Redesigning the X server for Hotplug Environments

Dave Airlie

Abstract

This paper describes a possible redesign of the X server core to suit current requirements.

1 Introduction

The current X server design dates back 20 years and currently some of the design decisions made back then are not suitable for the modern GPU applications. Requirements such as hotplug GPUs, dynamic GPU switching and multi-GPU environments are showing a number of shortcomings with the current design. However the current design is very extensible so this ability should not be removed. This document needs a lot more work.

2 The current X server design

2.1 Objects

The current server design has no clear separation between protocol and GPU objects. The main objects defined in the X server are:

1. Screen
2. Graphics Context (GC)
3. Drawable
4. Window
5. Pixmap
6. Colormap
7. Picture

These objects are passed between layers and in/out of the GPU drivers freely. The Screen, PictureScreen and GC currently contain most of the rendering related entry points into the driver, these entry points have defined results of operations on them, and underlying layers can hook into all the entry points using a wrapping mechanism. However there is no distinction between what is protocol related information and what is drawing related.

2.1.1 Screen

The screen object is the primary object containing the highlevel information about the protocol screen. The protocol decoding layer calls the per screen procedure entry points. These entry points generally have a default implementation that is wrapped by each protocol layer and rendering layer.

2.1.2 Graphics Context (GC)

The GC object contains the current X core protocol rendering context. A number of protocol operations allow modification of the graphics context, and subsequent rendering operations are executed using the current GC state. The GC defines a set of rendering operations that have default implementation that are wrapped by protocol and rendering layers.

2.1.3 Drawable

An X drawable is an object that drawing operations occur on. It is the base class for Pixmap and Windows. The X core protocol deals in terms of drawables and these are passed into the GC operations. Each rendering layer (hw or sw) has to do operate slightly differently on the drawable depending on whether its a pixmap or window.

2.1.4 Window

Windows are protocol objects and are a drawable subclass. The window structure contains all the information about a protocol window, this includes the clipping information for a window, and also links into the window hierarchy.

2.1.5 Pixmap

Pixmaps are protocol objects and are a drawable subclass. They represent a block of pixels with width/height/depth but no clipping information.

2.1.6 Colormap

Colormaps are protocol objects.

2.1.7 Picture

Pictures are part of the Xrender specification, and are protocol objects. They are associated with drawables (either Window or Pixmap), and are passed to the render entry points. Render entry points are stored in a per-screen object called the PictureScreen.

2.2 Driver API

The driver API is defined in terms of protocol objects. All the internals and protocol specific details are passed to each driver via the interface. When a driver creates a screen, a real X screen is created. Since the number of X protocol screens are defined once clients are connected you can not really add/remove protocol screens without Xinerama. Doing it with xinerama leads to another issue.

2.3 Xinerama

Xinerama works by grouping a number of separate X screens and presenting a single screen. The demulitplexing is done in the protocol layer, so every protocol that has knowledge of xinerama has two sets of protocol decoding code. The xinerama decoding calls into the lower layer decoding after reworking the incoming protocol packet with per-screen transformations and calls each underlying screen in turn. Xinerama creates instances of each object per underlying screen and stores them in a fake toplevel object. The identifier used for the fake toplevel object is the same as the identifier used for the object on the first underlying screen. Also a number of protocol only operations are never demulitplexed and only occur on the special 1st underlying screen. This leads to problems if you ever want to remove this screen in a hotplugging situation.

3 Reasons for redesign

3.1 GPU hotplugging

To try and add some idea of GPU hotplugging, the idea to use Xinerama to act as a frontend to the protocol then we could add/remove screens underneath it. However this falls down when you realise that unplugging the 1st screen is impossible due to the current design as too much information is stored in this screen. This also make GPU switching impossible as you can't remove the 1st screen. You could work around this by adding a fake first screen, but some experiments in this area showed that the code got really ugly quickly.

3.2 GPU offloading

Systems such as NVIDIA's Optimus and ATI's PowerXpress are designed around the concept of offloading rendering to a secondary GPU. However in the current

X architecture you cannot have a GPU driver loaded and useable that isn't providing a screen. If its providing a screen then it will show up on the protocol level, unless xinerama is enabled, however is xinerama is enabled then all rendering will be sent to both screens, and if you just want to offload some 3D rendering, interactions between Xinerama and DRI are very messy.

4 Dark Corners

This section just describes some identified areas that are known to need more investigation.

4.1 Overlays

Overlays are currently represented as a set of pixmaps attached to a window privately in the driver layers. However when a overlay window moves the overlay layer and underlay layer may have different clipping requirements. Also the layers may be represented as planar, one layer per pixmap, or packed, both layers in a single pixmap. In the packed case a planemask is needed to specify which layer is being addressed.

4.2 GLX

For direct rendered clients who are rendering to a shared buffer, the clipping information is required to be sent to the client side and kept up to date. This isn't the same as sending the driver clipping information with every drawing operation as the information is required to be kept up to date. This is mostly a protocol type operation so I think the protocol extension layer (GLX or DRI or NVGLX) can wrap the Window hierarchy like it does now to track this information. There may need to be some more info at the driver->extension level for this.

