Matthias Hopf <mhopf@suse.de> SUSE Linux Products GmbH / Novell

This paper may be distributed under the UVM „unchanged“ license or under the Creative Commons „no derivs 1.0“ license.

compiz: The Next Generation Desktop

With MacOS X and – finally – Windows Vista, the hardware accelerated 3D desktop has become a reality for many users in these operating systems. Partially unnoticed, partially hyped, the X windows system has been enhanced in a way that allows for a similar and even better experience with 3D desktops. The best known 3D desktop today is based on the compositing window manager "compiz", which also provides a flexible plugin structure for enhancing its feature set.

This talk will present the base system 3D desktops are using for their effects, and will then continue with compiz, some of its plugins, and its history. It will show, how these 3D desktop effects help new users to grasp standard desktop metaphors and behavior, allow for better usability and accessibility, and deliver tons of neat eye candy to the beholder.

Background – The Ideas Behind Compositing

The basic principles behind the X11 window system have remained the same for many years. With the development of Apple's rendering system Quartz [1] it looked like the X11 system wasn't able to compete even rudimentary. And with the first previews of Windows Vista [2] (named *Longhorn* at that time) it looked like Microsoft was completely able to compete very soon. But behind the scenes, the infrastructure for a similar rendering model was already designed and implemented in X11. This went on, largely unnoticed by the public, even though some alternative routes were investigated (e.g. Looking Glass [3]) as well, which showed very promising and good looking early results.

All these new or revised windowing systems have in common, that there is a fundamental change in the rendering semantics. All graphics commands generally create a set of primitives, that have to be rendered to the framebuffer, which is then used by the graphics card to send the final image to the monitor. During this process, the system has to make sure that only primitives – or parts of primitives – are displayed that are not obscured by other windows. So far the typical way to deal with this problem has been to clip the primitives against the obscurers and render the remaining portions directly to the framebuffer (see Figure 1). This model has a set of fundamental problems, the two most obvious being the inability to do correct semitransparent blending

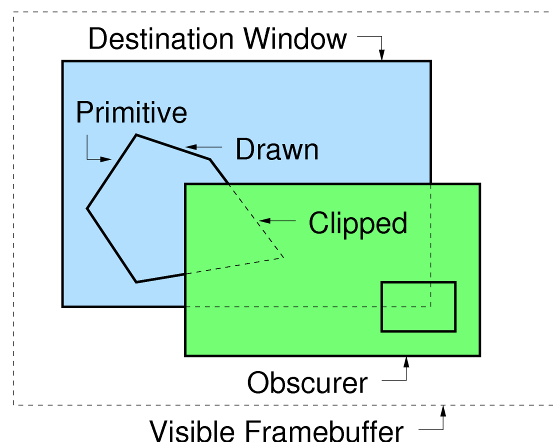

Figure 1: The original rendering model

of windows, and not being able to transform window contents in any way after rendering.

The remainder of this article will address the new so-called *compositing* model. In this model all primitives are rendered into per-window off-screen render buffers (that is, buffers that are *not* displayed on any monitor), and later – in a second step – composited several times per second into a final image by an additional process (see Figure 2). In both, Apple's and Microsoft's world, this process is an inherent part of the window system, while in X11 the protocol has been adapted to allow for an external process to do the compositing, very similar to window managers. This process is called the *composite manager*, and often it is integrated tightly into the window manager, creating a so-called *compositing window manager*. The most well-known compositing window manager nowadays is compiz.

