

A low-cost Linux based graphics cluster for cultural visualisation in virtual environments

Aidan Delaney and Karina Rodriguez-Echavarria

June 13, 2006

Abstract

We use visualisation technology to engage people with cultural heritage in much the same way that story and song have for millenia. The Internet has not yet lived up to its potential to distribute these high-fidelity, visually stimulating virtual environments. As such it is the responsibility of the cash-strapped museum sector to provide these experiences. This paper describes our approach of using the traditional stability of UNIX coupled with the low-cost of GNU/Linux on commodity hardware to provide an easy-to-use technology base for a non-IT sector. Unlike other clusters designed to provide for high-performance computing or highly available services, our cluster is designed for distributed rendering of graphics. We further demonstrate how we use chromium and distributed multi-headed X to address the dollar-cost of high-fidelity content production for cultural visualisation.

1 Introduction

Increasingly, cultural heritage is becoming an important application area for visualisation environments. These type of applications permit users, who are usually visitors to a site or museum, access to virtual graphic reconstructions of cultural heritage sites and artifacts that would normally be inaccessible due to location or fragile condition. The technology also provides the possibility of visiting places that no longer exist or which have evolved over time. However, developing visualisation environments, while maintaining low the costs of acquisition and maintenance of the hardware and software, is still a challenging problem. This is especially the case for museums and other heritage sites which are non-expert users of technology and do not have access to huge sources of funding and resources. These difficulties rely not only in software engineering issues but as well in integrating multiple hardware units (i.e. CPUs, GPUs, displays screens, etc.) to work seamless together [7].

Previous work in visualisation environments for cultural heritage has been mainly focused on single or multiple projection architectures, including CAVE based environment [8, 1, 2]. These environments usually consist of projection

screen installations, stereo surround sound and other devices for user to interact with the environment. A new type of installation which uses direct view screen technologies instead of projection systems is starting to emerge in this field. The causes of this new trend is caused by i) the drop in prices of screen technologies such as LCD, TFT and plasma displays which are making them an affordable hardware commodity; as well as ii) the awareness of the advantages that this type of technologies have when compared with projection technology. These advantages include higher resolutions offering crisper, brighter images. Furthermore such screens require less space and are non-intrusive, which makes them suitable to be mounted on a wall of a museum or other display space.

This paper will present a Linux-based architecture for an immersive visualisation environment using LCD screens for cultural heritage presentations. Furthermore, the paper will explore hardware and software issues for distributing the visualisation in the screen display. From this, the advantages of using Linux will be highlighted as well as other problems, which are still to be solved, will be identified. Finally, conclusions and further work will be described.

2 Immersive visualisation environment architecture

Clusters for 3d volume visualisation share characteristics that differ them from clusters for computation such as Beowulf clusters [4]. Such clusters

- are interactive i.e. offline rendering is not tolerated,
- are generally I/O bound rather than CPU bound with the host interconnects often being the bottleneck,
- exhibit locality of reference in three dimensions rather than a single dimension leading to different memory access patterns.

Clusters for rendering display walls are a subset of volume visualisation clusters. Other types of volume visualisation cluster may use the render nodes to render data and composite the results in a single small display. Our cluster is designed to provide a large array of reconfigurable monitors.

Our visualisation environment is an eight machine cluster (see Fig. 1). Each of these nodes are dual-processor AMD Athlon 64bit machines containing two nVidia 7800GTX PCI Express graphics cards. Each graphics card can drive two monitors meaning that a render node can drive four monitors in total. Render nodes have two gigabit ethernet cards, two gigabytes of RAM and are capable of driving four monitors. Currently nodes are homogeneous, however homogeneity is not a requirement for the functioning of the cluster. It is possible to substitute one node with another machine consisting of an Intel Pentium III class processor with three PCI graphics cards. We have not yet assessed the affect of such a substitution on the graphical throughput on the platform.



Figure 1: (Above) The visualisation environment in our lab. Each of the monitors are mounted on trussing custom designed to be reconfigurable. (Below) A scaled down view of a 8960×2304 pixel screenshot of the popular Planet Penguin Racer running on the cluster.

Each of our 24 TFT monitors has a native resolution of 1360×768 pixels which we run at 1600×1200 . The monitors are more rugged than standard desktop models and were chosen for their durability to contribute to the longevity of the development platform. A dual-core dual-processor AMD Athlon64 based machine with two nVidia 7800GTX graphics cards in SLI mode functions as the server of the cluster (the client in X or OpenSG terminology). Completing the development platform is a gigabit switch, a wireless router and various wireless devices which we use for multi-modal input.

