Bringing X.org's GLX Support Into the Modern Age

Ian D. Romanick
IBM
idr@us.ibm.com

Abstract[1]

GLX support in open-source X-Windows implementations has remained relatively unchanged since support was first added in XFree86 4.0.0. This has left several critical features, such as hardware accelerated indirect-rendering and pbuffers, unsupported. In this paper I present on-going work to remedy this situation. I will present the new infrastructure and possible future uses, work that remains to be done to complete the project, and impact of this work on other X-server components.

1 Introduction

In late 1998 SGI donated a large body of code to the XFree86 project to help bring full GLX support to the open-source X-server. The donated code implemented GLX 1.2 and several common GLX extensions. Even at this time, GLX 1.2 was on the verge of being out-dated due to the contemporaneous release of GLX 1.3 [11].

Because the code was donated rather than developed by the community, little knowledge of the inner workings of the code exists in the community. This has led to a general stagnation of the code. Today, few people know the code well enough to work on it, and the people that do know it have other tasks higher on their to-do list. As a result, functionality that was obsolescent in 1998 is entirely outdated in 2005.

One of the biggest problems in X.org's GLX code is that the client-side and server-side use a different "driver" architecture. The client-side cleanly divides duties between a loader (i.e., libGL) and a driver. On the server-side, the distinction is much less clear. Duties are less cleanly divided among two extension modules and the DDX. Unifying the two driver architectures would directly solve some of the deficiencies, such as the lack of hardware-accelerated indirect rendering, and would pave the way for solving others, such as lack of pbuffer support.

The remainder of this paper is divided into four sections. In the first two sections I present work that has already been done and work that remains to be done to modernize the GLX support on X.org. In the remaining two sections I present the impacts of these changes on other parts of the X server and on closed-source drivers.

[1] This paper was presnted at DDC 2005. It was supposed to appear in the *Proceedings of hte Linux Symposium 2005*, but, due to an error on the author's part, it did not.

2 Initial Steps Towards Modernization

Large portions of the OpenGL interface code are deterministically derived from the API. That is, all functions in the API that match a certain pattern will have similar code in parts of the interface. One clear example of this is the GLX protocol code. The protocol for `glVertex3fv` and `glNormal3fv`, which have identical parameter signatures and similar functionality, are nearly identical. This is no coincidence. The API is designed to have functions that do similar types of things look and operate similarly. This not only makes it easier for developers to learn new portions of the API, but also for implementors to implement the API. In many cases, simply knowing the parameter signature of a function is enough to generate interface or protocol code for that function.

2.1 Existing Code Generation

SGI's OpenGL Sample Implementation (SI) contains *.spec* description files that are used to generate portions of the code. These same .spec files are also used to generate the OpenGL and GLX header files published in the OpenGL extension registry [2]. Several of these header files are included in Mesa and X.org.

There are also scripts in the OpenGL SI to generate GLX protocol code from the .spec files. The code generated by these scripts was brought into the XFree86 code base with the rest of the SGI-contributed code. Unfortunately, the scripts and the .spec files were not.

Mesa releases before 6.3 used a file, called APIspec, to describe the portion of the OpenGL API known to Mesa. The format of this file was loosely based on the .spec files in the OpenGL SI. Several Python scripts in Mesa used this file to generate C and assembly source code. For example, both C and assembly versions of the static dispatch functions were generated.

Being created with a slightly different purpose in mind, Mesa's APIspec file was missing some data from SGI's .spec files. While APIspec listed all the parameters to each function, it did not differentiate between input parameters and output parameters. The .spec files contain this data, and it is required by the scripts that generate GLX protocol code.

This resulted in Mesa code generated from one set of data and scripts and X.org code generated from a different set of data and scripts. Since the scripts and data used by the X.org were not in X.org, adding support for new functions was extremely tedious and error prone. Clearly, some unification was needed.

2.2 Unification of Code Generation

Two paths forward were initially considered. One option was to add the missing data to APIspec and create new scripts to generate GLX protocol code. Analysis of the APIspec file and the existing scripts showed that each time a new field was added, several scripts would need to be modified. The modifications would be required even if the scripts did not need the new data. Without considerable reorganization of the existing scripts, modifying APIspec would have led to a maintenance nightmare.