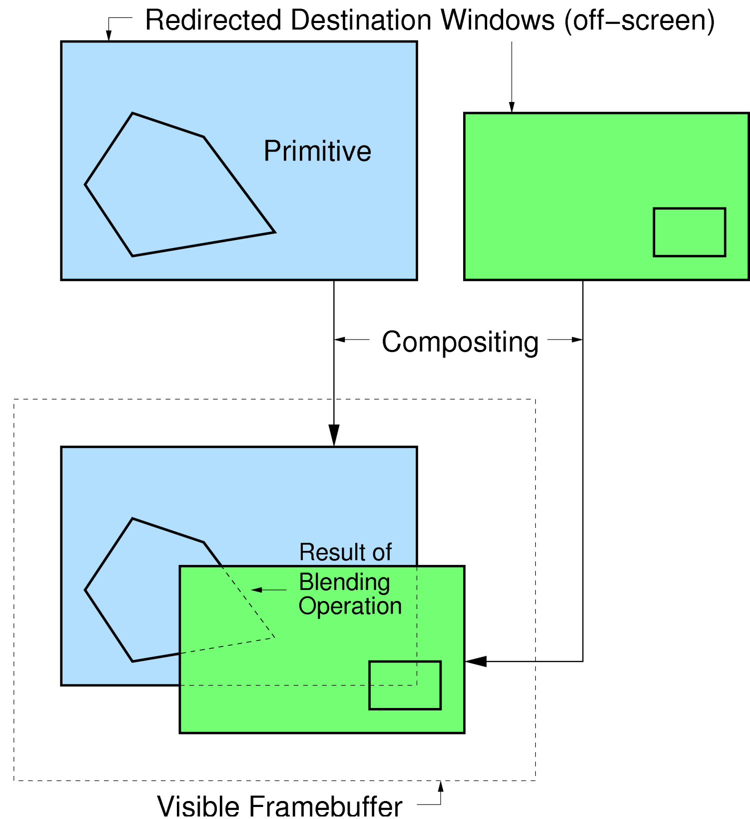

Figure 2: The compositing model

Composite & Render – Necessary Extensions

To achieve all this, all windows first have to be redirected to off-screen buffers, which are then composited either implicitly by the X server or explicitly by a composite manager into the visible frame buffer. This is exactly what the Composite extension does. It also makes the rendered window contents available to the composite manager in the form of pixmaps.

The first composite managers attempted to do the compositing itself using X11 calls alone. The original X11 rendering model did not have any notion of semitransparent images, let alone compositing techniques. The core functionality of the X model was already 20+ years old, and represented the primitives needed at that time, and not the much more elaborate primitives suitable for modern toolkits. For this the new Render extension was created, which adds new basic primitives for displaying images and polygons, along with a new glyph system for enhanced font display. All primitives can now be linked to data in the framebuffer using Porter-Duff operators, thus supporting the rendering of semitransparent surfaces (alpha blending) and fonts with antialiasing (pixel coverage). Many modern applications already make extensive use of antialiased fonts in particular. For more information about recent advancements in X11 outside the area of composited desktops see [4].

Compositing With OpenGL – Towards the 3D Desktop

Even with Render only simple affine 2D transformations are possible when compositing the window contents, and this greatly reduces the number possible effects. Thus, the first composite manager using OpenGL was born in 2005, glxcompmgr.

Humans are accustomed to understanding three-dimensional scenarios. It thus makes sense to project the GUI onto a three dimensional desktop, assuming the interactions with non-two-dimensional program representations are kept to a minimum. Though the idea of genuine 3D desktops sounds fascinating, real 3D interaction still poses a number of technical issues, is typically unintuitive, and thus still under investigation, especially in the virtual reality community.

Projecting two-dimensional pixel data onto three-dimensional objects is a standard application for OpenGL. At the same time, you get effects such as semitransparency more or less for free, as they are part of OpenGL's standard operations.

When a composite manager is active, more complexity is added to the graphics pipeline. As Figure 3 shows, the X server first redirects all window output to non-visible areas of the framebuffer. All X11 commands issued by an application are redirected to this memory space. This process occurs separately for each program. Then the composite manager binds these render buffers to textures, and draws the window contents using OpenGL primitives. The primitives are typically rectangles, but they can be more complex, three dimensional objects for transitions.

