USENIX Home • About USENIX • Events • Membership • Publications • Students

USENIX '04 Paper [USENIX '04 Technical Program]

Glitz: Hardware Accelerated Image Compositing using OpenGL

Peter Nilsson
Department of Computing Science Umeå University, Sweden
c99pnn@cs.umu.se

David Reveman
Department of Computing Science Umeå University, Sweden
c99drn@cs.umu.se

Abstract:

In recent years 2D graphics applications and window systems tend to use more demanding graphics features such as alpha blending, image transformations and anti-aliasing. These features contribute to the user interfaces by making it possible to add more visual effects as well as new usable functionalities. All together it makes the graphical interface a more hospitable, as well as efficient, environment for the user.

Even with today's powerful computers these tasks constitute a heavy burden on the CPU. This is why many proprietary window systems have developed powerful 2D graphics engines to carry out these tasks by utilizing the acceleration capabilities in modern graphics hardware.

We present Glitz, an open source implementation of such a graphics engine, a portable 2D graphics library that can be used to render hardware accelerated graphics.

Glitz is layered on top of OpenGL and is designed to act as an additional backend for cairo, providing it with hardware accelerated output.

Furthermore, an effort has been made to investigate if the level of hardware acceleration provided by the X Window System can be improved by using Glitz to carry out its fundamental drawing operations.

Introduction

There is a trend visible in the appearance of modern window systems and 2D graphics in general these days. They all become more and more loaded with graphical features and visual effects for each available product generation.

Unfortunately, these features means heavy computations that takes a lot of time when carried out by the general CPU. In the past this has meant a slowdown throughout the entire system as well as a significant limitation in the kind of visual effects that could be used.

In the field of 3D graphics, similar problems have been solved by implementing the drawing operations in dedicated 3D-graphics hardware. Hardware accelerated rendering means that the graphics hardware contains its own processor to boost performance levels. These processors are specialized for computing graphical transformations, so they achieve better results than the general purpose CPU. In addition, they free up the computer's CPU to execute other tasks while the graphics accelerator is handling graphics computations.

Modern window systems have developed 2D-graphics engines, which utilize the high performance rendering capabilities inherent in today's 3D-graphics hardware. In fact, much of the visual effects and advanced graphics that can be seen in these systems would not even be feasible without hardware accelerated rendering.

This paper presents Glitz, an open source implementation of a 2D graphics library that uses OpenGL[17] to realize a hardware accelerated high performance rendering environment.

Furthermore, these ideas have been applied to the X Window System (X)[16], to see if they could improve hardware acceleration of graphical applications and thereby making way for more advanced graphics.

The software that is developed in this project will primarily target users equipped with the latest in modern graphics hardware.

Traditional X Graphics

X was not originally designed for the advanced graphics that can be seen on modern desktops. The main reason is that graphics hardware available at the time was not fast enough. Application developers soon found the core graphics routines inadequate and the trend became to use client-side software rendering instead. Several steps have been taken to rectify this since then.

One major improvement was made with the introduction of the X Render

Extension (Render)[11]. Render has widely been accepted as the new rendering model for X. It brought the desired graphics operations to the applications and thereby filled in the gaps of the core protocol. Some of the features that Render supports are alpha compositing, anti-aliasing, sub-pixel positioning, polygon rendering, text rendering and image transformations. The core of Render is its image compositing model, which borrows fundamental notions from the Plan 9 window system[12]. Render provides a unified rendering operation, which supports the Porter-Duff[13] style compositing operators. All pixel manipulations are carried out through this operation. This provides for a simple and consistent model throughout the rendering system.

Render allows us to perform advanced graphics operations on server-side. Graphics operations that are performed on server-side can be accelerated by graphics hardware. XFree86's[9] Render implementation uses XFree86 Acceleration Architecture (XAA)[19] to achieve hardware accelerated rendering. XAA breaks down complex Render operations into simpler ones and accelerates them if support is provided by the driver, otherwise it falls back on software. To fall back on software means that all graphics computations are processed by the CPU. For most XFree86 drivers, image data lives in video memory, so for the CPU to be able to access this data it must first be fetched from video memory. The CPU can then perform its computations and the image data must then be transfered back to video memory. The result of such a software fall-back is most likely to be slower than if the operation would have been done on client-side with all image data already local to the CPU.

The ideal situation would be to have XAA Render hooks for all Render operations in every driver. This requires graphics driver developers to add XAA Render hooks in each driver, which results in a duplicated effort. Unfortunately, not many drivers have much support for XAA Render hooks at this point. This results in inconsistent acceleration between different rendering operations, different drivers and different hardware.

Glitz Fundamentals

The Render model seems to be ideal to build Glitz upon. It provides the necessary operations needed to carry out the rendering for modern 2D graphics applications. Hence Glitz is designed to exactly match the Render model semantics, adding an efficient and more consistent hardware acceleration of the rendering process.

