# Linux 环境下的图形显示原理（？？）

显示组

# 目录

# 第一章　Linux 环境下的图形系统

\author{赵自成 cl}
\date{\today}\maketitle
\graphicspath{{./}{graph/}}

## 1.1 Linux 下的图形系统简介

Linux/Unix 环境下最早的图形系统是 Xorg 图形系统，Xorg 图形系统通过扩展的方式以适应显卡和桌面图形发展的需要，然而随着软硬件的发展，特别是嵌入式系统的发展，Xorg 显得庞大而落后。

开源社区开发开发了一些新的图形系统，比如当前嵌入式系统上流行的 Andrio 的图形系统，以及不久前由开源 区提出的 Wayland 图形系统，

本节后续章节我们将先对 Xorg 图形系统进行介绍，然而由于 Wayland 和 Android 图形系统代表了图形系统的最新发展方向，因而我们将选择 Wayland 图形系统进行介绍，实际上 Wayland 图形系统和 Android 图形系统在很多方面是类似的，比如都使用 EGL 作为本地窗口接口，都使用 OpenGL ES 接口进行 3D 图形渲染等。

### 1.1.1 Framebuffer 驱动下的基本显示

Framebuffer 驱动提供基本的显示，framebuffer 驱动操作的硬件就是一个显示控制器和帧缓存（一片位于系统主存或者显卡显存）。Framebuffer 驱动向应用程序提供/dev/fbx 的设备接口，应用程序通过读写这个设备节点实现对显示控制器和帧缓存。

下面这个程序显示了应用程序操作操作 framebuffer 节点的过程。运行这个程序，将在屏幕右上方显示一个【绿色的（会有大小端的问题，所以不能说绿色）】正方形。

```
 8 int main ()
 9 {
10      int fd;
11      struct fb_var_screeninfo vinfo;
12      struct fb_fix_screeninfo finfo;
13      size_t screensize = 0;
14      int location;
15      char *fbp = NULL, *ptr;
16      int x, y, x0, y0;
17      int i,j;
18      int ret;
```

```
 19
 20        fd = open("/dev/fb0", O_RDWR);
.....
 25        ret= ioctl(fd, FBIOGET_FSCREENINFO, &finfo ) ;
 .....
 30        ret = ioctl(fd, FBIOGET_VSCREENINFO, &vinfo);
.....
 35        ret = ioctl(fd, FBIOPAN_DISPLAY,&vinfo);
.....
 40        screensize=vinfo.xres * vinfo.yres * vinfo.bits_per_pixel /8 ;
 41        fbp = (char *)mmap(NULL, screensize, PROT_READ|PROT_WRITE, MAP_SHARED,
fd, 0);
 48        x0 = 200;
 49        y0 = 200;
 50        ptr = fbp + y0 * finfo.line_length + x0 * vinfo.bits_per_pixel / 8;
 51        for( i = 0; i < 100; i++){
 52              char* tmp_ptr = ptr;
 53              for(j = 0; j < 100; j++){
 54                    *tmp_ptr++ = 100;
 55                    *tmp_ptr++ = 15;
 56                    *tmp_ptr++ = 200;
 57                    *tmp_ptr++ = 0;
 58              }
 59              ptr += finfo.line_length;
 60        }
 61        munmap(fbp,screensize);
 62        close(fd);
 63        return 0;
 64 }
```

应用程序对 framebuffer 的操作主要是通过 ioctl 和 mmap 完成的。mmap 将显存映射到用户空间。

25 行和 30 行分别获取当前 framebuffer 驱动的“固定参数”和“可变参数”，这两个参数包含了当前显示控制其的一些信息，可变参数主要是当前分辨率信息，固定参数主要是当前显存的地址。

35 行 FBIOPAN_DISPLAY 通常用于双缓存，但是这里使用还有其它意义，在后面讨论 drm 的 framebuffer 的时候还会具体讨论。

41 行将显存映射出来，51-59 行操作这片显存，往上绘制一个左上方坐标为（200,200），变长为 100 的正方形。

应用程序能够使用这些 ioctl 是因为内核提供了相应的接口，本书第 \verb|\ref{ch4}| 章将会描述如何写一个 framebuffer 驱动。

## 1.1.2 Xorg 环境下的图形系统

Linux/unix 环境下最早的图形系统是 xorg 图形系统，xorg 图形系统通过扩展的方式以适应显卡和桌面图形发展的需要，然而随着软硬件的发展，特别是嵌入式系统的发展，Xorg 显得庞大而落后。开源社区开发除了一些新的图形系统，比如当前嵌入式系统上流行的 Andrio 的图形系统，以及不久前由开源社区提出的 Wayland 图形系统，

X Window（这里我们说的 X、X window 和 Xorg 意义相同）在 1984 年由 MIT 研发，X Window 的主要特点便是:Server/Client 网络模型。不论是本地、远程的应用程序，都统一通过 Server/Client 模型来运作。X 的设计哲学之一是:提供机制，而非策略。

X Window 在推出之后快速演化,在 1987 年时候,其核心协议已经是第 11 版本了,简称 X11。这个版本已经将“提供机制,而非策略”这个哲学贯彻地非常彻底,以致于核心协议基本稳定,不需要特别大的改动。二十几年过去了了 X Window 的核心都没有特别大的变化,X Window 在核心层之外提供一个扩展层,开发者可以开发相应扩展,来实现自己的扩展协议,所以过去二十几年 X Window 除了继续完善核心协议、驱动以外,很大程度上都是扩展使它保持与时俱进：

多头显示支持由 Xinerama 扩展实现的;

多媒体视频回放的支持由 X Video 扩展实现的;

OpenGL 的 3D 支持通过 GLX 扩展来实现的;

Compiz 那样的合成桌面特效由 Composite 实现;

Keyboard 的支持通过 X Keyboard Extension(也就是 XKB)实现。

在传统的 Xorg 框架下面，我们考虑这样一个场景，当点击了某个应用程序窗口的按钮，会发生如果过程：

\begin{enumerate}

\item 用鼠标点击应用程序的按钮,这时内核收到了鼠标发来的事件,并将其通过 evdev 输入驱动发送至了 X Server。

\item 这时 X Server 可以判断哪个 Window 该收到这个消息，并将某座标按下按钮的消息发往 X Client，也就是这里说的应用程序。但事实上 X Server 并不知道它得到的窗口信息是不是正确的，因为当前的 Linux 桌面早已经不是 10 年前的那样了,现在是 Composite 即合成桌面的时代，合成桌面的一个特点便是:Compositor(如 Compiz，metacity)管理窗口的一切，X Server 只能知道屏幕的某个点收到了鼠标消息,却不知道这个点下面到底有没有窗口。

\item 假设应用场景没这么复杂,client 顺利地收到了消息,这时 client 要决定该如何做:按钮要有按下的效果。于是 client 再发送请求给 X Server，请求其绘制按钮按下的效果。

\item 当 X Server 收到消息后,它就准备开始做具体的绘图工作了:首先它告诉显卡驱动,要画怎么样一个效果,然后它也计算了被改变的那块区域,同时告诉 Compiz 那块区域需要重新合成一下。

\item Compiz 收到消息后,它将从缓冲里取得显卡渲染出的图形并重新合成至整个屏幕当然,Compiz 的合成动作,也属于渲染(render),也是需要请求 X Server,我要画这块,然后 X Server 回复:你可以画了。

\end{enumerate}

整个请求和渲染的动作过程,从 X Client->X Server,再从 X Server->Compositor,而且是双

向的,确实是比较耗时的,但是,事实还不是如此。介于 X Window 已有的机制,尽管 Compiz 已经掌管了全部最终桌面呈现的效果,但 X Server 在收到 Compiz 的渲染请求时,还会做一些本职工作,如:窗口的重叠判断、被覆盖窗口的剪载计算等等，这些都是无意义的重复工作,而且 Compiz 不会理会这些,Compiz 依然会在自己的全屏幕画布上,画着自己的动画效果。
【以上参考资料 揭开 Wayland 的面纱:X Window 的前生今世】
【以上参考资料 The Wayland display server Documentation 1.0】

我们注意到在步骤 3 和步骤 4，client 都是通过请求 X server 进行渲染的，如果是一个 3D 应用程序，这将导致 X cient 和 X server 之间有大量的绘图命令请求，为了解决这个问题，在 Xorg 的基础上提出了 dri 架构（direct rending infrustrum）。【请确认是这个原因】
【插入 DRI 架构的图】

在 dri 架构下，一个 3D client 程序的不再发送大量绘图命令给 X server，client 应用程序和 server 共享一片 buffer，client 应用程序链接到诸如 opengl 的渲染库，opengl 渲染库知道如何对硬件进行编程以及如何将渲染结果输出到 buffer 上，compiz 能够获取这片 buffer 并且在合成桌面的时候将其用作纹理。

【上面一段参考自 The Wayland display server Documentation 1.0】
【请参考更多资料介绍 DRI】
上图显示，在 dri 出现以前，opengl 程序发送 glx（X 的扩展协议）命令包给 X server，由 X server 控制 3D 硬件进行绘图，但是有了 dri 之后，应用程序可以直接访问硬件绘图，减少了通过 X 协议传输绘图命令包的过程，因而 3D 程序的渲染有了质的变化。

除了 DRI 外，X 将越来越多的事情移到了其他地方。
在 linux X Window 下,一般都通过 GTK+和 Qt 来进行了图形软件开发，而 gtk+通过 Cairo 来绘制图形。尽管在 Linux 平台下,Cairo、Qt 的发挥依然是基于 X Window 的,但 X Window 充其量仅仅是一个后端而已,并不是少它不行。同理,跨平台的 GTK+、Qt 也只是视 X 为其中所支持的后端之一,假如哪天 X 真的不在了,更换一个新后端,当前的 GNOME、KDE 也能完整的跑起来。

另外一个例子是模式设置，过去模式设置是有 X server 来做的，但是现在被移到了内核中，即所谓的 KMS（kernel mode setting）。
可以说,这 20 多年来,X 从什么都做已经到了做的越来越少。绝大多数的开发者开发图形应用程序,已经可以完全无视 X 的存在了,X 现在更像是一个中间人的角色。

## 1.1.3 嵌入式图形系统 DirectFB

DirectFB 图形系统是一个小的嵌入式系统，这个系统上 。。。

## 1.2 Linux 环境下的图形系统架构和显卡驱动

### 1.2.1 核内和核外驱动

Linux 是一个宏内核，设备驱动大部分都是包含在内核里面的，因此可以看到内核代码最庞大的部分是 drivers 目录，如果从 kernel.org 上面下载一个内核源码，直接编译，编译的时间大部分都耗在编译设备驱动上。

像微内核的操作系统，设备驱动不是内核的部分。

本书不会讨论微内核和宏内核的区别或者各自的优缺点。但是出于调试方便以及其他一些原因，Linux 操作系统上面的一些驱动是放在核外的。

一个主要的类别就是打印机、扫描仪这类设备，当前的打印机扫描仪通常都是通过 USB 接口连接到计算机上的，对于这些设备的 Linux 驱动，除了 USB 核心部分在内核，这些打印机扫描仪本身的驱动都是在核外的。Linux 上面的打印机使用 CUPS 系统\verb|附加 CUPS 连接|，CUPS 运行在核外，其驱动是按照 CUPS 的接口来开发的。Linux 上面的扫描仪使用的是运行在核外的 sane 系统，其驱动是以动态链接库的形式存在的。

另外一类核外的驱动是图形系统的驱动，由于图形系统、显卡本身比较复杂，而且由于一些历史原因，图形系统的驱动在核内核外都有，并且显卡驱动最主要的部分在核外，后面两节内容介绍图形系统的时候就能够看到。

### 1.2.2 Xorg/DRI 环境下图形系统驱动

（“内核部分驱动” “核外部分驱动”）

**内核 DRM 驱动**

当前的 Linux 系统上内核的显卡驱动称为 drm 驱动，在通常的 linux 内核发行版上，我们使用 lsmod 命令查看内核模块，能够看到如下信息：

```
radeon                933054  3
ttm                    45600  1 radeon
drm_kms_helper         22468  1 radeon
drm                   162230  5 radeon,ttm,drm_kms_helper
i2c_algo_bit            5055  2 i2c_gpio,radeon
// 应该还有 frambuffer 设备节点
```

我们的机器使用的是 ATI 的 radeon 显卡，上面显示了当前系统内核和显卡驱动相关的模块，drm 模块是内核 drm 驱动的基础架构，所有使用 dri 架构的显卡驱动都会加载这个内核模块，ttm 是 ttm 内核管理机制，drm_kms_helper 是内核模式的基础框架代码，i2c_algo_bit 是显卡上操作 i2c 设备使用的模块，显卡上的 i2c 设备主要包括了 connector，encoder 以及 pll 时钟芯片。Drm 内核驱动的代码在目录 drivers/gpu/drm 下面。

加载了 drm 驱动后，在/dev 目录下面会生成如下设备节点：

```
/dev/char/226:0 -> ../dri/card0
/dev/char/226:64 -> ../dri/controlD64
/dev/dri/card0
```

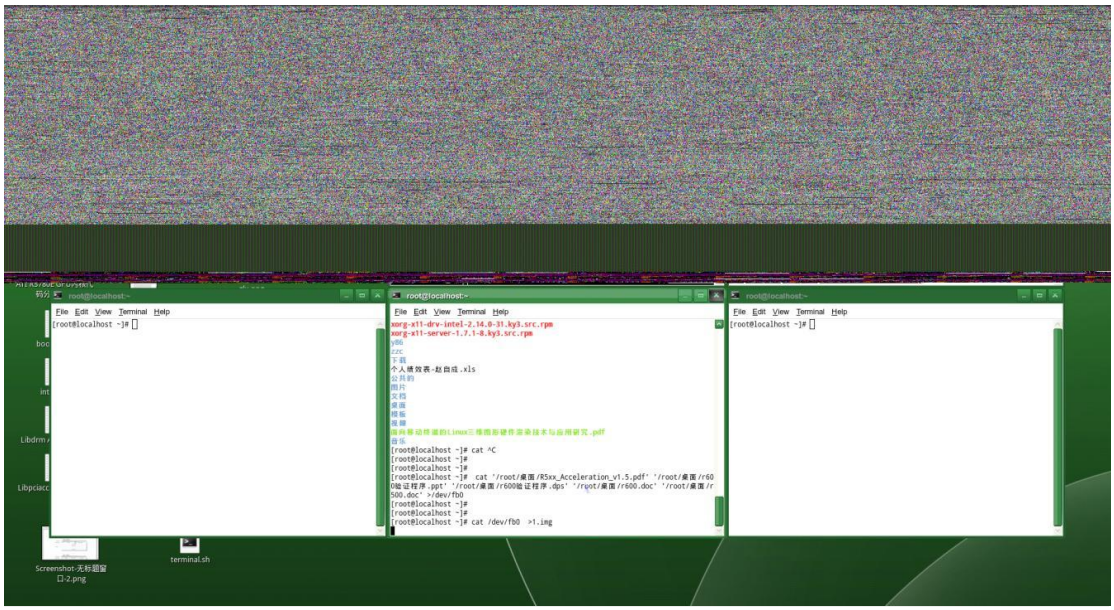
/dev/dri/controlD64

其中/dev/dri/card0 是操作 gpu 的接口，发送命令等操作都是通过对这个设备节点进行的。/dev/dri/controlD64 是 kms 相关的设备节点。

注意到这里除了 dri 目录下面的设备节点外，还有 frambuffer 设备节点，但是如果我们直接进行这样的操作“cat xx >/dev/fb0”，在屏幕上是看不到任何效果的，然而如果将 xorg.conf 的 Driver 修改成“fbdev”，重启 Xorg，然后执行“cat xxx >/dev/fb0”操作，就能看到类似下图【xxx】的情况，图片上面一部分是我们拷贝进去的内容。

这提示我们在 radeon 核外驱动正常运行的情况下，核外 radeon 驱动并不使用 framebuffer 驱动（实际上 drm 驱动注册的 frambuffer 设备只是给内核使用），在 radeon 核外程序正确运行后，不使用 framebuffer 管理的那片内存的内容作为显示输出，而是使用了另外一片内存。

然而前面\verb|\ref{framebuffer}|节给出的程序是能够在图形环境下正常运行的，这时因为我们在映射之前做了一个 PAN_DISPLAY 操作，PAN_DISPLAY 操作会去操作硬件修改 crtc 输出内容的显存地址，将其修改为 framebuffer 驱动使用的内存，这样操作后就能够看到效果了，如图\ref{drm 和 framebuffer}。



【还有许多内容需要添加】

**核外部分驱动**

Linux 下的图形驱动的主要部分是核外部分，核外部分包括了 xorg exa 驱动以及 mesa 3d 驱动，exa 是传统的 2D 加速框架，mesa 3d 驱动则是针对 3D 驱动的硬件加速。由于历史原因，显卡上最早只有 2D 部件，因而最早只有 2D 驱动，3D 功能是后来通过 mesa 添加的，因此即使现在硬件上 2D 功能是由 3D 部件实现的，但是 2D 驱动和 3D 仍然是分离的。

1）2D 加速驱动

Xorg 中有显示驱动，过去这个驱动里面包含了模式设置，终端切换操作，Xorg 通过一个称为 EXA 的扩展机将 2D 加速包含其中，而现在事实上的模式设置移植到了内核中，这个驱动除了维持 Xorg 的老的框架外，主要就是 2D 加速驱动中一些操作。

[参考 XFree86 design.pdf 详细介绍 xorg 驱动中的驱动，添加 xorg 驱动框架，exa 加速待后面论述]

2）3D 加速驱动}

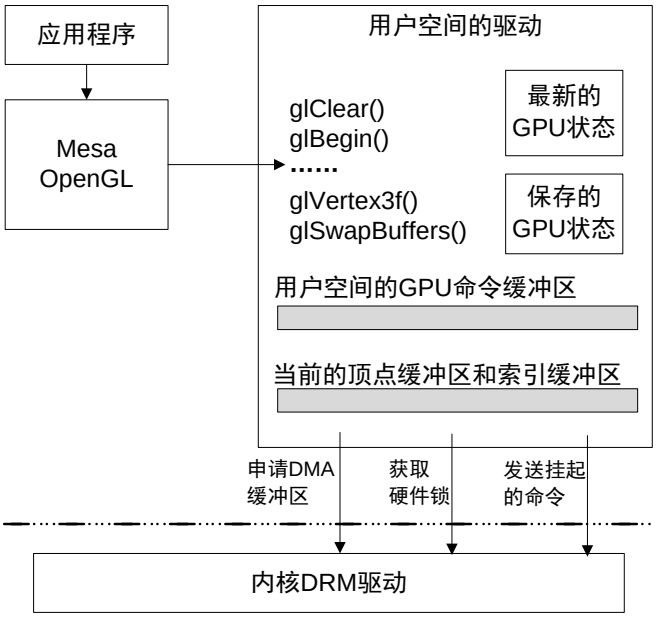
在 linux 环境中，我们可以通过 glxinfo 命令插卡 3D 硬件图形加速是否可用。比如在 radeon 显卡上 glxinfo 的输出包含以下内容：

OpenGL renderer string: Gallium 0.4 on AMD CEDAR

这里显示使用 AMD CEDAR 核心的显卡进行 opengl 3d 加速，如果硬件加速不可用，则应当是“vmware on llvmpipe”这类字眼。

GLX 是 X 协议的扩展，用于 OpenGL 和 X 的交互。

开源的 OpenGL 实现是 mesa，mesa 向上提供 OpenGL 接口，下层通过硬件的 mesa 驱动和硬件交互。




上图显示了一个 3D 应用程序运行的过程，OpenGL 绘图程序命令被用户空间的 mesa 驱动翻译成对应 GPU 的绘图命令放入命令缓冲区中，其他的如顶点信息/纹理信息/索引信息放入到相应缓冲中，mesa 驱动为每个应用程序保存了当前的绘图状态，当发生 3D 程序切换，当前状态被保存下来，当下次调度该程序运行的时候，先恢复该程序的绘图状态到硬件上，然后继续执行命令缓存中的命令。

当用户空间调用发送命令到内核的时候，内核驱动对硬件进行编程从该程序的命令缓冲区中取命令开始执行。

这个过程即是在 dri 框架下实现的，这里的所有绘制过程并不请求 X，openg 直接将命令发送给硬件。glx 只是在初始化窗口，申请 buffer 和切换 buffer 的时候才会和 X 交互。

这里讲的核外驱动，无论是 2D 驱动还是 3D 驱动，都需要内核 drm 驱动的支持，他们通过

## 1.2.3 DirectFB 图形系统驱动

## 1.3 本书主要内容和使用的代码

### 1.3.1 用于演示的内核驱动代码

本书后续章节使用代码有的是直接来自 linux 内核 radeon 驱动以及 xf86 的 ati 显卡驱动。Linux 内核代码使用的版本是 3.0.3，位于内核目录 drivers/gpu/drm/radeon 目录下面，加入 KMS 后的内核代码不同版本显卡驱动部分差别并不大。xf86 的 ati 驱动使用的版本是 6.14.2，libdrm 版本是 2.4.26。

另外为了演示显卡驱动的一些基本原理，我们自己编写了一个程序，当然这些原理很大程度上和图形学本身无关，而是涉及操作系统和设备总线这个层次的原理。

把内核的 radeon 驱动删除后，将我们的代码编成模块插入内核即可运行，内核模块自带了一些利用 GPU 进行绘图程序的，模块插入内核后就能够看到运行效果。

在理解这个代码前，读者需要对 linux 内核设备驱动和 pci 设备驱动有一定了解。

内核模块的入口函数是 kylin_radeon_probe。

```
2288 static int __init kylin_radeon_probe(struct pci_dev *pdev,
const struct pci_device_id *id)
...
2294        ret = pci_enable_device(pdev);
2295        if(ret) {
2296              printk("ERROR: pci_enable_device\n");
2297              goto error_enable;
2298        }
2299        pci_set_master(pdev);
2300        pci_set_drvdata(pdev,&rdev);
2301        rdev.pdev = pdev;
2302
2303        if (pci_set_dma_mask(pdev, DMA_BIT_MASK(40))) { //
2304              printk("radeon:No suitable DMA avaiable");
2305        }
2306        //map VRAM
2307        rdev.zone[PCIVRAM_ZONE].size = (unsigned long)pci_resource_len(pdev,0);
2308        rdev.zone[PCIVRAM_ZONE].cpu = ioremap(pci_resource_start(pdev,0),
                        rdev.zone[PCIVRAM_ZONE].size);
2309        if(NULL ==rdev.zone[PCIVRAM_ZONE].cpu) {
2310              printk("ERROR: cann't ioremap PCI VRAM\n");
2311              goto error_mapmem;
2312        }
2313        printk("[%s %d] vram mapped to addr: 0x%p, size: 0x%lx\n",
2314                        __func__, __LINE__, rdev.zone[PCIVRAM_ZONE].cpu,
2315                        rdev.zone[PCIVRAM_ZONE].size);
```

```
2316        rdev.zone[MMIO_ZONE].size = (unsigned long)pci_resource_len(pdev,2);
2317        rdev.zone[MMIO_ZONE].cpu = ioremap_nocache(pci_resource_start(pdev,2),
                                                       rdev.zone[MMIO_ZONE].size);
2318        printk("[%s %d] io mapped to addr: 0x%p, size: 0x%lx\n",
2319                        __func__, __LINE__, rdev.zone[MMIO_ZONE].cpu,
2320                        rdev.zone[MMIO_ZONE].size);
2321        if(NULL == rdev.zone[MMIO_ZONE].cpu) {
2322              printk("ERROR: cann't ioremap mmio register\n");
2323              goto error_mapio;
2324        }
```

