Simplifying Applications Use of Wall-Sized Tiled Displays

Espen Skjelnes Johnsen, Otto J. Anshus, John Markus Bjørndalen, Tore Larsen

Abstract
Computer users increasingly work with multiple displays that are arranged into a single row or a small matrix. The displays are usually driven from a single computer using one or more graphics cards. We also observe the emergence of larger, wall-sized, high-resolution displays composed of tiled displays or projectors driven by one or more computers.

Complexity and performance issues arise when applications need to drive several displays hosted by several computers. A common tradeoff is between simplicity, where the simplest solution can be to render the graphics on the application node, and performance, where the highest performance is often achieved by rendering at the display nodes.

We have proposed a display model and developed a system, The Wall Windowing System (W²S), where applications see multiple graphics cards and displays as one single resource while the rendering workload are shared by the available hardware. The primary abstractions are the virtual rendering surface, the integration space and the display node. An application renders to a virtual surface, which can be mapped to a specific location in the integration space. Display nodes act as viewports into regions of the integration space, showing the surface as a whole.

The architecture consist of a client side and a display side. Applications are built using a client side library and a modified Cairo graphics library. The display nodes interprets rendering instruction from clients.

To evaluate the performance, we have run benchmarks to compare our system to a distributed VNC setup. The results show that with W²S we achieved almost 40 fps, compared to less than 2 fps with DVNC.

1 Introduction
For several decades, we have seen a steady increase in the performance of computer hardware. One exception to this trend is the increase in resolution and size of displays, which has shown much less growth. This has lead to the development of tiled display walls, which are large high resolution displays built by aligning a matrix of smaller displays driven by a cluster of computers. For application programmers, the task of manually coordinating the displays is not trivial, and thus a system to automatically do this is needed.

The Wall Windowing System, or W²S for short, is an experimental distributed windowing system. It is designed to run effectively on large tiled display walls,

This paper was presented at the NIK-2008 conference; see http://www.nik.no/.
while providing users (i.e. application programmers) with the familiar single display abstractions and mechanisms. Figure 1 shows three applications using the system, running on a tiled display wall at the University of Tromsø.

2 Related Work

DVNC[4] is a modified VNC server/client setup. It consists of one central X11 server with a large framebuffer and multiple clients displaying regions of this framebuffer. Pixel data can be transferred directly between display nodes to reduce the load on the server. Despite this, the central server is still a potential bottleneck as it has to do all rendering of new content. Furthermore, DVNC does not utilize the processing power in the graphics hardware on the display nodes.

Distributed Multihead X¹ (Xdmx) is similar to DVNC from the user’s point of view, in that it gives the user a central X server to connect to. But instead of having a single framebuffer at the front end, the X server acts as a proxy, distributing graphics requests to the display nodes.

Chromium², as its predecessor WireGL[1], is a system for distributed rendering of OpenGL. It is highly configurable, supporting different types of tile sorting and compression algorithms. Chromium can be used standalone, or in combination with Xdmx, to support accelerated 3D rendering on tiled display walls. When used standalone, it will only support a single application at a time.

The Scalable Adaptive Graphics Environment[3] (SAGE) is a system which utilizes local rendering and lossless streaming of pixel data to display nodes. Unlike W²S it does not fully utilize the graphics hardware available at the display nodes.

[2] describes a “virtual graphics system” for tiled display walls, which uses the OpenGL API on the clients side for rendering into virtual layers. Unlike W²S’s surfaces, layers are not rendered off-screen and composed into the final framebuffer. Layers have to be redrawn on every update of the display, but to reduce usage of network bandwidth, rendering instructions are cached by the servers.
3 Display Model

Our display model is centered around a large abstract coordination system, called the integration space shown in Figure 2. Applications can map virtual rendering surfaces, eg. windows, into this space.

Display nodes act as viewports into the integration space, by projecting parts of it onto physical displays. By arranging the display nodes in a matrix, large tiled display walls can be built. Multiple display walls can be set up to show different or overlapping views of the integration space.

The integration space can be fully three dimensional, supporting effects like those introduced in Sun’s Project Looking Glass and effects emerging in modern desktop systems. Another option is to use a 2.5D space where surfaces just have a stacking order, as in traditional desktop systems.

4 Architecture and Design

W²S consists of two main parts (see figure 3): a set of display servers that run on the display nodes, and a client library that is linked with an applications code. Applications use the services provided by the client library to connect to the display servers over a network. The client library provides an API to manage surfaces in a high level fashion similar to other windowing systems.

Applications send streams of instructions to displays servers using a specially designed language. Transparently to the application programmer, the client library handles the tasks of distributing calls to the appropriate servers, and to ensure that display servers are continuously provided with necessary and up to date state information.