The Render model provides only low level fundamental graphics operations, not always suitable for direct use by application developers. A higher level graphics API is needed on top of the Render model to make it useful for this purpose. The

cairo library (formerly known as Xr[20]) is a modern, open source, cross-platform 2D graphics API designed for multiple output devices. With its PDF[3]-like 2D graphics API, it provides an attractive and powerful vector based drawing environment. Cairo uses a backend system to realize its multiple output formats. One thing missing thus far in cairo, is a backend that efficiently accelerates the rendering process with today's high performance graphics hardware. The backend interface of cairo has the same semantics as Render. Thus Glitz is designed to act as an additional backend for cairo providing this hardware accelerated output.

The output of Glitz is accelerated in hardware by using OpenGL for all rendering operations. Figure 1 illustrates these ideas by showing the layers involved when an application uses cairo to draw hardware accelerated graphics through Glitz.

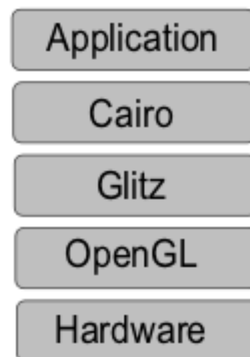

Figure 1: *Different layers involved when an application uses cairo to render graphics to an OpenGL surface.*

OpenGL can be used to accelerate 2D graphics output, just as with 3D. Most people think of OpenGL as a 3D graphics API, which is understandable because it was used primarily for 3D applications like visualizations and games in the past. However, it is just as well suited for 2D graphics of the nature discussed in this paper, where transformations and other visual effects play a central part, much like in traditional 3D applications. OpenGL is the most widely used and supported graphics API available today, it is well supported in hardware and very portable. It operates on image data as well as geometric primitives and offers the necessary operations needed for the creation of Glitz.

To sum up these ideas, Glitz is created to act as an interface on top of OpenGL, which provides the same consistent rendering model as Render. This interface is implemented in such a way that it takes advantage of the OpenGL hardware acceleration provided by modern graphics cards. The semantics of the library

are designed to precisely match the specification of Render. Having the same semantics as Render allows for a seamless integration with the cairo library that then provides an attractive environment for developing new high performance graphical applications.

Hopefully, the work presented in this paper will be useful in the design of a new generation of hardware accelerated 2D graphics applications for X and the open source community in general.

The OpenGL Rendering Pipeline

To fully utilize the hardware acceleration provided by OpenGL, familiarity with the order of internal operations used in OpenGL implementations is of great importance. This is often visualized as a series of processing stages called the OpenGL rendering pipeline. This ordering is not a strict rule of how OpenGL is implemented but provides a reliable guide for predicting what OpenGL will do.

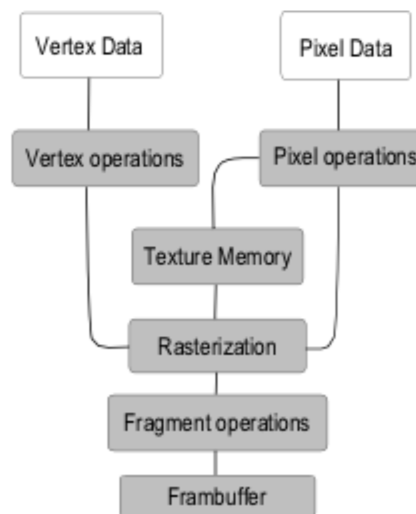

Figure 2: *The OpenGL Rendering Pipeline*

The latest generations of graphics hardware allow the application to replace the conventional vertex operations and fragment operations processing stages shown in figure 2, with application defined computations. These computations are defined with assembler-like languages that an OpenGL implementation compiles into vertex and fragment programs. The vertex and fragment programs provide an interface for directly accessing hardware and have been proven very useful in the development of Glitz.

Related Work

Some of the proprietary window systems have created their own graphics engines that can perform hardware accelerated rendering in a similar manner to the model discussed here. The one, that has probably attracted most attention is Apple's Quartz Extreme[5] compositing engine used in Mac OS X[6]. The user interface in Mac OS X is loaded with advanced graphics effects of the nature discussed in this paper. They all seem to run smoothly without bringing too much load on the CPU.

Microsoft is also developing something similar in their Avalon[2] graphics engine. It will be a fundamental part for hardware accelerated 2D graphics in the next windows version, currently being developed under the name Windows Longhorn[1].

Glitz is not the first Open Source graphics library that has been layered on top of OpenGL. An example, Evas[14]; a hardware accelerated canvas API, which is part of the Enlightenment Foundation Libraries. Glitz is unique compared to these libraries by using an immediate rendering model designed for the latest hardware extensions. Immediate data is resident in graphics hardware and off-screen drawing is a native part of the rendering model.

Implementation and Design

The development of Glitz and the other parts have been made in an entirely open fashion. Input from the open source community has been regarded.

Library Structure

As OpenGL layers are available for various platforms and systems, the library is designed to be usable with any of various OpenGL layers. Different OpenGL layers can be used by plugging them in to the core of Glitz through a virtualized backend system. Each backend needs to provide a table of OpenGL function pointers and few additional functions, which will allow for the core to perform its rendering operations unaware of the structure of the underlying OpenGL layer.