The second option that was considered was to simply use the .spec files from the OpenGL SI. At the time, it was not known that the scripts used to generate the original GLX protocol code were available. Without those scripts, new scripts would have to be written from scratch. Even given the scripts from the OpenGL SI, the existing scripts in Mesa were written in Python,

and the scripts from the OpenGL SI were not. Maintaining scripts in two different languages would prevent code sharing. In addition, the .spec files were missing some data, such as dispatch offsets for newer functions, that were present in APIspec.

In the end, neither of these options was selected. Instead a new XML-based data file was created [10]. Since the data was stored in XML, adding new data fields in the future would not require modification to other scripts. Only the scripts that needed the new data fields would have to be updated. In addition, numerous tools and libraries already existed for operating on XML data.

APIspec was converted to an XML format, using an awk script and some hand editing, and new scripts were developed to process it. These scripts, which are currently in their second iteration, are designed in a layered manner. At the bottom are generic Python classes to parse XML and create objects representing different elements of the OpenGL API[2]. For example, there are separate classes to represent OpenGL functions, enumerants, and data types.

These classes are used by different "application" scripts. These scripts use the data stored in these objects to generate C or assembly source code. In this respect, the structure is very similar to the old scripts that used APIspec.

The key difference is that, thanks in part to the use of XML, it is possible to add elements to the XML that the basic parsing classes will simply ignore. The basic parsing classes can then be subclassed to utilize the added elements. The XML schema used, for example, contains lots of data that is only of interest when generating GLX protocol related code. The basic parsing classes completely ignore this data, but layered classes process and store it. These layered classes are used by the GLX protocol generator scripts.

[2]XML-XSLT could have been used, but using XSLT would have made it much more difficult to implement some of the optimizations in the generated code that the Python scripts implement

2.3 Client-Side GLX Protocol Code Generation

Outside the general dispatch mechanism, another area where automatically generated code can be used is in the GLX protocol code. There are three portions of the GLX protocol code that are particularly well suited to generated code: initialization of the GL dispatch table, sending and receiving GLX protocol, and determining the size of variable-length GLX packets.

Of the approximately 26,000 lines of code specific to libGL, 10,000 lines are generated by scripts. This is done by approximately 2,300 lines of GLX-specific Python code.

2.4 Future Uses of API Data

In addition to the existing uses of the API data for generating code, there are three general areas where these techniques could be readily applied. Not only would the application of scripted code generation improve the maintainability of Mesa and X.org's GLX code, but it would provide a gentle introduction to the code base for newcomers.

Within core Mesa, much of the code for handling display lists shows the same repetitive pattern as much of the GLX protocol code. This makes this portion of Mesa a good candidate for code generation. It is likely that as much as 60% to 70% of the 8,400 lines of `dlist.c` could be replaced with generated code.

There is an additional element that makes generation of display-list code interesting. Only a portion of the functions in the OpenGL API can be included in display-lists. The XML description does not include this information: it would need to be added. Several of the base classes would also need to be updated to store the information.

The XML description contains nearly all of the data required to generate test vectors for much of the API. Functions that have a variable-length parameter list, such as glTexParameterfv, and functions that return variable-length response vectors, such as glGetTexParameterfv, would be prime targets for exhaustive test vectors. These tests would help detect cases where changes to one of the generator scripts subtly breaks some rarely used portion of the generated code. Such defects crept in several times during the development of the exist GLX protocol code generator scripts.

Having a large body of test vectors would raise the confidence level of developers making changes to the scripts. However, the test vectors alone are not enough. Without the addition of interoperability testing, protocol errors can still be introduced. This will be even more true when server-side code is generated by scripts. The risk is that a change to one of the scripts could infect the client-side and server-side code with the same bug. Even without the use of generated code on the server, there is still some risk [3].