Render nodes mount `/usr/local` from the server via NFS and have their `passwd` file updated via cron. The server runs Ubuntu 6.06 LTS (Dapper Drake) and the render nodes Ubuntu 5.10 (Breezy Badger). The normal `gdm` startup on the render nodes has been modified to start `Xorg` on `display : 0` immediately after boot. `Display : 0` combines a column of three monitors using Xinerama thus eight machines are used to drive twenty four monitors. Aside from the software in `/usr/local` the software on the render nodes is as distributed by Canonical. This considerably lowers the maintenance involved in keeping software patched and up-to-date. The server provides static leased IP addresses to all render nodes along with their hostname and acts as the Internet gateway for the subnet.

Our hardware platform has only recently been finished. During its construction we encountered several subtle bugs. An example of which was the failure of our Marvell (sky2.ko based) gigabit ethernet cards handling tcp packets longer than 1468bytes even after negotiating the ethernet standard MTU of 1500 bytes with the gigabit switch. As such we consider the hardware platform to be an initial prototype rather than an optimised production system.

3 Visualisation display flow

We employ two contrasting methods for visualisation display flow in the novel interfaces laboratory. Both intend to be transparent to the programmer by distributing the data generated by a single instance of a visualisation application and distributing it across the cluster for rendering. They differ in that the first, chromium, is intended to run unmodified OpenGL applications whereas OpenSG provides a C++ framework for developing applications.

3.1 Chromium and Distributed Multiheaded X

Chromium [5] is an application-independent method to present an entire network of render nodes as a single OpenGL context. It achieves this by replacing the OpenGL (Mesa) runtime libraries on the client machine and filtering each OpenGL request through a series of chromium stream processing units (SPUs). In our case the client uses the `tilesort` SPU to distribute an OpenGL stream to the render nodes. The render nodes, who receive their configuration from the server, use the `render` SPU to unmarshall the stream and process it. The major advantage of chromium is that existing OpenGL applications need not be modified in order to distribute the rendering workload.

```

import sys
sys.path.append( '../server' )
from mothership import *

app = sys.argv[1]

cr = CR()
cr.MTU( 10*1024*1024 )

TILE_COLS = 8
TILE_ROWS = 1
HOSTS = ['mercury', 'venus', 'mars', 'jupiter',
         'saturn', 'uranus', 'neptune', 'plutus']

TILE_WIDTH = 1600
TILE_HEIGHT = 3600

tilesortspu = SPU('tilesort')
tilesortspu.Conf('use_dmx', 1)
tilesortspu.Conf('retiler_on_resize', 1)
tilesortspu.Conf('bucket_mode', 'Non-Uniform Grid')
tilesortspu.Conf('draw_bbox', 0)
tilesortspu.Conf('scale_images', 0)

clientnode = CRApplicationNode( )
clientnode.StartDir( 'crbindir' )
clientnode.SetApplication( app )
clientnode.AddSPU( tilesortspu )
clientnode.Conf('track_window_size', 1)
clientnode.Conf('track_window_position', 1)

for row in range(TILE_ROWS):
    for col in range(TILE_COLS):
        n = row * TILE_COLS + col

        renderspu = SPU( 'render' )
        renderspu.Conf('display_string', HOSTS[n] + ':0')
        renderspu.Conf('render_to_app_window', 1)
        renderspu.Conf( 'window_geometry', [1.1*col*TILE_WIDTH, 1.1*row*TILE_HEIGHT, TILE_WIDTH, TILE_HEIGHT] )

        servernode = CRNetworkNode( HOSTS[n] )
        servernode.AddTile( col*TILE_WIDTH, (TILE_ROWS-row-1)*TILE_HEIGHT, TILE_WIDTH, TILE_HEIGHT )

        servernode.AddSPU( renderspu )
        servernode.Conf('optimize_bucket', 0)
        servernode.Conf('use_dmx', 1)
        cr.AddNode( servernode )

        tilesortspu.AddServer( servernode, protocol='tcpip', port=7000 + n )

cr.AddNode( clientnode )
cr.Go()

```

Figure 2: The chromium mothership configuration script from the novel interfaces lab cluster. An application such as Planet Penguin Racer is invoked as follows `$ crmothership dmx.conf 'which ppracer'`

Distributed Multiheaded X (Xdmx) provides a single logical X11 server by proxy forwarding X requests to multiple distributed X servers [3]. It does for multiple screens attached to multiple hosts what Xinerama does for multiple screens attached to a single host. Pairing chromium with Xdmx provides us with a general purpose desktop tiled across twenty four monitors with the added advantage of multiple input device support. A single purpose turnkey installation of an OpenGL application, say in a museum, would not require Xdmx.