The core of Glitz is built as a separate library with a minimal set of dependencies. Each backend is built into its own library and different applications can use different backend libraries with the same core library.

The advantages of having a virtualized backend system and different backend libraries instead of just choosing the code to compile using preprocessor macros

are important. It makes the link between the OpenGL implementation and the core of the library more flexible. It allows for the core of the library to be compiled for multiple backends.

As of now Glitz has two backends, for GLX[7] and AGL[4]. GLX is the OpenGL layer used on Unix[18] like systems to provide a glue layer between OpenGL and X. AGL is an OpenGL layer available in Mac OS. Backends for other OpenGL layers can be added in the future.

The Render protocol describes an immediate rendering model. This means that the application itself maintains the data that describes the model. For example, with Render you draw objects by completely specifying what should be drawn. Render simply takes the data provided by the application and immediately draws the appropriate objects.

The opposite is to have a retained rendering model. A rendering model is operating in retained mode if it retains a copy of all the data describing a model. Retained mode rendering requires a completely specified model by passing data to the rendering system using predefined structures. The rendering system organizes the data internally, usually in a hierarchical database.

Principal advantages of immediate mode rendering includes a more flexible model and immediately available data that is not duplicated by the rendering system. However, it is more difficult to accelerate the immediate rendering model, because you generally need to specify the entire model to draw a single frame, whether or not the entire model has changed since the previous frame.

Off-screen Drawing

Off-screen drawing is an essential part of an immediate mode 2D graphics API. Support for off-screen drawing in OpenGL has been around for a long time on IRIX[8] systems and other workstations, but it is not until recently that it has become a standard feature on the regular home desktop computer.

Pixel buffers or so called pbuffers are what make off-screen rendering possible in OpenGL. Pbuffers are allocated independently of the frame-buffer and usually stored in video memory. The process of rendering to pbuffer is accelerated by hardware in the same way as rendering to the frame-buffer. However, as the pbuffer is a relatively new feature in the OpenGL world, it is not yet supported by all hardware and all drivers. When support for off-screen drawing is missing, the application using Glitz will have to handle this on its own. Even though Glitz is primarily designed for modern graphics hardware, it is important to be able

to fall back on software rendering in cases where Glitz is not able to carry out off-screen drawing operations. For example, the cairo library handles this gracefully using image buffers.

User-Provided Immediate Data

As all representation of pixel data within Glitz reside in the graphics hardware's memory, application generated images must be transmitted to the hardware in some way. For this purpose Glitz provides two functions, one for transmitting pixel data to graphics hardware and one for retrieving pixel data from graphics hardware.

Image Compositing

The general composite operation is the fundamental part of the rendering model. It takes two source surfaces and one destination surface, where the second of the two source surfaces is the optional mask surface. Figure 3 illustrates this operation.

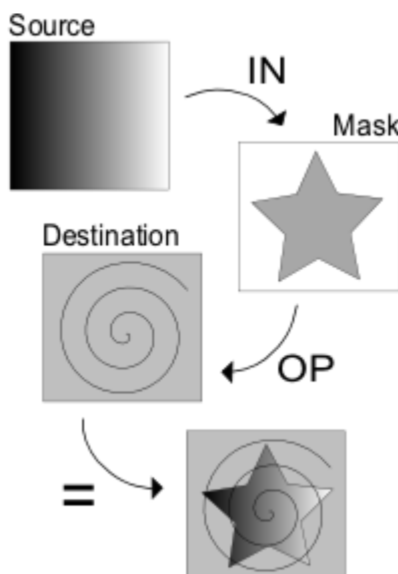

Figure 3: *src IN mask OP dst*

To composite one surface onto another with OpenGL, texturing of a rectangular polygon is used. This means that the source surfaces must be available as textures. The default method for making a surface available as a texture is to have the graphics hardware copy the pixel data from a drawable into a texture image. As this is a relatively slow operation, Glitz does its best to minimize the area and number of occasions for this copy operation. On some hardware, a feature called render-texture is available that allows Glitz to completely

eliminate the need for this copy operation and instead use a pbuffer directly as a texture.

The optional mask surface that can be provided to the general composite operation creates some additional complications. The source surfaces must first be composited onto the mask using the Porter-Duff in-operator and the result must then be composited onto the destination. The default method for handling this is to create an intermediate off-screen surface, which can be used for compositing using the in-operator. This surface can then be composited onto the destination with an arbitrary operator for the correct result.

The best way to do this would be to perform compositing with a mask surface directly without the creation of an intermediate surface. Even though the fixed OpenGL pipeline does not seem to allow for such an operation, Glitz is able to do this on hardware that support fragment programs. Fragment programs allow for fragment level programmability in OpenGL, and in combination with multi-texturing, Glitz can perform composite operations with a mask surface very efficiently.