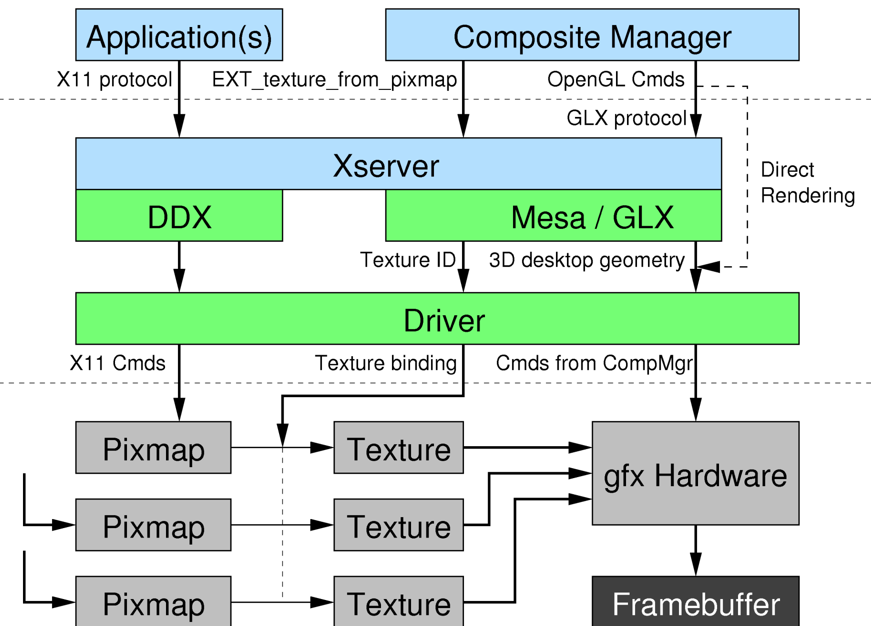

Figure 3: Pipeline using an OpenGL composite manager

Xgl vs. AIGLX – Misunderstood Endeavors

Another important X server component that desperately needs reworking is the hardware acceleration architecture, which is responsible for efficient hardware representation of graphic commands. The previous XAA architecture is built around core requests, and thus it is difficult to extend. The architecture outlived its usefulness and needs replacing. The most promising alternatives are EXA and OpenGL.

In 2005 David Reveman had an almost finished version of Xgl, an X server implementing all rendering calls by using glitz, an OpenGL accelerated backend

of cairo, which in turn is actually a userland representation and abstraction of the Render extension in library form.

In contrast to popular claims, Xgl does not accelerate the execution of OpenGL programs. On the contrary, only indirect rendering is possible for technical reasons at this time of writing. In other words, OpenGL commands are handed to Xgl via the GLX protocol before being passed on to the graphics hardware. Indirect rendering is much slower than direct rendering for programs that need to generate large numbers of polygons (games) or textures (video).

Xgl is also *not* responsible itself for the breathtaking effects we have seen so much of recently, however, it was the first system that allowed programmers to create an OpenGL based composite manager, which is then responsible for these effects [5][6].

In order to have an efficient OpenGL compositing system, the composite manager has to be able to bind the off-screen window pixmaps as textures. All solutions that are available today do this by providing the so-called `EXT_texture_from_pixmap` GLX extension. The textures have to reside in the memory space of the process that is doing the rendering, because otherwise they cannot be bound to the graphics card efficiently. In order to do that three solutions are possible:

1. Implement pixmap and texture space sharing in the OpenGL and X11 drivers
   This is the most complex model, but allows direct rendering in the composite manager and in additional OpenGL processes.

2. Implement Indirect OpenGL rendering in the X server
   In this case all graphics commands are sent to the graphics card by the X server, so the pixmaps and textures are in the same memory space by definition. The implementation is also very driver dependent.

3. Implement a middle layer on top of the X server that converts X11 into OpenGL commands
   In this case the middle layer is the actually acting X server, and it implements indirect OpenGL rendering.

When David Reveman started to implement an OpenGL based composite manager, he found that implementing the necessary GLX extension in Xgl was relatively straightforward, so option 3 was implemented and glxcompmgr saw the light of the day.

Almost in parallel to this endeavor the X.org community decided that it was about time to finally implement hardware accelerated indirect rendering in the X server, a feature that was available with sgi machines under IRIX – and also with Linux machines using the proprietary NVIDIA driver – for already quite some time. This was called the AIGLX project (**A**ccelerated **I**ndirect **GLX**). In the this process the implementation of `EXT_texture_from_pixmap` was almost a side effect, realizing option 2.

Only the proprietary NVIDIA driver so far realizes the full potential, by making `EXT_texture_from_pixmap` available even with direct rendering, implementing option 1.

So in effect AIGLX and Xgl were *not* competing projects aiming for 3D desktops as the same major goal, but they addressed different issues that both made the

implementation of the single most important feature for 3D composite managers possible. Unfortunately, neither of the three presented solutions is the ultimate answer right now, each of them suffers a number of problems.

Practically Xgl works extremely well nowadays, even with drivers that do not support the AIGLX extension, e.g. the proprietary ATI driver. But there was a lot of fuzz during the early days, when Novell decided to develop Xgl in-house in a non-open way, before it was released to the public again. While technically a good idea (you don't have to keep the same publishing quality during times of big changes, and there was no one developing on Xgl except David) it drove the open source community away from this solution. Also, at the current time Xgl cannot natively access the hardware; instead it relies on a system that initializes the framebuffer and provides an OpenGL interface. Right now, that is the popular Xorg server; in other words, Xgl opens a window that covers the whole screen on the Xorg server. After this has happened, X applications can connect to Xgl, while the standard X server only has to deal with the Xgl client throughout the whole session. Many people consider this combination to be a resource hog and not a very clean straightforward solution.