Each render node in the network runs an X session and runs the chromium server (crserver). An execution of an application on the chromium framework is controlled by the chromium mothership. The mothership configuration file is a Python script (see fig. 2) which, in our case, details how to tile the application across the eight render nodes. Our configuration is little different from the example `dmx.conf` provided in the chromium distribution.

3.2 OpenSG

OpenSG [6] is an application framework with transparent support for rendering an OpenSG scenegraph across a network of render nodes. Scene graphs in OpenSG contain the following elements

Nodes: These are used only to define the hierarchy of the graph. They store their children (if any) and a core. Every node can only have one parent, but of course as many children as necessary. If for some reason a new parent is assigned to a node, the former parent will be replaced by the new one.

Cores: The core is the place where the spacial information is stored, i.e. geometry, transformations, etc. These can be referenced by as many nodes as desired, which facilitates efficient sharing of data such as large textures.

An OpenSG scenegraph of a scene displaying three geometries stores the following information:

- The root is the chassis geometry which has three children nodes: one for each geometry transformation.
- Before defining the geometry of the monkey face, the orientation and position of each of them is stored in three different transform cores attached to three different transform nodes.
- The geometry of the three monkey faces are then stored in only one core and attached to three different nodes.

In practice scene graphs are stored in a file in a format such as 3DS, DXF or VRML and loaded using the `SceneFileHandler`. A custom scene file loader can easily be defined for proprietary formats. The anatomy of an OpenSG based application is straightforward and an example application is provided (see fig. 3). `display()` is registered by the boilerplate GLUT code as a handler for the display and idle events. The matrix attached to the root node is modified to spin the geometry (lines 14-19) which is loaded from a file (line 59).

It is more straightforward to modify a GLUT based OpenSG application to render on multiple nodes than the other subclasses of `ClusterWindowBase` such as the QT, Win32 and X implementations. We change the window type from `GLUTWindow` to `MultiDisplayWindow`, tell the application which servers to contact and by which method to contact them. The available methods are one of *Multicast*, *SockStream* and *SockPipeline*. Mouse and keyboard interaction are handled using the familiar GLUT functions. In total nine lines of code need to be modified to distribute the application over a visualisation cluster. The end result can be quickly smoke-tested on a single machine running multiple OpenSG servers (see fig. 4).

```

1 #include <OpenSG/OSGConfig.h>
2 #include <OpenSG/OSGGLUT.h>
3 #include <OpenSG/OSGGLUTWindow.h>
4 #include <OpenSG/OSGSimpleSceneManager.h>
5 #include <OpenSG/OSGSceneFileHandler.h>
6
7 OSG_USING_NAMESPACE
8
9 SimpleSceneManager *mgr;
10 TransformPtr spin;
11 int rot = 0;
12
13 void display(void) {
14     Matrix m;
15     m.setTransform(Quaternion(Vec3f(1,0.5,0), ++rot * (3.14159/64)));
16     beginEditCP(spin);
17     spin->setMatrix(m);
18     endEditCP(spin);
19     mgr->redraw();
20 }
21
22 void reshape(int w, int h) {
23     mgr->resize(w, h);
24     glutPostRedisplay();
25 }
26
27 int main(int argc, char **argv) {
28     osgInit(argc,argv);
29     glutInit(&argc, argv);
30     glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH | GLUT_DOUBLE);
31
32     GLUTWindowPtr gwin = GLUTWindow::create();
33     gwin->setId(glutCreateWindow("Spinning Monkey"));
34
35     glutReshapeFunc(reshape);
36     glutDisplayFunc(display);
37     glutIdleFunc(display);
38
39     NodePtr root = Node::create();
40     spin = Transform::create();
41
42     Matrix m;
43     m.setIdentity();
44
45     beginEditCP(spin);
46     spin->setMatrix(m);
47     endEditCP(spin);
48
49     // load the scene
50     NodePtr scene;
51     Char8 filename[] = "Data/monkey.wrl";
52     scene = SceneFileHandler::the().read(filename);
53
54     beginEditCP(root);
55     root->setCore(spin);
56     root->addChild(scene);
57     endEditCP(root);
58
59     mgr = new SimpleSceneManager;
60     mgr->setWindow(gwin);
61     mgr->setRoot(root);
62     mgr->showAll();
63
64     gwin->init();
65
66     glutMainLoop();
67
68     return 0;
69 }

