Proceedings of the Linux Symposium

Volume Two

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.

Roadmap to a GL-based composited desktop for Linux

Kevin E. Martin
Red Hat, Inc.
kem@redhat.com

Keith Packard
Intel, Inc.
keith.packard@intel.com

Abstract

Over the past several years, the foundation that will lead to a GL-based composited desktop has been laid, but there is still much work ahead for Linux. Other OSes already have or are well on their way toward having a solution in this space. We need a concerted effort across every level of the OS—from the applications through the toolkits and libraries into the X server and the kernel—if we are to be successful.

In this paper, we examine the key technologies required, solve the limitations of the current X server design, and bring a GL-based composited desktop to fruition. For each of these technologies we will present current development status, explain how they fit together to create the GL-based composited desktop, and outline a roadmap for how to complete the remaining tasks.

1 Desktop design limitations

The current X server design is starting to show its age. Recent developments have shown that it's possible to create a GL-based composited desktop, but in order to effectively take advantage of the new technologies we describe in this paper, we must first understand the key limitations of the current design.

First, the current desktop has been designed around a 2D display device, while the silicon on graphics chips has shifted dramatically to 3D support. Integrating 3D into the desktop has long been the goal, but until recently it has not been possible. Other operating systems have also recognized this paradigm shift—Apple is using OpenGL through its Quartz [1] compositor architecture and Sun has a research project called Looking Glass [7] to experiment with using Java3D on their desktop.

A second limitation is that all drawing operations are being rendered directly into the front buffer. What this means is that users can see rendering artifacts while the desktop scene is being constructed—i.e., the intermediate states are visible. Most drawing operations are very fast, so it usually appears as a visually displeasing blur before the final image is visible, but sometimes is it much worse and you can see individual elements being drawn. Traditionally, toolkits have had to work around this problem by drawing directly to host-memory pixmaps and then copying the finished image to the screen.

A third limitation has been the static nature of the desktop states and the transitions between those states are either instantaneous or have very primitive transition animations. For example, when minimizing or un-minimizing windows, they simply pop into or out of existence or very simple window outlines are drawn in

sequence from the window to the icon in the panel that show the transition.

Below, we describe an incremental approach to making the new technology that addresses these limitations available in the open source community. The process we describe is to evolving the existing Xorg X server and its extensions to provide the new technology. In this way, we can minimize regressions for the existing installed base while still making a huge impact on what is possible.

2 Building on the past

The GL-based composited desktop is built on top of several key technologies that have been developed over the past several years. In this section, we describe three projects—the DRI, Composite, and Luminocity—which are necessary to understand the new technologies.

2.1 Direct rendering infrastructure

Throughout most of the 1990s, the only open source implementation of OpenGL was Mesa [9], which was a software-only client-side library that implemented the OpenGL interface. Then, in late 1998, Precision Insight began developing the Direct Rendering Infrastructure (DRI) [5], which brought open source hardware accelerated 3D graphics to the Linux platform. With this development, we took a huge step forward on the path to addressing the first limitation.

The way the DRI worked was that when an application requested a direct-rendering context, libGL would query the X server to see if a hardware-specific driver was available and if one was available, it would dynamically load that driver and initialize the internal dispatch table to use the driver for hardware accelerated rendering. In this way, applications could be written to the OpenGL library interface (or one of its toolkits) and not have to have hardware-specific knowledge.

But, as the name implies, the DRI was implemented to handle direct rendering. The goal was to eventually use the exact same hardware-specific driver code to handle accelerated indirect rendering as well, but in the initial implementation, indirect rendering was used—the software Mesa code.

2.2 Composite

With the relatively recent development of the Composite extension [8], developers now have the ability to redirect window contents to off-screen storage—i.e., pixel data that would normally have been drawn directly to an on-screen window can instead be drawn to a host-memory or off-screen pixmap. The pixel data can then be copied to the display buffer as needed to update what the user sees on his screen as his desktop. So, Composite effectively gives us the ability to double-buffer window data, an ability that has long been used by OpenGL applications to eliminate visual artifacts, and addresses the second limitation of the current desktop design in such a way that toolkits and individual apps do not have to implement their own solutions.

The ability to double-buffer window contents is not new to the X world—the double-buffer extension (DBE) allowed individual apps to double-buffer their output. What makes Composite unique is that it allows an external application, the Composite Manager, to control when windows are redirected and how their pixel data are copied to the display buffer instead of requiring each application to have direct knowledge of DBE.