The final area where generation of code could be trivially applied is, of course, to the GLX protocol code on the server. Just like the client-side, thousands of line of code on the server-side were generated by scripts in the OpenGL SI. Some scripts have already been developed to generate server-side code but have not been contributed to the project.

3 Required Server-Side Changes

The ultimate goal of the server-side changes is to load the “client-side” DRI drivers in the server. Doing so would immediately enable hardware-accelerated indirect-rendering. Before this can be done, a number of changes to server-side GLX code are required.

Roughly speaking, the server-side interfaces between libglx [6] and libGLcore [5] need to be made to match the interfaces between libGL and the client-side drivers. This is a large task that must be divided into a number of smaller steps.

For the remainder of this discussion, libGL and libglx will be referred to as the “loader”, and the client-side DRI driver and libGLcore will bew referred to as the “driver”. This terminology reflects the parallels between the components on the client and server-side.

3.1 Server-Side GLX Protocol Code Generation

The first step is to replace the code that had been generated by scripts in the OpenGL SI with code generated by scripts that use the new XML data. This includes code for receiving GLX protocol requests, sending GLX protocol replies, validating the sizes of GLX protocol packets, and dispatching GL commands to the driver. For perspective, in X.org 6.8 this accounts for 26,000 lines of the 30,000 total lines of server-side GLX code.

There is another reason to start with this change. A number of the other steps involve making modifications to the routines that decode the incoming GLX protocol and dispatch the function calls to the driver. By writing the code generation scripts first, these changes will

require only small changes to the scripts rather than changes to each individual function.

Fortunately, most of the server-side protocol code is similar to the client-side protocol code. The server-side scripts can be created using the client-side scripts as a start.

3.2 Server-Side Dispatch Table

On the client-side, functions are dispatched to the driver using a table of function pointers. This allows the driver a great deal of flexibility. Drivers can change the pointers for individual functions on the fly based on changes to GL state. Drivers also have the option of leaving pointers for unsupported functions unset.

On the server-side, functions are dispatched to the driver by calling statically exported functions. This is problematic for a number of reasons [7]. Not all servers support all functions. For example, some versions of Darwin do not support `glPointParameterfARB`. The result is ugly conditional compilation based on the target platform. If a dispatch table were used, the driver would leave the pointer unset. The dispatch routine would detect this and return an error to the client. While there may still be conditional compilation involved, it would be in the platform-specific code rather than the platform-independent code.

Because the driver has to statically export all of its own dispatch functions, it is impossible to load multiple drivers. If a user had two different graphics cards in their system, each would need a different driver loaded. The symbol names in the driver loaded second would conflict with the symbol names in the first, and it would fail to load.

Each driver exports dispatch functions and the loader contains external references to these functions. This adds extra relocation information to the object files, which increases load time and wastes disk space. All of this could be avoided by using a dispatch table.

3.3 Rearranging Visual Initialization

The code path for initializing GLX visuals is confusing and convoluted. The process begins in the function GlxWrapInitVisuals. This function inserts its own function in the visual initialization pipeline and calls `__glXglDDXExtensionInfo`. This second function is located in the driver. In a system where a different driver is loaded for each device, it is impossible for the loader to call into the driver at this early point! There is even a comment in the code stating that it is a hack.

The next step happens when the DDX calls `GlxSetVisualConfigs`. This function immediately passes all of the incoming data to the driver's setVisualConfigs function. In addition, one of the parameters passed in is essentially unused by any of the open-source drivers. The driver's setVisualConfigs function only stores the pointers passed in to it. No processing is done there.

The process continues in `GlxInitVisuals`. Unsurprisingly, this function immediately calls the driver's initVisuals function. This function does the work of creating new X visuals that can be returned to the client.

The process concludes in `GlxExtensionInit`. After performing some generic initialization for the module, this function calls `__glXScreenInit` For each screen on the display, this function calls the driver's screenProbe function. The primary purpose of this function is to make copies of the visual information for each screen and correct any data (e.g., red, green, and blue

color bit masks) that may have been incorrectly set by other layers.

It took over a week of continuous code spelunking to achieve this level of understanding of the control flow. Part of the problem is that there is code in the driver that clearly belongs in the loader. However, shuffling that code around only solves part of the problem.