Image Transformations

Image transformation is a natural part of OpenGL and is efficiently done on all available hardware and with all available OpenGL implementations. Glitz transforms the vertex coordinates of the rectangular polygon used for texturing, and OpenGL will then in hardware handle fetching of correct source fragments for this polygon.

When using fragment programs for direct compositing with mask surfaces, some transformations cannot be done since the source surface and the mask surfaces share vertex coordinates. If this is the case, Glitz will be forced to not use direct compositing.

Repeating Patterns

To provide for solid colors and repeating patterns, surfaces have a `repeat' attribute. When set, the surface is treated as if its width and height were infinite by tiling the contents of the surface along both axes.

Normally OpenGL only supports tiling of textures with dimensions that are power of two sized. If surface dimensions are of this size Glitz can let OpenGL handle the tiling for maximum efficiency. For surfaces that do not have power of two sized dimensions, Glitz will repeat the surfaces manually by performing multiple texturing operations.

Some OpenGL implementations support tiling of none power of two sized textures as well. If this is the case, Glitz will let OpenGL handle tiling of all surfaces.

Polygon Rendering

Glitz supports two separate primitive objects; triangles and trapezoids. Triangles are specified by locating their three vertices. Trapezoids are represented by two horizontal lines delimiting the top and bottom of the trapezoid, and two additional lines specified by arbitrary points. These primitives are designed to be used for rendering complex objects tessellated by higher level libraries.

Glitz only supports imprecise pixelization. Precise pixelization is not supported since OpenGL has relatively weak invariant requirements of pixelization. This is because of the desire for high-performance mixed software and hardware implementations. Glitz matches the following set of invariants for imprecise polygons.

- Precise matching of abutting edges
- Translational invariance
- Sharp edges
- Order independence

Hence the visual artifacts associated with polygon tessellation and translation are minimized.

Anti-aliasing

Aliasing is a general term used to describe the problems that may occur whenever an analog signal is point sampled to convert it into a digital signal, and the sample rate is to low. The number of samples do not contain enough information to represent the actual source signal. Instead the samples seem to represent a different signal of lower frequency, called an aliased signal.

In computer graphics, aliasing translates to the problems related to point sampling an analogous mathematical representation of an image into discrete pixel positions. With the currently available display devices it is simply not feasible to sample a non aliased signal, the resolution of the screen (the number of samples) is simply not high enough.

The results of aliasing are called artifacts. The most common artifacts in

computer graphics include jagged profiles, disappearing or improperly rendered fine detail and disintegrating textures. The most obvious one, and the one that most applies to 2D graphics, is the jagged profile artifact. Figure 4 illustrates an aliased graphical image suffering from a jagged edge.

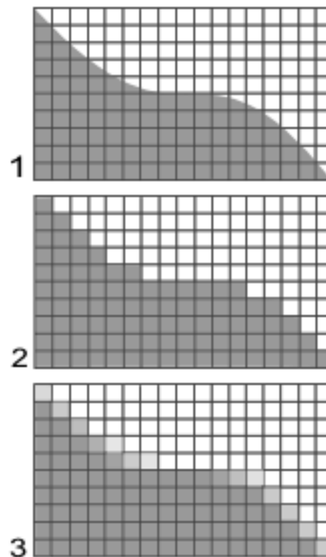

Figure 4: *(1) The mathematical representation of an edge (2) The edge, point sampled to screen pixels (3) The anti-aliased edge*

Anti-aliasing, naturally enough, is the name for techniques designed to reduce or eliminate this effect, usually by shading the pixels along the borders of graphical elements. There are several techniques that can be used to achieve anti-aliased graphics rendering with the OpenGL API. The most common techniques include:

- OpenGL's built in polygon smooth hint
- Multi-pass using accumulation buffering
- Multi-pass using blending
- Full-scene anti-aliasing using software super-sampling
- Full-scene anti-aliasing using hardware assist

All these anti-aliased drawing techniques are approximations. Each has its advantages and disadvantages. Different methods are suitable in different application contexts and have various support in graphics hardware and drivers. These methods have been investigated and evaluated with regards to performance and the result of the actual visual output. The challenge is to find a technique, or a combination of techniques, that will be able to provide nice

anti-aliasing of the rendered scene in an as wide spectra of hardware and drivers as possible. As important as a nice on-screen result might be, performance issues are given a high priority in the search for a suitable anti-aliasing model. Another important criteria has been that they do not all fit well into the immediate rendering model used in Glitz.

The anti-aliasing model chosen for Glitz is flexible and other techniques can easily be added for special cases later on. The current implementation uses hardware assisted full-scene anti-aliasing.

This technique has been found suitable because it has a functional easy to use interface in OpenGL (through the multi-sample extension) and it fits well into Glitz without complicating the structure of the rendering model. It is relatively fast on current hardware and it produces adequate results in real-time rendering. The trend among graphics hardware manufacturers seems to be to favor multi-sampling over other anti-aliasing techniques in new products.