In addition, there are many other benefits from using the Composite extension because it does not dictate how the window contents will be drawn to the display buffer. Various special effects can be used to render the window contents. For example, the window contents can be stretched to fit the screen or shrunk to fit into a window's icon if stretch operators are available. Other effects such as tranlucent windows or drop shadows can be implemented if alpha-blending is available. Many such effects were demonstrated with the simple compositing manager, `xcompmgr`. More complex effects could be implemented if the composite manager was implemented with OpenGL.

2.3 Luminocity

In late 2004, some of the developers at Red Hat began the Luminocity project [2] to experiment with using OpenGL in a composite manager. The basic idea behind Luminocity was to create OpenGL textures from each redirected window and then render rectangles to the framebuffer using those textures. This is similar to what Apple was doing with Quartz and Sun was doing in the Looking Glass project.

A significant difference between Luminocity and previous composite managers (e.g., `xcompmgr`) was that it handled both windowing and compositing operations in the same process. By combining the two, Luminocity could not only copy window data to the screen and render static effects like drop shadows, but it could also animate various state transitions. For example, Red Hat created the wobbly window effect, where windows were modeled with by simple spring system so that dragging a window around would distort it as if you were pulling on one of the springs.

Since the only open source hardware-accelerated OpenGL available at that time was through the DRI, Luminocity was developed to use direct rendering. However, this quickly led them to discover one of the primary performance problems: the number of data copies required to get the redirected window pixel data into a texture that could be used by the hardware were killing performance. Luminocity first had to copy the redirected window data from a host-memory pixmap in the X server to the client application, which required a slow XGetImage call, or a copy of the redirected pixmap data to a shared memory pixmap. The image data could then be reformatted to send to the OpenGL driver, which might also have to reformat the pixel data (depending on which driver and what data format was supported by the driver), and then the driver would upload the texture to the video card. Ultimately, we want to get to the point where no data copying is necessary—i.e., a redirected window could be drawn to a pixmap resident in the framebuffer and have the pixmap format be the same as what the driver requires so that it can be used directly by the hardware's 3D engine.

3 Roadmap to the new desktop

With the DRI, the Composite extension, and Luminocity, the three main limitations of the current desktop design were addressed, but in order to turn these solutions into something that performs well, supports the myriad of X extensions, and is robust enough to use in an enterprise environment, much more work is needed. Many new technologies are currently under development in the X, DRI, Mesa, kernel, toolkit, and desktop communities. Below we survey the technologies that will allow us to achieve our GL-based composited desktop goal.

3.1 Accelerated indirect rendering

As noted earlier, indirect rendering was left completely unaccelerated in the initial DRI project. The plan had always been to implement accelerated indirect rendering using the same card-specific driver code that is loaded on the client-side by libGL; however, it was not a simple task, and the driving issue to make this happen did not occur until the GL-based composited desktop became feasible.

The software Mesa driver used in the initial DRI work was based on the libX11 version of Mesa, which translated OpenGL requests into X11 drawing commands. This code, which previously called Xlib functions directly, was modified to instead call the equivalent internal X server function. This version of the client-side GL code was called GLcore.

The interface used to initialize and call into GLcore were the `__GLinterface` and `__GLdrawablePrivate` structs, which are part of the OpenGL sample implementation (SI) [12]. However, the interface to the DRI card-specific driver code was based on Mesa internals, which is quite different than the GLcore interface based on the SI. In order to use the same driver code on both with the client-side DRI and the server side GLX code, this impedance mismatch had to be solved and was one of the reasons that it took so long to implement accelerated indirect rendering.

The AIGLX project is currently under development in the Xorg community, and its goals are to solve the impedance mismatch between the client and server-side driver code while still allowing unaccelerated indirect rendering code with the software Mesa driver when no card-specific driver is available or when the user requests it. This work is both part of and built on top of the GLX client-side code rewrite [11]. The initial development stage of AIGLX is part of the X11R7.1 release.

In this project, a new abstraction layer [3] based on the DRI interface was developed to provide the glue logic between the server-side GLX extension code and the card-specific driver. The new interface provides three objects: `__GLXscreen`, `__GLXcontext`, and `__GLXdrawable`. Methods for allocating the DRI-specific objects and calling into the card-specific driver are contained entirely within the abstraction layer, which are called the DRI provider.