```

Figure 3: A listing of an introductory application which loads a VRML model and rotates it. It is predominately boiler plate code where the actual rotation occurs in the `display()` method

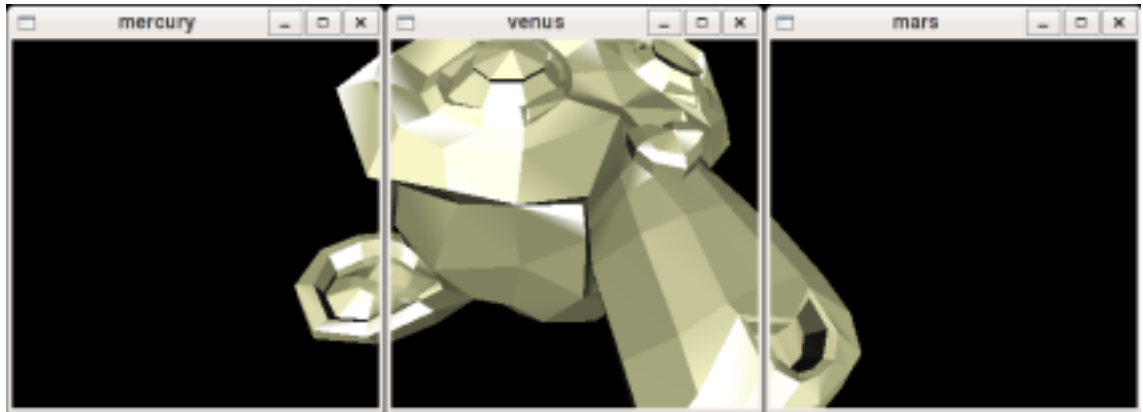


Figure 4: An OpenSG application being smoke-tested on a development system.

4 Conclusions

We have developed a volume visualisation cluster suitable for developing solutions to problems in the domain of cultural heritage. The cluster is constructed from commodity components which due to lack of reliance on homogeneity can be upgraded piecewise in the future. The monitor wall can be reconfigured into a cave-like structure and the software can be dynamically adapted to support this.

Two methods of developing cultural visualisation applications are supported on the hardware. One which runs unmodified GLUT applications the other which supports GLUT-style development but requires application modification. Network traffic performance under Chromium appears to be susceptible to changes in the MTU. This is of particular significance to our cluster due to issues with our network cards.

Linux provides a solid base on which to develop our applications. It is sufficiently robust to recover from all too common system lockups caused by the proprietary graphics driver. The dpkg based management system in Ubuntu minimises render node maintenance time and allows us to concentrate on development.

5 Acknowledgments

This work has been conducted thanks to the contribution of a grant from the HFCE. We would also like to thank our colleagues Craig and Tudor for innumerable enlightening conversations.

References

- [1] D. Christopoulos A. G. Gaitatzes and M. Roussou. Reviving the past: Cultural heritage meets virtual reality. In *VAST2001: Virtual Reality, Archaeology, and Cultural Heritage*, 2001.
- [2] Niklas Rber Bert Freudenberg, Maic Masuch and Thomas Strothotte. The computer-visualistik-raum: Veritable and inexpensive presentation of a virtual reconstruction. In *VAST2001: Virtual Reality, Archaeology, and Cultural Heritage*, 2001.
- [3] Rickard E. Faith and Kevin E. Martin. Client-to-server dmx extension to the x protocol. <http://dmx.sourceforge.net/DMXSpec.txt>, 2004.
- [4] Marcelo Eduardo Magallón Gherardelli. Hardware accelerated volume visualization on pc clusters. Master's thesis, Universität Stuttgart, 2004.
- [5] Ren Ng Randall Frank Sean Ahern Peter D. Kirchner Greg Humphreys, Mike Houston and James T. Klosowski. Chromium: A stream-processing framework for interactive rendering on clusters. 2002.
- [6] The OpenSG Homepage. www.opensg.org.
- [7] Edmond Boyer Jrmie Allard, Clment Mnier and Bruno Raffin. Running large vr applications on a pc cluster: the flowvr experience. In *11th Eurographics Workshop on Virtual Enviroments*, 2005.
- [8] S. Rapp and I. Weber. Lumenactive: A novel presentation tool for interactive installations. In *VAST2005: Virtual Reality, Archaeology, and Cultural Heritage*, 2005.