On the other hand, the AIGLX approach does still not work correctly with XVideo and OpenGL application support. Applications using these types of primitives will not be composited correctly, which can lead to rather strange effects. Fullscreen applications are typically not affected, though. Both, XVideo and OpenGL, work fine with Xgl – with XVideo being accelerated by OpenGL fragment shaders, but OpenGL applications only being able to use indirect rendering.

The NVIDIA proprietary driver had issues with OpenGL and Composite being activated at the same time. This has been pretty much resolved with newer driver versions, so technically they are working very well. But the drivers are not open source, and thus inherently clash with the open source community. While the X11 license allows for binary only drivers, the GPL of the kernel does not, and due to the (rather large) binary only kernel blob of the driver a lot of kernel developers consider the proprietary drivers being illegal, as they infringe the kernel license.

compiz – Merging Window Manager and Compositing

It turned out very soon, that window and composite managers would have to communicate a lot for many useful features that involve e.g. window positioning, or the display of small representations of windows. Therefore it was only reasonable to join these two processes, resulting in the first composting window manger, compiz [7].

Without this combination many features that actually improve the usability of desktops could not be implemented efficiently, reducing composite managers to pure eye candy.

In a compiz session, you might not notice that OpenGL is used for output. Shadows and slight semi-transparent window decorations are the only hints you get. But when you toggle to another virtual desktop, the three-dimensional nature of the desktop becomes very obvious (Figure 4).

In order be very flexible in terms of usability like window handling and the used feature set, a powerful plugin architecture was implemented, with only the most basic functionality remaining in the core. At the time of writing 27 plugins are part of the 0.5.0 release of compiz, with more being developed outside the main trunk. Table 1 lists the core plugins and their intention.

Figure 4: Desktop switching with compiz' cube plugin

An external program renders the window decorations and hands them over to compiz. This makes it easy to integrate themes or different sets of widgets. At present, there are window dressing programs for Gnome and KDE. The community is also working on a general themed decoration plugin.

OpenGL-based programs are often criticized for being eye candy and nothing more. However, this technology really does support useful developments in the field of accessibility aids for users with sensory impairments (using a zoom plugin, for example), and it supports the development of selection aids such as Exposé (scale plugin), which have proved very useful in the Mac world. Window thumbnails for application switching (Figure 5) are also very useful, especially if they are capable of showing the current live output from the program.

annotate	Allows free-hand drawing on screen
blur	Blurs semitransparent backgrounds
clone	Eases use of multi-monitor setups
cube	Standard cube workspace switcher
dbus	Im-/exports settings via dbus
decoration	Interface to decoration managers
fade	Fades windows in/out
fs	Im-/exports settings via fuse
gconf	Im-/exports settings via gconf
ini	Im-/exports settings via textfile
inotify	Internal file notification API
minimize	Animates minimization
move	Implements window movement
place	Implements initial window placement
plane	Alternative workspace switcher
png	Internal png file format support
regex	Support for window matching rules
resize	Implements window resizing
rotate	Implements the cube rotation effect
scale	Exposé-like effect
screenshot	Saves screenshots
svg	Internal svg file format support
switcher	Application switcher
video	Internal video support
water	Raindrops and water effects
wobbly	Wobbly windows using springs
zoom	Zooming effect

Table 1: The current core compiz plugins

compiz vs. beryl – Split and Reunification

Early in 2006 Quinn Storm joined the compiz development, and kept all kinds of experimental patches in her own CVS tree. Soon after that, there were some arguments on the compiz mailing list about patches that implemented eagerly wanted features into compiz, but weren't aligned to the plugin structure very well. David wanted to keep the core clean and implement everything in plugins,

while others were more focused on having the features implemented early.

This climaxed in June 2006, when Quinn announced a fork of compiz. This happened mostly due to still missing Xinerama support in upstream compiz, and due to licensing issues (compiz is using the BSD-alike X11 license, while Quinn and other contributors wanted to stick to the GPL. Shortly after that, the fork was renamed to Beryl, initially in order to avoid name clashes. Given the extremely open nature of the Beryl project, which happily accepted patches that weren't considered clean enough for compiz, a striving community built around it, which contributed a lot of code – though sometimes of doubtful quality, but also including a set of attractive plugins.

