A brief introduction to the Linux graphics stack

JULY 29, 2014

This post attempts to be a brief and simple introduction to the Linux graphics stack, and as such, it has an introductory nature. I will focus on giving enough context to understand the role that Mesa and 3D drivers in general play in the stack and leave it to follow up posts to dive deeper into the guts of Mesa in general and the Intel DRI driver specifically.

A bit of history

In order to understand some of the particularities of the current graphics stack it is important to understand how it had to adapt to new challenges throughout the years.

You see, nowadays things are significantly more complex than they used to be, but in the early times there was only a single piece of software that had direct access to the graphics hardware: the X server. This approach made the graphics stack simpler because it didn't need to synchronize access to the graphics hardware between multiple clients.

In these early days applications would do all their drawing indirectly, through the X server. By using Xlib they would send rendering commands over the X11 protocol that the X server would receive, process and translate to actual hardware commands on the other side of a socket. Notice that this "translation" is the job of a driver: it takes a bunch of hardware agnostic rendering commands as its input and translates them into hardware commands as expected by the targeted GPU.

Since the X server was the only piece of software that could talk to the graphics hardware by design, these drivers were written specifically for it, became modules of the X server itself and an integral part of its architecture. These userspace drivers are called DDX drivers in X server argot and their role in the graphics stack is to support 2D operations as exported by Xlib and required by the X server implementation.

Recent Posts

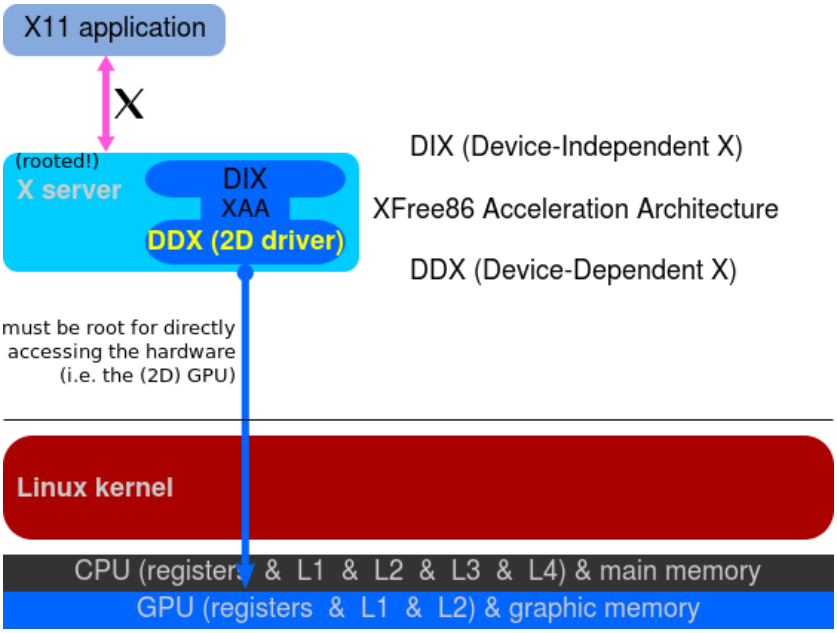
VK_KHR_shader_float16_int8 on Anvil
Intel Mesa Vulkan driver now supports shaderInt16
Frame analysis of a rendering of the Sponza model
Improving shader performance with Vulkan's specialization constants
Intel Mesa driver for Linux is now Vulkan 1.1 conformant

Recent Comments

Hans-Kristian on VK_KHR_shader_float16_int8 on Anvil
Iago Toral on Intel Mesa Vulkan driver now supports shaderInt16
oscar on Intel Mesa Vulkan driver now supports shaderInt16
J.A. on Intel Mesa driver for Linux is now OpenGL 4.6 conformant
J.A. on OpenGL terrain renderer: rendering the terrain mesh

Archives

December 2018
May 2018
April 2018
March 2018
February 2018
October 2017
July 2017
January 2017
November 2016



DDX drivers in the X server (image via wikipedia)

In my Ubuntu system, for example, the DDX driver for my Intel GPU comes via the xserver-xorg-video-intel package and there are similar packages for other GPU vendors.

3D graphics

The above covers 2D graphics as that is what the X server used to be all about. However, the arrival of 3D graphics hardware changed the scenario significantly, as we will see now.