Since not all graphics cards have card-specific 3D drivers and since several other servers (e.g., Xnest) that provide GLX support cannot use hardware drivers, the GLcore module must remain available and the top level of the GLcore module had to be rewritten to use the new interface. This allows it to be used in place of the card-specific drivers when needed or desired, and is called the GLcore provider.

To initialize the GL module for each screen, a stack of GL providers are called and the first provider that returns a non-NULL `__GLXscreen` claims that screen. This mechanism allows for future GL modules to implement their own `__GLXprovider` and hook into the provider stack.

Future development will be needed to add support for GLX 1.3 (see below) and to continue reworking GLX visual initialization [11].

3.2 GLX 1.3 support

Much of the support for GLX 1.3 has already been added to the client and sever-side code, but several key pieces are currently missing. In particular, support for pbuffers will need to be implemented, which requires more advanced memory management than we currently have.

3.2.1 Memory management

OpenGL applications can require lots of off-screen video-card or agp memory for their buffers (e.g., front, back, depth, vertex, etc.) as well as for their textures. The initial DRI implementation used a shared buffer allocation scheme which pre-allocated the front, back, and depth buffers. This allocation scheme was possible since windows were clipped by the X server, and it was the X server's responsibility to determine what memory resources were given to the shared buffers, textures, and off-screen pixmaps at server initialization time. However, this scheme is no longer adequate and needs to be reevaluated for several reasons explained below.

First, with GLX 1.3, a new shared resource—the pbuffer—was added, which allows off-screen rendering for both direct and indirect rendered contexts. To claim support for GLX 1.3, pbuffer support is required, which means that dynamic allocation of off-screen memory resources is required and the simple allocation scheme from the initial DRI implementation is inadequate.

Second, GLXPixmaps were unaccelerated in the initial DRI implementation, and in order to implement hardware acceleration, the buffers associated with them need to be dynamically allocated/freed in off-screen memory as pixmaps are created/destroyed. Note that direct rendering to GLXPixmaps is not required, but it is greatly desired for use with the Composite and texture-from-pixmap extensions.

Finally, with the Composite extension, it is now possible to redirect GLX windows. Those redirected windows are no longer clipped by the normal X window stacking order, so it not possible to share the pre-allocated buffers. In addition, redirecting windows greatly increases the off-screen memory requirements if hardware-accelerated rendering is desired (which is especially true for OpenGL applications). For example, if a user is running his desktop at 1600∗1200 at 32BPP and he open his web browser in a full-screen window, the additional memory required for that one window is 7.3MB. If that same user opens a full-screen OpenGL application that also has a back and 32-bit depth buffer, then the memory requirement jumps to nearly 22MB! And this does not account for any textures that the app might use.

Each of these issues can be solved with a more advanced memory management framework that can be shared by all processes that need to access video and agp memory—e.g., the X server, direct rendered clients, and the kernel's direct rendering manager (DRM). The new framework generalizes all allocations to private buffers so that textures, color and ancillary buffers, pbuffers, pixmaps, and other buffers (e.g., FBOs and VBOs) are treated the same and can be allocated from the same memory pools. Additional basic requirements include being able to dynamically allocate the buffers as required by the client and being able to evict other clients' buffers while still guaranteeing that their contents are preserved. This work is currently under development by Tungsten Graphics [13].

With this new memory management framework, it will become possible to implement several other GLX extensions including texture-from-pixmap and framebuffer objects, both of which are very useful to a GL-based composited desktop.

3.2.2 Texture from pixmap extension

With AIGLX we now have the ability to render directly from within the X server process; however, we still need to be able to use the window pixel data that was redirected to a pixmap with

the Composite extension as a texture. This is what the texture from pixmap GLX extension provides (TFP).

The simple approach, as used by Luminocity, is to copy the data either through the protocol via XGetImage or through a shared-memory pixmap into the client's address space and then the direct-rendered composite manager could use that data as the source for a `glTexImage2D` or `glDrawPixels` call. However, this does not work in practice due to the high overhead of copying pixel data to and from video memory. A better approach is to keep the pixmap data in the X server address space where it was rendered and use it directly as the source for a texture operation. `GLX_EXT_texture_from_pixmap` provides the interface to make that happen.

