. Prentice Hall

[8] Tom McReynolds, David Blythe:
“Advanced graphics programming using OpenGL”
ISBN: 1-55860-659-9

[9] Peter S. Pacheco
“Parallel programming with MPI”
ISBN: 1-55860-339-5. Pearson Prentice Hall

[10] Barry Wilkinson, Michael Allen:
“Parallel programming – Techniques and applications using networked workstations and parallel computers”
ISBN: 0-13-191865-6. Morgan Kaufmann

[11] Erik Yuzwa:
“Game Programming in C++ from start to finish”
ISBN: 1-58450-432-3. Charles River Media

[12] W. Richard Stevens, Stephen A. Rago:
“Advanced programming in the UNIX Environment Second Edition”
ISBN: 0-201-43307-9. Addison Wesley

[13] Gerd Hesina, Dieter Schmalstieg, Anton Fuhrmann And Werner Purgathofer:
“Distributed Open Inventor: A practical approach to Distributed 3D graphics”
Vienna University of Technology, Austria.

[14] Martin Naef, Edouard Lamboray, Oliver Staadt, Markus Gross:
“The blue-c Distributed Scene Graph”.
Computer Graphics Laboratory, Swiss federal institute of technology Zurich.

[15] Greg Humphreys, Mike Huston, Ren Ng, Randall Frank, Sean Ahern, Peter D Kircher, James T. Klosowski.

"Chromium: A stream-processing framework for interactive rendering on clusters"
Stanford University, Lawrence Livermore National Laboratory, IBM T.J. Watson Research Center.

[16] www.openscenegraph.com:
"An Overview of Scene Graphs and OpenSceneGraph – draft"

[17] Jürgen Döllner, Klaus Hinrichs
"A generalized scene graph"
Institut für Informatik, Universität Münster, Germany

[18] Cary Sandvig and Jesse Schnell:
"Panda3D"
Disney

[19] Dirk Reiners, Gerrit Voß and Johannes Behr
"OpenSG: Basic Concepts"
OpenSGForum, Camtech, ZDGV

[20] Jan Ohlenburg, Iris Herbst, Irma Lindt, Thorsten Fröhlich and Wolfgang Broll
"The Morgan Framework: Enabling Dynamic Multi-User AR and VR projects."
Fraunhofer Institute for Applied Information Technology.

[21] John Rohlf and James Helman:
"IRIS Performer: A high performance multiprocessing toolkit for real-time 3D graphics"
Silicon Graphics Computer Systems.

[22] Martin Naef, Oliver Staadt, Markus Gross:
"blue-c API: A multimedia and 3D Video Enhanced Toolkit for collaborative VR and Telepresence"
Computer graphics laboratory, Swiss federal institute of technology Zurich.
Computer science department, University of California.

[23] Blair MacIntyre and Steven Feiner:
"A distributed 3D graphics library"
Department of computer science, Columbia University.

[24] Paul S. Strauss and Rick Carey:
"An Object – Oriented 3D graphics toolkit"
Silicon Graphics Computer Systems

[25] Jason But
"Socketcc - A C++ Socket Library for Linux. An IPv4/IPv6 class-based sockets C++ library"
http://www.ddj.com/184405070

[26] Jason But
"*Socketcc"*
http://www.ctie.monash.edu.au/SocketCC/

[27] Replica Software
"ReplicaNet"
http://www.replicanet.com/

[28] Andrew S. Tanenbaum and Maarten van Steen.
"*Distributed Systems. Principles and paradigms"*
ISBN: 0-13-121786-0

[29] Gerard Tel.
"Introduction to distributed algorithms"
ISBN: 0-521-79483-8
[30] www.wikipedia.com
Visitor pattern
«http://en.wikipedia.org/wiki/Visitor_pattern»
[31] Marshall Cline
Miscellaneous technical issues
http://www.parashift.com/c++-faq-lite/misc-technical-issues.html
[32] www.wikipedia.com
"Comparison between OpenGL and Direct3D"
http://en.wikipedia.org/wiki/Comparison_of_Direct3D_and_OpenGL
[33] David Dewey
" Introduction to the Mandelbrot Set. A guide for people with little math experience."
http://www.ddewey.net/mandelbrot/
[34] www.wikipedia.com
"Immediate mode"
http://en.wikipedia.org/wiki/Immediate_mode
[35] www.wikipedia.com
"Retained mode"
http://en.wikipedia.org/wiki/Retained_mode
[36] Christian Pirchheim
"Visual programming of user interfaces for distributed graphics applications"
http://studierstube.icg.tu-graz.ac.at/thesis/pirchheim_thesis.pdf
[37] Tamer Fahmy
"Pivy – embedding a dynamic scripting language into a scene graph library"
http://pivy.coin3d.org/
[38] Kai Li, Hahn Chen et al.
"Early experiences and challenges in building and using a scalable display wall system"
http://www.cs.princeton.edu/~rudro/cga00.pdf
IEEE Computer Graphics and Applications, vol 20(4), pp 671-680, 2000.
[39] Rudrajit Samanta et. Al.
"Load balancing for multi-projector rendering systems"
http://www.cs.princeton.edu/~rudro/hwws99.pdf
SIGGRAPH/Eurographics Workshop on Graphics Hardware, Los Angelos, California - August, 1999.
[40] CentOS
"CentOS"
http://www.centos.org/
[41] Red Hat, Inc
"Red Hat Linux"
http://www.redhat.com/

[42] Jenkins Software
"RakNet"
http://www.rakkarsoft.com/
[43] Sun
"Java 3D"
http://java.sun.com/products/java-media/3D/
[44] Kitware
"VTK – The Visualization toolkit"
www.vtk.org
[45] http://www.ogre3d.org/
"OGRE"
[46] Systems in motion
"COIN3D"
http://www.coin3d.org/