这段代码用于初始化 pci 设备并映射 pci 设备的寄存器空间和内存空间。Radeon 显卡的 bar 0 为内存空间，bar 2 为寄存器空间。

后续章节我们将根据具体功能逐步说明后续代码。

## 1.3.2 R500 和 R600 DirectFB 驱动代码

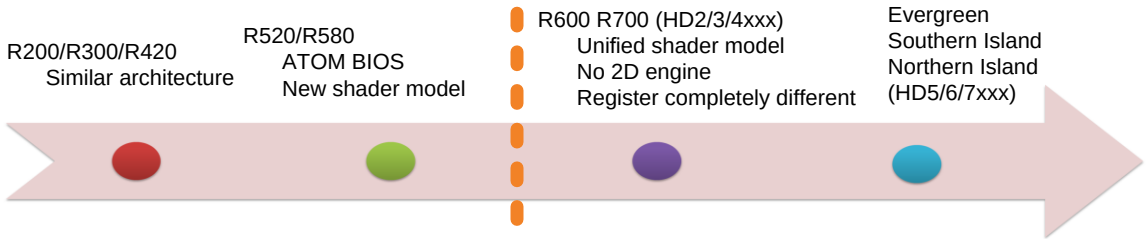
# 第 2 章 AMD 显卡工作原理

\author{赵自成 cl}
\date{\today}\maketitle
\graphicspath{{./}{graph/}}

早期的显卡仅用于显示用，我们可以称为显示控制器;后来显卡中加入了 2D 加速部件，这些部件用于做拷屏，画点，画线等操作;随着游戏、三维模拟以及科学计算可视化等需要，对 3D 的需求逐渐增加，大规模的 3D 程序场景涉及大量的图形学计算，而在早期，图形绘制工作由 CPU 来完成，要达到真实感和实时效果，只能绘制一些简单的线框模型，连最基本的填充都无法实现，更不用说真实感绘制中的光照和纹理贴图。上世纪 80 年代，斯坦福大学的 Jim Clark 教授率先提出用专用集成电路技术实现一个专用的 3D 图形处理器的设想，随后与他的学生创立了 SGI 公司，并于 1984 年推出了世界上第一个通用图形工作站 IRIS1400。图形处理器（GPU）的出现使得图形绘制的概念彻底发生了变化。\verb|参考文献 面向移动设备的真实感图形处理器，另外请参考 Global Illumination on GPU 前面两章的描述 |

## 2.1 AMD 显卡简介

### 2.2.1 AMD 显卡体系结构发展历程




图\ref{radeonprocess}显示了 AMD GPU 核的变化过程，R100 是一款固定功能流水线的显卡，R200 为可编程处理器，R300 在 R200 的急促上发生了比较大的变化，此后的 R400 和 R300 差别不大。到 R500 的时候 GPU 除了有 vbios 外，还引入了 atombios，atombios 是一段比较简单的脚本，和具体平台无关，使用特定的解释器解释执行，然而 GPU 3D 核较 R300 变化不大，寄存器变化也比较少（主要变化在 pixel shader 部分）。

R600 在 R500 的基础上发生很大的变化，2D 部件被废除，3D 部件改成了 unified shader 架构，GPU 的寄存器和原来完全不同，由于体系结构的变化，对硬件编程也和原来有了很大变化。R700 GPU 是 R600 的优化版，R7xx 的编程和 R6xx 的编程基本上是一样的，后续的 Evergreen、Southern Island 和 Northern Island 都是这种 unified shader 架构的延续，因此理解 R600 的编程后将比较容易理解后续 GPU 核的编程。

本书的后续部分针对 R600 核心进行描述，总体来说，R600 硬件上重要的变化包括【个

人观点】:

\begin{enumerate}
\item 不再包含 2D 加速部件，所有的加速都使用 3D 部件完成
\item 使用 Shader Model 4.0，包含 Geometry shader【详情请查阅关于 shader model 4.0 和 geometry shader 的相关资料】
\item 使用 Unified shader 架构，vertex shader、fragment shader（以及 geometry shader）不再分成单独的部件，而是被统一起来。
\end{enumerate}

除了以上变化之外，还有一点值得注意的是，R6xx 显卡 GPU 核的计算都是基于标量的，而不是像以前的那样基于矢量的【参考 ATI Radeon? HD 2000 programming guide】

对于软件编程而言寄存器读写方式上也有比较大的变化，过去对 R5xx 编程的时候，所有的对硬件的编程都是可以命令包或者普通的直接写寄存器的方式进行的，但是 R6xx 的部分寄存器是不能够通过直接写寄存器的方式进行编程的，必须通过 3 型命令包的方式进行【目前看到的代码都是这样的，R6xx 寄存器手册上有很多寄存器地址都是一样的，但是含义不同，这些寄存器偏移都大于等于 0x8000】，后续的代码会看到这方面的具体情况。

2.2 显示输出
\subsection{模式、Vbios 和 Atomios}
\subsection{Framebuffer 驱动中的模式设置}
\subsection{Xorg 图形环境下的 Radeon 显卡模式设置}

## 2.3 显存管理

### 2.3.1 VRAM 内存和 GTT 内存

显卡使用的内存分为两部分，一部分是显卡自带的显存称为 VRAM 内存，另外一部分是系统主存称为 GTT 内存（graphics translation table 和后面的 GART 含义相同，都是指显卡的页表，GTT 内存可以就理解为需要建立 GPU 页表的显存）。在嵌入式系统或者集成显卡上，显卡通常是不自带显存的，而是完全使用系统内存。通常显卡上的显存访存速度数倍于系统内存，因而许多数据如果是放在显卡自带显存上，其速度将明显高于使用系统内存的情况。

某些内容是必须放在 vram 中的，比如最终用于显示的“帧缓存”，以及后面说的页表 GART （graphics addres remapping table），另外有一些比如后面将介绍的命令环缓冲区（ring buffer）是要放在 GTT 内存中的。另一方面，VRAM 内存是有限的，如果 VRAM 内存使用完了，则必须将一些数据放入 GTT 内存中。

通常 GTT 内存是按需分配的，而且是给设备使用的，比如 radeon r600 显卡最多可以使用 512M 系统内存，一次性分配 512M 连续的给设备用的内存在 linux 系统中是不可能成功的，而且即使可以成功，有相当多的内存是会被浪费掉的。按照按需分配的原则，使用多少就从系统内存中分配多少，这样得到的 GTT 内存在内存中肯定是不连续的。GPU 同时需要使用 VRAM 内存和 GTT 内存，最简单的方法就是将这两片内存统一编址（这类似 RISC 机器上 IO 和 MEM 统一编址），VRAM 是显卡自带的内存，其地址一定是连续的，但是不连

续的 GTT 内存如果要统一编址，就必须通过页表建立映射关系了，这个页表被称为 GTT（Graphics Translation Table，地址转换表）或者 GART（Graphics address remapping table），这也是这些内存被称为 GTT 内存的原因。

和 CPU 端地址类似，我们将 GPU 使用的地址称为“GPU 虚拟地址”，经过查页表之后的地址称为“GPU 物理地址”，这些地址是 GPU 最终用于访存的地址，由于 GPU 挂接在设备总线上，因此这里的“GPU 物理地址”就是“总线地址”，当然落在 vram 区域的内存是不用建页表的，这一片内存区域的地址我们只关心其“GPU 虚拟地址”。

R600 显卡核心存管理有关的寄存器如表\ref{memreg}示，目前并没有找到完整的描述这些寄存器的手册，表中的数据根据我们阅读代码获取到。

| 寄存器名称 | 地址 | 功能 |
|---|---|---|
| R600 CONFIG MEMSIZE | 0x5428 | VRAM 大小 |
| MC VM FB LOCATION | 0x2180 | VRAM 区域在 GPU 虚拟地址空间的起始地址和长度 |
| MC_VM_SYSTEM_APERTURE_LOW_ADDR | 0x2190 | VRAM 区域在 GPU 虚拟地址空间的起始地址 |
| MC_VM_SYSTEM_APERTURE_HIGH_ADDR | 0x2194 | VRAM 区域在 GPU 虚拟地址空间的结束地址 |
| VM L2_CNTL | | GPU L2 Cache 控制寄存器 |
| MC_VM_L1_TLB_MCB | | GPU TLB 控制寄存器 |
| VM_CONTEXT0_PAGE_TABLE_START _ADDR | 0x1594 | GTT 内存的起始地址 |
| VM_CONTEXT0_PAGE_TABLE_END_ADDR | 0x15B4 | GTT 内存的结束地址 |
| VM_CONTEXT0_PAGE_TABLE_BASE_ADDR | 0x1574 | GPU 页表基地址 |
| VM_CONTEXT0_CNTL | 0x1410 | GPU 虚拟地址空间使能寄存器 |
| VM_CONTEXT0_PROTECTION_ FAULT_DEFAULT_ADDR | 0x1554 | GPU 页故障寄存器 |
| RADEON_PCIE_TX_DISCARD_ RD_ADDR_LO/HI | | |
| RADEON_PCIE_TX_GART_ERROR | | |
| | | |

【这一段代码修改成使用 R600 显卡描述】

在 Radeon 显卡中，VRAM 内存涉及到“visiable vram”和“real vram”两个说法，visiable vram 是可以使用 pci 设备内存映射方式映射出来的内存，这部分内存可供软件访问，而显卡的 vram 还有一部分是不可见的，不能被软件直接访问[是 GPU 自身使用的？]，这部分内存加上 visiable ram 共同构成显卡的 real vram。有一些显卡的 vram 是可以全部被访问到的，比如 Loogson 2A【？？】机器使用的北桥芯片集成的 RS780 显卡。

在我们使用的 R580 显卡上，通过读取 pci 配置空间可以获取到 visiable vram 大小为 256M，读取 RADEON_CONFIG_MEMSIZE 获取 real vram 大小为 512M，于是 vram 长度为 512M，将 vram 起始地址设置为 0x0,那么结束地址为 0x1fffffff，然后将起始地址和结束地址写入 R_000004_MC_FB_LOCATION（在 r600 上是）寄存器：

\begin{verbatim}
```
rv515_mc_wreg(R_000004_MC_FB_LOCATION,
    S_000004_MC_FB_START(rdev->vram_start >> 16) |
    S_000004_MC_FB_TOP(rdev->vram_end >> 16));
```
\end{verbatim}

S_000004_MC_FB_START 和 S_000004_MC_FB_TOP 两个宏的意思可以查阅我们的演示代码或者内核 radeon 驱动代码。

而后是设置 GTT 内存和 GART。GTT 的大小是由驱动自己确定的，GTT 大小确定后，GART 占用的内存也就确定了。在 R580 显卡上将 GTT 起始地址设为 0x20000000，紧接在 VRAM 后，为 GART 分配好内存后，使能页表机制的时候告知 GPU GTT 的位置和 GART 的位置：

```
 1 uint32_t table_addr;
 2 uint32_t tmp = RADEON_PCIE_TX_GART_UNMAPPED_ACCESS_DISCARD;
 3 rv370_pcie_wreg(rdev, RADEON_PCIE_TX_GART_CNTL, tmp);
 4 rv370_pcie_wreg(rdev, RADEON_PCIE_TX_GART_START_LO, rdev->gtt_start);
 5 tmp = rdev->gtt_end & ~RADEON_GPU_PAGE_MASK;
 6 rv370_pcie_wreg(rdev, RADEON_PCIE_TX_GART_END_LO, tmp);
 7 rv370_pcie_wreg(rdev, RADEON_PCIE_TX_GART_START_HI, 0);
 8 rv370_pcie_wreg(rdev, RADEON_PCIE_TX_GART_END_HI, 0);
 9 table_addr = rdev->zone[GART_ZONE].gpu;
10 rv370_pcie_wreg(rdev, RADEON_PCIE_TX_GART_BASE, table_addr);
11 /* FIXME: setup default page */
12 rv370_pcie_wreg(rdev,RADEON_PCIE_TX_DISCARD_RD_ADDR_LO,rdev->vram_start);
13 rv370_pcie_wreg(rdev, RADEON_PCIE_TX_DISCARD_RD_ADDR_HI, 0);
14 /* Clear error */
15 rv370_pcie_wreg(rdev, RADEON_PCIE_TX_GART_ERROR, 0);
16 tmp = rv370_pcie_rreg(rdev, RADEON_PCIE_TX_GART_CNTL);
17 tmp |= RADEON_PCIE_TX_GART_EN;
18 tmp |= RADEON_PCIE_TX_GART_UNMAPPED_ACCESS_DISCARD;
19 rv370_pcie_wreg(rdev, RADEON_PCIE_TX_GART_CNTL, tmp);
```

代码 4,6-8 行设置 gtt 内存起始地址，10 行设置页表基地址，16-19 行使能页表。

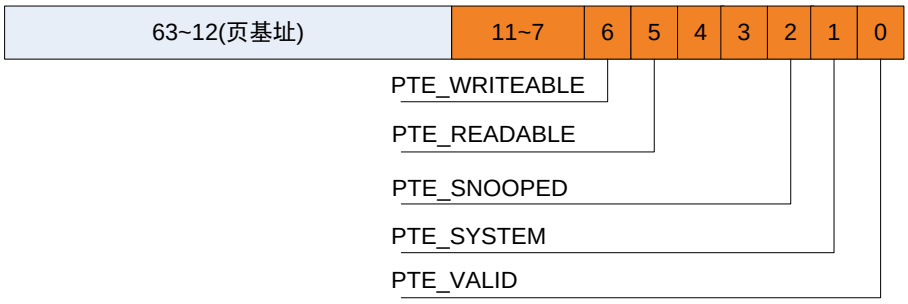
在我们演示程序中，为了简化，GPU VRAM 的使用是静态指定的。这部分代码在 gpu_virtual_location 函数中。比如为 r600 blit 过程对 shader 进行编程的代码分配内存：

```
288         rdev.zone[BLIT_ZONE].size = ALIGN(BLIT_BUF_SIZE, PAGE_SIZE);
289         rdev.zone[BLIT_ZONE].gpu =rdev.zone[GTT_TABLE_ZONE].gpu -
rdev.zone[BLIT_ZONE].size;
290         rdev.zone[BLIT_ZONE].ptr = rdev.zone[BLIT_ZONE].gpu -
rdev.zone[VRAM_ZONE].gpu +rdev.zone[VRAM_ZONE].ptr;
```

所有分配的内存都要同时记录两个地址：用于 GPU 访问的 GPU 虚拟地址空间的地址和用于软件访问的 CPU 虚拟地址空间的地址。

## 2.3.2 Radeon 显卡页表机制

GTT 是显卡的页表，相比于 CPU 使用的 3 级页表，radeon GPU 使用的页表比较简单，radeon GPU 使用的是 1 级页表[是否可配置]，页表大小为 4K，那么页表项的后面 12 位（$2^{12}$=4k）为标志位。在早期的 radeon GPU 中，GPU 使用的页表页表项是 32 位的，到 r600 之后 GPU 页表项为 64 位(图\ref{r600pte})。




页表项的 12 位标志位中只有后 6 位有用，定义如下：

```
#define R600_PTE_VALID          (1 << 0)
#define R600_PTE_SYSTEM         (1 << 1)
#define R600_ PTE_SNOOPED       (1 << 2)
#define R600_PTE_READABLE       (1 << 5)
#define R600_PTE_WRITEABLE (1 << 6)
```

GPU 页表在 GPU VRAM 内存中，VM_CONTEXT0_PAGE_TABLE_BASE_ADDR 和 VM_CONTEXT0_PAGE_TABLE_END_ADDR 两个寄存器表明了页表在 vram 中的位置。由显卡的一个寄存器表明其在 GPU vram 中的位置，下面这个函数实现了填页表的过程。

```
static inline void set_gpu_page(uint64_t dma_addr, uint32_t index)
{
    void __iomem *ptr = (void *)rdev.zone[GTT_TABLE_ZONE].cpu;
    dma_addr &= 0xfffffffffffff000ULL;
    dma_addr |= R600_PTE_VALID | R600_PTE_SYSTEM | R600_PTE_SNOOPED;
    dma_addr |=R600_PTE_READABLE |R600_PTE_WRITEABLE;
    writeq(dma_addr, ((void __iomem *)ptr) + (index * 8));
}
```

上述函数有两个参数，dma_addr 是分配的系统内存经过映射后的总线地址，这个地址用于设备访问主存，也是我们上文说的“GPU 物理地址”，后面一个参数 index 是页表项索引。代码中 ptr 是页表所在的内存在 CPU 虚拟地址空间中的地址，r600 的页表项为 64 位，r500 及以下的页表为 32 位。

我们看一片内存的分配和映射情况。

第三章我们将使用一个称为 ring buffer 的一片内存，这片内存用于放置命令，cpu 将命令放置到这一片内存中，gpu 从这一片内存中拿数据。

```
1316 int radeon_ring_init(struct pci_dev *pdev)
1317 {
1318         uint32_t tmp_addr;
1319         struct page *cpu_page;
1320         dma_addr_t dma_addr;
1321         dma_addr_t tmp_dma_addr;
1322         int j, i;
1323
```

```
1324        rdev.cp_gpu_addr = rdev.zone[RING_ZONE].gpu;
1325        // 1M shoulb be contiguous
1326        cpu_page = alloc_pages(GFP_DMA32|GFP_KERNEL|__GFP_ZERO,
1327                            get_order(rdev.zone[RING_ZONE].size));
1328        if(NULL == cpu_page) {
1329              printk("ERROR:    cann't alloc page for ring \n");
1330              goto alloc_error;
1331        }
1332        rdev.cp_ring = phys_to_virt(page_to_phys(cpu_page));
1333        dma_addr = pci_map_page(pdev, cpu_page, 0,
1334                            rdev.zone[RING_ZONE].size, PCI_DMA_BIDIRECTIONAL);
1335        if(0 == dma_addr){
1336              printk("ERROR: map ring buffer error\n");
1337              goto map_error;
1338        }
1339        rdev.cp_dma_addr = dma_addr;
1340        tmp_addr = rdev.zone[RING_ZONE].gpu;
1341        for (j = 0; j < rdev.zone[RING_ZONE].size/PAGE_SIZE; j++) {
1342              tmp_dma_addr = dma_addr + j * PAGE_SIZE;
1343              tmp_dma_addr &= 0xfffffffffffff000ULL;
1344                            tmp_dma_addr   |=   R600_PTE_VALID   |   R600_PTE_SYSTEM   |
R600_PTE_SNOOPED;
1345              tmp_dma_addr |= R600_PTE_READABLE |R600_PTE_WRITEABLE;
1346
1347              for(i = 0;i < PAGE_SIZE/GPU_PAGE_SIZE; i++) {
1348                    set_gpu_page(tmp_dma_addr + i * GPU_PAGE_SIZE,
1349                          (tmp_addr - rdev.zone[GTT_ZONE].gpu)>>12UL);
1350                    tmp_addr += GPU_PAGE_SIZE;
1351              }
1352
1353        }
1354        mb();
1355        if (pcie_tlb_flush()) {
1356              printk("ERROR: flush firstly\n");
1357              goto flush_error;
1358        }
1359
1360        rdev.cp_ptr_mask = (rdev.cp_ring_size / 4) - 1;
1361        rdev.cp_ring_free_dw = rdev.cp_ring_size / 4;
1362        return 0;
1363
1364 flush_error:
1365        pci_unmap_page(pdev, dma_addr, rdev.zone[RING_ZONE].size,
1366                            PCI_DMA_BIDIRECTIONAL);
```

```
1367 map_error:
1368        free_pages((unsigned long)rdev.cp_ring,
1369                   get_order(rdev.zone[RING_ZONE].size));
1370 alloc_error:
1371        // wati to do??
1372        return -1;
1373 }
```

\ref{vramgtt}节我们说过，在 GPU 端的虚拟地址空间中，内存（显存）是静态划分的，因而 1324 行是使用预先定好的地址。

由于 ring 环必须在物理内存上连续，而且需要按照页（GPU 页）对齐。Radeon GPU 页为 4K，主流硬件平台上主机操作系统上的页大小都是 4K 的整数倍，因而按照 CPU 页对齐就可以了。

得到的 cpu_page 是个 struct page 类型，1332 行获取这片内存读应的虚拟地址。

1333-1334 行映射这片内存到 pci 地址空间中，获取到这片内存的总线地址。

1341-1353 行的内容为填充页表的过程，由于主机操作系统的页大小和 GPU 页大小可能不一致，因而要将一个 GPU 页拆分成$PAGE_SIZE/GPU_PAGE_SIZE$个 GPU 页，对每个 GPU 页大小的内存块分别建立映射。由于最终是 GPU（设备）访问内存，页表里面填的都是总线地址。

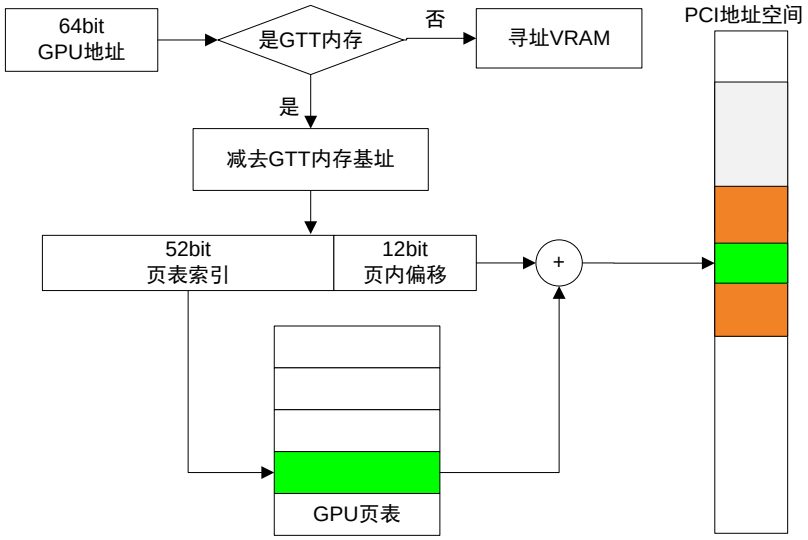
1354-1358 行保证页表项被填充到了页表中。

【是否需要这张图和这部分内容】

结合图\ref{memallocate}来说明内存的映射关系。

这里的主机操作系统页大小为 16K。现在需要在 GTT 内存申请一片 16K（一个 CPU 页）内存，先从系统内存中分一片 16K 大小的 mem1，然后将 mem1 分成 4（PAGE_SIZE/GPU_PAGE_SIZE）段，每一段和一个 GPU 页对应，一个 CPU 页对应 4 个 GPU 页，对应页表中的 4 项。然后我们需要一片更大的内存 mem2，mem2 在物理上没有连续的要求，但是 mem2 比较大，一次分不出这么大的内存，我们将 mem2 分成两次（或者多次）分配，每次映射一部分，通过页表映射后不连续的物理地址在 GPU 地址空间中变得连续了。

在页表机制初始化完成后，R600 显卡 GPU 访存按照图\ref{gpuaddressing}显示的过程进行。




如果是 GTT 内存，则需要查 GPU 页表，根据 64 位地址（在当前的驱动中实际上只用了 32 位）的前面 50 位定位 GPU 页表项，根据页表项内容的后 12 位与上 0 即是内存在 PCI 设备空间中的“页基址”，“页基址”加上原来 64 位地址的后 12 位（页内偏移）就得到对应的总线地址。

注意到由于 vram 和 GTT 统一编址，而 vram 并不参与这里的页表地址转换过程，因而需要有减去 GTT 内存基址的过程，本节代码 1349 行将 tmp_addr 减去 rdev.zone[GTT_ZONE].gpu 就是这个原因。当然上图的整个过程是 GPU 硬件做的，对软件透明。

```
1348                    set_gpu_page(tmp_dma_addr + i * GPU_PAGE_SIZE,
1349                            (tmp_addr - rdev.zone[GTT_ZONE].gpu)>>12UL);
```

在分配完所有内存后，我们的 R600 显存使用情况如下图示：

```
\begin{verbatim}
\begin{figure} %  修改插图
   \centering
   % Requires \usepackage{graphicx}
   \includegraphics[scale=1.0]{memusage}\\
   \caption{显存使用情况}\label{memusage}
\end{figure}
\end{verbatim}
```

从下往上看，Fb 是帧缓存，blit 内存主要是 r600 上 blit 操作对 shader 编程的一些代码，然后是 gpu 页表，atombios 是显卡中带的一段代码，显卡可执行这段代码对显卡进行初始化，我们先将代码读到这片内存中，然后执行之，在模式设置一节将会讨论 atombios。

以上是必须放在 vram 内存上的内容，后面 gtt 内存。wb 是 write back buffer 的简称，ring buffer 和 ib（indirect buffer）会在下章讨论。