4.3 Colormaps?

NVIDIA pointed out they use Window info to store a per-window colormap. Need to nail down if there is a nicer way to do this.

5 Proposed new design

5.1 Identifying the layers and splitting points.

The role of the X server protocol layer is to deal with all the X protocol related information. The role of the GPU driver is to render the information passed to it from the server. In the current X server tree, the fb sw renderer and the acceleration architectures would be considered the boundary of the driver rendering interface. All other layers (dix, mi, Xext, damage etc.) would be considered protocol related layers.

5.2 What needs changing

5.2.1 Splitting the protocol and driver objects

Currently the objects defined above mix information that is relevant to the protocol and to the gpu driver in one single set of structs. There is no distinction between any layer of the stack in terms of whether its dealing in protocol or driver related information. Generally the bottom layer of the wrapping stack for the Screen pointers can aid in telling whether they are protocol or driver related. Generally most of the operations that end in the mi layer are protocol related and any that end in the fb layer are drawing related. However in some cases the fb layer will then call back into the mi layer thus making the job a lot harder. The main areas where this overlap is messy is around Window copy operations. So the main core of the design is splitting the current set of objects into two sets, and having a per-protocol set and a per-driver set.

5.2.2 Removing Windows from the rendering interface

Currently rendering is done to drawables, a drawable may actually be a window or a pixmap. The driver acceleration architectures and fb sw layer have to work out information from the drawable type to decide where to draw an object etc. Ideally we could remove knowledge of Windows from the accel/fb layers and the driver would just see Pixmaps and clipping information for each operation. Currently render operations come via GC operations and PictureScreen operations, so these would need to be moved to the per-driver objects.

5.2.3 Pushing down Xinerama

With a split between protocol and driver objects, the Xinerama multiplexing can be lowered to the boundary layer between protocol and driver objects. This would remove the duplicate protocol processing and the duplication of stored information like window hierarchies and allow dynamic addition/removal of drivers post protocol processing.

5.3 Object interrelationships.

ProtocolScreen object would replace the current Screen object in the core X server. It would have a 1..n relationship with the Screen objects coming from the drivers. In a non-xinerama setup, this would be a 1..1 relationship, in a xinerama setup this would be a 1..n. There are also circumstances where a Screen object may exist without a corresponding ProtocolScreen link, for things like GPU offloading.

ProtocolWindow objects would have a 1..n (possibly nxn for overlay) Pixmap relationship. Each protocol window would be backed with a per-screen pixmap.

ProtocolGCOps would be the current GCops structure and there would be a per-screen set of GCops that would these operations would be translated into. So protocol extensions requiring to wrap the GC operations would wrap the

ProtocolGCOps, and rendering and acceleration layers would wrap the GCops. (TODO: GCFuncs)

ProtocolPicture objects would have a 1..n Protocol relationship. Each picture would eventually be backed with a per-screen pixmap

6 Implementation plan

This section contains some rough ideas on how to implement this.

6.1 Requirements/Limitations on implementation

- Try not to break the X server too much.
- Try to keep as much working as possible at any one time
- Try to keep bisection mostly working.
- It seems likely that a first implementation will have to break all these rules and once working a plan to get from master -> there could be drawn up.
- Try not to break the driver API really badly. This means that the objects the driver sees should remain with current naming and new naming should be used for the protocol objects. However this will still require changes in extensions that are shipped with drivers like non-X.org using GLX etc.

6.2 Impedance layer

The author has been considering the possibilty of introducing an impedance layer into the current tree to aid splitting it out. This would work as follow,

1. Rename all current Screen/GC/Picture rendering and driver related entry points.
2. All these renamed entry points would terminate in the impedance layer.
3. The impedance layer would rework the information it gets and call into the driver entry points that are stored alongside.

The impedance layers job would be to allow slow migration path from old to new. With out-of-tree drivers still being able to use the wrapped APIs. The impedance layer would be responsible for doing all Drawable to underlying Pixmap conversion, no Windows would pass into the lowlevel rendering layers, this includes conversion of rendering operations relying on Drawable->x, Drawable->y. In later iteratons the impedance layer would be where the xinerama pushdown would occur. The impedance layer would define new driver entry points and trim down the current driver API. It would also avoid the re-entry problem of the mi layer calling into the fb layer calling into the mi layer. As all protocol related clipping etc would be worked out before the fb layer is ever call to do a rendering operation. This will require new APIs between the impedance layer and the rendering layers.

The plan would be to migrated fb and exa to the impedance layer first as proof. XAA will probably be a problem.

7 Prototype Proposed structures

- ProtocolScreen
- ProtocolDrawable
- ProtocolPixmap
- ProtocolWindow
  - Like current Window, but store at least one PixmapPtr in it, instead of hiding it in the fb privates
- ProtocolGCOps
- ProtocolPicture

Driver structures:

- Screen
  - GetCopyAreaFunction - used to move copyarea out of GCOps - to protocol level
  - GetCopyPlaneFunction - used to move copyplane out of GCops - to protocol level
  - PixmapWindowFixup - used to fixup per-window pixmaps like background/border - used to move CWA up.
  - PixmapCopyRegion - used to copy regions of pixmaps around - to move CopyWindow to protocol level
- Pixmap
- GCOps
- Picture