Full-scene anti-aliasing using hardware assist is typically implemented as multi-sampling, sometimes super-sampling. This is a very fast model that works for all primitives, interrelationships, and rendering models. It is also well supported in current hardware, since a couple of graphics card generations back. Some extra memory is required, but typically less than for software super-sampling or accumulation buffering. It yields decent-quality results but some people may not find them acceptable for small text. This does not affect the choice in this case however, as anti-aliasing of text will preferably be handled by an external font rendering library. On high end systems this technique has potential for generating extremely high quality results with a relatively low cost. Unfortunately, it is not always available for off-screen buffers (pbuffers).

The other techniques have been discarded mainly due to poor hardware support, high memory consumption, bad performance or poor results.

Indirect Polygons

Glitz has two different methods for rendering indirect polygons. Using an intermediate off-screen surface or using a stencil buffer.

The first method creates an off-screen surface containing only an alpha channel. The polygons are then rendered into this intermediate surface, which is used as mask when compositing the supplied source surface onto the destination surface. This method requires off-screen drawing support, and anti-aliased polygon edges can only be rendered if off-screen multi-sample support is available.

Whenever a stencil buffer is available, it will be used for drawing indirect polygons. The polygons are then rendered into the stencil buffer and the stencil buffer is used for clipping when compositing the supplied source surface onto the destination surface. This method for drawing indirect polygons is faster and does not require off-screen drawing support. When rendering to on-screen surfaces only on-screen multi-sample support is needed for anti-aliased polygons.

Indirect polygons can be used for pattern filling of complex objects.

Direct Polygons

Glitz is able to render polygons directly onto a destination surface. Each polygon vertex has a specific color associated with it and colors are linearly interpolated between the vertices. Direct polygons have the advantages of not requiring an intermediate off-screen surface or stencil buffer and are therefore faster, and supported on more hardware. Direct polygons might not produce the same results as indirect polygons when the alpha color component is not equal to one and should as a result not be used for complex objects with these properties. The more general indirect polygons should instead be used in these cases.

Text Rendering

Current version of Glitz has no built in text support. Glyph rasterization and glyph management could however be handled by the application or a higher level library. For efficient text rendering, glyph-sets with off-screen surfaces containing alpha masks, should be used. With external glyph management, Glitz renders text at approximately 50000 glyphs per second on the test setup described in section Results.

Built in text handling is planned for future versions of the library and tests have indicated that this should increase glyph rendering speed to around 200000 glyphs per second.

Clipping

Render can restrict read and writes to a drawable using a clip-mask. Clients can create this clip-mask on their own or implicitly generate it using a set of rectangles. Glitz has a similar clipping interface but the clip-mask cannot be created by the application, it must always be implicitly generated from a set of

rectangles, triangles and trapezoids. With Glitz, clipping only restricts writing to a surface. Glitz's clipping interface cannot restrict reading of a surface.

Programmatic Surfaces

Glitz allows you to create programmatic surfaces. A programmatic surface does not contain any actual image data, only a minimal set of attributes. These attributes describe how to calculate the color for each fragment of the surface.

Not containing any actual image data makes initialization time for programmatic surfaces very low. Having a low initialization time makes them ideal for representing solid colors.

Glitz also support programmatic surfaces that represent linear or radial transition vector patterns. A linear pattern defines two points, which form a transition vector. A radial gradient defines a center point and a radius, which together form a dynamic transition vector around the center point. The color of each fragment in these programmatic surfaces is fetched from a color range, using the fragments offset along the transition vector.

A color range is a one dimensional surface. The color range data is generated by the application and then transfered to graphics hardware where it can be used with linear and radial patterns. This allows applications to use linear and radial patterns for a wide range of shading effects. For example, linear color gradients and Gaussian shading. By setting the *extend* attribute of a color range to *pad*, *repeat* or *reflect*, the application can also control what should happen when patterns try to fetch color values outside of the color range.

Most programmatic surfaces are implemented using fragment programs and they will only be available on hardware supporting the fragment program extension.

Figure 5 shows the results from using programmatic surfaces for linear color gradients.

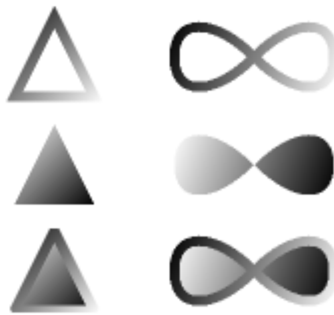

Figure 5: *Programmatic surfaces used for linear color gradients*

Convolution Filters

Convolutions can be used to perform many common image processing operations including sharpening, blurring, noise reduction, embossing and edge enhancement.

A convolution is a mathematical function that replaces each pixel by a weighted sum of its neighbors. The matrix defining the neighborhood of the pixel also specifies the weight assigned to each neighbor. This matrix is called the convolution kernel.

Glitz supports user defined convolution kernels. If a convolution kernel has been set for a surface, the convolution filter will be applied when the surface is used as a source in a compositing operation. The original source surface remains unmodified.