## 2.3.3 Linux 显存管理 API

\ref{vramgtt}节中我们为了简单起见，使用的 vram 内存是静态分配的，但是在 linux 内核中是有一套完善的内存管理机制的。

和操作系统里面的系统内存管理一样，这套机制比较复杂，我们这里不详细描述这套机制的具体实现，而是简单描述如何在核内核外获取和使用显存。

**内核部分**

在 radeon 内核驱动代码 radeon_device_init 函数中有如下代码：
drivers/gpu/drm/radeon/radeon_device.c radeon_device_init

```
810         if (radeon_testing) {
811                 radeon_test_moves(rdev);
812         }
```

810 行是一个全局变量开关，当这个开关开启的时候，驱动会做一个拷屏操作，这段代码在 drivers/gpu/drm/radeon/radeon_test.c 文件中，radeon_test_moves 做些数据拷贝操作，包括从 vram 到系统主存和系统主存到 vram 之间的数据拷贝（这个是我们直接够在 radeon 内核驱动代码中运行并且能够看到效果的命令处理过程）。如果我们利用 radeon 驱动提供的软硬件环境进行编程，可以将代码放在这个地方。

```
1       struct radeon_bo *vram_obj = NULL;
2       struct radeon_bo *gtt_obj = NULL;
3       uint64_t vram_addr, gtt_addr;
4       unsigned    size;
5       void *vram_map, *gtt_map;
6
7       size = 1024 * 768 * 4;
8       r = radeon_bo_create(rdev, size, PAGE_SIZE, true,
9                            RADEON_GEM_DOMAIN_VRAM, &vram_obj);
10      if (r) {
11              DRM_ERROR("Failed to create VRAM object\n");
12              goto out_cleanup;
13      }
14      r = radeon_bo_reserve(vram_obj, false);
15      if (unlikely(r != 0))
16              goto out_cleanup;
17      r = radeon_bo_pin(vram_obj, RADEON_GEM_DOMAIN_VRAM, &vram_addr);
18      if (r) {
19
20              DRM_ERROR("Failed to pin VRAM object\n");
21              goto out_cleanup;
22      }
```

```
23      r = radeon_bo_kmap(vram_obj, &vram_map);
24      if (r) {
25          DRM_ERROR("Failed to map VRAM object\n");
26          goto out_cleanup;
27      }
28
29      r = radeon_bo_create(rdev, size, PAGE_SIZE, true,
30                           RADEON_GEM_DOMAIN_GTT, &gtt_obj);
31      if (r) {
32          DRM_ERROR("Failed to create GTT object\n");
33          goto out_cleanup;
34      }
35      r = radeon_bo_reserve(gtt_obj, false);
36      if (unlikely(r != 0))
37          goto out_cleanup;
38      r = radeon_bo_pin(gtt_obj, RADEON_GEM_DOMAIN_GTT, &gtt_addr);
39      if (r) {
40          DRM_ERROR("Failed to pin GTT object\n");
41          goto out_cleanup;
42      }
43      r = radeon_bo_kmap(gtt_obj, &gtt_map);
44      if (r) {
45          DRM_ERROR("Failed to map GTT object\n");
46          goto out_cleanup;
47      }
48
49 out_cleanup:
50      if (vram_obj) {
51          if (radeon_bo_is_reserved(vram_obj)) {
52              radeon_bo_unpin(vram_obj);
53              radeon_bo_unreserve(vram_obj);
54          }
55          radeon_bo_unref(&vram_obj);
56      }
57      if(gtt_obj){
58          if(radeon_bo_is_reserved(gtt_obj)){
59              radeon_bo_unpin(gtt_obj);
60              radeon_bo_unreserve(gtt_obj);
61          }
62          radeon_bo_unref(&gtt_obj);
63      }
```

在 gem 内存管理机制中，所有分配的显存以上代码显示了创建两个 buffer object（bo），分别从 vram 和 gtt 内存中分配内存空间，以及释放内存空间和 bo 的过程。

Buffer object 是显卡对显存管理的基本结构，是对一片内存的抽象，不同的显卡会有不同的定义，但是基本原理和内容大致相同，radeon 显卡驱动中使用的是 radeon_bo 结构来管理和描述一片显存。

1-2 行，这里我们有两个 bo 对象（分配两片显存），一片内存来自 vram，另外一片来自 gtt 内存。

8 行，创建并初始化一个 bo，分配显存。参数如下：

\begin{description}

\item[rdev] radeon_device 结构体指针

\item[size] 该 bo 的大小

\item[True] 来自内核还是用户空间的请求，如果是内核，则分配 bo 结构过程是不可中断的，并且从用户空间和内核空间访问这篇显存的时候虚拟地址和物理地址间的映射关系是不同的，因而这里要做区分

\item[RADEON_GEM_DOMAIN_VRAM] 显存位于 vram 还是 gtt 内存，radeon 驱动中定义了 3 中类型的显存:\\

\#define RADEON_GEM_DOMAIN_CPU \qquad\quad 0x1 \\

\#define RADEON_GEM_DOMAIN_GTT \qquad\quad 0x2 \\

\#define RADEON_GEM_DOMAIN_VRAM \qquad 0x4 \\

RADEON_GEM_DOMAIN_CPU 暂不清楚是何用途，后面两个表示内存分别来自 gtt 内存和 vram。

\item[vram_obj] bo 指针，返回的 bo 结构体

\end{description}

14 行，reserve（保留）bo，（表明当前 bo 已经被使用，不允许其他代码使用？？）。如果 bo 已经被 reserve，那么这里的要等到 bo 被 unreserve 之后才能使用。（确认？？）

17 行，获取 bo 代表的显存的 GPU 虚拟地址，GPU 将使用这个地址访问内存，后面我们让 GPU 访存的时候用的都是这类型的地址。

23 行，映射 bo 代表的显存空间，该函数的第二个参数返回映射后的虚拟地址，驱动将使用这个访问这片内存。

29-47 行代码和上面说的原理相同，不同的是这片内存来自 GTT 内存，在 API 函数内部处理的时候区别会比较大，但是使用 API 时只有只有显存类型这个参数不同。

50-56 行释放内存和 bo 结构。

**用户空间**

用户空间通过 libdrm 获取显存。下面这段代码显示了核外如何获取和使用显存：

```
1       int ret;
2       struct kms_bo *bo;
3       unsigned bo_attribs[] = {
4           KMS_WIDTH,      0,
5           KMS_HEIGHT,     0,
6           KMS_BO_TYPE, KMS_BO_TYPE_SCANOUT_X8R8G8B8,
7           KMS_TERMINATE_PROP_LIST
8       };
9       bo_attribs[1] = width;
10      bo_attribs[3] = height;
```

```
11      ret = kms_bo_create(kms, bo_attribs, &bo);
12      if (ret) {
13          fprintf(stderr, "failed to alloc buffer: %s\n", strerror(-ret));
14          return NULL;
15      }
16      ret = kms_bo_get_prop(bo, KMS_PITCH, stride);
17      if (ret) {
18          fprintf(stderr, "failed to retreive buffer stride: %s\n",   strerror(-ret));
19          kms_bo_destroy(&bo);
20          return NULL;
21      }
22      ret = kms_bo_map(bo, &virtual);
23      if (ret) {
24          fprintf(stderr, "failed to map buffer: %s\n", strerror(-ret));
25          kms_bo_destroy(&bo);
26          return NULL;
27      }
28      return bo;
```

这段代码和内核中的代码很相似，读者根据调用的函数的函数名就应该能够理解其含义了。

如果是 radeon 显卡页可以使用如下代码：

```
1       struct radeon_bo *bo;
2
3       *stride =   width * 4;
4       bo = radeon_bo_open(bufmgr, 0, *stride * height, 1<<12UL,
5                           RADEON_GEM_DOMAIN_VRAM, 0);
6       if (bo == NULL) {
7           fprintf(stderr, "error open bo\n");
8           return NULL;
9       }
10      radeon_bo_map(bo, 1);
11      virtual = bo->ptr;
12      return bo;
```

【这些代码来自 libdrm 或者一个介绍 libdrm 的网站，本文档描述并不完整】

## 2.4 命令环机制

Radeon 显卡使用命令环机制，命令环是 GTT 内存中分出来的一片内存，驱动程序往命令环中填充数据，填充完后通知 GPU 命令已经写入命令，GPU 的命令处理器 CP（Command

Processor）开始执行命令。

在前面的描述中我们即是通过 ring 环内存的使用来说明如何在系统中分配内存以及建立映射关系的。

## 2.4.1 命令处理器

【请根据我们当前的理解适当修改】

命令处理器（Command Processor，CP）是面向图形控制器的可编程的专用计算引擎，主要是处理环形缓冲和间接线性缓冲的命令流，即获取和翻译 PROMO4 命令流。

图形控制器中的 CP 主要完成以下任务：

\begin{enumerate}

\item 接收驱动程序的命令流。驱动程序要么将命令流先写入系统内存，然后由 CP 通过总线主设备访问方式进行获取；要么就直接通过 PCI 设备访问方式将命令流写给 CP。当前支持三种命令流：环形缓冲命令流、间接缓冲 1 命令流和间接缓冲 2 命令流。

\item 解析命令流，将解析后的数据传输给图形控制器的其他内部特征模块，内部特征模块包括 3D 图形处理器、2D 图形处理器、视频处理器或者 MPEG 解码器。每时钟间隔内，数据传输可以是 32 位、64 位、96 位或者 128 位；其中 64 位、96 位和 128 位数据传输必须在向量写模式下进行。而且，向量写模式只有在命令流处于拉模式下才有效。推模式下只能进行 32 位的数据传输。

\item CP 内部有两通用 DMA 引擎，一个用于图形用户接口相关任务，一个专门用于视频采集任务。DMA 引擎要求窗口源地址和目的地址都必须进行字节对齐。

\end{enumerate}

CP 接受驱动程序的命令流有两种方式，对 GPU 的 CP 而言，称为“推模式”和“拉模式”。推模式就是可编程 IO 模式，即驱动程序通过 PCI 总线访问直接操作 GPU；通过这种方式，驱动程序可以将寄存器写序列或命令包序列直接发送给 GPU。

\begin{enumerate}

\item 寄存器写序列。设置图形控制器的处理引擎状态，然后启动引擎。一般说来，通过某个寄存器的写可以触发引擎。这种情况常常用于调试。

\item 命令包序列。以压缩方式将命令信息传输给 GPU；然后，GPU 中的智能控制器将命令包转换成对其他处理引擎的寄存器写。

\end{enumerate}

在拉模式下，CPU 先将命令信息写入系统主存；然后 GPU 使用总线主设备访问方式，读取系统主存。因此拉模式需要解决的重要问题是，CPU 和 GPU 如何管理和访问系统主存中的共享缓冲。

推模式的命令缓冲受限与 GPU 片上存储空间；而拉模式就不存在这样的限制。推模式在某些整体系统性能上也有自己的优点，例如访存带宽比较少。另外，实现 GPU 片上命令缓冲溢出到帧缓冲技术，也可以缓解推模式的缓冲受限状况；但是，这也要求帧缓冲带宽能满足命令缓冲读写的需求。

CP 在推拉两种模式之间的切换必须谨慎处理。切换常常不能成功；一般说来，整机系统在复位过程就选择好了模式（拉模式会被程序员优先选择），在整个系统运行过程中不进行模式切换。

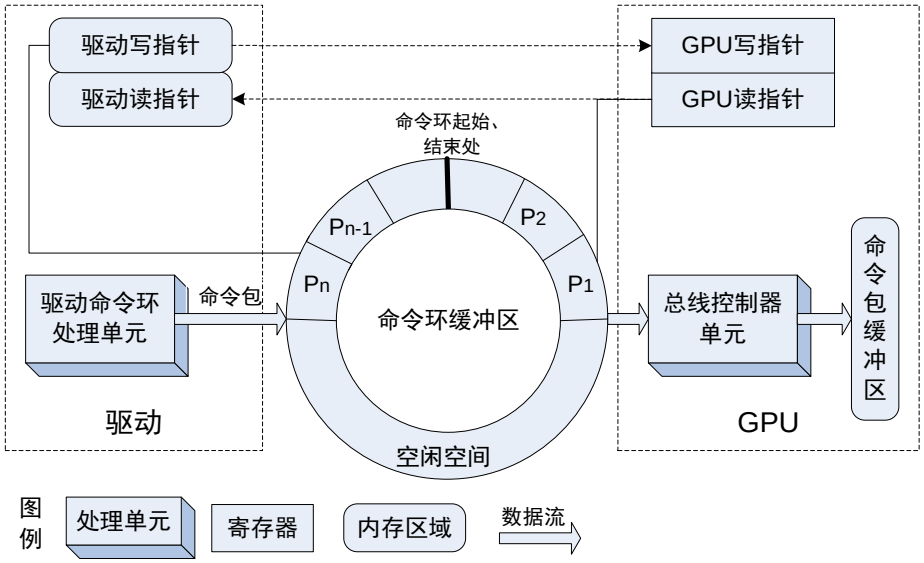
在后面的代码中，我们使用的推模式主要是"寄存器写序列"，使用命令包的时候使用拉

模式。【是否正确】

## 2.4. 2 命令环缓冲区

在拉模式【？？】下，驱动程序在系统内存中为命令包申请一块缓冲区。GPU 会根据这些命令包去执行屏幕绘图等操作。这种命令缓冲区按照环形方式进行管理，是 CPU 和 GPU 共享的一片系统主存，CPU 负责写入命令包，GPU 负责读取和解析命令包。因为 CPU 和 GPU 看到的环形缓冲区状态必须是一致性，所以 CPU 和 GPU 都要共同维护和管理环形缓冲区的状态：基地址、长度、写指针和读指针。为了使 Ring Buffer 能够正常工作，CPU 和 CPU 必须维护这种状态的一致性。Ring Buffer 基地址和大小是在系统第一次启动时已经初始化好的，之后一般也不会改变。一个简单的工作是初始化这种状态的读指针和写指针的拷贝。另一方面，当操作 Ring Buffer 时， 读指针和写指针的修改非常频繁；为了维护环形缓冲区的状态一致性，当写操作者（CPU）更新写指针时，它必须发这个值给图像控制器写指针的读拷贝。同样的，当读操作者（GPU）更新读指针时，它必须发这个值给读指针的写拷贝。CPU 填写命令包、递增写指针后将写指针内容发送给 GPU；当 GPU 抽取命令包、递增读指针后将读指针内容发送给 CPU。无论是 CPU 还是 GPU 都是从低地址开始进行填写或抽取操作的，一旦到了环形缓冲区的结束处，又从环形缓冲区起始处继续；同时，GPU 从队列头取 CPU 写入的命令包，一旦到达队列尾，又从头开始。

见图\ref{ringbuffer}。

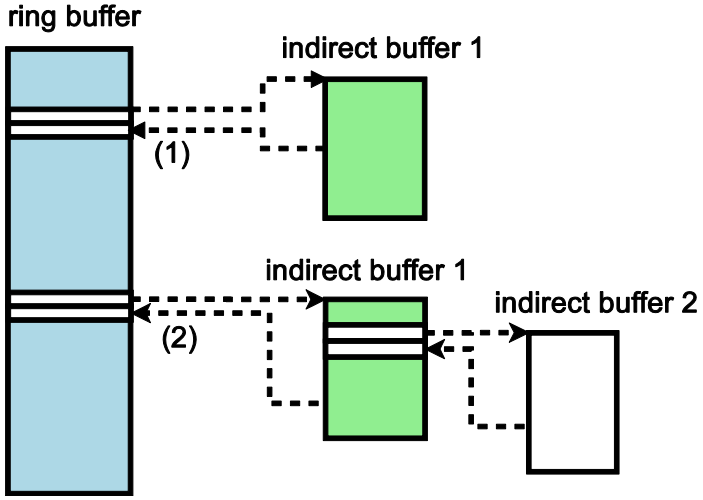



| 状态 | 说明 | 初始化值 | 运行动态变化 |
|---|---|---|---|
| 基地址 | 环形缓冲区起始地址，包括两种形式：内核虚地址，供 CPU 使用；PCI 地址，供 GPU 使用。 | 驱动申请缓冲区时就固定了 | 不变 |
| 长度 | 环形缓冲区长度。 | | 不变 |
| 写指针 | CPU 将要写入的单元索引 | 零 | 递增变化 |
| 读指针 | GPU 正在读取的单元索引 | | 递增变化 |

表\ref{ringbufferstate}显示了相关状态的变化情况。

## 2.4.3 间接缓冲区

在系统主存中，除了环形缓冲区之外，CP 还可以从间接缓冲 1 和间接缓冲 2 中获取命令包。这个过程是这样完成的：在主命令流中（ring buffer）有一个设置 CP 的间接缓冲 1 地址和大小的寄存器。写间接缓冲 1 的寄存器触发 CP 从提供的地址取新的命令流。从主命令流解析最后一个命令包来设置间接缓冲 1 地址和大小的寄存器；然后 CP 开始从间接缓冲 1 中取数据。间接缓冲 1 的数据流可能要设置 CP 的间接缓冲 2 地址和大小的寄存器。和之前的过程一样，写间接缓冲 1 大小的寄存器触发 CP 从提供的地址中获取新的命令流。从间接缓冲 1 流中解析最后一个包来设置间接缓冲 2 地址和大小的寄存器。CP 从间接缓冲 2 取一定数量的数据一直到间接缓冲末尾；然后从间接缓冲 1 返回它的命令包解析。CP 从间接缓冲 1 中取数据一直到间接缓冲 1 的末尾，然后从主命流（ring buffer）返回它的命令包解析。【请适当修改本段后后面内容保证用词统一】




图\ref{indirectbuffer}显示了两个场景，场景（1）是 ring 环执行过程中碰到了运行 indirect buffer 1 的指令，于是 CP 取 indirect buffer 1 中的命令运行，运行完了之后跳回到 ring buffer 中。场景（2）显示了在 indirect buffer 1 运行的过程中还碰到了跳转到 indirect buffer 2 的指令，因而 CP 跳转到 indirect buffer 2 中执行指令，执行完成后跳回到 indirect buffer 1 中。

这个过程有点类似函数调用。程序在运行过程中遇到函数调用，则会使用跳转指令跳到被调用函数入口，执行完函数后跳回到原来的程序位置继续执行。

## 2.4.3 内核命令环缓冲区机制的实现

在 Linux 内核 radeon 驱动中有一个 ring test 过程用于验证 ring buffer 可以正常运行，如果 ring test 通过，那么 GPU 和 CPU 交互的部分已经配置正确，可以正常工作了。

Ring buffer 机制几乎在所有类型的芯片上都是一样的，区别只是 r600 以后的芯片 ring buffer GPU 端读写指针的寄存器地址发生了变化。因而 Linux 内核驱动针对不同 GPU 核实现 ring buffer 机制以及 ring test 过程的代码几乎是完全相同的。

```
2287 int r600_ring_test(struct radeon_device *rdev)
2288 {
2289        uint32_t scratch;
2290        uint32_t tmp = 0;
2291        unsigned i;
2292        int r;
2293
2294        r = radeon_scratch_get(rdev, &scratch);
2295        if (r) {
2296              DRM_ERROR("radeon: cp failed to get scratch reg (%d).\n", r);
2297              return r;
2298        }
2299        WREG32(scratch, 0xCAFEDEAD);
2300        r = radeon_ring_lock(rdev, 3);
2301        if (r) {
2302              DRM_ERROR("radeon: cp failed to lock ring (%d).\n", r);
2303              radeon_scratch_free(rdev, scratch);
2304              return r;
2305        }
2306        radeon_ring_write(rdev, PACKET3(PACKET3_SET_CONFIG_REG, 1));
2307         radeon_ring_write(rdev,  ((scratch  -  PACKET3_SET_CONFIG_REG_OFFSET)  >>
2));
2308        radeon_ring_write(rdev, 0xDEADBEEF);
2309        radeon_ring_unlock_commit(rdev);
2310        for (i = 0; i < rdev->usec_timeout; i++) {
2311              tmp = RREG32(scratch);
2312              if (tmp == 0xDEADBEEF)
2313                    break;
2314              DRM_UDELAY(1);
2315        }
2316        if (i < rdev->usec_timeout) {
2317              DRM_INFO("ring test succeeded in %d usecs\n", i);
2318        } else {
2319              DRM_ERROR("radeon: ring test failed (scratch(0x%04X)=0x%08X)\n",
2320                    scratch, tmp);
2321              r = -EINVAL;
2322        }
2323        radeon_scratch_free(rdev, scratch);
2324        return r;
2325 }
```

以上代码显示了 r600 及其以上 GPU 核内核驱动做 ring test 的代码。

2294 行获取一个可用的 scratch 寄存器，scratch 寄存器是功能未定义的寄存器，由（驱动）软件定义其功能。

2299 行使用 mmio 的方式直接向寄存器中写入值“0xCAFEDEAD”，此时该 scratch 寄存

器的内容为 0xCAFEDEAD。

2300 行向内核驱动中的 ring buffer 机制申请 3 个 dword（gpu 命令都是以 4 字节为单位计的），同时由于会有多个程序并发访问 ring buffer，这里还会对 ring buffer 加锁。

2306-2308 行代码向刚才申请到的 ring buffer 内存中写入 3 个 dword 的命令，关于 GPU 命令在下一章会详细介绍，这里的命令的意思是向刚才的 scratch 寄存器中写入值“0xDEADBEEF”。

2309 行提交命令，上面三行代码写的命令写入 ring buffer 后并不会被执行，直到调用 radeon_ring_unlock_commit 之后命令才会被执行。

2310-2314 行是一个通过轮询的方式检查 scratch 寄存器的过程，如果上面的命令正常运行，那么 scratch 寄存器的值将会是“0xDEADBEEF”，否则命令没有正常运行，ring test 失败。

总结上面的过程，在 radeon 内核驱动使用了下面三个函数就可以操作 ring buffer:

| API 函数 | 功能 | 参数 |
|---|---|---|
| radeon_ring_lock | 申请 ring buffer 内存并锁住 ring buffer，如果 ring buffer 被用完，则更新 CPU 端的读指针。 | N 为申请的 dwords 数目 |
| radeon_ring_write | 向 ring buffer 写入命令和命令参数，这里只更新 CPU 端的写指针 | |
| radeon_ring_unlock_commit | 更新 GPU 端的写指针，释放 ring buffer 锁 | |

需要提及的是 scratch 寄存器，scratch 寄存器是 GPU 预留给软件使用的寄存器，r300 以前的显卡只 5 个 scratch 寄存器，其后的显卡有 7 个寄存器，GPU 本身并不依赖这些寄存器对其进行编程，软件可以自定义其功能。上面这段代码仅仅用于验证命令是否正确执行，然而后面的轮询过程却对我们有所启发：软件发送了命令之后什么时候直到命令被执行完成了？可以按照这里面的做法，在命令尾部再添加一条写 scratch 寄存器的命令（当然必须保证往 scratch 寄存器写入的值和 scratch 寄存器原来的值不一样），而后轮询该 scratch 寄存器，如果这个寄存器被写入了我们要求其写入的值，那么就可以确定命令已经执行完了。这里实际上定义了一个软硬件同步的机制，后面中断机制的章节会讨论驱动中 fence 机制的实现，fence 机制是使用中断实现的，但是那里面使用了我们上面提到的思想。

Ring buffer 机制的实现涉及到上表中的 3 个函数。为了简化，下面使用我们自己的代码来说明 ring buffer 机制的实现，我们自己的代码简化了许多内容。

```
1120 int radeon_ring_lock(struct radeon_device *rdev, unsigned ndw)
1121 {
1122         int r;
1126         r = radeon_ring_alloc(rdev, ndw);
1133         return r;
1134 }

1099 int radeon_ring_alloc(struct radeon_device *rdev, unsigned ndw)
1100 {
1101         ndw = (ndw + rdev->cp_align_mask) & ~rdev->cp_align_mask;
1102         while (ndw > (rdev->cp_ring_free_dw - 1)) {
1103                 radeon_ring_free_size(rdev); /* get the free size */
```

```
1104            if (ndw < rdev->cp_ring_free_dw) {
1105                    break;
1106            }
1112    }
1113    rdev->cp_count_dw = ndw;
1117    return 0;
1118 }
```

```
1074 void radeon_ring_free_size(struct radeon_device *rdev)
1075 {
1076    /* if we have enabled writeback buffer */
1077    /* rdev->cp_rptr = le32_to_cpu(rdev->wb[RADEON_WB_CP_RPTR_OFFSET/4]);*/
1078    if(rdev->family >= CHIP_R600) {
1079            rdev->cp_rptr = mmio_read(R600_CP_RB_RPTR);
1080    }
1081    else {
1082            rdev->cp_rptr = mmio_read(RADEON_CP_RB_RPTR);
1083    }
1084    // now all the packets cost very small mem and will not come here
1085    rdev->cp_ring_free_dw = (rdev->cp_rptr + (rdev->cp_ring_size / 4));
1086    rdev->cp_ring_free_dw -= rdev->cp_wptr;
1087    rdev->cp_ring_free_dw &= rdev->cp_ptr_mask;
1088    if (!rdev->cp_ring_free_dw) {
1089            rdev->cp_ring_free_dw = rdev->cp_ring_size / 4;
1090    }
1091 }
```

ndw 是需要申请的大小，CP 处理命令的时候有对其要求，当前 radeon 驱动中都是要求 16 个 ndw 对其的，1101 行根据对其要求调整 ndw。rdev->cp_ring_free_dw 记录了 ring buffer 中剩余的空间大小，如果空间不够了，调用 radeon_ring_free_size 查询 GPU 的读指针，并更新 CPU 的读指针（1077/1079/1082），直到有足够空间为止。