As noted above, the ideal solution is to have the graphics card render the window contents into an off-screen buffer, which would then be used directly (i.e., with no copying or conversion) as the input to the hardware texture engine. To implement this solution, we will need additional infrastructure work (e.g., memory management) as well as additional card-specific driver work. Intermediate solutions are also possible.

One intermediate TFP solution is to redirect window data into host-memory pixmaps and call the texture operations directly through the new AIGLX abstraction layer interface to the Mesa/DRI card-specific driver. By rendering directly to host-memory pixmaps, we bypass the "read from framebuffer" operation, which is very slow—especially on agp hardware. This intermediate TFP solution is what is currently implemented and provides reasonable performance for the initial window/composite manager and toolkit work.

3.2.3 Framebuffer objects

The `GL_EXT_framebuffer_object` (FBO) extension [6], which was recently approved by the OpenGL Architectural Review Board 'superbuffers' working group, defines a way to render to destination buffers that are not the traditional front display buffer (e.g., depth or stencil buffers) and, further, it allows the destination to be other off-screen areas that can be used as a texture source. By allowing FBOs to be used both as an OpenGL render target and at a later time as a texture source, this extension provides the basic framework required to implement redirected OpenGL windows.

The proposed memory management work described above lays the groundwork for FBOs and makes the FBO implementation significantly easier because it generalizes the notion of *buffers*—i.e., it treats window-system framebuffers, textures, and FBOs the same. However, there is still significant infrastructure and card-specific driver work needed to generalize how the various buffers are used.

Once the memory management and FBO work is complete, redirected GLXWindows can be internally emulated by framebuffer complete FBOs within the X server for indirect rendering similar to how the Composite and TFP extension emulates X windows with X pixmaps. Additional work will be required for direct rendering to ensure that the DRI can handle emulated GLXWindows.

An additional issue is that since the Composite extension allows for redirection to be dynamic, AIGLX and the DRI will need to provide a mechanism for migrating from GLXWindows to FBOs that masquerade as GLXWindows and vice versa. However, the first implementation might require OpenGL apps to be restarted if an existing GLXWindow is redirected.

3.3 Composite overlay windows

There are a few cases where window output should not be redirected off screen; the most obvious being the output of the compositing manager itself. Early compositing managers painted their output directly to the root window, bypassing any compositing computations.

However, a GL-based compositing environment makes using the root window problematic. The existing GLX implementation assigns specific rendering abilities to each Visual: double buffering, alpha channel, etc. Usually, the root window is assigned a visual with minimal capabilities to avoid excess resource consumption. Without a way to assign appropriate resources, a GL-based compositing manager would have to accept whatever capabilities were assigned by the X server vendor.

In addition, these early 2D compositing managers painted their output to the root window in 'IncludeInferiors' mode; this mode bypasses the normal clipping which would otherwise obscure the rendering from areas of the screen covered by application windows. While core X and the Render extension both provide this IncludeInferiors mode, GLX does not, making it impossible to avoid the normal clipping.

Both of these problems can be solved. The FBconfigs mechanism from GLX 1.3 allows applications to assign alternate capabilities to GL contexts created for existing windows. And GLX could easily be extended to support IncludeInferiors drawing modes.

However, it's also quite easy to work around these limitations and leave most of the system unchanged. Create a special 'overlay' window that lies above all regular application windows and then create the compositing manager window as a child of the overlay window. This permits arbitrary selection of a Visual and eliminates all of the clipping issues.

The one remaining issue is dealing with mouse input, which now wants to bypass the overlay window and act on the real application windows. This is done by using the Shape extension to set the Input shape on the overlay window to an empty region, effectively eliminating the overlay window from participating in mouse events.

It is quite possible that this overlay window mechanism will eventually be superseded by the other mechanisms described above, in the meantime, this modest addition to the composite extension will serve for now.

3.4 Input transformation

While the Composite extension provides full control over the presentation of window content to the user, it completely ignores mouse input. If the composite manager doesn't precisely align window contents with their 'native' positions on the screen, chaos will ensue as the user can no longer use the position of the cursor to guide his or her mouse interactions.

To provide this complementary capability, the system must provide some mechanism for client control over the mapping from cursor coordinates to locations within the window hierarchy. The Compositing Manager must be given full control over the translation of root-relative coordinates to the position of the cursor within the appropriate window.

While the Composite extension's output redirection mechanism is reasonably simple to understand, the same is not true for input transformation. It may be that this author hasn't yet found straightforward semantics that would make this all "just work," or it may be that this is harder to implement than the output side.