In addition to having code in the wrong place, two different problems are attacked by a single set of routines. These routines create the set of GLX visuals exported by the server to clients. They also create the data structures used internally by the drivers to describe the format of data in the framebuffer. This reflects the GLX 1.2 nature of the code contributed by SGI, as GLX 1.2 lacks support for fbconfigs.

In addition to getting all the code in the right place, these two processes need to be separated. The separation of these processes on the client-side GLX code provides a good starting model for the server-side, but there are additional issues. On the client-side, the loader receives a list of supported visuals from the server[3]. When it boot-straps the driver, the driver returns a list of fbconfigs that it can support. The loader the unifies the two lists and only returns fbconfigs to the application that are supported by both the client and the server. This works because the list of visuals originates in the DDX on the server and the list of fbconfigs originates in the DRI driver on the client, both of which have hardware-specific knowledge.

The server does not have this same luxury. On the server, the 3D driver is the software rendering engine in Mesa, which has no hardware-specific information. It therefore cannot create a list of supported fbconfigs.

[3]The client can receive either visuals or fbconfigs from the server. If visuals are received, they are translated to fbconfigs by the loader before being passed to the driver.

The solution is conceptually simple, but involves heavy modification of the visual creation process. The DDX creates a skeleton list of GLX visuals and passes the list to `GlxSetVisualConfigs`. Instead of calling into the driver, the loader stores the data locally. When `GlxInitVisuals` is called, the device-independent work that would have been done in the driver is done in the loader. At this point the loader knows whether or not 3D rendering will be performed by the software renderer by the presence (or lack thereof) of skeleton GLX visuals supplied by the DDX. This is all the information the loader needs to create a viable set of fbconfigs.

When the driver is actually loaded by the loader, it is passed the list of fbconfigs created by the loader. Most hardware drivers will ignore the fbconfigs passed in, just like on the client-side. However, a software driver would make a copy of the fbconfigs and return it as its set of supported fbconfigs. From that point, the process operates much like on the client-side.

3.4 Using the Loader-Driver Interface From the Client-Side

As can be seen from the previous section, the interface between the loader and the driver on the server side is quite different from the interface on the client-side. Once the visual initialization process is fixed, the majority of the remaining work is the methodical process of changing data structures, changing parameter lists, and shuffling code around.

There is still one remaining technical hurdle. Some additional facilities provided by the client-side loader need to be brought to the server-side. For example, drivers on the client-side can inform the loader of the set of GLX extensions that they support. There is a fair amount of framework for doing this that does not yet exist on the server.

3.5 Impact on Other Server-Side Code

Given the depth of the proposed server-side changes, there are bound to be some casualties in other parts of the code. While it is not possible to foresee all the impacts, some of the larger impacts are easily visible.

At each step of the way, the non-Unix GL drivers will be broken. At the very least, the Darwin and cygwin ports will be impacted as much as the Unix driver. Some of the changes, such as moving from using static dispatch functions to a dispatch table, may clean up portions of the code in those components. Other changes, such as changing the visual initialization process, may prove complicated.

In addition, there will likely be some changes to the interface between the DDX and the loader. Several aspects of this interface are designed to meet possible future needs, that never materialized. For example, along with the skeleton GLX visuals, the DDX passes a pointer for each visual to driver private data. However, no driver makes any use of these pointers. In all of the open-source drivers, the pointers are NULL. Several other aspects of the interface are equally useless.

4 IHVs

The deficiencies in X.org's GLX support have been as much of a problem for IHVs' closed-source drivers as they have for open-source. Some IHVs have gone so far as to supply their own server-side GLX modules to get around these problems. As deficiencies are resolved, there are several things that IHVs can do make use of the new facilities and help the process along.

4.1 Get Engaged in the Process

With very few exceptions, X.org's GLX infrastructure has been designed with no input from IHVs. This is tragic. Free Software issues aside, IHVs are a part of the larger X-Windows community. They are also customers of X.org's infrastructure. Being forced to design solutions without customer input is a process doomed to fail. Having customer input is no guarantee of success, but it does improve the odds.