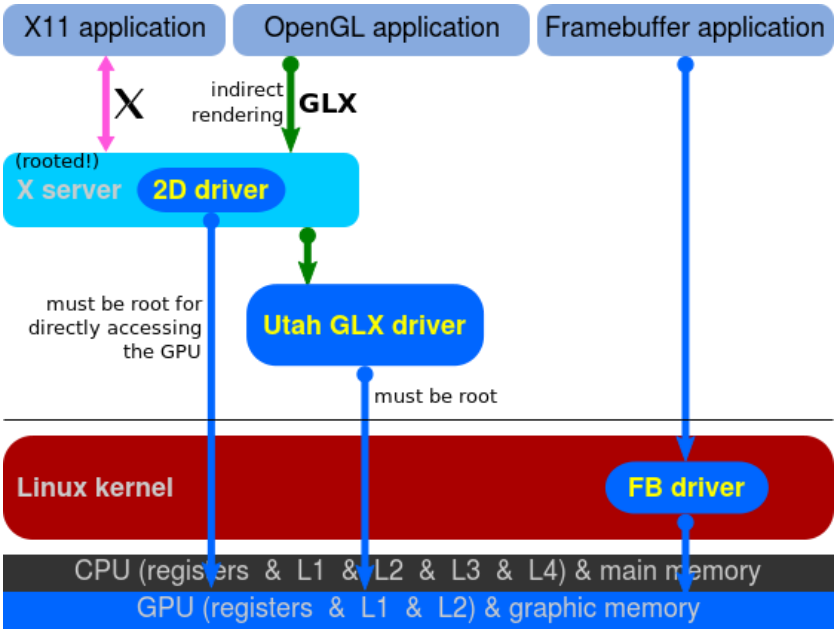
In Linux, 3D graphics is implemented via OpenGL, so people expected an implementation of this standard that would take advantage of the fancy new 3D hardware, that is, a hardware accelerated *libGL.so*. However, in a system where only the X server was allowed to access the graphics hardware we could not have a *libGL.so* that talked directly to the 3D hardware. Instead, the solution was to provide an implementation of OpenGL that would send OpenGL commands to the X server through an extension of the X11 protocol and let the X server translate these into actual hardware commands as it had been doing for 2D commands before.

We call this *Indirect Rendering*, since applications do not send rendering commands directly to the graphics hardware, and instead, render indirectly through the X server.

October 2016
August 2016
July 2016
July 2015
May 2015
April 2015
March 2015
November 2014
September 2014
August 2014
July 2014
February 2014
January 2014
December 2013
June 2013
May 2013
June 2011
May 2011
August 2010
June 2010
May 2010
April 2010
March 2010
February 2010
October 2009
July 2009
June 2009
March 2009
June 2008
April 2008
October 2007
July 2007
June 2007
May 2007
March 2007
February 2007
November 2006
October 2006
September 2006
August 2006
July 2006
June 2006
March 2006

Categories

graphics
Maemo
Uncategorized
webkit



OpenGL with Indirect Rendering (image via wikipedia)

Unfortunately, developers would soon realize that this solution was not sufficient for intensive 3D applications, such as games, that required to render large amounts of 3D primitives while maintaining high frame rates. The problem was clear: wrapping OpenGL calls in the X11 protocol was not a valid solution.

In order to achieve good performance in 3D applications we needed these to access the hardware directly and that would require to rethink a large chunk of the graphics stack.

Enter Direct Rendering Infrastructure (DRI)

Direct Rendering Infrastructure is the new architecture that allows X clients to talk to the graphics hardware directly. Implementing DRI required changes to various parts of the graphics stack including the X server, the kernel and various client libraries.

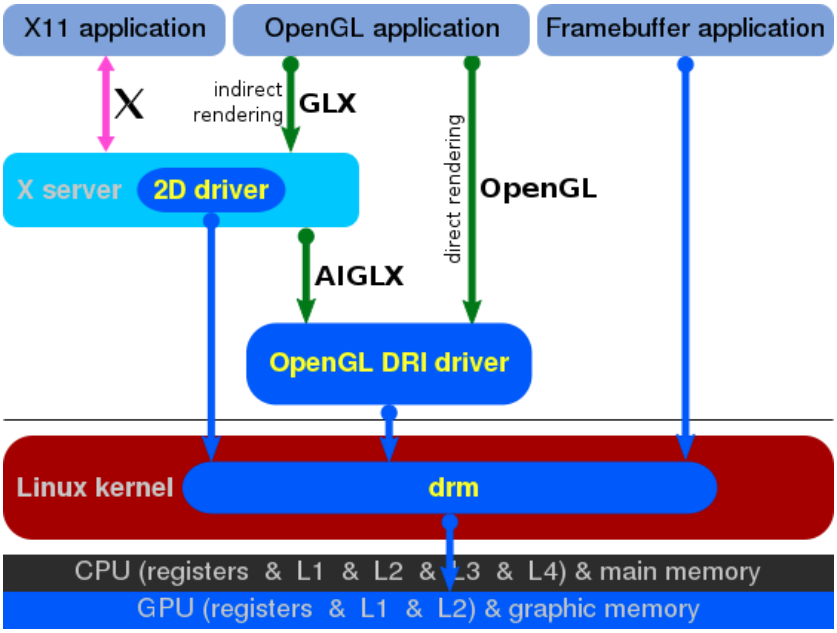
Although the term DRI usually refers to the complete architecture, it is often also used to refer only to the specific part of it that involves the interaction of applications with the X server, so be aware of this dual meaning when you read about this stuff on the Internet.