每一次命令执行完成后，CPU 并不会去查询 GPU 读指针以及更新 CPU 的读指针，而是等到 ring buffer 没有空间之后才会查询，这样查询 GPU 读指针的频率就比较低了。

注意到 1077 行是使用了 write back buffer 的情况，write back buffer 的原理是：可以将寄存器“对应“到某个内存地址上，GPU 将不再写寄存器而是写到指定的内存中，驱动将读写这个内存地址而不是原来的寄存器。我们的代码中没有使用 write back buffer。

```
1132 static inline
1133 void radeon_ring_write( struct radeon_device *rdev, uint32_t v) // write struct rdev
1134 {
1135    rdev->cp_ring[rdev->cp_wptr++] = v;
1136    rdev->cp_wptr &= rdev->cp_ptr_mask;
1137    rdev->cp_count_dw--;/* in radeon driver it is used for debug */
1138    rdev->cp_ring_free_dw--;
1139 }
```

1135 行向当前 ring buffer 位置写入命令或者命令参数。

1138 行更新剩余的 ring buffer 空间。

```
1142 void radeon_ring_unlock_commit(struct radeon_device *rdev)
1143 {
1144        unsigned count_dw_pad;
1145        unsigned i;
1146
1147             /* We pad to match fetch size */
1148        count_dw_pad = (rdev->cp_align_mask + 1) -
1149                                    (rdev->cp_wptr & rdev->cp_align_mask);
1150        for (i = 0; i < count_dw_pad; i++) {
1151             radeon_ring_write(rdev,    2 << 30);
1152        }
1153        mb();
1154        if(rdev->family == CHIP_RS780) {    // notice
1155             mmio_write(CP_RB_WPTR, rdev->cp_wptr);
1156             (void)mmio_read(CP_RB_WPTR);
1157        }
1158        if(rdev->family == CHIP_R580) {
1159             mmio_write(RADEON_CP_RB_WPTR, rdev->cp_wptr);
1160             (void)mmio_read(RADEON_CP_RB_WPTR);
1161        }
1162 }
```

发送命令，每一次发送的 gpu 命令及参数数目都必须是对齐的，在 radeon_ring_alloc 函数分配内存的时候是对齐的，但是填充完命令的时候，由于这些对齐的地方保存有 ring buffer 回绕之前的内容，必须把这些内容清空才不会导致 GPU 运行命令出错。1151 行使用 2 型命令包填充这些对齐位置的内存（GPU 命令包章节将详细讨论命令包格式）。1154-1161 行更新 GPU 写指针（更新写指针后读该寄存器保证内容已经写入该寄存器）更新写指针后 GPU 将开始执行命令。

### 2.4.5 内核间接缓冲区机制的实现

Linux 内核中完成 ring test 后，会有一个 indirect buffer test 过程。这个过程和 ring test 过程完成的操作一样，写 scratch 寄存器。

```
2660 int r600_ib_test(struct radeon_device *rdev)
2661 {
2662        struct radeon_ib *ib;
2663        uint32_t scratch;
2664        uint32_t tmp = 0;
2665        unsigned i;
2666        int r;
2667
2668        r = radeon_scratch_get(rdev, &scratch);
.....
2673        WREG32(scratch, 0xCAFEDEAD);
```

```
2674        r = radeon_ib_get(rdev, &ib);
......
2679        ib->ptr[0] = PACKET3(PACKET3_SET_CONFIG_REG, 1);
2680        ib->ptr[1] = ((scratch - PACKET3_SET_CONFIG_REG_OFFSET) >> 2);
2681        ib->ptr[2] = 0xDEADBEEF;
2682        ib->ptr[3] = PACKET2(0);
2683        ib->ptr[4] = PACKET2(0);
2684        ib->ptr[5] = PACKET2(0);
2685        ib->ptr[6] = PACKET2(0);
2686        ib->ptr[7] = PACKET2(0);
2687        ib->ptr[8] = PACKET2(0);
2688        ib->ptr[9] = PACKET2(0);
2689        ib->ptr[10] = PACKET2(0);
2690        ib->ptr[11] = PACKET2(0);
2691        ib->ptr[12] = PACKET2(0);
2692        ib->ptr[13] = PACKET2(0);
2693        ib->ptr[14] = PACKET2(0);
2694        ib->ptr[15] = PACKET2(0);
2695        ib->length_dw = 16;
2696        r = radeon_ib_schedule(rdev, ib);
......
2703        r = radeon_fence_wait(ib->fence, false);
......
2708        for (i = 0; i < rdev->usec_timeout; i++) {
2709            tmp = RREG32(scratch);
2710            if (tmp == 0xDEADBEEF)
2711                break;
2712            DRM_UDELAY(1);
2713        }
.....
2721        radeon_scratch_free(rdev, scratch);
2722        radeon_ib_free(rdev, &ib);
2723        return r;
2724 }
```

2668-2673 行的内容和 ring test 的过程一样。

2674 行从系统中获取一个 indirect buffer，ib->ptr 中记录了 indirect buffer 在内存中的位置。

2679-2694 向 indirect buffer 中填充命令和参数，这里填写的命令和参数与 ring test 中填写的命令和参数是相同的，当然这里也有对齐要求。

2696 行将填写好的 indirect buffer 添加到调度队列中。

2703 行涉及 fence 机制，在中断机制一节中我们将详细介绍。

仍然使用我们简化后的演示代码来说明 indirect buffer 机制的实现。

```
1962 int gpu_ib_pool_init (struct radeon_device *rdev)
```

```
1963 {
1964        uint32_t tmp_addr;
1965        struct page *cpu_page;
1966        dma_addr_t dma_addr;
1967        dma_addr_t tmp_dma_addr;
1968        int j, i;
1969        int size = rdev->zone[IB_ZONE].size;
1970        struct pci_dev *pdev = rdev->pdev;
1971
1972        rdev->ib_gpu_addr = rdev->zone[IB_ZONE].gpu;
1973        cpu_page = alloc_pages(GFP_DMA32|GFP_KERNEL|__GFP_ZERO,
1974                               get_order(size));
1975
......
1980        rdev->ib = phys_to_virt(page_to_phys(cpu_page));
1981
1982        dma_addr = pci_map_page(pdev, cpu_page, 0, size,
1983                               PCI_DMA_BIDIRECTIONAL);
......
1988        rdev->ib_dma_addr = dma_addr;
1989        tmp_addr = rdev->zone[IB_ZONE].gpu;
......
2005        for (i = 0; i < IB_POOL_SIZE; i++) {
2006            unsigned offset;
2007            offset = i * 64 * 1024;
2008            rdev->ib_pool[i].gpu_addr = rdev->ib_gpu_addr + offset;
2009            rdev->ib_pool[i].ptr = (uint32_t*)((void*)rdev->ib + offset);
2010            rdev->ib_pool[i].dma_addr = rdev->ib_dma_addr + offset;
2011            rdev->ib_pool[i].free = true;
2012            rdev->ib_pool[i].length_dw = 0x0;
2013            rdev->ib_pool[i].idx = i;
2014        }
2015        return 0;
2023 }
```

Indirect buffer 位于 GTT 内存中，其内存的分配和页表建立过程和第 2 章内容的 ring buffer 分配过程相同。和 linux 内核 radeon 驱动一样，这里我们给出了 IB_POOL_SIZE（16）个 indirect buffer，这些 buffer 是在一块连续的物理内存上分割出来的，2005-2014 行设置号地址将这些 indirect buffer 添加到 ib_pool 中。

演示代码中的 gpu_ib_get 从 ib_pool 的 16 个 indirect buffer 中选出一个可用的 buffer，在实际的 linux 内核代码中 gpu_ib_get 还涉及 fence 事件和互斥操作。

```
1582 int radeon_ib_schedule(struct radeon_devie *rdev, struct radeon_ib *ib)
1583 {
1584        int r = 0;
1585        if (ib->length_dw == 0) {
```

```
1586            printk(KERN_ERR "[%s %d] couldn't schedule IB\n", __func__, __LINE__);
1587            return -EINVAL;
1588        }
1589        /* 64 bits is enough */
1590        r = radeon_ring_lock(64);
1591        if(r){
1592            printk(KERN_ERR "[%s %d] allocate ring buffer error\n",
1593                __func__, __LINE__);
1594        }
1595        radeon_ring_ib_execute(rdev, ib);
1596        // we do not have fence and ib will not protected by fence
1597        // so we do not set ib free here and will free late
1598        //radeon_fence_emit(rdev, ib->fence);
1599        //ib->free = true;
1600        radeon_ring_commit(rdev);
1601        return 0;
1602 }

2189 void radeon_ring_ib_execute(struct radeon_device *rdev, struct radeon_ib *ib)
2190 {
2191        radeon_ring_write(rdev, PACKET3(PACKET3_INDIRECT_BUFFER, 2));
2192        radeon_ring_write(rdev, ib->gpu_addr & 0xFFFFFFFC);
2193        radeon_ring_write(rdev, upper_32_bits(ib->gpu_addr) & 0xFF);
2194        radeon_ring_write(rdev, ib->length_dw);
2195 }
```

Indirect buffer 要能够正常运行，必须将其插入到 ring buffer 的代码中去，这就类似在汇编代码中插入"call xx"指令进行函数调用一样。radeon_ring_ib_execute 函数添加的命令就相当于函数调用时使用的 call 指令。

在后面介绍了 GPU 命令包后读者就能够明白这段代码了。每个 indirect buffer 的地址和大小都是在初始化的时候固定的，但是通常往 indirect buffer 中填写的内容的长度是不确定的，在使用的时候必须同时告诉 indirect buffer 的地址和当前使用的长度（2192-2194 行）。调用 radeon_ring_commit 之后，ring buffer 的内容开始执行，此时执行到 indirect buffer 指令后进入 indirect buffer 中的指令执行，这就是图\ref{indirectbuffer}显示的场景 1。

## 2.5 Radeon GPU 命令包

命令换缓冲区一节介绍了 GPU 的命令环机制，在命令环机制下，可以使用推模式和拉模式两种方式实现对 GPU 的编程。本章我们介绍使用命令包的拉模式对 GPU 进行编程。

## 2.5.1 PM4 命令包格式

radeon 显卡可以运行在 PM4 模式下，在这种模式下，不需要直接向寄存器中写数据执行绘图操作，而是在系统内存中准备 PM4 格式的命令包并让硬件（显卡的微引擎）执行绘图命令。

1 型当前定义了 4 中命令包，分别是 0 型/1 型/2 型和 3 型命令包，命令包由两部分组成，第一部分是命令包头，第二部分是命令包主体，命令包头为请求 GPU 执行的具体操作，命令主体为执行该操作需要的数据。

【本节内容参卡 R5xx Acceleration v1.5.pdf 29-33 页内容 R600 的命令包格式是一样的，此处加以说明】

下面将给出这些命令包的定义，并给出内核使用的实例。

**1）0 型命令包**

0 型命令包用于写连续 N 个寄存器。

包主体部分是依次往这些寄存器写的值。包头各个部分的意义为：

| BITS | Field Name | Description |
|---|---|---|
| 12:0 | BASE_INDEX | 要写的连续寄存器的第一个寄存器地址，最大地址 0x7FFF |
| 14:13 | 保留位 | |
| 15 | ONE_REG_WR | 0 表示将包主体的数据依次写入寄存器中<br>1 表示所有数据写入同一个寄存器 |
| 29:16 | COUNT | 要写的寄存器数目 N-1 |
| 31:30 | TYPE | 0 型包包类型为 0 |

Linux 内核代码./drivers/gpu/drm/radeon/r600.c r600_fence_ring_emit 函数有如下语句：

```
radeon_ring_write(rdev, PACKET0(CP_INT_STATUS, 0));
radeon_ring_write(rdev, RB_INT_STAT);
```

PACKET0 定义如下：

```
#define PACKET0(reg, n) ((PACKET_TYPE0 << 30) |   \        包类型 0 型命令包
                        (((reg) >> 2) & 0xFFFF) |          \        寄存器偏移基地址
                        ((n) & 0x3FFF) << 16)                        要写的寄存器数目
```

所有类型的数据包 31~30bit 为包类型标识符，0 型数据包的类型标识符为 0,其 30bit 为 PACKET_TYPE0（0x0）,29~16bit 为命令写的寄存器数量-1（(n) & 0x3FFF) << 16），上面例子只写一个寄存器，其值为 0。第 14~132bit 为保留位，12~0bit ((reg) >> 2) & 0xFFFF)为第一个寄存器偏移地址，由于使用 0 型包可以访问的所有寄存器都是 4 字节的，寄存器地址都是 4 字节对其的，所以低 2 位为 0。

**2）1 型命令包**

1 型命令包用于写两个寄存器，部分寄存器使命令无法访问到，这个时候需要使用 0 型命令。【什么地方这样说的】

1 型命令包主体为分别向包头定义的两个寄存器写入的值。1 型命令包包头定义如下：

| BITS | Field Name | Description |
|---|---|---|
| 10:0 | REG_INDEX1 | 第一个寄存器的地址 |

| | | |
|---|---|---|
| 21:11 | REG_INDEX2 | 第二个寄存器的地址 |
| 29:22 | RESERVED | 保留位 |
| 31:30 | TYPE | 1 型命令包的类型为 0x1 |

由于 1 型数据包可以用 0 型数据包代替而且 1 型数据包并不能够访问到所有寄存器，在 linux-2.6.32.20 的内核 radeon 驱动中并没有使用 1 型命令包。

**3) 2 型命令包**

2 型命令包是一个空命令包，用于填充对齐命令。2 型命令包没有包主体，其包头最高两位为 0x2，其它位无意义。

2 型命令包不做任何操作，仅用于填充用，填充 ring buffer 的时候有对齐要求，2.6.32.20 内核 radeon 驱动对齐要求是 16 个 dword（16×4 字节），在命令没有 16 dword 对齐的时候，就需要使用 2 型命令包填充。

drivers/gpu/drm/radeon/radeon_ring.c radeon_ring_commit 函数用于通知 GPU 从 ring buffer 中取数据并执行，该函数包含如下代码：

```
count_dw_pad = (rdev->cp.align_mask + 1) - (rdev->cp.wptr & rdev->cp.align_mask);
for (i = 0; i < count_dw_pad; i++) {
    radeon_ring_write(rdev, 2 << 30);
}
```

第一句用于计算对齐命令需要的 dword 数目，后面的 for 循环用于填充 2 型命令。2 型命令仅有个命令头部，并且只有 31~30bit 有效。

**4）3 型命令包**

3 型命令包是最功能最丰富的包，主要执行绘图命令，图形的主要功能都是通过这个包实现的。

3 型命令包主体的内容由包头的 IT_OPCODE 决定。

| BITS | Field Name | Description |
|---|---|---|
| 7:0 | Reserved | 保留位 |
| 15:8 | IT_OPCODE | 操作码 |
| 29:16 | COUNT | 包主体 DWORDS 数目-1 |
| 31:30 | TYPE | 3 型包类型为 0x1 |

3 型命令是主要的命令包，涵盖了寄存器设置/绘图命令/同步等主要操作。

以下是一个使用 3 型命令包设置寄存器的例子，这段代码来自 drivers/gpu/drm/radeon/r600.c 的 r600_ib_test 函数：

```
ib->ptr[0] = PACKET3(PACKET3_SET_CONFIG_REG, 1);
ib->ptr[1] = ((scratch - PACKET3_SET_CONFIG_REG_OFFSET) >> 2);
ib->ptr[2] = 0xDEADBEEF;
ib->ptr[3] = PACKET2(0);
ib->ptr[4] = PACKET2(0);
```

```
.....
ib->ptr[15] = PACKET2(0);
ib->length_dw = 16;
```

这段代码使用了 indirect buffer，但是填充的命令和 ring buffer 中填充的命令是一样的。

```
#define PACKET3(op, n)  ((PACKET_TYPE3 << 30) |                 \
(((op) & 0xFF) << 8) |                  \
((n) & 0x3FFF) << 16)
```

3 型命令头部包含了操作码 op 和数据数目（以 dword 计）。上面例子中 PACKET3(PACKET3_SET_CONFIG_REG, 1) PACKET3_SET_CONFIG_REG 表明这次命令包用于设置寄存器，1 表明后面有 2 个 dword 数据，分别是 (scratch - PACKET3_SET_CONFIG_REG_OFFSET) >> 2（寄存器地址）和 0xDEADBEEF（往寄存器中写的值）。后面是用于对齐用的 2 型包。

上面的代码用于写 scratch 寄存器，scratch 在后面的事件处理机制起着很重要的作用。

下面使用一个更加复杂的命令包来说明 3 型包的使用，下面的这个命令包用于执行一个简单的 2D 操作。r600 显卡是 ATI 推出的第一款使用统一着色器的 GPU，r600 及其以后的显卡不包含单独的 2D 单元，而是使用 3D 部件执行 2D 操作。为了简单起见，这里我们使用 r500 显卡上的填充矩形的命令包。

```
radeon_ring_write(rdev, PACKET3(PACKET3_PAINT_MULTI, 6));
radeon_ring_write(rdev,
RADEON_GMC_DST_PITCH_OFFSET_CNTL |
RADEON_GMC_DST_CLIPPING | // important
RADEON_GMC_BRUSH_SOLID_COLOR |                  // 13 << 4
(RADEON_COLOR_FORMAT_ARGB8888 << 8) |         // << 8
RADEON_GMC_SRC_DATATYPE_COLOR |                 // 4 << 12
RADEON_ROP3_P |                                 // << 16
RADEON_GMC_CLR_CMP_CNTL_DIS);                  // 1 << 28
radeon_ring_write(rdev, ((pitch / 64) << 22) | (fb_offset>>10));
radeon_ring_write(rdev, 0 | (0 << 16));                         // SC_TOP_LEFT
radeon_ring_write(rdev, (fb_w -1) | ((fb_h -1) << 16));        // SC_BOT_RITE
radeon_ring_write(rdev, color); // this is color
radeon_ring_write(rdev, (x << 16) | y);
radeon_ring_write(rdev, (w << 16) | h);
radeon_ring_write(rdev, PACKET0(RADEON_DSTCACHE_CTLSTAT, 0));
radeon_ring_write(rdev, RADEON_RB2D_DC_FLUSH_ALL);
radeon_ring_write(rdev, PACKET0(RADEON_WAIT_UNTIL, 0));
radeon_ring_write(rdev, RADEON_WAIT_2D_IDLECLEAN|
RADEON_WAIT_HOST_IDLECLEAN|
RADEON_WAIT_DMA_GUI_IDLE);
```

3 型包根据他们 IT_OPCODE 的不同，其 IT_BODY 差别很大，如果 IT_OPCODE 的最高位为 1（通常是 2D 绘图命令），那么 PACKET 还需要加入 GUI control。R500 上的 2D

绘图命令有如下格式：
HEADER
GUI_CONTROL
SETUP_BODY
DATA_BLOCK

其中 Header 部分对应 3 型命令包的头，GUI_CONTROL/SETUP_BODY 共同构成了当前绘图环境的配置，这两部分加上 DATA_BLOCK 共同构成了 3 型包的 IT_BODY 部分。上面的代码第一句表明该命令包执行的是矩形绘制（PAINT_MULTI 可以同时绘制多个矩形，这里我们只绘制了一个矩形）。

第二句对应 GUI_CONTROL，GUI_CONTROL 为 32bit，内容为当前绘制环境的标志，下表给出了代码中使用的一些标志（如果是 blit 操作，除了表中的 DSTxx 参数外，还需要设置对应的 SRCxx 参数），关于这些标志更详细的信息可以参考“R5xx Acceleration v1.5.pdf”35-36 页相关内容。

| bit | Field name | Description |
|---|---|---|
| 1 | DST_PITCH_OFFSET | 绘图目标区域的 PITCH 值和该区域在 GPU 虚拟地址空间中的偏移，如果该为被置为 1,则需要在 SETUP_BODY 中指定该参数。 |
| 3 | DST_CLIPPING | 设置绘图区域的裁剪参数，如果该位置为 1,则需要在 SETUP_BODY 中设置 SC_TOP_LEFT 和 SC_BOTTOM_RIGHT 参数。 |
| 7:4 | BRUSH_TYPE | 绘图时使用的 brush 类型，brush 类型需要根据这里给出的类型在 SETUP_BODY 中填 brush 包，不同的 BRUSH_TYPE 对应的 brush 包不同 |
| 11:8 | DST_TYPE | 绘图目标区域的像素类型:<br>1 :- (reserved)<br>2 :- 8 bpp pseudocolor<br>3 :- 16 bpp aRGB 1555<br>4 :- 16 bpp RGB 565<br>5 :- reserved<br>6 :- 32 bpp aRGB 8888<br>7 :- 8 bpp RGB 332<br>8 :- Y8 greyscale<br>9 :- RGB8 greyscale (8 bit intensity, duplicated for all 3 channels. Green channel is used on writes)<br>10 :- (reserved)<br>11 :- YUV 422 packed (VYUY)<br>12 :- YUV 422 packed (YVYU)<br>13 :- (reserved) |
| | | |

在上面示例程序中，以上标志位均被设置，并且 BRUSH_TYPE 被设置为 14，DST_TYPE 设为 32 位真彩色。
根据 GUI_CONTROL 的设置，SETUP_BODY 中需要设置以下参数：
DST_PITCH_OFFSET

SC_TOP_LEFT
SC_BOTTOM_RIGHT
BRUSH_PACKET

上面代码中的 3-6 行即是对这些参数的设置。更多参数的可以参考“R5xx Acceleration v1.5.pdf”37 页的内容。

下面对这些参数进行介绍：

**DST_PITCH_OFFSET**

包括了三部分，31:30 位是和 tiling 相关的标志位，29：22 位是以 64 字节为单位的 pitch 值，21：0 位是 DST 绘图区域（在 xorg 中称为 pixmap）以 1KB 为单位在显存中的偏移，这里提示我们，在分配内存的时候必须是 1K 对齐的，否则在使用的时候会出问题，后面讨论 directfb 的时候将会碰到这个问题。对于这个参数，上面代码填的是

radeon_ring_write(rdev, ((pitch / 64) << 22) | (fb_offset>>10));

**SC_TOP_LEFT 和 SC_BOTTOM_RIGHT**

指定绘图区域的裁剪区域，裁剪区域是个矩形，SC_TOP_LEFT 指定裁剪区域左上方坐标，2D 绘图时以屏幕左上方的点为原点，从左往又为 X 轴正方向，从上往下为 Y 轴正方向。

```
radeon_ring_write(rdev, 0 | (0 << 16));                    // SC_TOP_LEFT
radeon_ring_write(rdev, (fb_w -1) | ((fb_h -1) << 16)); // SC_BOTTOM_RIGHT
```

fb_w 和 fb_h 为当前绘图区域的长和宽，这里我们指定的裁剪区域就是整个绘图区域。

**BRUSH_PACKET**

在 GUI_CONTROL 中指定的 brush type 为 RADEON_GMC_BRUSH_SOLID_COLOR, 关于 brush type 和对应的值请参考“R5xx Acceleration v1.5.pdf”38 页的内容，这里指定的类型为 13，对应的 BRUSH_PACKET 格式为 4 字节，内容为绘图使用的前景色。

radeon_ring_write(rdev, color); // the foreground color

后面两句代码是 DATA_BLOCK 部分，对应绘图使用的参数。

```
radeon_ring_write(rdev, (x << 16) | y);// 矩形左上角坐标
radeon_ring_write(rdev, (w << 16) | h);// 矩形宽和高
```