The client library provides an API to include rendering code directly in a host language, similar to how one would embed SQL for database queries. Extension libraries can be used to provide the same functionality on a higher level of abstraction. We have implemented a W²S backend for the Cairo graphics library which enable us to use the Cairo API to render 2D vector graphics into virtual surfaces.

---

1http://dmx.sourceforge.net
2http://chromium.sourceforge.net
3http://www.sun.com/software/looking_glass/
4Cairo is a cross platform library for 2D rendering used by modern widgets sets such as Gtk+ and Qt to enable resolution independent rendering of GUI elements. We describe the Cairo backend in the Implementation section.
Figure 3: The architecture of W²S. Each client is directly connected to a client thread in the display server that runs a virtual machine for that client. The client’s virtual machine is connected to a second virtual machine that is executed by the graphics thread.

**Display Servers**

Instructions sent from clients are handled by virtual machines in the display servers. Each client connection is served by a dedicated VM running in its own OS thread. Instructions are either executed immediately or compiled into procedures for later execution. In addition, each connection also has a virtual machine dedicated to executing rendering instructions. All graphics VMs are running in a single OS thread and scheduled in round-robin fashion. This way we avoid multi-threading problems with current OpenGL libraries and the graphics context switches are under full software control. The latter is important as switching context on the graphics card has a rather high overhead, and should be minimized.

**Extensibility**

One important aspect of the display servers is the ability to extend the functionality by dynamically loading modules. One such module existing already is the server-side support for Cairo rendering. Modules could be implemented to support 3D rendering and efficient handling of compressed images and video streaming.

**Client/Server Protocol**

Two different protocols may be used for communication between clients and display servers. One is text based, i.e. sending source code directly, and the other uses 16-bit binary tokens representing instructions. To avoid the management overhead involved in assigning tokens statically (i.e. keeping client and server code in sync), the system uses dynamic token assignment controlled by the client. To bootstrap this process, communication is started in text mode. The client would initially assign tokens to a few basic instructions and then switch to binary mode. New tokens are then assigned on demand by the client when new instructions are needed for the first time, using the binary protocol.

To support numbers (and other data types) in binary mode, an approach similar to how
strings are supported in traditional Forth is used; special instructions indicates that data should be read from the input stream. Instructions like int32 and double-float are used to read binary encoded numbers. Other kind of objects, like raster images, are handled in the same.

**Client Library**

The client library provides basic abstractions for the application programmer to handle multiple displays as a unified single display, by hiding the details of communication with individual display nodes. The library has an API to create and manipulate virtual surfaces similar to those found in other windowing systems, like X11 and Windows. It also has support to simplify the creation of other libraries that can be used for accessing the display nodes, in particular the code objects and state synchronization described in the following sections. Such libraries could typically be backends to existing 2D and 3D APIs (e.g. Cairo and OpenGL) and libraries to efficiently display images and play videos. Most such client side libraries would also require plugin modules to be created for the display servers.

W²S does not provide any support for input event handling, but leaves this up to the application. Nor is there any direct support window for decorations, but the client library has functionality to assist the application programmer with this. The API consists of hooks for adding window decoration functions and some predefined functions to draw standard decorations.

5 Implementation Details

**Cairo Backend**

Cairo is cross platform library for 2D rendering based on the Porter-Duff image composing model. We have created a W²S backend for Cairo which enables the application programmer to use the Cairo API to emit rendering instructions to display servers. Only a few new API entry points had to be introduced, the most notable being cairo_w2s_surface_create. In addition to the the client side Cairo backend, a Cairo module has to be loaded on each of the display servers. This module binds the Cairo API to the the W²S instruction language. The client side Cairo backend can be made very simple, as it only has to forward Cairo calls to the display servers using the W²S instruction language.

Two different techniques are used to optimize polygon handling in the W²S Cairo backend. Tile sorting based on polygon bounding boxes, is used to send polygons only to nodes where they most likely would be visible. This can greatly reduce the network load for applications with a relative large number of smaller polygon, as most polygons only have to be sent to one or a few nodes.

With larger polygons, tile sorting does not give the same gain, as polygons would typically span multiple nodes. For such applications, the backend supports per-node polygon clipping. With this technique, polygons are clipped against the visible viewport of each tile, before being sent to the server.

[^http://cairographics.org]
6  EXPERIMENTS AND RESULTS

Client Library

Code Objects

After having switched to binary mode, the client library will compile instructions from embedded source code into code objects. The same code object can be sent to multiple display nodes and also compiled once and reused multiple times. By adding filtering functions, the actual stream of instructions sent to display nodes can be tailored on an individual basis. This is used by the Cairo backend to implement per-tile clipping of polygons.

State Synchronization

Display nodes may fail or deliberately be taken off-line at any time, reappearing at any later time. When a display node comes back online, we need to ensure that it has the correct state, e.g. that surfaces and rendering contexts have been created, pre-compiled code is available, necessary fonts are downloaded and so on. A robust system should handle this in a way that is transparent to the application programmer. To support this, W²S has an automatic state synchronization mechanism called state objects.