In Glitz, convolution filtering is implemented using fragment programs and is only available on hardware with fragment program support. The alternative would be to use OpenGL's imaging extension, which would require a transfer of all pixels through OpenGL's pixel pipeline to an intermediate texture. Even though the alternative method would be supported by older hardware, Glitz uses fragment programs in favor of performance.

This is an example of a convolution kernel representing a gaussian blur filter.

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Figure 6 shows an image before and after applying a gaussian filter using the convolution kernel above.

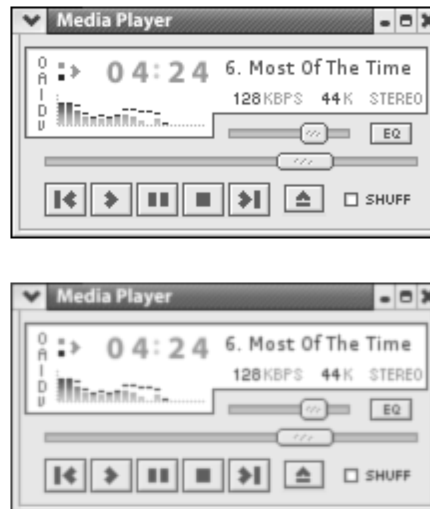

Figure 6: *An image before and after applying a Gaussian convolution filter*

A Cross-platform OpenGL Layer

Glitz's backend system works as an abstraction layer over the supported OpenGL layers and has genuine support for off-screen drawing.

In addition to the 2D drawing functions, Glitz also provides a set of functions that make it possible to use Glitz as a cross-platform OpenGL layer. The following three functions allow the application to use ordinary OpenGL calls to draw on any Glitz surface.

- glitz_gl_begin (surface)
- glitz_gl_end (surface)
- glitz_get_gl_texture (surface)

An application can initiate ordinary OpenGL rendering to a Glitz surface by calling the glitz_gl_begin function with the surface as parameter. The surface can be either an on- or off-screen surface. After a call to glitz_gl_begin, all OpenGL drawing will be routed to the glitz surface. The glitz_gl_end function must be called before any other Glitz function can be used again.

An application can use both Glitz's 2D drawing functions and ordinary OpenGL calls on all surfaces as long as all OpenGL calls are made within the scope of glitz_gl_begin and glitz_gl_end.

glitz_get_gl_texture allows the application to retrieve a texture name for a Glitz surface. The application can use this texture name with ordinary OpenGL calls.

```
offscreen_surface =
  glitz_glx_surface_create (display, screen,
                            GLITZ_STANDARD_ARGB32,
                            width, height);

glitz_fill_rectangle (GLITZ_OPERATOR_SRC,
                      offscreen_surface,
                      clear_color,
                      0, 0, width, height);

/* draw things to offscreen surface using
   glitz's 2D functions ... */

texture =
  glitz_get_gl_texture (offscreen_surface,
                        &name, &target,
                        &tex_width,
                        &tex_height);

glitz_gl_begin (onscreen_surface);

/* set up projection and modelview
   matrices ... */

glEnable (target);
glBindTexture (target, name);

glBegin (GL_QUADS);
glTexCoord2d (0.0, 0.0);
glVertex3d (-1.0, -1.0, 1.0);
glTexCoord2d (tex_width, 0.0);
glVertex3d (1.0, -1.0, 1.0);
glTexCoord2d (tex_width, tex_height);
glVertex3d (1.0, 1.0, 1.0);
glTexCoord2d (0.0, tex_height);
glVertex3d (-1.0, 1.0, 1.0);
glEnd ();

glitz_gl_end (onscreen_surface)

glitz_surface_destroy (offscreen_surface);
```

Figure 7: Rendering 3D graphics with a Glitz surface as texture

Figure 7 shows an example that render 2D graphics to an off-screen surface and then use it as a texture when drawing in 3D.

Applications, libraries and toolkits that use Glitz as rendering backend will get both 2D and 3D support with the ability two use all 2D surfaces as textures for 3D rendering.

Results

Right now the library has not been tested that much in real applications since it is relatively early in the development process. Some test and benchmark utilities have been developed to analyze library functionality with respect to accuracy and performance.

Accuracy

The quality of the visual output from Glitz compares quite well to the corresponding output from XFree86's Render implementation (Xrender). Most objects tessellated by the cairo library are rendered without noticeable differences compared to Xrender using Glitz. However, in some complex objects a slight variation can be seen between Glitz's output and Xrender's output. Figure 8 and 9 illustrates the output inconsistencies for aliased rendering. The infinite sign is tessellated into 370 trapezoids by the cairo library, which are then rendered using Glitz and Xrender. Both figures are magnified to better show the output differences.

Figure 8: *Aliased OpenGL output*

Figure 9: *Aliased xrender output*

Anti-aliased rendering may introduce some additional inconsistencies in the output between Glitz and Xrender. The number of samples used for multi-sampling has a big effect on the anti-aliasing quality. On older hardware, anti-aliasing is not even guaranteed, as it depends on relatively new OpenGL extensions. Nevertheless, if Glitz is run on fairly modern graphics hardware, very similar results are achieved with anti-aliased output. Figure 10 and 11 show the same infinite sign and illustrates the output inconsistencies for

anti-aliased rendering.