PAINT_MULTI 命令包的 DATA_BLOCK 部分定义如下表示：

| Ordinal | Field name | Description |
|---|---|---|
| 1 | [DST_X1 \| DST_Y1] | 第 1 个矩形左上角的坐标，高 16 为 X 轴坐标，低 16 位为 Y 轴坐标 |
| 2 | [DST_W1 \| DST_H1] | 第 1 个矩形的宽和高 |
| ... | | |
| 2n-1 | [DST_Xn \| DST_Yn] | 第 n 个矩形左上角的坐标 |
| 2n | [DST_Wn \| DST_Hn] | 第 n 个矩形的宽和高 |

# 2. 5.3 R500 显卡使用命令包实现 2D 加速

下面给出了一些代码，读者根据前面的介绍并参考“R5xx Acceleration v1.5.pdf”是很容易理解的，将这些代码添加到 drivers/gpu/drm/radeon/radeon_test.c 文件中并调用这些函数，在开启 radeon_testing 的情况下就能在启动阶段看到效果。

**画线**

POLYLINE 的 op_code 为 0x95，用于绘制折线,其 3 型命令包数据块部分见表 4-6。

| Ordinal | Field Name | Description |
|---|---|---|
| 1 | [Y0 \| X0] | 折线的起始点坐标，X0 为低 16 位表示 X 轴坐标，Y0 为高 16 位表示 Y 轴坐标。 |
| 2 | [Y1 \| X1] | 第二个点的坐标 |
| ... | | |
| N+1 | [Yn \| Xn] | 第 N 个点的坐标 |

```
1 void r5xx_draw_line_2d(struct radeon_device *rdev, uint64_t fb_location,
2                        int *points, int num ,int color, int fb_w, int fb_h)
3 {
4       int r;
5       struct radeon_fence *fence = NULL;
6       int ndw = 32 + 6 + num;// ?? 32 is enough
7       int i = 0;
8
9       r = radeon_fence_create(rdev, &fence);
10      if (r) {
11            DRM_ERROR("Failed to create fence\n");
12            goto out_cleanup;
13      }
14      r = radeon_ring_lock(rdev, ndw);
15      radeon_ring_write(rdev, PACKET3(PACKET3_POLYLINE, 4 + num));
16      radeon_ring_write(rdev,
17                RADEON_GMC_DST_PITCH_OFFSET_CNTL |
18                RADEON_GMC_DST_CLIPPING | // important
19                RADEON_GMC_BRUSH_SOLID_COLOR |              // 13 << 4
20                (RADEON_COLOR_FORMAT_ARGB8888 << 8) |     //    << 8
21                RADEON_GMC_SRC_DATATYPE_COLOR |          // ??    4 << 12
22                RADEON_ROP3_P |                                  // << 16
23                RADEON_GMC_CLR_CMP_CNTL_DIS);
24      radeon_ring_write(rdev, ((fb_w * 4 / 64) << 22) | (fb_location >>10));
25      radeon_ring_write(rdev, 0 | (0 << 16));
26      radeon_ring_write(rdev, (fb_w -1) | ((fb_h -1) << 16));
27      radeon_ring_write(rdev, color);
28      for( i = 0; i < num; ++i){
```

```
29                  radeon_ring_write(rdev, *points++);
30          }
31          radeon_ring_write(rdev, PACKET0(RADEON_DSTCACHE_CTLSTAT, 0));
32          radeon_ring_write(rdev, RADEON_RB2D_DC_FLUSH_ALL);
33          radeon_ring_write(rdev,
34                    RADEON_WAIT_2D_IDLECLEAN |
35                    RADEON_WAIT_HOST_IDLECLEAN |
36                    RADEON_WAIT_DMA_GUI_IDLE);
37
38          if(fence) {
39                r = radeon_fence_emit(rdev, fence);
40          }
41          radeon_ring_unlock_commit(rdev);
42          r = radeon_fence_wait(fence, false);
43          if (r) {
44                DRM_ERROR("Failed to wait for fence\n");
45                goto out_cleanup;
46          }
47
48 out_cleanup:
49          if(fence) {
50                radeon_fence_unref(&fence);
51          }
52 }
```

注意到这里调用了三个函数处理 fence：
radeon_fence_create，创建一个 fence;
radeon_fence_emit，在提交 ring buffer 之前发送 fence;
radeon_fence_wait，等待 fence。
在中断机制章节中会详细介绍。

**画矩形（省略了出错处理代码）**

使用 PAINT_MULTI 可以绘制矩形，可以在一次命令中绘制多个矩形，其 IT_OPCODE 为 0x9a，其数据主体部分如表 4-7 示。

| Ordinal | Field Name | Description |
|---|---|---|
| 1 | [DST_X1 \| DST_Y1] | 第一个矩形左上角坐标<br>DST_Y1：低 16 位，y 轴坐标，范围是-8192~8191，14 和 15 位是 13 位的拷贝<br>DST_X1：高 16 位，x 轴坐标，范围是-8192~8191，30 和 31 位是 29 位的拷贝 |
| 2 | [DST_W1 \| DST_H1] | 第一个矩形的宽和高，无符号整形值。<br>DST_H1: 低 16 位，高<br>DST_W1: 高 16 位，宽 |
| ... | | |

| | | |
|---|---|---|
| 2n-1 | [DST_Xn \| DST_Yn] | 第 n 个矩形左上角坐标，定义和第一个点相同 |
| 2n | [DST_Wn \| DST_Hn] | 第 n 个矩形的宽和高，定义和第一个点相同 |

表 4-7　PAINT_MULTI

```
 1 void r5xx_draw_rectangl_2d(struct radeon_device *rdev, uint64_t fb_location,
 2                            int x, int y, int w, int h, int color, int fb_w, int fb_h)
 3 {
 4       int r;
 5       int ndw = 32 + 6;// ?? 32 is enough
 6       struct radeon_fence *fence = NULL;
 7
 8       r = radeon_fence_create(rdev, &fence);
.....
13       r = radeon_ring_lock(rdev, ndw);
.....
18       radeon_ring_write(rdev, PACKET3(PACKET3_PAINT_MULTI, 6));
19       radeon_ring_write(rdev,
20                 RADEON_GMC_DST_PITCH_OFFSET_CNTL |
21                 RADEON_GMC_DST_CLIPPING | // important
22                 RADEON_GMC_BRUSH_SOLID_COLOR |              // 13 << 4
23                 (RADEON_COLOR_FORMAT_ARGB8888 << 8) |     //    << 8
24                 RADEON_GMC_SRC_DATATYPE_COLOR |          // ??    4 << 12
25                 RADEON_ROP3_P |                                  // << 16
26                 RADEON_GMC_CLR_CMP_CNTL_DIS);                // 1 << 28
27
28       radeon_ring_write(rdev, ((fb_w * 4 / 64) << 22) | (fb_location >>10));
29       radeon_ring_write(rdev, 0 | (0 << 16));
30       radeon_ring_write(rdev, (fb_w -1) | ((fb_h -1) << 16));
31
32       radeon_ring_write(rdev, color); // this is color
33       radeon_ring_write(rdev, (x << 16) | y);
34       radeon_ring_write(rdev, (w << 16) | h);
35
36       radeon_ring_write(rdev, PACKET0(RADEON_DSTCACHE_CTLSTAT, 0));
37       radeon_ring_write(rdev, RADEON_RB2D_DC_FLUSH_ALL);
38       radeon_ring_write(rdev, PACKET0(RADEON_WAIT_UNTIL, 0));
39       radeon_ring_write(rdev,
40                 RADEON_WAIT_2D_IDLECLEAN |
41                 RADEON_WAIT_HOST_IDLECLEAN |
42                 RADEON_WAIT_DMA_GUI_IDLE);
43
44       r = radeon_fence_emit(rdev, fence);
.....
49       radeon_ring_unlock_commit(rdev);
50       r = radeon_fence_wait(fence, false);
```

```
.....
 55 out_cleanup:
.....
 59 }
```

**Blit**

只给出了发送的命令，其它部分和上面绘制矩形绘制线段相同。

```
  1 void r6xx_blit_2d(struct radeon_device *rdev,
  2                    uint64_t src_ad, uint64_t dst_addr,
  3                    int src_x, int src_y, int dst_x, int dst_y,
  4                    int src_w, int src_h, int fb_w, int fb_h)
  5 {
  6         int r;
  7         int ndw;
  8         struct radeon_fence *fence = NULL;
  9         ndw = 64 + 10;
 10
.....
 21         radeon_ring_write(rdev, PACKET3(PACKET3_BITBLT_MULTI, 8));
 22         radeon_ring_write(rdev,
 23                           RADEON_GMC_SRC_PITCH_OFFSET_CNTL |
 24                           RADEON_GMC_DST_PITCH_OFFSET_CNTL |
 25                           RADEON_GMC_SRC_CLIPPING |
 26                           RADEON_GMC_DST_CLIPPING |
 27                           RADEON_GMC_BRUSH_NONE |
 28                           (RADEON_COLOR_FORMAT_ARGB8888 << 8) |
 29                           RADEON_GMC_SRC_DATATYPE_COLOR |
 30                           RADEON_ROP3_S |
 31                           RADEON_DP_SRC_SOURCE_MEMORY |
 32                           RADEON_GMC_CLR_CMP_CNTL_DIS |
 33                           RADEON_GMC_WR_MSK_DIS);
 34 // SRC_PITCH_OFFSET
 35         radeon_ring_write(rdev, ((fb_w * 4/64) << 22) | (src_addr >> 10));
 36 // DST_PITCH_OFFSET
 37         radeon_ring_write(rdev, ((fb_w * 4/64) << 22) | (dst_addr >> 10));
 38 // SRC_SC_BOT_RITE
 39 //    radeon_ring_write(rdev, (0x1fff) | (0x1fff << 16));
 40         radeon_ring_write(rdev, (fb_w -1) | ((fb_h -1) << 16));
 41 // SC_TOP_LEFT
 42         radeon_ring_write(rdev, 0 | (0 << 16));
 43 // SC_BOT_RITE
 44 //    radeon_ring_write(rdev, (0x1fff) | (0x1fff << 16));
 45         radeon_ring_write(rdev, (fb_w -1) | ((fb_h -1) << 16));
 46 // [SRC_X1 | SRC_Y1]
```

```
 47         radeon_ring_write(rdev, (src_x << 16) | src_y);
 48 // [DST_X1 | DST_Y1]
 49         radeon_ring_write(rdev, (dst_x << 16) | dst_y);
 50 // [SRC_W1 | SRC_H1]
 51         radeon_ring_write(rdev, (src_w << 16) | src_h);
.....
}
```

## 2.6 中断机制

在 CPU 看来，可以将中断分成两种方式：硬件中断和软件中断（说法不准确？？），比如网卡产生的中断称为硬件中断，而如果是软件使用诸如"int 0x10"（X86 平台上）这样的指令也能够产生中断，称为软件中断，硬件中断是异步的，其发生的时机是不可预测的，但是软件中断是同步的，CPU 是“确切”知道其发生的时机的。

同样的，在 GPU 开来，中断也可以分成“硬件中断”和“软件中断”两类，比如热插拔事件或者 vblank 事件都会产生“硬件中断”，这些事件在 GPU 看来是异步的，GPU 不知道这些事情何时发生。

GPU 也可以使用类似 CPU 的 int 指令那样产生中断，考虑这样一种情形：驱动向硬件发送了绘图命令后必须等到硬件执行完了这些命令后才能继续执行后面的代码，否则硬件的上一次命令没有执行完就继续执行下一次命令会导致错误，显卡可以采用软中断机制，在完成绘图命令后执行一个类似“int xx”的命令产生中断，这里 GPU 是“确切”知道中断发生的时机的----即在绘图命令完成的时候。

前面章节反复提到的 fence 就是这种“软件中断”的具体应用。本章除非特别指明，所有出现“软件中断”的地方都是指上文提及的“软件通过对 GPU 进行编程而使 GPU 产生的中断”，需要和 linux 内核中断机制中的“软中断”或者在某些场合下将“信号”当成“软件中断”的情况加以区别。

本章将通过分析 linux 内核 radeon 驱动的中断部分的代码阐明 radeon GPU 的中断机制。这里主要选用 r600 核心的代码，其它核心的中断机制大体上是一样的。

内核 radeon 驱动 R600 显卡中断相关的代码可以分为两个部分，一部分是驱动程序初始化；包括中断控制器初始化，中断状态初始化，中断 ring 环初始化等。另一部分是中断处理函数。

### 2.6.1 中断初始化

## 2.6.2 软中断

在 GPU 命令一节\ref{gpu_command}中我们看到，fence 是按照下面的步骤使用的：

radeon_fence_create->radeon_fence_emit->radeon_fence_wait

radeon 驱动中的 fence 机制用于同步 GPU 和 CPU，Fence 机制的实现依赖 GPU 产生的软中断和 scratch 寄存器。

CP 完成一个绘图操作后执行产生中断的命令，向 CPU 发送一次中断信号，这里的“产生中断的命令”其实就是写 CP_INT_STAT 寄存器。

在 radeon 驱动代码中，完成向 ring buffer 中填充绘图命令后，会调用 radeon_fence_emit 函数（参考 GPU 命令包章节的代码），在 r600 显卡上最终调用 r600_fence_ring_emit 函数，该函数中有如下代码：

```
2327 void r600_fence_ring_emit(struct radeon_device *rdev,
2328                            struct radeon_fence *fence)
......
2347        /* Emit fence sequence & fire IRQ */
2348        radeon_ring_write(rdev, PACKET3(PACKET3_SET_CONFIG_REG, 1));
2349        radeon_ring_write(rdev, ((rdev->fence_drv.scratch_reg -
PACKET3_SET_CONFIG_REG_OFFSET) >> 2));
2350        radeon_ring_write(rdev, fence->seq);
2351        /* CP_INTERRUPT packet 3 no longer exists, use packet 0 */
2352        radeon_ring_write(rdev, PACKET0(CP_INT_STATUS, 0));
2353        radeon_ring_write(rdev, RB_INT_STAT);
```

这里让 GPU 执行的命令(2352-2353 行代码)类似我们在操作系统中让 CPU 执行的“int xx”指令，这两句代码的意思是写 CP_INT_STATUS 寄存器，但是注意到寄存器 CP_INT_STATUS 是一个中断状态寄存器，驱动通过 MMIO 的方式是无法写这个寄存器的，但是如果 CP 写这个寄存器就会产生“软件中断”。[当前观察到的现象是这样的]

通常硬件会有一些寄存器用于表示中断相关信息，在硬件产生中断的时候将相关信息写入寄存器中，驱动读取这些寄存器就能知道和中断相关的具体信息。Radeon GPU 中除了有这类寄存器外（[表明中断类型？？]），scratch 寄存器可以派上用场。

在内核 radeon 驱动中，每一个 fence 都被分配了唯一的 ID 号（seq），在 radeon_fence_emit 中有如下代码：

```
71 int radeon_fence_emit(struct radeon_device *rdev, struct radeon_fence *fence )
......
80        fence->seq = atomic_add_return(1, &rdev->fence_drv.seq);
```

2340-2350 行代码 fence$\rightarrow$seq 的值被写入一个 scratch 寄存器，所以当绘图命令完成中断产生之前 scratch 寄存器就会是被置为这个唯一的 ID 号，读取 scratch 寄存器就能够知道是哪一次绘图命令产生的中断。

Fence 中断处理函数是 radeon_fence_poll_locked。首先读取 fence 编号，知道是那一次 fence 操作产生的中断，当产生中断的 fence 编号是最后一个编号时，需要将最后一个 fence 编号赋值为当前编号，同时更新 fence 定时器。如果不是最后一个 fence 编号产生的中断，就需要判断定时器，然后唤醒 fence 中断队列。

### 2.6.3 中断环

【这里说明后续我们没有使用这里完整的中断机制】

# 第三章 AMD R6xx 显卡图形加速编程

\author{赵自成 tyh}
\date{\today}\maketitle
\graphicspath{{./}{graph/}}

## 3.1 实时计算机图形学基础

### 3.1.1 图形流水线

### 3.1.2 模型视图变换

### 3.1.3 光照模型

### 3.1.4 纹理和着色

### 3.1.5 OpenGL 编程接口和 GLSL 着色语言

【这里介绍简单的 GLSL 程序，后续需要使用】

较早的 GPU 和 OpenGL 基于固定的图形渲染流水线，OpenGL（1.x 版本）的实现是一个“状态机”，用户通过 3D API 提供的函数设置好相应的状态，例如变换矩阵、材质参数、光源参数、纹理混合模式等，然后传入顶点流。图形硬件则利用内置的固定渲染流水线和渲染算法对这些顶点进行几何变换、光照计算、光栅化、纹理混合、雾化操作，最终将处理结果写入帧缓冲区。这种渲染体系限制用户只能使用图形硬件中固化的各种渲染算法。这虽然可以很好的满足对渲染质量要求不高的应用，但难以满足那些需要更高的灵活性和更真实的渲染质量的实时图形应用。

随着 3D 应用的发展，用户已经不再满足于基于顶点的近似 Phong 模型光照计算(这是早期 OpenGL 采用的光照计算模型)和简单的多纹理混合，传统的图形硬件无法满足用户对于更加真实的 3D 渲染的需求，于是有了可编程顶点处理器和可编程片段处理器的出现。

相应的软件也因为这种变化发生了改变，可编程处理器的出现导致了 GLSL 这样的语言的出现。我们使用一个简单的例子；

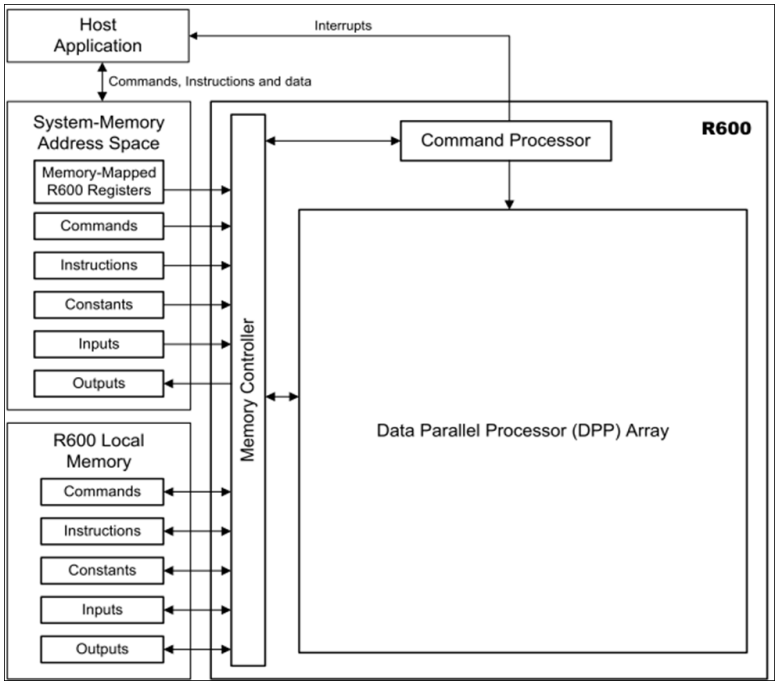
## 3.2 R6xx 显卡核心的 3D 引擎

【前面已经介绍 AMD 显卡】

R6xx 核心是 AMD 一款非常重要的 GPU 核心，这个核心引入了统一处理器架构，因而其寄存器和指令集和以前的 GPU 都完全不同，对其编程也有一定区别。

### 3.2.1 R6xx 显卡核心 3D 引擎组成

图\ref{r600block}显示了 R600 GPU 核心的硬件逻辑图，R600 GPU 包含并行数据处理阵列（DPP array）、命令处理器、内存控制器以及其他逻辑部件，R600 的命令处理器【或者称为微引擎？】读取驱动编写命令并解析命令以，R600 还要将硬件产生的“软中断”发送给 CPU。R600 的内存控制器能够访问 R600 GPU 核上的所有内存（VRAM 内存，或者称本地内存）以及用户配置的系统内存，为了满足 GPU 读写的需要，R600 GPU 还要完成 DMA 控制器的功能。




CPU 上运行的程序不能够直接写 R600 GPU 的本地内存，但是 CPU 程序能够命令 R600 将数据或者程序拷贝到 R600 本地内存或者从本地内存拷贝数据到系统内存上。

完整的能够在 R600 上运行的程序包含两个部分：一部分是在主机（CPU）上运行的程序，一部分是在 R600 处理器上运行的程序，处理图形应用时这部分程序称为 Shader 程序，进行 GPU 通用计算时，这部分程序称为 Kernel 程序。

## 3.2.2 R6xx 显卡核心图形处理流水线

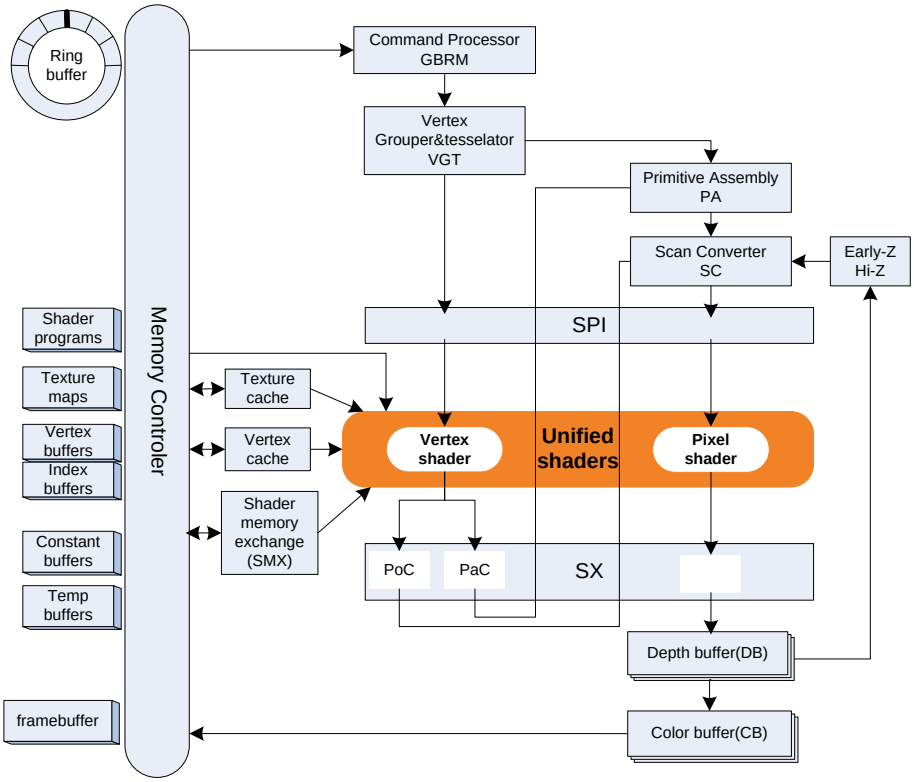
前面章节已经描述了实时计算机图形学里面的图形处理流水线，本节针对 R600 描述在 R600 GPU 上的图形处理过程。图\ref{r600pipeline}显示了 R600 GPU 处理图形数据的过程。

**1） 命令处理**

命令处理器处理环形命令缓冲区和间接缓冲区中的命令流，命令处理器处理这些命令流的时候通常会产生一系列的写寄存器活动（后面会讨论一些寄存器的读写问题，有些寄存器只能使用命令流的形式写，而且无法读）。驱动设置好 index buffer（GTT 内存）并将 index buffer 告知硬件，被命令触发后，“顶点组装和细分器（vertex grouper and tesselator， VGT）”根据 index buffer 的地址将索引数据 DMA 到 SPI （shader pipe interpolator）上，同时将图元的连接信息发送给“图元组装器”（primitive assembly）。