Another important part of DRI is the Direct Rendering Manager (DRM). This is the kernel side of the DRI architecture. Here, the kernel handles sensitive aspects like hardware locking, access synchronization, video memory and more. DRM also provides userspace with an API that it can use to submit commands and data in a format that is adequate for modern GPUs, which effectively allows userspace to communicate with the graphics hardware.

Notice that many of these things have to be done specifically for the target hardware so there are different DRM drivers for each GPU. In my Ubuntu system the DRM module for my Intel GPU is provided via the libdrm-intel1:amd64 package.

Meta

Log in
Entries RSS
Comments RSS
WordPress.org



OpenGL with Direct Rendering (image via wikipedia)

DRI/DRM provide the building blocks that enable userspace applications to access the graphics hardware directly in an efficient and safe manner, but in order to use OpenGL we need another piece of software that, using the infrastructure provided by DRI/DRM, implements the OpenGL API while respecting the X server requirements.

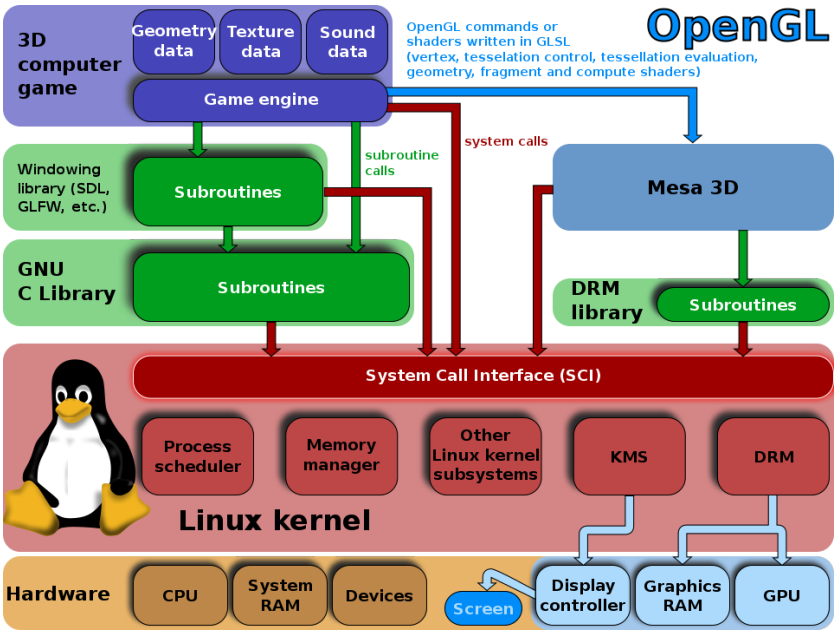
Enter Mesa

Mesa is a free software implementation of the OpenGL specification, and as such, it provides a *libGL.so*, which OpenGL based programs can use to output 3D graphics in Linux. Mesa can provide accelerated 3D graphics by taking advantage of the DRI architecture to gain direct access to the underlying graphics hardware in its implementation of the OpenGL API.

When our 3D application runs in an X11 environment it will output its graphics to a surface (window) allocated by the X server. Notice, however, that with DRI this will happen without intervention of the X server, so naturally there is some synchronization to do between the two, since the X server still owns the window Mesa is rendering to and is the one in charge of displaying its contents on the screen. This synchronization between the OpenGL application and the X server is part of DRI. Mesa's implementation of GLX (the extension of the OpenGL specification that addresses the X11 platform) uses DRI to talk to the X server and accomplish this.

Mesa also has to use DRM for many things. Communication with the graphics hardware happens by sending commands (for example "draw a triangle") and data (for example the vertex coordinates of the triangle, their color attributes, normals, etc). This process usually involves allocating a bunch of buffers in the graphics hardware where all these commands and data are copied so that the GPU can access them and do its work. This is enabled by the DRM driver, which is the one piece that takes care of managing video memory and which offers APIs to userspace (Mesa in this case) to do this for the specific target hardware. DRM is also required whenever we need to allocate and manage video memory in Mesa, so things like creating textures, uploading data to

textures, allocating color, depth or stencil buffers, etc all require to use the DRM APIs for the target hardware.



OpenGL/Mesa in the context of 3D Linux games (image via wikipedia)

What's next?

Hopefully I have managed to explain what is the role of Mesa in the Linux graphics stack and how it works together with the Direct Rendering Infrastructure to enable efficient 3D graphics via OpenGL. In the next post we will cover Mesa in more detail, we will see that it is actually a framework where multiple OpenGL drivers live together, including both hardware and software variants, we will also have a look at its directory structure and identify its main modules, introduce the Gallium framework and more.

BOOKMARK THE PERMALINK.

← A tour around the world of Mesa and Linux graphics drivers

Diving into Mesa →

ONE COMMENT

J.A. July 31, 2014 at 8:41 am

A nice complementary introduction about X11 can be also found at http://magcius.github.io/xplain/article/, which contains real-time demo of how X server works.

Powered by WordPress & simpleX.