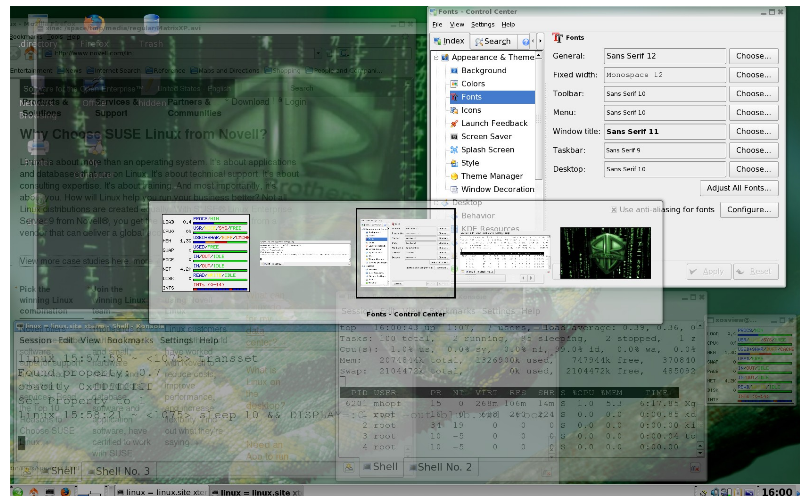

Figure 5: Application switching with live thumbnails.

In late March, only few months after the X.org developer's conference 2007, where things still looked irreconcilable, the main contributors on Beryl announced that they had reached consensus inside Beryl *and* compiz, that the two projects should reunite. This process is still ongoing, and will take a while to complete. Finally, the core will be the core of compiz modulo a few extensions needed by Beryl plugins, while the plugins of both projects will be hosted combined in a new project that is yet to be named.

The Future – Interaction in Composited Environments

Currently, parts of the community resources are surely bound by the reunification process, but development of core features certainly goes on. Major short- to mid-term goals of the main developers are:

- porting the core to use XCB instead of Xlib
  XCB is a new library that uses the same wire protocol as Xlib to talk to the X server, but a much enhanced and latency reduced API. This is considered to be the future of X11 programming. Newer versions of Xlib already use XCB internally.
- adding software cursors
  This allows to add effects on the cursor image and to always provide flicker-free cursors. Mostly done already.
- adding a video interface
  This will improve video playback, by applications not needing XVideo any more (which doesn't work well with AIGLX yet), and by removing one memcpy() of the video data. This partially done already.
- adding drawing synchronization
  This will help applications to synchronize better with desktop activities.

- adding a retained mode drawing interface
  Quite some applications already have to render things directly in the composited environment (e.g. the window decorators), so a more general interface is needed that allows for easy extensibility.
- adding input transformations
  That is *the* major missing piece holding back a lot of new possible usability mechanisms.

So far there has been no way in compiz to interact with applications that are not rendered in a simple 1:1 mapping (i.e. that are in their regular windowed state). Other projects like Looking Glass [3] or Metisse [8] have some simple input transformation almost since day one, but the solution in compiz should be more sophisticated and flexible, thus more difficult to design and implement. In the final solution the user should be able to work with any application on the screen, and if it was wrapped around a torus...

Conclusions – The Next Generation Desktop is Here

compiz has grown up in the last year, and is undoubtedly the most advanced and usable compositing window manager available today. Due to an active developer community it advances fast, and its flexible plugin architecture has shown to help a lot in developing new and exiting effects without compromising core code quality. The system has been stable enough for quite some time with some drivers that it is suitable for production use.

Packages for compiz are available in all major distributions, and many allow running compiz on top of Xgl, Xorg with AIGLX, or Xorg with binary NVIDIA drivers out-of-the-box.

Bibliography

[1] Apple's Quartz Extreme, http://www.apple.com/macosx/features/quartzextreme, 2005.

[2] Microsoft's Aero, http://www.microsoft.com/windows/products/windowsvista/features/details/aero.mspx, 2007.

[3] Project Looking Glass, https://lg3d-core.dev.java.net/, 2004.

[4] Matthieu Herrb and Matthias Hopf, New Evolutions in the X Window System, http://www.openbsd.org/papers/eurobsd2005/herrb-hopf.pdf, OpenBSDCon, 2005.

[5] Matthias Hopf, OpenGL-beschleunigter Desktop mit Xgl und Compiz, Linux Magazin, May 2006, pp. 42-45.

[6] Matthias Hopf, An OpenGL-accelerated desktop with Xgl and Compiz, Linux Magazine, July 2006, pp. 24-26.

[7] compiz - The Compositing Window Manager, http://www.compiz.org/, 2005.

[8] The Metisse Project, http://www.mandriva.com/projects/metisse, 2007.