By becoming involved in community discussions, IHVs also remain abreast of interface and functional changes coming in future releases. This gives the IHV opportunity to make the necessary changes to their code base to remain compatible with the release. End users do not want to wait months after an X.org release to have updated drivers for their hardware. That is the kind of thing that makes people buy from a different vendor.

Community involvement does not require donations of code or risk to company IP. The only resource involved is an engineer's time. While this is a valuable commodity, the amount of time required to keep up on infrastructure updates is minor.

4.2 Migrate to the Updated Client-Side Interface

All of the closed-source drivers that already use DRI use an older version of the loader-driver interface. Several aspects of this interface have been deprecated for over a year. It is expected that all of these interfaces will be removed for the X.org 7.0 release. Migrating client-side drivers to the most recent version of the loader-driver interface is a good first step towards getting a DRI driver ready to be loaded in the server.

The full details of the changes to the loader-driver interface are beyond the scope of this paper. The rationale behind the changes falls broadly into one of two categories. The changes were made to either support additional GLX functionality, such as `GLX_SGI_make_current_read` [1], or to support use of DRI drivers in non-X-Windows environments [4]. Fortunately, these same changes provide additional interfaces needed to communicate information between the 3D driver and the rest of the X-server.

As previously mentioned, these interfaces will undergo some additional changes before X.org 7.0 is released. For the most part, this means that deprecated portions of the older interface will be removed and deficiencies [8, 9] in the existing interface will be fixed.

5 Conclusion

The future for GLX support in X.org is looking bright. There is finally a path forward to resolve the functional deficiencies in the infrastructure. Resolving these issues will both improve the state of open-source driver support and pave the way for improved closed-source driver support.

To provide the required GLX support to their customers, IHVs are forced to replace portions of the GLX infrastructure with their own code. This hurts the IHV, hurts their customers, and hurts the X.org community at large. Early and frequent IHV involvement in the infrastructure is key to keeping the interfaces and the implementation honest.

A large amount of work has already been done to bring the client-side DRI drivers into the server. A lot of work remains. With a concerted effort by X.org developers and thoughtful input from IHVs, server-side loading of DRI drivers should be an easily attainable goal for the X.org 7.1 release.

References

[1] Silicon Graphics Inc. `GLX_SGI_make_current_read` extension specification. URI `http://oss.sgi.com/projects/ogl-sample/registry/SGI/make_current_read.txt`.

[2] Silicon Graphics Inc. OpenGL extension registry. URI `http://oss.sgi.com/projects/ogl-sample/registry/`.

[3] Bugzilla #3539: `glXGetFBConfigsSGIX` incompatible with Solaris/IRIX. URI `http://bugs.freedesktop.org/show_bug.cgi?id=3539`.

[4] DRI Project. DRI without X. URI `http://dri.freedesktop.org/wiki/DriWithoutX`.

[5] DRI Project. GLcoreExtension. URI `http://dri.freedesktop.org/wiki/GLcoreExtension`.

[6] DRI Project. GLXExtension. URI `http://dri.freedesktop.org/wiki/GLXExtension`.

[7] Ian Romanick. Bugzilla #2996: libglx / libGLcore should use a dispatch table. URI `http://bugs.freedesktop.org/show_bug.cgi?id=2996`.

[8] Ian Romanick. Bugzilla #3377: `_glapi_add_entrypoint` does not function properly for offset -1. URI `http://bugs.freedesktop.org/show_bug.cgi?id=3377`.

[9] Ian Romanick. Bugzilla #3378: `_glapi_add_entrypoint` does not generate offset for drivers. URI `http://bugs.freedesktop.org/show_bug.cgi?id=3378`.

[10] Ian Romanick. Updates to `glapi` generator scripts, 2004. URI `http://marc.theaimsgroup.com/?l=mesa3d-dev&m=108490587932147&w=2`.

[11] Paula Womack and Jon Leech (ed.). *OpenGL Graphics with the X Window System (Version 1.3)*, 1998.