**2） 顶点处理**

所有的着色处理（shader processing，也即 GPU 核上进行的各种运算）都是在统一着色器块（unified shader block）中进行的，同一着色器块包含 Sequencer （SQ）和 Shader pipe（SP）模块，每一个 shader 程序都能够访问一些通用寄存器，这些通用寄存器是在 shader 程序运行之前（由 SPI）动态分配的，SPI 会往这些寄存器中加载合适的参数，这些参数包括顶点数据的基地址。然后 SPI 会为 SQ 启动程序执行过程，shader 程序需要做的第一件事情就是取顶点数据，然后针对这个顶点运行 shader 程序，顶点处理 shader 的输出被放置到“着色器输出缓存”（shader export，SX）中（由于 pixel 处理过程和 vertex 处理过程在同样的硬件上运行，所以 pixel shader 的输出也是放到 SX 中），R600 的顶点处理过程的输出包含两部分：Position Cache 放置顶点的坐标信息，Parameter Cache 放置顶点的其他属性信息。



在没有 Geometry shader 的情况下，顶点数据的处理按照下面流程进行：

- VGT \footnote{本节内容翻译自 R6xx R7xx Acceleration.pdf}一个指向 index buffer 的指针（如果是立即模式，index buffer 是硬件临时指定的？），VGT 遍历所有的索引并一个个发送到 SPI 里面
- SPI 将其输入缓存中的索引数据组成称为 wavefront 的向量（wavefront 最多 64 个顶点）
- 当 wavefront 准备好后，SPI 根据驱动提供的大小（驱动向 SQ_PGM_RESOURCE_VS 寄存器写入的）分配 GPR（通用寄存器，所有的通用寄存器都是 32*4=128bit 的，可以存放一个四维的浮点数向量）和线程空间\footnote{前面需要解释线程空间}（thread space），然后这些索引被放入到 GPR 中（GPR 的 id 是由谁指定的），shader core 被通知一个新的 wavefront 准备好了；
- Shader core 对 Wavefront 上的每个顶点运行顶点处理程序
  - 顶点处理器根据 GRP 中的索引取出顶点数据（使用 fetch 指令或者单独的 fetch 程序）
  - 顶点数据被取到 GPR 中
  - Shader 程序的其他部分继续运行
  - Shader 程序在 SX 的 position Cache 中分配空间，并将顶点的坐标信息（XYZW）输出到这片空间
  - Shader 程序在 SX 的 Parameter cache 中分配空间，将顶点的其他属性信息（颜色，纹理）发送到这片空间，程序退出
- SPI 被告知一个 wavefront 的所有顶点处理完毕，SPI 释放掉 GPR

配置好渲染状态后，除了第 4）步外，上面的过程对用户（驱动程序）来说是透明的，第 4）步是按照用户编写的 shader 的程序的流程执行的。

一个 vertex shader 程序至少要包含 a)、b)、d)三个过程，vertex shader 首先通过索引取到顶点数据，然后对这些顶点数据进行运算，运算后的结果分别放入 position cache 和 parameter cache 中供后续的部件使用。c）步骤根据 shader 程序的功能，可能有也可能没有。

**3） 片段处理**

顶点处理完成后，顶点数据被发送到“图元组装器”（PA）中进行图元组装（注意到过程 1 已经将顶点连接信息发送给了图元组装器），【同时还有一个 setup 的过程，查阅相关资料理解 setup 主要做什么工作？？】，PA 的输出被发送给“扫描转换器”（scan convert，SC）进行扫描转换（光栅化过程，差值计算），SC 会检查“深度缓存”（Depth buffer，DB）确定片段的可用性，这个检查过程会进行 Early Z、Re-Z 和 HiZ 处理（可以这么理解，SC 检查 z buffer，如果片段的深度值比 depth buffer 中的值还要大，则该片段被遮挡，在没有开启 blending 的情况下，这个片段可以扔掉，后续就不用处理了，如果开启了 blending 的，则还要进行后续处理）。光栅化出来的片段被发送给 SPI，再进入到 shader core 里面进行最后的片段处理。片段处理器会进行取纹理、ALU 计算以及内存读写操作。最后，shader memory exchange 作为 shader core 的 scratch 内存读写 cache【？？我们暂时没有用到？？】，完成后，片段的几何信息【在屏幕坐标系中的坐标？？】和颜色信息通过 SX （vertex shader 也会输出到 SX 中）发送到 DB 和 CB 中进行最后的处理。

顶点数据经过顶点处理流程后，进入光栅化阶段，由 Scan Converter 对各属性数据进行差值，形成片段数据，片段数据流经片段处理阶段着色，形成可选的像素（或者称为片段 fragment）。

由于 R600 是统一处理器架构，顶点处理和片段处理都在同样的硬件上进行，因此片段

处理的过程和顶点处理的过程是类似的。

R600 片段处理阶段（包括光栅化阶段）按照如下流程进行：

- “图元组装器”（PA）从 SX 的 position buffer 中读出顶点的坐标信息，从 VGT 中读取出顶点连接信息，有了这两方面的信息后，就可以组装出图元；
- 组装出来的图元被发送到 SC 进行初步的扫描转换；（初步的扫描转换做了些什么工作，将大图元分成小的块 tile）
- 初步扫描转换出来的块（tiles，有合适的翻译？？）被送到 SPI 中进行最后的插值
  - SPI 分配 GPR 和线程空间（thread space）（根据驱动指定的大小）；
  - SC 和 SPI 从 SX 的 parameter cache 中读取顶点的属性数据；
  - SPI 针对顶点属性计算出插值出来的每个像素的属性
  - 将插值后的属性加载到 GPR
- Shader core 被告知一个 pixel wavefront 到来，准备执行 pixel shader
- Shader core 对 wavefront 里面的每一个片段运行 pixel shader，pixel shader 的结束部分包含将片段属性（颜色）输出到 SX 的指令
- SPI 被告知 wavefront 里面的所有片段处理完成，SPI 释放 GPR 和线程空间

Pixel shader 程序将计算的结果输出到 SX 后将会被送往指定的 Render Target，可以一次最多配置 8 个 Render Target。

**4） 最后的渲染**

Pixel shader 的输出被放到 DB 和 CB 中进行最后的处理（对应图 1、图 2 和图 3 中的 raseter operation 和 merging 过程），这部分处理包括 alpha 测试、深度测试和最后的融合（blending）。

# 3.3 R600 显卡核心的 3D 引擎编程

## 3.3.1 R600 3D 引擎基本状态编程

从图\label{r600pipeline}可以看到，3D 图形处理流水线需要流经多个硬件单元才能得到最后的渲染结果，所有的硬件单元必须被正确编程，才能得到正确的结果。

本节使用 exa 驱动的代码详细说明如何对 3D 引擎进行编程。

EXA 驱动挂接在 xorg 可加载驱动中，我们采用的代码版本是 xf86-video-ati-20110727【查阅该版本号】，

在启用 KMS 的情况下，Xorg 的 R600 驱动按照如下的调用路径加载 EXA 驱动：RADEONScreenInit_ KMS ---> RADEONAccelInit ---> R600DrawInit。

R600DrawInit 函数中先为 radeon_accel_state 结构体中的 ExaDriverPtr 结构体的加速回调函数赋值，这些函数就是实现 EXA 加速所必须的函数，这些 2D 加速包括填充矩形（Solid）、块拷贝（Copy）以及混合（Composite）。

调用 exaDriverInit 将 exa 驱动注册到 Xorg 系统中。

然后是调用 R600AllocShaders 为 Shader 分配显存，Shaders 是 GPU 上运行的程序必须放在 VRAM 内存上：

```
accel_state->shaders_bo = radeon_bo_open(info->bufmgr, 0, size, 0,
                                         RADEON_GEM_DOMAIN_VRAM, 0);
```

加载 Shader 程序，将为 Shader 分配的 VRAM 映射到用户空间，然后以直接写内存的方式将 Shader 程序加载到 VRAM 上。然后取消映射：

```
ret = radeon_bo_map(accel_state->shaders_bo, 1);
......
accel_state->solid_vs_offset = 0;
R600_solid_vs(ChipSet, shader + accel_state->solid_vs_offset / 4);
......
radeon_bo_unmap(accel_state->shaders_bo);
```

在所有的 EXA 加速过程中，每一个加速过程都注册了 3 个函数，分别是 R600PrepareXXX、R600XXX 和 R600DoneXXX，这 3 个函数分别进行 3D 引擎初始化、输入数据触发 3D 引擎以及 3D 引擎的清理工作。

接下来通过 Copy 加速过程描述 3D 引擎的编程过程。

首先是 3D 引擎的初始化 R600PrepareCopy，Copy 操作涉及源和目的。获取源和目的的图像的 pitch 值，这里的 pitch 是图像一行的像素数目（包括可能的空白对齐区域，注意有些时候 pitch 值指以字节计一行图像占用的内存大小）：

```
dst_obj.pitch = exaGetPixmapPitch(pDst) / (pDst->drawable.bitsPerPixel / 8);
src_obj.pitch = exaGetPixmapPitch(pSrc) / (pSrc->drawable.bitsPerPixel / 8);
```

获取代表图像显存的显存对象：

```
src_obj.bo = radeon_get_pixmap_bo(pSrc);
dst_obj.bo = radeon_get_pixmap_bo(pDst);
```

获取源和目的图像的格式信息：

```
src_obj.width = pSrc->drawable.width;
src_obj.height = pSrc->drawable.height;
src_obj.bpp = pSrc->drawable.bitsPerPixel;
src_obj.domain = RADEON_GEM_DOMAIN_VRAM | RADEON_GEM_DOMAIN_GTT;

dst_obj.width = pDst->drawable.width;
dst_obj.height = pDst->drawable.height;
dst_obj.bpp = pDst->drawable.bitsPerPixel;
dst_obj.domain = RADEON_GEM_DOMAIN_VRAM;
```

注意到上面代码的 domain 域，源可能来自 VRAM 或者 GTT，但是 dst 是显示区域，一定只可能是 VRAM。后面还有一段代码用于计算图片的高度和大小。

调用 R600SetAccelState 函数，该函数只是做了一些软件上的赋值操作，注意 copy_vs_offset 和 copy_ps_offset 都是 GPU 虚拟地址。

```
if (!R600SetAccelState(pScrn,
                       &src_obj,
                       NULL,
                       &dst_obj,
                       accel_state->copy_vs_offset, accel_state->copy_ps_offset,
```

```
                rop, planemask))
    return FALSE;
```

主要的 3D 引擎初始化工作在 R600DoPrepareCopy 函数调用里面。

```
R600DoPrepareCopy(pScrn);
```

这部分内容我们将同时结合《Radeon R6xx/R7xx 3D Register Reference Guide》《Radeon R6xx/R7xx Acceleration》两份文档进行说明。《Radeon R6xx/R7xx Acceleration》的“3D Engine Programming”部分较为完整的描述了需要编程的部分。

radeon_vbo_check 函数获取可用的 vertex buffer，如果没有可用的 vertex buffer，将会重新分配一个。

```
radeon_vbo_check(pScrn, &accel_state->vbo, 16);
```

radeon_cp_start 用于清空当前命令流中的命令【需要在合适的地方解释命令流 CS command stream】：

```
radeon_cp_start(pScrn)：
if (info->cs) {
    if (CS_FULL(info->cs)) {
        radeon_cs_flush_indirect(pScrn);
    }
     accel_state->ib_reset_op = info->cs->cdw;
}
```

r600_set_default_state 设置 3D 引擎的初始化状态，对 R600 硬件的配置主要在这个地方。

```
void r600_set_default_state(ScrnInfoPtr pScrn, drmBufPtr ib):

    if (accel_state->XInited3D)
        return;
    r600_start_3d(pScrn, accel_state->ib);

    sq_conf.ps_prio = 0;
    sq_conf.vs_prio = 1;
    sq_conf.gs_prio = 2;
    sq_conf.es_prio = 3;

    switch (info->ChipFamily) {
        case CHIP_FAMILY_R600:
        sq_conf.num_ps_gprs = 192;
        sq_conf.num_vs_gprs = 56;
        sq_conf.num_temp_gprs = 4;
        sq_conf.num_gs_gprs = 0;
        sq_conf.num_es_gprs = 0;
        sq_conf.num_ps_threads = 136;
        sq_conf.num_vs_threads = 48;
        sq_conf.num_gs_threads = 4;
```

```
            sq_conf.num_es_threads = 4;
            sq_conf.num_ps_stack_entries = 128;
            sq_conf.num_vs_stack_entries = 128;
            sq_conf.num_gs_stack_entries = 0;
            sq_conf.num_es_stack_entries = 0;
        break;
        r600_sq_setup(pScrn, ib, &sq_conf);
```

accel_state 中的 XInited3D 域记录了 3D 引擎是否已经初始化过，如果 EXA 驱动已经初始化过 3D 引擎，则无需再度初始化。调用 r600_start_3d 函数

启动 3D 引擎的，然后调用 r600_sq_setup 对 3D 引擎的 Sequencer 进行配置。

r600_start_3d 函数启动 3D 引擎的命令环，在 GPU 核心小于 RV770 的情况下，执行如下命令

```
  void r600_start_3d(ScrnInfoPtr pScrn, drmBufPtr ib)：

      BEGIN_BATCH(5);
      PACK3(ib, IT_START_3D_CMDBUF, 1);
      E32(ib, 0);

      PACK3(ib, IT_CONTEXT_CONTROL, 2);
      E32(ib, 0x80000000);
      E32(ib, 0x80000000);
      END_BATCH();
```

上面的代码如果转换成相应的内核代码会是这样的:

```
  radeon_ring_lock(rdev, 5);
  radeon_ring_write(rdev, CP_PACKET3(START_3D_CMDBUF, 1));
  radeon_ring_write(rdev, 0x0);
  radeon_ring_write(rdev, CP_PACKET3(IT_CONTEXT_CONTROL, 2));
  radeon_ring_write(rdev, 0x80000000);
  radeon_ring_write(rdev, 0x80000000);
```

手册上并没有给出 START_3D_CMDBUF 和 IT_CONTEXT_CONTROL 的具体含义，【猜测这里 START_3D_CMDBUF 是将关闭掉 Command Buffer。】

接下来是 Sequencer 的配置，Sequencer 可以认为是【？？r600 isa 手册有一个简单的描述，用此描述】GPU 的一个控制单元【对应图\ref{r600pipeline}中的 SPI】，Sequence 控制 Shader 程序的运行。Sequencer 相关的寄存器参见《Radeon R6xx/R7xx 3D Register Reference Guide》第三节“General Shader Registers”。

```
static void r600_sq_setup(ScrnInfoPtr pScrn, drmBufPtr ib, sq_config_t *sq_conf):
......
      BEGIN_BATCH(8);
      PACK0(ib, SQ_CONFIG, 6);
      E32(ib, sq_config);
      E32(ib, sq_gpr_resource_mgmt_1);
      E32(ib, sq_gpr_resource_mgmt_2);
```

```
    E32(ib, sq_thread_resource_mgmt);
    E32(ib, sq_stack_resource_mgmt_1);
    E32(ib, sq_stack_resource_mgmt_2);
    END_BATCH();
```

注意到这里写寄存器的方式，注意到 PACK0 的定义：

```
#define PACK0(ib, reg, num)                                                    \
do {                                                                           \
    if ((reg) >= SET_CONFIG_REG_offset && (reg) < SET_CONFIG_REG_end) {     \
    PACK3((ib), IT_SET_CONFIG_REG, (num) + 1);                  \
    E32((ib), ((reg) - SET_CONFIG_REG_offset) >> 2);         \
    } else if ((reg) >= SET_CONTEXT_REG_offset && (reg) < SET_CONTEXT_REG_end) { \
    PACK3((ib), IT_SET_CONTEXT_REG, (num) + 1);                 \
    E32((ib), ((reg) - SET_CONTEXT_REG_offset) >> 2);           \
    } else if ((reg) >= SET_ALU_CONST_offset && (reg) < SET_ALU_CONST_end) { \
    PACK3((ib), IT_SET_ALU_CONST, (num) + 1);               \
    E32((ib), ((reg) - SET_ALU_CONST_offset) >> 2);                 \
    }
    ......
    else {                                          \
    E32((ib), CP_PACKET0 ((reg), (num) - 1));               \
    }                                               \
} while (0)
```

地址小于 SET_CONFIG_REG_offset（0x8000）的寄存器必须使用 3 型命令包写，这些寄存器只有 GPU 自己能够访问，因此访问这些寄存器必须使用 GPU 命令让 GPU 自己，不能使用通常直接写寄存器的方式，我们上面这段代码改写成：

```
  radeon_ring_lock(rdev, 8);
  radeon_ring_write(rdev, CP_PACKET3(IT_SET_CONFIG_REG, 7));
  radeon_ring_write(rdev, (SQ_CONFIG - IT_SET_CONFIG_REG) >> 2);
  radeon_ring_write(rdev, sq_config);
  radeon_ring_write(rdev, sq_gpr_resource_mgmt_1);
  radeon_ring_write(rdev, sq_gpr_resource_mgmt_2);
  radeon-ring_write(rdev, sq_thread_resource_mgmt);
  radeon_ring_write(rdev, sq_stack_resource_mgmt_1);
  radeon_ring_write(rdev, sq_stack_resource_mgmt_2);
  radeon_ring_unlock(rdev);
```

这是一个 3 型命令包，包头 IT_SET_CONFIG_REG 表明这个命令用于“Write Register Data to a Location on Chip”，即写片上的寄存器，这个命令包的第第二个 DWORD 是寄存器相对于 IT_SET_CONFIG_REG（0x8000）的偏移（以 DWORD 计），命令包后续的内容是往 offset 开始的连续几个寄存器里面写入的值，这里总共写了 6 个寄存器。这里配置的是地址为 0x8c00（SQ_CONFIG）开始的几个寄存器。在寄存器手册搜索地址为 0x8c00、0x8c04、0x8c08、0x8c0c、0x8c10、0x8c12 的这几个寄存器可以查看详细说明，这里对 SQ 设置的参数同时需要考虑硬件的能力和软件的需求。对 SQ 的配置包括通用寄存器数目的配置、线程数目的配置以及堆栈资源的分配。

目前为止，碰到的寄存器有三类，使用的时候要注意区分：

- CPU 可访问的寄存器
  这类寄存器可以映射到进程地址空间，驱动程序可以直接访问
- GPU 内部寄存器
  这类寄存器不能被驱动程序直接访问，只能由 GPU 的命令处理器访问，驱动程序只能通过写 GPU 命令的方式写这些寄存器
- Shader 通用寄存器
  这类寄存器用于执行 Shader 程序的时候使用，类似 x86 上的 eax、ebx 这样的寄存器.

```
sq_conf.num_ps_gprs = 192;
sq_conf.num_vs_gprs = 56;
sq_conf.num_temp_gprs = 4;
```

以上代码为每个【确认 per simd 是什么意思，似乎不是这个意思，所有运行的 thread 实例共能够使用这么多通用寄存器】PS 程序分配的寄存器数目为 192 个，为 VS 程序分配的通用寄存器数目为 56。

```
sq_conf.num_ps_threads = 136;
sq_conf.num_vs_threads = 48;
```

同时【？？】运行 ps 程序的线程为 136 个，同时运行 vs 程序的线程数目为 48 个。

EXA 程序中不需要使用 GS 和 ES，Shader 也不需要使用堆栈，对于 GS、ES 和堆栈的配置跳过。

```
PACK0(ib, SQ_VTX_BASE_VTX_LOC, 2);
E32(ib, 0);
E32(ib, 0);
```

上面代码向 SQ_VTX_START_INST_LOC 和 SQ_VTX_BASE_VTX_LOC 两个寄器中写入 0，这两个寄存器的含义涉及到 GPU 取顶点数据
的寻址过程，GPU 取顶点数据的时候按照下面的方式寻址：

fetch_addr = (index + index_offset) * stride + base + offset

其中 index 是通用寄存器中的值，为顶点的索引值，后面讨论 GPU 指令的时候还会涉及到；index_offset 中的值即来自 SQ_VTX_START_INST_LOC 和 SQ_VTX_BASE_VTX_LOC 两个寄存器，可以根据 Shader 程序中的相应位选择使用哪个寄存器的值，stride 是设置顶点资源的时候设置的，为一个顶点的全部数据占的字节数。base 也是设置顶点资源的时候配置的，为顶点内存（vertex buffer）的地址（GPU 虚拟地址），offset 指明取顶点的哪部分数据，后面讨论 GPU 指令集的时候还会讨论。

然后是对 GS 和 ES 之间的 ring 的配置【查阅？？】，exa 驱动并没有使用，略过。

接下来是对 Depth test、Stencil test 和 alpha test 的配置：

```
EREG(ib, DB_DEPTH_CONTROL, 0);
PACK0(ib, DB_RENDER_CONTROL, 2);
E32(ib, STENCIL_COMPRESS_DISABLE_bit | DEPTH_COMPRESS_DISABLE_bit);
E32(ib, FORCE_SHADER_Z_ORDER_bit);

EREG(ib, DB_ALPHA_TO_MASK,((2 << ALPHA_TO_MASK_OFFSET0_shift) |
```

```
                                (2 << ALPHA_TO_MASK_OFFSET1_shift) |
                                (2 << ALPHA_TO_MASK_OFFSET2_shift) |
                                (2 << ALPHA_TO_MASK_OFFSET3_shift)));
    EREG(ib, DB_SHADER_CONTROL, ((1 << Z_ORDER_shift) | /*
EARLY_Z_THEN_LATE_Z */
                    DUAL_EXPORT_ENABLE_bit)); /* Only useful if no depth export */

    PACK0(ib, DB_STENCIL_CLEAR, 2);
    E32(ib, 0); // DB_STENCIL_CLEAR
    E32(ib, 0); // DB_DEPTH_CLEAR

    PACK0(ib, DB_STENCILREFMASK, 3);
    E32(ib, 0); // DB_STENCILREFMASK
    E32(ib, 0); // DB_STENCILREFMASK_BF
    E32(ib, 0); // SX_ALPHA_REF
```

往寄存器 DB_DEPTH_CONTROL 写入 0 将关闭 depth test 和 stencil test。\\

【DB_RENDER_CONTROL？？】，关闭掉压缩功能？？（具体查阅寄存器手册）。\\

【DB_SHADER_CONTROL？？】\\

【】

对 viewport、windows、clip、scissor 等的配置【查阅？？】。

对光栅化部件的编程【查阅相关资料详细描述】:

```
    PACK0(ib, PA_SC_LINE_CNTL, 9);
    E32(ib, 0); // PA_SC_LINE_CNTL
    E32(ib, 0); // PA_SC_AA_CONFIG
    E32(ib, ((2 << PA_SU_VTX_CNTL__ROUND_MODE_shift) | PIX_CENTER_bit | //
PA_SU_VTX_CNTL
            (5 << QUANT_MODE_shift))); /* Round to Even, fixed point 1/256 */
    EFLOAT(ib, 1.0);                // PA_CL_GB_VERT_CLIP_ADJ
    EFLOAT(ib, 1.0);                // PA_CL_GB_VERT_DISC_ADJ
    EFLOAT(ib, 1.0);                // PA_CL_GB_HORZ_CLIP_ADJ
    EFLOAT(ib, 1.0);                // PA_CL_GB_HORZ_DISC_ADJ
    E32(ib, 0);                     // PA_SC_AA_SAMPLE_LOCS_MCTX
    E32(ib, 0);                     // PA_SC_AA_SAMPLE_LOCS_8S_WD1_M

    EREG(ib, PA_SC_AA_MASK, 0xFFFFFFFF);
```

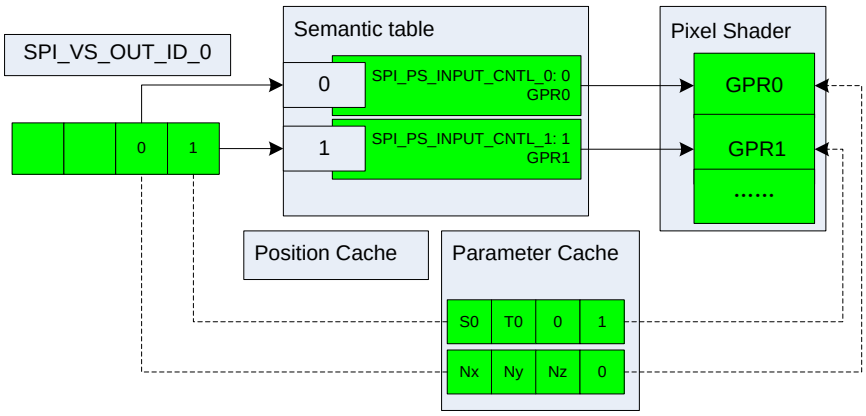
对 semantic table 的编程：

```
    /* default Interpolator setup */
    EREG(ib, SPI_VS_OUT_ID_0, ((0 << SEMANTIC_0_shift) |
                        (1 << SEMANTIC_1_shift)));
   PACK0(ib, SPI_PS_INPUT_CNTL_0 + (0 << 2), 2);
    /* SPI_PS_INPUT_CNTL_0 maps to GPR[0] - load with semantic id 0 */
```

```
E32(ib, ((0       << SEMANTIC_shift)  |
        (0x01 << DEFAULT_VAL_shift)|
        SEL_CENTROID_bit));
/* SPI_PS_INPUT_CNTL_1 maps to GPR[1] - load with semantic id 1 */
E32(ib, ((1       << SEMANTIC_shift)  |
        (0x01 << DEFAULT_VAL_shift)|
        SEL_CENTROID_bit));
```