Figure 10: *Anti-Aliased OpenGL output (using 4 samples multi-sampling)*

Figure 11: *Anti-Aliased Xrender output*

Even though these results appear somewhat different in these magnified figures, the performance gained by using these OpenGL accelerated anti-aliasing techniques by far makes up for this. In most cases the generated results are close to indistinguishable.

Performance

A number of test and demo applications have been created during the development of Glitz to verify performance and functionality. This section presents results from a benchmark utility named rendermark. Rendermark compares the rendering performance of Glitz, Xrender and Imlib2[15] by doing a set of basic operations a repeated number of times. Comparison with Imlib2 is interesting as it is promoted as the fastest image compositing, rendering and manipulation library for X. Imlib2 performs all its rendering operations on client-side image buffers, so no on-screen rendering results are available for Imlib2.

Table 1 lists the tested output formats and table 2 shows the test setup.

Table 1: *Rendermark Output Formats*

im-off	Imlib2 off-screen
xr-on	Xrender on-screen
xr-off	Xrender off-screen

gl-on	GL on-screen
gl-off	GL off-screen

Table 2: *Test Setup*

CPU	1GHz Pentium 3
OS	Linux 2.4.22
X11	XFree86 4.3.0
GPU	Nvidia GeForce FX-5600 (Nvidia's binary driver)

Each test is repeated a thousand times and the total time is shown in the tables.

Image Compositing

Image compositing performance is very important. For Xrender and Glitz this tests the composite primitive used for basically all rendering operations. Good performance here means good performance throughout the whole system.

- *over.* blends one image onto another using the over operator.
- *scale.* blends one image onto another using the over operator with additional scaling factors. Bilinear filtering is used.
- *blend.* blends one image onto another using the over operator with half opacity.
- *blur.* blends one image onto another using the over operator with a blur filter. For Glitz, this means applying a 3x3 mean blur convolution filter. The version of Xrender used for this test does not support convolution filters, and the test is therefore skipped.

Table 3 shows the image compositing results.

Table 3: *Seconds to complete composite test (lower is better)*

	im-off	xr-on	xr-off	gl-on	gl-off

over	3.809	3.870	3.850	0.109	0.107
scale	16.444	85.504	86.924	0.126	0.132
blend	4.349	73.222	69.613	0.264	0.263
blur	26.499	-	-	3.089	3.078

Color blend

This test evaluates color blend performance by drawing rectangles.

Table 4 shows the color blend results.

Table 4: *Seconds to complete color blend test (lower is better)*

	im-off	xr-on	xr-off	gl-on	gl-off
rect	3.824	75.346	73.617	0.108	0.105

Polygon Drawing

Polygon drawing is extensively used when rendering vector graphics. These tests show the performance when rendering the simplest polygon type, the triangle.

- *tri1.* Draws a set of triangles, each in a separate rendering operation.
- *tri2.* Draws a set of triangles in one single rendering operation. This type of operation is often used for rendering complex objects, which have been tessellated into (in this case) triangles. Imlib2 skips this test as it lacks support for it.

Table 5 shows polygon drawing results.

Table 5: *Seconds to complete triangle drawing test (lower is better)*

	im-off	xr-on	xr-off	gl-on	gl-off
tri1	2.977	66.555	63.191	0.072	0.072
tri2	-	2.078	1.950	0.030	0.030

Gradient Drawing

Tests linear color gradient performance. Xrender skips this test as it lacks support for it.

Table 6 shows gradient drawing results.

Table 6: *Seconds to complete gradient drawing test (lower is better)*

	im-off	xr-on	xr-off	gl-on	gl-off
grad	6.281	-	-	1.065	1.117

Hardware Accelerated Xrender

Nvidias's[10] binary XFree86 drivers contains an experimental feature that allows the driver to hardware accelerate the Render extension on XFree86's X server. Some Render operations are known to perform extremely good with this feature turned on.

Table 7: *Seconds to complete test (lower is better)*

	xr-on	xr-off	gl-on	gl-off
over	0.113	0.185	0.109	0.107
scale	84.659	86.217	0.126	0.132
blend	0.116	0.181	0.264	0.263

blur	-	-	3.089	3.078
rect1	0.066	0.159	0.111	0.108
rect2	0.070	0.228	0.113	0.118
tri1	3.300	3.085	0.072	0.072
tri2	1.597	1.688	0.030	0.030
grad	-	-	1.065	1.117

Table 7 shows that nvidia's driver performs well compared to Glitz in the cases where no transformations are used. In cases where transformations are used, Glitz is much faster than nvidia's driver, which most likely falls back on software rendering.