3.5 Window/composite manager

Luminocity was a toy window/composite manager which allowed developers to rapidly prototype various effects and experiment with using OpenGL in a composited desktop. Luminocity could have been developed into a fully functional window manager, but this would have involved re-creating the years of work that went into developing Metacity. Instead, what was learned during the Luminocity project was applied to and re-implemented in Metacity.

The approach taken was to create a new OpenGL scene-graph based compositing library, called `libcm`, that encapsulated the methods used by the rest of the window manager to draw the desktop. Metacity could then hook various state transition animations into the scene-graph as needed.

By making the full OpenGL interface available, arbitrarily complex animations can be created that are only limited by what we can dream and what the hardware is capable of. Some common effects that have already been developed include various minimization, maximization, menu fade in/out, drop shadows, window transparency, and workspace switching. Many others can be developed as the need arises.

It should be noted that while most of the technologies described above are critical to the GL-based composited desktop, they have been developed to be completely general-purpose and can be used independently by all developers. For example, `compiz` [10] is another window/composite manager which takes a different approach, but works using the standard Xorg X server with the open source technologies currently under development [4].

4 Building X on OpenGL

Another X.org project, Xgl, is focused on replacing the rendering infrastructure within the X server with calls to OpenGL. By eliminating custom 2D rendering code, the goal is to gain access to the often highly optimized OpenGL implementation for the video card, reducing the amount of code necessary to support each card while improving performance at the same time.

While a GL-based X server doesn't seem very closely related to the work presented here, Xgl uses its access to the OpenGL API to provide accelerated indirect GLX functionality, including an implementation of the TFP extension. The result is an X server which also supports OpenGL-based compositing managers.

The key difference is that while the work presented here is an incremental addition to the existing X server architecture, Xgl represents a complete re-implementation of the X server input and drawing infrastructure. As all current OpenGL implementations run within the confines of a 2D window system, for Xgl to run, another window system must be running 'underneath' it. The eventual goal of the Xgl project is to replace the underlying window system with lightweight hardware management mechanisms.

5 Conclusion

We have surveyed many new technologies that will allow the Linux and other communities to implement a GL-based composited desktop. As of this writing, the initial implementations of AIGLX and TFP are scheduled to be included with X11R7.1 and a technology preview, which redirects windows to host-memory pixmaps is available at:

http://fedoraproject.org/wiki/RenderingProject/aiglx

Work on input transformation, advanced memory management, redirecting extensions (e.g., Xv, GL, DRI), frame buffer objects, FBconfigs, and full GLX 1.3 support are all currently in progress with the expectation that they will start appearing in the upstream development source code over the next several months. As each new technology appears, the window/composite manager, toolkits, and other desktop features will be updated to take advantages of the new features. The future of a GL-based composited desktop for Linux is looking very bright.

References

[1] Apple Computer. Quartz extreme. http://www.apple.com/macosx/features/quartzextreme/.

[2] Red Hat. Luminocity. http://live.gnome.org/Luminocity.

[3] Kristian Høgsberg. Aiglx update. http://lists.freedesktop.org/archives/xorg/2006-February/013326.html.

[4] Kristian Høgsberg. Compiz on aiglx. http://lists.freedesktop.org/archives/xorg/2006-March/013577.html.

[5] Precision Insight. Direct rendering infrastructure. http://dri.freedesktop.org/wiki/.

[6] Jeff Juliano and Jeremy Sandmel. Framebuffer object extension to opengl. http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.%txt.

[7] Sun Microsystems. Project looking glass. http://www.sun.com/software/looking_glass/.

[8] Keith Packard. Composite extension. http://cvs.freedesktop.org/xlibs/CompositeExt/protocol?view=markup.

[9] Brian Paul. Mesa 3d graphics library. http://www.mesa3d.org/.

[10] David Reveman. Compiz. http://en.opensuse.org/Compiz.

[11] Ian D. Romanick. Bringing x.org's glx support into the modern age. http://www.cs.pdx.edu/~idr/publications/ddc-2005.pdf.

[12] SGI. Opengl sample implementation. http://oss.sgi.com/projects/ogl-sample/.

[13] Keith Whitwell and Thomas Hellstrom. New dri memory manager and i915 driver update. http://www.tungstengraphics.com/xdevconf2006.pdf.