图形处理流水线上，每一个过程都是相对独立的，这里有一些过程是由应用程序（或者驱动程序）决定的，比如输入数据的格式，顶点数据根据具体应用的不同会有不同，比如最简单的应用只包含顶点坐标，复杂一点的可能还包含顶点颜色、纹理坐标、法向量数据等，再比如可编程处理器的 Shader 程序是应用程序或者驱动程序确定的，输出数据的格式是由应用程序或者驱动程序决定的，必须通过某种方式将这些"可编程"部分流出的数据告诉下一个阶段的部件。

先看输入的顶点数据和顶点处理器流出的数据。在早期的固定流水线的显卡上，应用程序或者驱动必须告诉显卡输入数据具体包含哪些分量，比如 R100/R200 显卡是通过 VAP_VTX_FMT_0/1 这两个寄存器配置顶点数据的格式的，这两个寄存器都有很多位，每一位都对应某一属性数据，如果这一位被置 1，则表明输入数据包含这一属性，而且输入数据的各个属性之间必须按照规定的先后顺序排列。到 R300 以后，可编程处理器被引入，此时可编程处理器是不需要知道数据属性的，因为顶点处理程序是由程序员或者编译器编写的，程序员或者编译器是知道数据的格式的，此时 VAP_VTX_FMT_0/1 这两个寄存器就被废除了，取而代之的是一个表明顶点数据数据大小（占用的字节数）的寄存器，取顶点数据的硬件只需根据每个顶点数据的大小从顶点数据缓冲区中取出一个顶点的全部数据然后跳到下一个顶点取下一个顶点的数据就可以了，顶点数据送到 Shader 运行，编写 Shader 的程序员或者编译器知道数据的格式，然后对数据进行处理。在没有 Fetch shader 的情况下，R600 对顶点数据格式的处理和 R300 是类似的，后面设置顶点数据资源的时候将会看到某个寄存器的 Stride 位记录的就是顶点数据的大小。如果有 Fetch shader，这必须配置一个 semantic table，EXA 驱动里面没有使用 Fetch Shader，但是后面的 Vertex shader 和光栅化部件以及 Pixel Shader 有类似的 semantic table，上面一段程序的 semantic table 就是针对这个设计的。




图\ref{semantic}显示了一个完整的 semantic 的数据转化过程，SPI_VS_OUT_ID 寄存器共有 10 个，每个寄存器有是个域，每个域里代表 semantic table 中的一项，根据 SPI_VS_OUT_CONFIG 寄存器的配置，SPI_VS_OUT_ID 系列寄存器的每一个域代表一个向量或者向量的一个分量，这里配置的是代表一个向量，EXA 驱动用到的顶点属性比较少，因此需要的 semantic table 也不大。SPI_VS_OUT_ID_0 的第 0 个域代表 Parameter Cache 中的第 0 个向量，第 1 个域代表 Parameter Cache 中的第 1 个域，一次类推，为了能够更清

晰的说明问题，图中显示的内容和 EXA 驱动稍微有点差异，SPI_VS_OUT_ID_0 寄存器的第 0 个域填充的是 1，表明需要查询 semantic table 的第 1 项，semantic table 是由 SPI_PS_INPUT_CNTL 一系列共 32 个寄存器组成的，每个寄存器代表 semantic table 的一项，SPI_PS_INPUT_CNTL_1 寄存器的 SEMANTIC 位的内容为 1，对应 Pixel Shader 的 GPR1，因此光栅化部件处理完，将像素的纹理信息放到 Pixel Shader 的 GPR1 中，Pixel Shader 程序从 GPR1 中可以取到纹理数据，同样的，法向量信息被放置到 GPR0 中。【Position 数据也可能进入到 Pixel shader 中？semantic table 还需要包含 Position 的相关条项？？查阅相关寄存器】

和 semantic table 有关还有一处代码，

```
r600_set_spi(pScrn, accel_state->ib, (1 - 1), 1);
void
r600_set_spi(ScrnInfoPtr pScrn, drmBufPtr ib, int vs_export_count, int num_interp)
{
    RADEONInfoPtr info = RADEONPTR(pScrn);

    BEGIN_BATCH(8);
    /* Interpolator setup */
    EREG(ib, SPI_VS_OUT_CONFIG, (vs_export_count << VS_EXPORT_COUNT_shift));
    PACK0(ib, SPI_PS_IN_CONTROL_0, 3);
    E32(ib, (num_interp << NUM_INTERP_shift));
    E32(ib, 0);
    E32(ib, 0);
    END_BATCH();
}
```

首先是写 SPI_VS_OUT_CONFIG 寄存器，这里 VS_PER_COMPONENT 位为 0，表明一个向量在 semantic 中占一个条目，VS_EXPORT_COUNT 为 0，表明 Vertex Shader 输出了 1 个向量，在 BLT 过程中，就是输出了一个纹理坐标这一个向量。

然后是写 SPI_PS_IN_CONTROL_0、SPI_PS_IN_CONTROL_1 和 SPI_INTERP_CONTROL_0 四个寄存器，SPI_PS_IN_CONTROL_0 的 NUM_INTERP 表明有一个属性需要进行差值，【手册上说需要包括位置信息，那么这里应该有位置坐标和纹理坐标两个属性需要差值？？】这个寄存器里面还有一个比较重要的位 POSITION_ENA，表明位置信息是否也加载到 PS 中，【如果这一位开启的话，是否 SPI_PS_IN_CONTROL_0 的 NUM_INTERP 位和 semantic table 都需要修改？？】

接下来是对雾效果的处理，EXA 驱动中并没有使用雾效果，因此这里相关配置全部填充的是 0。

然后是对 Fetch Shader 的的配置：

```
r600_fs_setup(pScrn, ib, &fs_conf, RADEON_GEM_DOMAIN_VRAM);
```

前面有代码将 fs_conf 变量置为 0：

```
memset(&fs_conf, 0, sizeof(shader_config_t));
```

调用 r600_fs_setup 实际上是将 FS 的配置清空为 0，EXA 驱动中不使用 Fetch Shader，Fetch Shader 的配置和 Vertex Shader、Pixel Shader 等的配置基本相同，后面使用的时候还会

提到 Vertex Shader。

然后是 VGT 的配置，VGT 是最早接触顶点数据的部件，VGT 中的信息在图元组装的时候也会被使用，VGT 包含了两方面的内容：一部分是 Vertex Group，另外一部分是 Tesselator，关于 VGT 的配置的细节，这里不进行论述，后面 DirectFB 驱动的代码基本上按照默认参数设置，读者可以参考寄存器手册。其中 PA_SU_POINT_SIZE 和 PA_SU_LINE_CNTL 可能会用到，分别用于控制点的大小和线的粗细。

至此 r600_set_default_state 函数已经全部完成。

```
r600_set_generic_scissor(pScrn, accel_state->ib, 0, 0, accel_state->dst_obj.width,
    accel_state->dst_obj.height);
r600_set_screen_scissor(pScrn, accel_state->ib, 0, 0, accel_state->dst_obj.width,
    accel_state->dst_obj.height);
r600_set_window_scissor(pScrn, accel_state->ib, 0, 0, accel_state->dst_obj.width,
    accel_state->dst_obj.height);
```

以上代码用于设置裁剪范围，【这里为什么会有 generic screen 和 window 的区别】

## 3.3.2 Shader 程序的配置

接下来进入的是 r600_vs_setup 函数对 Vertex Shader 的配置：

```
  void
r600_vs_setup(ScrnInfoPtr pScrn, drmBufPtr ib, shader_config_t *vs_conf, uint32_t domain)
{
    RADEONInfoPtr info = RADEONPTR(pScrn);
    uint32_t sq_pgm_resources;

    sq_pgm_resources = ((vs_conf->num_gprs << NUM_GPRS_shift) |
            (vs_conf->stack_size << STACK_SIZE_shift));

    if (vs_conf->dx10_clamp)
    sq_pgm_resources |= SQ_PGM_RESOURCES_VS__DX10_CLAMP_bit;
    if (vs_conf->fetch_cache_lines)
    sq_pgm_resources |= (vs_conf->fetch_cache_lines << FETCH_CACHE_LINES_shift);
    if (vs_conf->uncached_first_inst)
    sq_pgm_resources |= UNCACHED_FIRST_INST_bit;

    /* flush SQ cache */
    r600_cp_set_surface_sync(pScrn, ib, SH_ACTION_ENA_bit,
                    vs_conf->shader_size, vs_conf->shader_addr,
                    vs_conf->bo, domain, 0);
```

```
    BEGIN_BATCH(3 + 2);
    EREG(ib, SQ_PGM_START_VS, vs_conf->shader_addr >> 8);
    RELOC_BATCH(vs_conf->bo, domain, 0);
    END_BATCH();

    BEGIN_BATCH(6);
    EREG(ib, SQ_PGM_RESOURCES_VS, sq_pgm_resources);
    EREG(ib, SQ_PGM_CF_OFFSET_VS, 0);
    END_BATCH();
}
```

Shader 程序配置的相关寄存器的详细信息见寄存器手册的“Shader Program Setup Registers”章节，所有 Shader 程序的配置都是三个寄存器：

- SQ_PGM_START_xx
  Shader 程序的起始地址
- SQ_PGM_RESOURCES_xx
  为 Shader 程序分配的资源情况
- SQ_PGM_CF_OFFSET_xx
  Shader 程序第一条 CF 指令相对 Shader 程序起始地址的偏移

上面代码是对 Vertex Shader 程序的配置，SQ_PGM_START_VS 寄存器中写入的是 Vertex Shader 程序的 GPU 地址，SQ_PGM_CF_OFFSET_VS 写入的是 0，Shader 程序的第一条 CF 指令在程序起始地址处。SQ_PGM_RESOURCES_xx 寄存器中包含有 NUM_GPRS【和 SQ 配置处的 GPR 有什么关系】，必须配置足够的寄存器数目才能使程序运行，如果这里配置的寄存器数目 NUM_GPRS 为 5，那么 Shader 程序中可访问的寄存器为 GPR[0...4]，如果访问 GPR5，程序会运行出错。

```
r600_ps_setup(pScrn, accel_state->ib, &ps_conf, RADEON_GEM_DOMAIN_VRAM);
```

PS 程序的配置和 VS 程序的配置类似，不再赘述。

## 3.3.3 资源的配置

这里的资源包括两部分：顶点资源和纹理资源。【还有常量资源等等，是否需要说明】

**1) 纹理资源的配置**

先看纹理资源的设置。函数调用进入了 r600_set_tex_resource 函数，纹理配置相关的寄存器在寄存器手册“Shader Vertex Resource Constants”章节。总共 7 个寄存器，

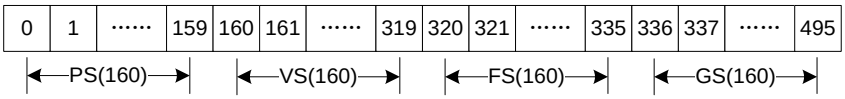
- SQ_VTX_CONSTANT_WORD0
  配置纹理图的维度、PITCH 值以及纹理图宽度，
- SQ_TEX_RESOURCE_WORD1
  记录了纹理图的高度、深度和像素格式，
- SQ_TEX_RESOURCE_WORD2
  记录纹理图在显存中的地址（GPU 地址），
- SQ_TEX_RESOURCE_WORD3
  记录 mipmap 在显存中的地址，EXA 驱动进行 BLIT 操作的时候，纹理图是以原图的大

小贴上去的，实际上不使用 mipmap，这个寄存器中写入的值和 SQ_TEX_RESOURCE_WORD2 写入的值是一样的，

- SQ_TEX_RESOURCE_WORD4
  配置纹理图像素格式
- SQ_TEX_RESOURCE_WORD5
  【功能暂不清楚】
- SQ_TEX_RESOURCE_WORD6
  【功能暂不清楚】

这里需要注意到纹理资源的 id 问题，配置以上寄存器的时候并没有指定纹理 id，寄存器地址实际上已经表明了 id 了，在寄存器手册上上面几个寄存器都有一个后缀_0，这个 0 就已经表明了纹理的 id 号了，从 r600_set_tex_resource 函数的代码能够清楚看到这一点：
PACK0(ib, SQ_TEX_RESOURCE + tex_res->id * SQ_TEX_RESOURCE_offset, 7);

这里的寄存器地址是在 SQ_VTX_CONSTANT_WORD0_0 的基础上加了一个偏移，关于 RESOURCE_ID 的问题，后面设置顶点资源的时候也有，【是否应该这样理解：vs 中有多个自己的顶点资源，这些资源都被编号，ps 也有自己的顶点资源（ps 要顶点资源干嘛？还是因为为了和 texture 统一才这样做，texture 在 vs 和 ps 中都是可用的，可是 fs 呢，fs 为什么会有自己的资源），也被编号，并且这些资源可以共享，比如一个 tex 资源在 vs 中使用，也在 ps 中使用，则在 ps 中一个编号为 0 的 tex 资源，在 vs 中有一个编号为 160 的 tex 资源，都是指向同一个 tex 资源？？】




SQ_TEX_RESOURCE = SQ_TEX_RESOURCE_WORD0_0,/* 160 PS, 160 VS, 16 FS, 160 GS */
SQ_TEX_RESOURCE_ps_num = 160,
SQ_TEX_RESOURCE_vs_num = 160,
SQ_TEX_RESOURCE_fs_num = 16,
SQ_TEX_RESOURCE_gs_num = 160,
SQ_TEX_RESOURCE_all_num = 496,
SQ_TEX_RESOURCE_offset = 28,
SQ_TEX_RESOURCE_ps = 0,
SQ_TEX_RESOURCE_vs = SQ_TEX_RESOURCE_ps + SQ_TEX_RESOURCE_ps_num,
SQ_TEX_RESOURCE_fs = SQ_TEX_RESOURCE_vs + SQ_TEX_RESOURCE_vs_num,
SQ_TEX_RESOURCE_gs = SQ_TEX_RESOURCE_fs + SQ_TEX_RESOURCE_fs_num,

注意到几个宏定义，SQ_TEX_RESOURCE_WORD0_0 为这一系列寄存器的起始地址，每一个资源对应 7 个寄存器，故偏移量为 7*4 = 28（SQ_TEX_RESOURCE_offset）。
PS 的第一块纹理的 id 号为 0，写入寄存器地址为 0x38000~0x38018，PS 第二块纹理的 id 号为 1，写入寄存器地址为 0x38000+28 ~ 0x38018+28，VS 的第一块纹理的 id 号为 160，写入的寄存器地址为 0x38000+28*160 ~ 0x38018+28*160。

接下来是设置采样器，可以参考寄存器手册上的介绍，这里不详细描述。

后面还有设置顶点资源的代码 test_r600_set_vtx_resource。

**2) 顶点资源和索引的配置**

在 R5xx 显卡中，输入数据的属性可以是按照向量的方式写入到 VAP_PORT_DATA 寄存器然后到达 IVM 中，顶点位置和其他属性数据的排列方式可以是任意的，驱动程序只需要通过 VAP_VTX_SIZE 寄存器告知硬件顶点数据的大小。也可以使用索引方式，将顶点数据放置到 VRAM 的某一个位置，然后通过 VAP_PORT_IDX 寄存器将索引写入，到显卡上。

在 R6xx 显卡上只支持索引模式，用户必须先将顶点数据放置到 vertex buffer 中，vertex buffer 是驱动从 GTT 上分配出来的用于放置顶点数据的一片内存，然后将索引写入显卡，因此 vertex shader 的输入都是索引，因此在 vertex shader 中必须调用事先编程好的 fetch shader 或者直接使用取顶点数据指令将顶点位置和顶点属性取出来【从主存 DMA 到显卡上】【细节还需要详细理解】，然后才能对这些数据进行处理【R3xx 索引模式下是似乎不需要这样做，似乎无论使用索引模式还是立即模式，硬件都会将数据准备好，因此 vertex shader 看到的就是顶点数据，确认？？】。

和 R5xx 显卡一样，R6xx 显卡输入的顶点属性数据顺序也是可以由驱动自己定义的，在使用 fetch shader 的情况下，必须编程一个 semantic table 告知硬件顶点属性的排列顺序【semantic table 如何编程】，如果直接是在 vertex shader 程序中取顶点数据，则编写 vertex shader 的用户或者编译器知道数据的格式，不需要 semantic table【参考 Radeon R6xx/R7xx Acceleration 2.5 Shader Linkage】。

**EXA 驱动中关于 vertex buffer 的分配**

R600DoPrepareCopy 中有使用 vertex buffer 的分配过程，在尚未分配 vertex buffer 的时候，vertex buffer 按照如下过程分配：

R600DoPrepareCopy → radeon_vbo_check → r600_vb_no_space → radeon_vbo_get → radeon_vbo_get_bo

最后的分配过程是这样的

```
dma_bo->bo = radeon_bo_open(info->bufmgr, 0, VBO_SIZE,
                            0, RADEON_GEM_DOMAIN_GTT, 0);
```

这里使用的是 GTT 内存。

R600Copy 函数调用了 R600AppendCopyVertex，R600AppendCopyVertex 函数中表明了 vertex buffer 的使用过程：

```
static void
R600AppendCopyVertex(ScrnInfoPtr pScrn, int srcX, int srcY, int dstX, int dstY, int w, int h)
{
    float *vb;

    vb = radeon_vbo_space(pScrn, 16);

    vb[0] = (float)dstX;
    vb[1] = (float)dstY;
    vb[2] = (float)srcX;
    vb[3] = (float)srcY;

    vb[4] = (float)dstX;
```

```
    vb[5] = (float)(dstY + h);
    vb[6] = (float)srcX;
    vb[7] = (float)(srcY + h);

    vb[8] = (float)(dstX + w);
    vb[9] = (float)(dstY + h);
    vb[10] = (float)(srcX + w);
    vb[11] = (float)(srcY + h);

    radeon_vbo_commit(pScrn);
}
```

radeon_vbo_space(pScrn, 16)从 vertex buffer 中获取 16【应该是出于对齐的原因】个 dword 大的内存并获取到指向申请到的这块内存的指针（具体过程大致是如果 vertex buffer 不够用了，就重新申请，如果还没有做映射，则做一次映射，然后根据整个 bo 的起始映射地址加上偏移得到可以使用的 vertex buffer 的地址）即上面代码中的 vb 指针，然后填上顶点数据。这里是要做 blit 操作，源内存中的内容被看做是纹理，在目的地址处绘制矩形，然后将纹理应用到这个矩形上【这里是绘制矩形，但是只是用了三个顶点的坐标，似乎是自动计算了第四个坐标，为了避免误解后续我们写 dfb 驱动的时候使用改成绘制 QUAD_LIST 这样就可以显示的指定完整的四个坐标】。

radeon_vbo_commit(pScrn)计算了一下当前剩余的空间，并将指针移动到剩余的空间上（有点类似前面的 radeon_ring_lock 和 radeon_ring_commit 的过程）。

R600DoCopy 函数按照如下的调用顺序完成硬件顶点资源的设置：

R600DoCopy → r600_finish_op → set_vtx_resource，

```
    /* Vertex buffer setup */
    accel_state->vb_size = accel_state->vb_offset - accel_state->vb_start_op;
    vtx_res.id                  = SQ_VTX_RESOURCE_vs;
    vtx_res.vtx_size_dw         = vtx_size / 4;
    vtx_res.vtx_num_entries = accel_state->vb_size / 4;
    vtx_res.mem_req_size        = 1;
    vtx_res.vb_addr             = accel_state->vb_mc_addr + accel_state->vb_start_op;
    vtx_res.bo                  = accel_state->vb_bo;
```

这里的 id 号和纹理资源的 id 号类似。EXA 驱动里面只有一个顶点资源，被编号为 SQ_VTX_RESOURCE_vs（160），顶点大小为 16（x、y 坐标加上 s、t 纹理坐标共 4x4=16 字节）。vtx_res.vb_addr 为 vertex buffer 的地址。【vtx_res.vtx_num_entries 似乎有点问题】

将上面这些信息设置到硬件里：

```
set_vtx_resource(pScrn, accel_state->ib, &vtx_res, RADEON_GEM_DOMAIN_GTT);
void set_vtx_resource(ScrnInfoPtr pScrn, drmBufPtr ib, vtx_resource_t *res, uint32_t domain)
{
    RADEONInfoPtr info = RADEONPTR(pScrn);
    uint32_t sq_vtx_constant_word2;

    sq_vtx_constant_word2 = ((((res->vb_addr) >> 32) & BASE_ADDRESS_HI_mask) |
                             ((res->vtx_size_dw << 2) <<
```

```
SQ_VTX_CONSTANT_WORD2_0__STRIDE_shift) |
                (res->format                                    <<
SQ_VTX_CONSTANT_WORD2_0__DATA_FORMAT_shift) |
                (res->num_format_all                            <<
SQ_VTX_CONSTANT_WORD2_0__NUM_FORMAT_ALL_shift) |
                (res->endian                                    <<
SQ_VTX_CONSTANT_WORD2_0__ENDIAN_SWAP_shift));
    if (res->clamp_x)
        sq_vtx_constant_word2 |= SQ_VTX_CONSTANT_WORD2_0__CLAMP_X_bit;

    if (res->format_comp_all)
        sq_vtx_constant_word2                                   |=
SQ_VTX_CONSTANT_WORD2_0__FORMAT_COMP_ALL_bit;

    if (res->srf_mode_all)
        sq_vtx_constant_word2                                   |=
SQ_VTX_CONSTANT_WORD2_0__SRF_MODE_ALL_bit;

    BEGIN_BATCH(9 + 2);
    PACK0(ib, SQ_VTX_RESOURCE + res->id * SQ_VTX_RESOURCE_offset, 7);
    E32(ib, res->vb_addr & 0xffffffff);                 // 0: BASE_ADDRESS
    E32(ib, (res->vtx_num_entries << 2) - 1);           // 1: SIZE
    E32(ib, sq_vtx_constant_word2); // 2: BASE_HI, STRIDE, CLAMP, FORMAT, ENDIAN
    E32(ib, res->mem_req_size << MEM_REQUEST_SIZE_shift);   // 3:
MEM_REQUEST_SIZE ?!?
    E32(ib, 0);                                 // 4: n/a
    E32(ib, 0);                                 // 5: n/a
    E32(ib,                     SQ_TEX_VTX_VALID_BUFFER         <<
SQ_VTX_CONSTANT_WORD6_0__TYPE_shift);       // 6: TYPE
    RELOC_BATCH(res->bo, domain, 0);
    END_BATCH();
}
```

以上写的是 Radeon R6xx/R7xx 3D Register Reference Guide 第 6 部分 Shader Vertex Resource Constants 的几个寄存器

SQ_VTX_CONSTANT_WORD0_0 0x38000
SQ_VTX_CONSTANT_WORD1_0 0x38004
SQ_VTX_CONSTANT_WORD2_0 0x38008
SQ_VTX_CONSTANT_WORD3_0 0x3800c
SQ_VTX_CONSTANT_WORD6_0 0x38018

往第 0 个寄存器中写入的是 vertex buffer 地址（GPU 虚拟地址）的高 32 位，往第 1 个寄存器里面写入的全部顶点占用的内存大小（以 DWORD 计），往第 2 个寄存器中写入的是 vertex buffer 地址的低 32 位、stride 值、endian 等信息。往第 3 个寄存器写入的值暂不清楚其含义。【手册并未给出】。中间还有两个寄存器未定义，编程的时候需要往这两个寄存器里面写 0。