Conclusion

During the development of Glitz we have found that with the OpenGL API and the extensions available today, along with the wide range of hardware supporting them, a Render-like interface on top of OpenGL is viable and very efficient. This is an important conclusion as the desire for having an X server running on top of OpenGL grows rapidly.

The benchmark results points out Glitz's remarkable rendering performance. Even Imlib2's highly optimized rendering engine is no where near Glitz's performance.

Although performance is of high importance, the greatest advantage with Glitz is that it provides a general way for accelerating the Render imaging model.

Future Work

Today, the existing implementation of the library supports all the basic functionality, which where initially set up for the project. However, there are still some important features missing, and the software is in an early stage of development with a lot of work remaining to make it stable and optimized with regards to performance and accuracy.

The following list contains those features that most importantly need to be addressed in future versions of the library.

- *Text rendering.* Built in text rendering will allow much higher glyph rendering speeds and remove complex glyph management from the application.
- *Sub-pixel rendering.* Sub-pixel rendering can be used to effectively increase the horizontal resolution of LCD displays. This will require support for compositing each color component with different masks.

The future will most certainly demand new features from the library, since it is an area of continuous development.

Visions

The X desktop seems to be going into a new era and cairo is definitely the 2D graphics API that will be used in tomorrow's X applications. The support for hardware accelerated surfaces in cairo might then be of great importance. Plans for the creation of an X server that will use OpenGL for all rendering are currently being made and this library, or the work behind the library, can hopefully be usable for this purpose.

Acknowledgments

We would like to thank Keith Packard, Carl Worth, and all of the people involved in the development of cairo for being helpful and encouraging. We would also like to thank our internal supervisor Berit Kvernes, along with the staff at the department of Computing Science at Umeå University, for supporting us in this project by approving it for financial funding in terms of study allowances.

Availability

All source code related to this project is free software currently distributed under the MIT license. The license of Glitz will follow that of cairo in case of changes.

The source can be retrieved via anonymous pserver access from the cairo CVS (`anoncvs@cvs.cairographics.org:/cvs/cairo`). The current status of Glitz and some additional information is available at `http://www.freedesktop.org/software/glitz`.

Bibliography

1

Microsoft Corporation.
Online Resources: Windows Longhorn, December 2003.
http://msdn.microsoft.com/longhorn.

2
Microsoft Corporation.
Online Resources: Avalon, April 2004.
http://msdn.microsoft.com/Longhorn/understanding/pillars/avalon/default.aspx.

3
Adobe Systems Inc., editor.
PDF Reference: Version 1.4.
Addison-Wesley, 3rd edition, 2001.

4
Apple Computer Inc.
Online Resources: AGL, April 2004.
http://developer.apple.com/opengl/.

5
Apple Computers Inc.
Online Resources: Quartz Extreme, Faster Graphics, December 2003.
http://www.apple.com/macosx/features/quartzextreme.

6
Apple Computers Inc.
Online Resources: Mac OS X, April 2004.
http://www.apple.com/macosx.

7
Silicon Graphics Inc.
Online Resources: GLX, April 2004.
http://www.sgi.com/software/opensource/glx/.

8
Silicon Graphics Inc.
Online Resources: IRIX, April 2004.
http://www.sgi.com/software/irix/.

9
XFree86 Project Inc.
XFree86: an open source X11-based desktop infrastructure, April 2004.
http://www.xfree86.org.

10

Online Resources: NVIDIA.
Nvidia, April 2004.
http://www.nvidia.com.

11
Keith Packard.
A New Rendering Model for X.
In *FREENIX Track, 2000 Usenix Annual Technical Conference*, pages 279-284, San Diego, CA, June 2000. USENIX.

12
Rob Pike.
draw - screen graphics.
Bell Laboratories, 2000.
Plan 9 Manual Page Entry.

13
Thomas Porter and Tom Duff.
Compositing Digital Images.
Computer Graphics, 18(3):253-259, July 1984.

14
Rasterman.
Online Resources: EVAS, April 2004.
http://enlightenment.org/pages/evas.html.

15
Rasterman and the imlib2 development team.
Imlib2: An image processing library, December 2003.
http://www.enlightenment.org/pages/imlib2.html.

16
Robert W. Scheifler and James Gettys.
X Window System.
Digital Press, third edition, 1992.

17
Mark Segal, Kurt Akeley, and Jon Leach (ed).
The OpenGL Graphics System: A Specification.
SGI, 1999.

18
K. Thompson.
Unix implementation.

The Bell System Technical Journal, 57(6):1931-1946, July-August 1978.

19
Mark Vojkovich and Marc Aurele La France.
XAA.HOWTO.
Technical report, The XFree86 Project Inc., 2000.

20
Carl D. Worth and Keith Packard.
Xr: Cross-device Rendering for Vector Graphics.
2003 ottawa linux symposium, July 2003.

This paper was originally published in the Proceedings of the 2004 USENIX Annual Technical Conference,
June 27-July 2, 2004, Boston, MA, USA
Last changed: 10 June 2004 aw

USENIX '04 Technical Program
USENIX '04 Home
USENIX home