A state object is compiled in the same way as code objects. To ensure that all nodes (or a given subset) have the necessary state information, a call to w2s_state_emit is inserted in the application code. The client library will then emit the state code to nodes which doesn’t have this already.

State objects can be extended after the initial emit. This feature is used in the Cairo backend to download glyphs on demand to display nodes by adding state objects representing fonts.

6 Experiments and Results

To compare the performance of W²S to DVNC, we have run some benchmarks based on the Cairngears demo application. This was done using a 28 node display wall cluster (Intel Pentium 4 EM64T, 3.2 GHz, 2 GB RAM running the Rocks cluster distribution of Linux) connected to 28 projectors with a resolution of 1024 × 768 pixels arranged in a 4 × 7 matrix. The application were running on a separate machine with the same hardware configuration. All network connections were with gigabit Ethernet, using a single switch. Table 1 summarizes the results.

The first W²S row shows the result of running the application natively on a single tile with a window size of 768 × 768 pixels. This is used to establish a reference for the other experiments, which were all run with a window size of 3000 × 3000.

The second row shows the result of using W²S without any of the optimizations described in Section 5 enabled, resulting in a frame rate of 31.0 and a network usage of approximately 70 kB/frame.

With tile sorting enabled, the network usage per frame dropped about 50%, but the framerate only increased slightly. This is probably due to the Cairo renderer in the display servers dropping polygons outside the viewport with little overhead, so the total amount of processing is more or less the same.

With client side clipping enabled, the network usage decreased even further, to 12 kB/frame, and the frame rate increased to 39. This is even better than running on a single tile, and is due to the rendering load being distributed among multiple nodes. Unfortunately, clipping introduces some CPU overhead, but since it uses multiple threads, it should benefit by running on multi-core CPUs.
7 CONCLUSIONS

<table>
<thead>
<tr>
<th></th>
<th>FPS</th>
<th>Bytes/Frame</th>
<th>App. CPU Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single tile</td>
<td>31.87</td>
<td>67.5%</td>
<td></td>
</tr>
<tr>
<td>W²S non optimized</td>
<td>31.00</td>
<td>71736</td>
<td>7.4%</td>
</tr>
<tr>
<td>W²S with tile sorting</td>
<td>32.85</td>
<td>35658</td>
<td>6.5%</td>
</tr>
<tr>
<td>W²S with clipping</td>
<td>39.05</td>
<td>12497</td>
<td>25.8%</td>
</tr>
<tr>
<td>W²S with both</td>
<td>39.53</td>
<td>7173</td>
<td>18.7%</td>
</tr>
<tr>
<td>VNC</td>
<td>1.95</td>
<td></td>
<td>39.7%</td>
</tr>
<tr>
<td>Xdmx</td>
<td>0.08</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Performance results comparing frames per second (FPS), bytes sent per frame and application CPU usage for VNC and for W²S using a number of techniques to reduce network traffic.

When both optimizations are enabled, the network usage is only 1/10th of the unoptimized case. The frame rate does not increase significantly compared to the clipping only benchmark as the extra data sent with only clipping enabled, is not taxing the gigabit network, and the extra data is discarded by the display servers.

When running with DVNC, the frame rate achieved was less than 2 fps. The CPU utilization was almost 40%, due to the VNC server not being hardware accelerated, so the application has to do most of the rendering in software.

We also tried to run the experiment with Xdmx, without any success. The displays did not show any output (despite the network load from the proxy server being 100MB/s) and the frame rate was very low. We do not know yet whether this is caused by the Xdmx server not being set up properly or if it doesn’t scale to the number of nodes in our display wall.

7 Conclusions

With this work we have shown that it is possible to create a fully distributed windowing system for tiled display walls, which have good performance due to the utilization of graphics hardware in the display nodes, and gives the user the familiar single display abstractions.

Experience also showed, very much to our surprise, that it was possible to take existing programs using the Cairo API and run them unmodified with W²S, except for the window system specific code to setup surfaces and handle events.

To get an indication of the performance of the system, we did some simple experiments by running the Cairogears demo application. The results showed that the frame rate achieved with W²S was about 40 fps which is even better compared to the program running natively on a single tile. When running through VNC, the frame rate was less than 2 fps.

8 Acknowledgements

The authors wish to thank Tor-Magne Stien Hagen for assistance with the execution of the experiments, and Daniel Stødle for being the first user of W²S. This work has been sponsored by the Norwegian Research Council, projects No. 159936/V30, SHARE – A Distributed Shared Virtual Desktop for Simple, Scalable and Robust Resource Sharing across Computer, Storage and Display Devices, and No. 155550/420 – Display Wall with Compute Cluster.
9 References

References