**3）常量资源的配置**

Copy 过程中没有使用常量资源，但是 Solid 过程使用了常量，将颜色作为常量传递给 Shader。

颜色应该是作为顶点属性放置在顶点输入数据里面的，但是由于 Solid 操作整个矩形内填充的是同一种颜色，因此可以将颜色作为常量传递给 Shader。

关于如何设置常量，以及常量如何被访问，读者可以参考 EXA 驱动中的 r600_set_alu_consts 函数和 Solid 的 Shader 程序代码。

## 3.3.4 输出

r600_set_render_target(pScrn, accel_state->ib, &cb_conf, accel_state->dst_o

render target 称为渲染目标，在早期的显卡上，渲染的结果只能输出在显示区域上，后来的显卡引入了 Multiple render target 的概念，多目标渲染，即渲染结果可以同时输出到显存的多个位置，在 R600 上最多可以配置 8 个 render target，也就是说一次渲染的结果可以输出到显存上的最多 8 个不同的位置。

这部相关的寄存器在寄存器手册"Color Buffer Registers"章节，这里的很多寄存器都有八套，目前我们只使用一个 render target，只是用于显示输出。

首先关心的是输出地址，EXA 驱动向 CB_COLOR0_BASE 寄存器中写入的就是 framebuffer 的地址，CB_COLOR0_SIZE 寄存器中存放的是 Render target 的大小，这里的大小是按照 TILE 来计算的，TILE 大小为 8*8，CB_COLOR0_SIZE 寄存器 PITCH_TILE_MAX 域放置的是一行的 TILE 数目减 1（width/8），SLICE_TILE_MAX 域存放的是整个 render target 的 tile 数目减 1（width*height/(8*8) -1）。

CB_COLOR_INFO 寄存器配置 render target 的格式，目前比较重要的位有：
ENDIAN 指定输出是否进行大小端转换，ARRAY_MODE 指定 render target 的 tiling 格式，FORMAT 指定输出像素的格式。

总体上看，从图形处理流水线的源头开始，需要准备好 vertex buffer 和 index，在立即模式下，index 可以直接编程在命令中，通过配置寄存器告诉 GPU vertex buffer 的位置，在启动 GPU 流水线之前，还需要将 vertex shader 程序和 pixel shader 程序加载到 vram 中，并通过配置寄存器告示 GPU shader 程序的位置，在 vertex shader 和 pixel shader 之间还需要配置光栅化部件以及 semantic table，在 pixel shader 的输出端配置 render target，这样整个 GPU 的编程就算完成了。

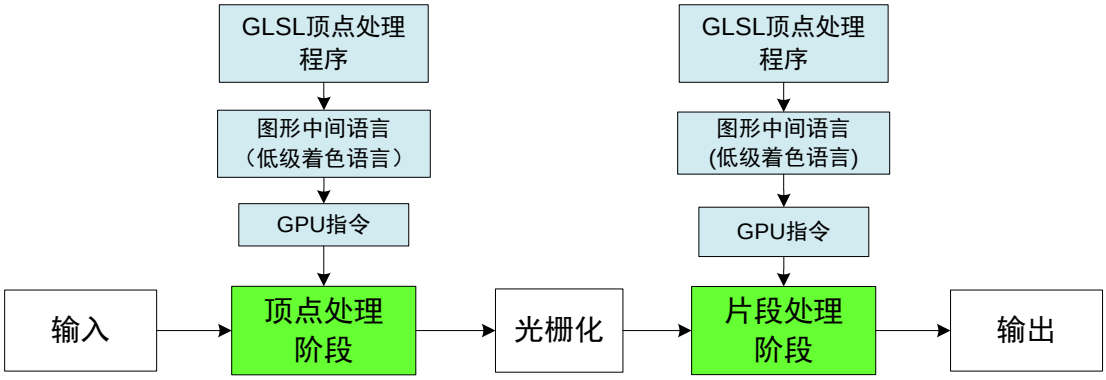
GPU 流水线启动之后，就按照“R6xx 显卡核心图形处理流水线”章节描述的过程处理数据，并将最后的结果输出到 render target 中。

## 3.4 中间语言

GLSL 语言（以下高级着色语言就是指 GLSL）是语法类似 C 的高级语言，在 GLSL 规

范中，GLSL 语言被先翻译成教低级的类汇编语言（ILSL，当前 mesa 中是 tgsi？？，以下简称低级着色语言）【查阅此处以确定该信息，tgsi 是否和该处处于同样的地位，如果是，后续介绍了 arbvp arbfp 之后请添加材料说明 tgsi】，然后被翻译成硬件特定的指令集。OpenGL 体系管理委员会于 2002 年 6 月和 2002 年 9 月分别通过了两个官方扩展：ARB_VERTEX_PROGRAM 与 ARB_FRAGMENT_PROGRAM 来统一对低级着色语言的支持，GLSL 语言被编译成针对这两个扩展的低级着色语言（因此这两个扩展可以看成是 GLSL 运行的虚拟机），显卡厂商的驱动将低级着色语言翻译成 GPU 指令。这两个扩展的 1.0 版本分别是 arbvpl0 和 arbfpl0，这两个扩展的 2.0 版本分别是 arbvp20 和 arbfp2。

目前，在 Mesa 上，GLSL 首先被编译器翻译成 tgsi 中间语言，然后显卡特定的驱动将这些 tgsi 语言的代码编译成 GPU 指令，这个过程如图\ref{mesapipeline}示。




【添加相关内容】

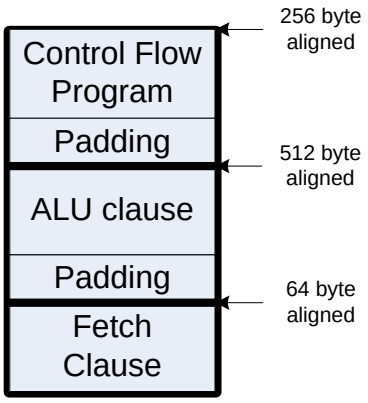
## 3.5 R6xx 显卡核心指令集（多添加内容还是精简？？）}

在主机（CPU）上运行的程序都是按照代码的先后顺序顺序执行的，比如下面这样的代码：

\footnote{此处添加代码}\begin{verbatim}
  这里插入一段 X86 汇编指令
\end{verbatim}

在 CPU 上运行的程序，所有的访存指令和运算指令按照代码的堆叠顺序执行（不考虑指令集并行），如果有跳转指令则跳转到相应位置。

但是 GPU 主要用于运算，没有包含复杂的控制程序，因此 R600 Shader 程序和 CPU 程序比较显得比较简陋，必须有专门的代码指示程序的执行顺序，并且指示程序运行顺序的指令（Control Flow 指令，后面称为 CF 指令）、运算指令（后面称 ALU 指令）和访存指令（在 R600 GPU 中称为 Fetch 指令）必须按照类别存放，同一种类型的指令放在一起，不同类型的指令按照某种顺序存放，同一类型的指令（不包括 CF 指令）构成一个 Clause。图\ref{shaderlayout}显示了 R600 GPU Shader 程序在内存中存放的形式。




R600 的指令包含 Control Flow（后面简称 CF）指令，ALU（运算）指令，Vertex Fecth（取顶点）指令和 Texture Fetch（取纹理）指令。指令的格式称为 Microde Format。

每一个 Shader 程序（Pixel Shader 或者 Vertex Shader）包含两部分，一部分是 CF 指令，另一部分是 Clause。 这些 Clause 由 CF 指令初始化（或者不恰当的理解成 Clause 由 CF 指令调用）。

R600 的每一条指令的格式（为了保持和手册上术语的一致，后面将使用 Microcode Format 这个词）都包含了 2 个或者 4 个 DWORD（CF 和 ALU 为 2 个 DWORD，Vertex Fetch 和 Texture Fetch 为 4 个 DWORD），这些 Microcode Format 可以在《R600 Family Instruction Set Architecture》手册上查阅到。

## 3.5.1 Vertex Shader 程序实例

下面使用一个具体的实例来说明，下面的程序来自我们的 R600 Direct FB 驱动 BLT 过程的 Vertex Shader 程序（请参考后续章节），按照图\ref{shaderlayout} 的要求，这段程序被分成了两部分，第一部分是 CF 指令，共四条指令，指令（0）~指令（3）（指令（3）为空指令，用于对齐），第二部分为取顶点指令，共两条指令，指令（4）~ 指令（5），分别用于取顶点位置坐标和纹理坐标。

```
        CF_DWORD0(ADDR(4)),                                    //  （0）  addr 0
        CF_DWORD1(POP_COUNT(0),
                CF_CONST(0),
                COND(SQ_CF_COND_ACTIVE),
                I_COUNT(2),                                    // 2 instructions
                CALL_COUNT(0), END_OF_PROGRAM(0), VALID_PIXEL_MODE(0),
                CF_INST(SQ_CF_INST_VTX),                WHOLE_QUAD_MODE(0),
BARRIER(1)),

        CF_ALLOC_IMP_EXP_DWORD0(ARRAY_BASE(CF_POS0),        //  （1）  addr 1
                TYPE(SQ_EXPORT_POS), RW_GPR(1), RW_REL(ABSOLUTE),
                INDEX_GPR(0), ELEM_SIZE(0)),
        CF_ALLOC_IMP_EXP_DWORD1_SWIZ(SRC_SEL_X(SQ_SEL_X),
SRC_SEL_Y(SQ_SEL_Y),
                SRC_SEL_Z(SQ_SEL_Z),                    SRC_SEL_W(SQ_SEL_W),
R6xx_ELEM_LOOP(0),
```

```
                BURST_COUNT(0), END_OF_PROGRAM(0), VALID_PIXEL_MODE(0),
                CF_INST(SQ_CF_INST_EXPORT_DONE),     WHOLE_QUAD_MODE(0),
BARRIER(1)),

        CF_ALLOC_IMP_EXP_DWORD0(ARRAY_BASE(0),                  //（2） addr 2
                TYPE(SQ_EXPORT_PARAM), RW_GPR(0), RW_REL(ABSOLUTE),
                INDEX_GPR(0), ELEM_SIZE(0)),
        CF_ALLOC_IMP_EXP_DWORD1_SWIZ(SRC_SEL_X(SQ_SEL_X),
                SRC_SEL_Y(SQ_SEL_Y), SRC_SEL_Z(SQ_SEL_Z),
                SRC_SEL_W(SQ_SEL_W), R6xx_ELEM_LOOP(0),
                BURST_COUNT(0), END_OF_PROGRAM(1), VALID_PIXEL_MODE(0),
                CF_INST(SQ_CF_INST_EXPORT_DONE), WHOLE_QUAD_MODE(0),
                BARRIER(0)),

        VTX_DWORD_PAD,                                          //（3） addr 3
        VTX_DWORD_PAD,

        VTX_DWORD0(VTX_INST(SQ_VTX_INST_FETCH),                 //（4） addr 4
                FETCH_TYPE(SQ_VTX_FETCH_VERTEX_DATA),
FETCH_WHOLE_QUAD(0),
                BUFFER_ID(0), SRC_GPR(0), SRC_REL(ABSOLUTE),
                SRC_SEL_X(SQ_SEL_X), MEGA_FETCH_COUNT(16)),
        VTX_DWORD1_GPR(DST_GPR(1), DST_REL(0), DST_SEL_X(SQ_SEL_X),
                DST_SEL_Y(SQ_SEL_Y), DST_SEL_Z(SQ_SEL_0),
                DST_SEL_W(SQ_SEL_1), USE_CONST_FIELDS(0),
                DATA_FORMAT(FMT_32_32_FLOAT),
                NUM_FORMAT_ALL(SQ_NUM_FORMAT_SCALED),
                FORMAT_COMP_ALL(SQ_FORMAT_COMP_SIGNED),
                SRF_MODE_ALL(SRF_MODE_ZERO_CLAMP_MINUS_ONE)),
        VTX_DWORD2(OFFSET(0),
#ifdef __BIG_ENDIAN
                ENDIAN_SWAP(SQ_ENDIAN_8IN32),
#else
                ENDIAN_SWAP(SQ_ENDIAN_NONE),
#endif
                CONST_BUF_NO_STRIDE(0),
                MEGA_FETCH(1)),
        VTX_DWORD_PAD,

        VTX_DWORD0(VTX_INST(SQ_VTX_INST_FETCH),                 //（5）addr 6
                FETCH_TYPE(SQ_VTX_FETCH_VERTEX_DATA),
FETCH_WHOLE_QUAD(0),
                BUFFER_ID(0), SRC_GPR(0), SRC_REL(ABSOLUTE),
                SRC_SEL_X(SQ_SEL_X), MEGA_FETCH_COUNT(8)),
```

```
        VTX_DWORD1_GPR(DST_GPR(0), DST_REL(0),
                DST_SEL_X(SQ_SEL_X), DST_SEL_Y(SQ_SEL_Y),
                DST_SEL_Z(SQ_SEL_0), DST_SEL_W(SQ_SEL_1),
                USE_CONST_FIELDS(0), DATA_FORMAT(FMT_32_32_FLOAT),
                NUM_FORMAT_ALL(SQ_NUM_FORMAT_SCALED),
                FORMAT_COMP_ALL(SQ_FORMAT_COMP_SIGNED),
                SRF_MODE_ALL(SRF_MODE_ZERO_CLAMP_MINUS_ONE)),
        VTX_DWORD2(OFFSET(8),
#ifdef __BIG_ENDIAN
                ENDIAN_SWAP(SQ_ENDIAN_8IN32),
#else
                ENDIAN_SWAP(SQ_ENDIAN_NONE),
#endif
                CONST_BUF_NO_STRIDE(0),
                MEGA_FETCH(0)),
        VTX_DWORD_PAD,
```

上面程序的运行过程如图\ref{blitshader}示，图中标记了两个线程，此刻两个线程正在对两个顶点数据进行处理。下面结合这张图详细描述程序运行的过程。

- CF 指令（0）

指令（0）的 ADDR 位指示程序从地址 4 处的指令（指令（4））开始运行，I_COUNT 位指示共执行 2 条指令（指令（4）和指令（5）），执行完后回到指令（0），指令（0）的 END_OF_PROGRAM 位表明程序还没有结束，继续执行指令（1）。

- Vertex Fetch Clause

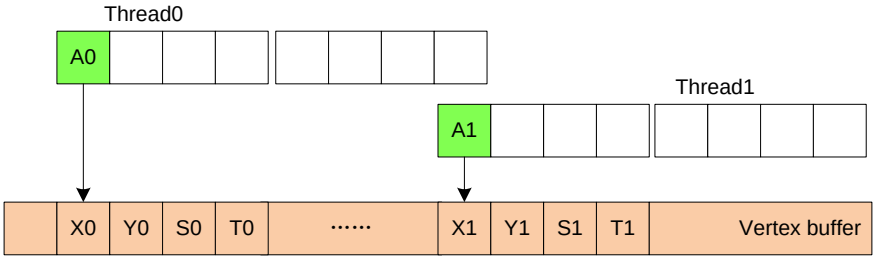
CF 的第一条指令指明程序会从第指令（4）处开始执行，指令（4）和指令（5）构成一个 Vertex Fetch Clause，两条指令一起完成取顶点数据，这里是 BLT 操作，顶点数据包括顶点的位置坐标和纹理坐标。由于是 2D 操作，因此这里的坐标的有效分量只有两个。\\ 指令（4）的 VTX_INST 位表明改指令是一条取数据的指令，从 BUFFER_ID 为 0 的内存处取顶点（FETCH_TYPE）数据，SRC_GPR 为索引号所在的寄存器位置，一次取的数据量为 16 字节（一个四元向量的大小），取出来的数据被放置在编号为 1 的寄存器中（DST_GPR），DST_SEL_X(SQ_SEL_X)表明取出来的向量的 X 分量放置到目的寄存器第一个 DWORD 位置处，Y 分量按照相同的方法放置（DST_SEL_Y(SQ_SEL_Y)），目的寄存器的第三个 DWORD 处被置为 0，第四个 DWORD 处被置为 1（可以使用 0,1 或者 0.5【确认】）。指令（5）和指令（4）类似，由于所有顶点属性数据已经取完，因此原来存在于 GPR0 的地址不再需要，可以覆盖掉。

当然其实我们使用一条指令就可以将 X、Y、S 和 T 四个分量全部取出来，这里为了演示寄存器的使用所以取了两次。

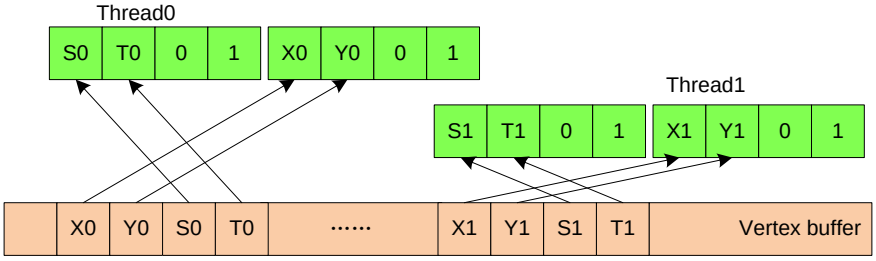
- CF 指令（1）和指令（2）

这是两条输出指令，指令所做的工作如图\ref{blitshader3}示，指令（1）用于输出顶点的位置坐标（TYPE(SQ_EXPORT_POS)），这条指令从 GPR1（RW_GPR(1)）中读取数据，将数据输出到 Position Buffer 0 中（ARRAY_BASE(CF_POS0)）。输出的时候还有一个 Swizzle 操作，这条指令的 Swizzle 操作没有变换向量各个分量。【此处介绍 Swizzle 操作】

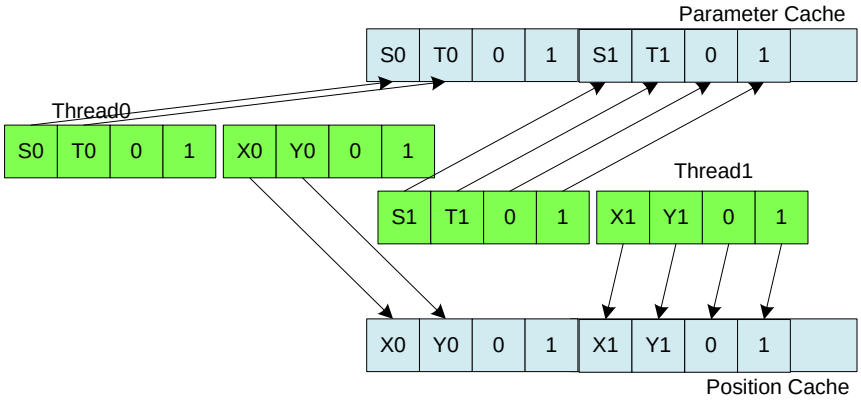
指令（1）的 END_OF_PROGRAM 标志表明程序还没有结束，因此继续执行指令（2），指令（2）的 END_OF_PROGRAM 位表明程序至此结束（后续如果还写有指令将不会执行）。




a)




b)




在程序运行之前，顶点数据的地址已经放在通用寄存器的 GPR0 中（输入数据的寄存器配置【查阅此处】），见图\ref{blitshader1}。【这里结合前面的公式说明取顶点数据的过程】

# 3. 5.2 Pixel Shader 程序的实例

```
CF_DWORD0(ADDR(2)),          // 指令（0）    addr 0
CF_DWORD1(POP_COUNT(0),
        CF_CONST(0),
        COND(SQ_CF_COND_ACTIVE),
        I_COUNT(1),
        CALL_COUNT(0),
        END_OF_PROGRAM(0),
        VALID_PIXEL_MODE(0),
        CF_INST(SQ_CF_INST_TEX),
        WHOLE_QUAD_MODE(0),
```

```
        BARRIER(1)),

    CF_ALLOC_IMP_EXP_DWORD0(ARRAY_BASE(CF_PIXEL_MRT0),// 指令（1）
addr 1
        TYPE(SQ_EXPORT_PIXEL),
        RW_GPR(0x6),            // i changed it to 0x6
        RW_REL(ABSOLUTE),
        INDEX_GPR(0),
        ELEM_SIZE(1)),
    CF_ALLOC_IMP_EXP_DWORD1_SWIZ(SRC_SEL_X(SQ_SEL_X),
        SRC_SEL_Y(SQ_SEL_Y),
        SRC_SEL_Z(SQ_SEL_Z),
        SRC_SEL_W(SQ_SEL_W),
        R6xx_ELEM_LOOP(0),
        BURST_COUNT(1),
        END_OF_PROGRAM(1),
        VALID_PIXEL_MODE(0),
        CF_INST(SQ_CF_INST_EXPORT_DONE),
        WHOLE_QUAD_MODE(0),
        BARRIER(1)),

    TEX_DWORD0(TEX_INST(SQ_TEX_INST_SAMPLE),  // 指令（2） addr 2
        BC_FRAC_MODE(0),
        FETCH_WHOLE_QUAD(0),
        RESOURCE_ID(0),
        SRC_GPR(0),
        SRC_REL(ABSOLUTE),
        R7xx_ALT_CONST(0)),
    TEX_DWORD1(DST_GPR(0x6),  //  i changed it to 0x6
        DST_REL(ABSOLUTE),
        DST_SEL_X(SQ_SEL_X), /* R */
        DST_SEL_Y(SQ_SEL_Y), /* G */
        DST_SEL_Z(SQ_SEL_Z), /* B */
        DST_SEL_W(SQ_SEL_W), /* A */
        LOD_BIAS(0),
        COORD_TYPE_X(TEX_UNNORMALIZED),
        COORD_TYPE_Y(TEX_UNNORMALIZED),
        COORD_TYPE_Z(TEX_UNNORMALIZED),
        COORD_TYPE_W(TEX_UNNORMALIZED)),
    TEX_DWORD2(OFFSET_X(0),
        OFFSET_Y(0),
        OFFSET_Z(0),
        SAMPLER_ID(0),
        SRC_SEL_X(SQ_SEL_X),
```

SRC_SEL_Y(SQ_SEL_Y),
SRC_SEL_Z(SQ_SEL_0),
SRC_SEL_W(SQ_SEL_1)),
TEX_DWORD_PAD,

这里总共三条指令，其中指令（0）和指令（1）是两条 CF 指令，指令（2）是一条取纹理的指令。

指令（0）表明程序将从 addr 为 2 的指令（2）处开始执行，指令（2）是一条 texture fetch 指令，这条指令根据 GPR0 中给出的纹理坐标（SRC_GPR(0)，根据前面 semantic 的配置，被差值的纹理坐标存放在 GPR0 中）从 id 号为 0 的纹理资源中取出纹理值，放入到 GPR6 中（DST_GPR(0x6)）。

取纹理操作完成后，执行指令（1），指令（1）是一条输出指令，将取到的纹理直接放到 Render target 0（ARRAY_BASE(CF_PIXEL_MRT0)）上去，如果是 OpenGL 程序源写 GLSL 程序，和上面程序功能类似的代码应该是这样的：

【xxxxx】

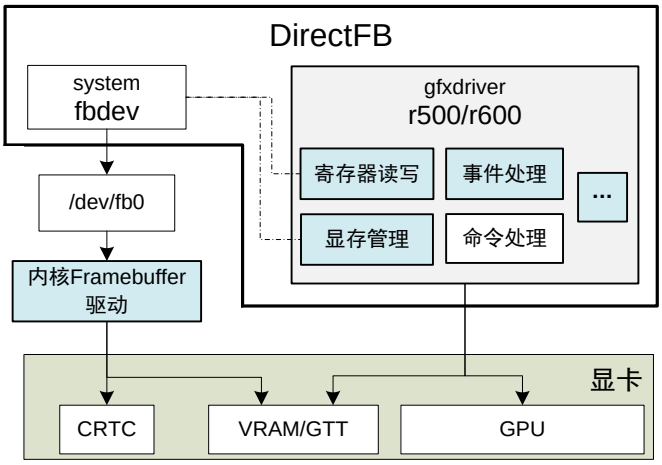
# 第四章 DirectFB 图形系统和图形驱动编写

\author{赵自成 xk}
\date{\today}\maketitle
\graphicspath{{./}{graph/}}

## 4.1 DirectFB 图形系统分析？？（本章详细分析）

## 4.2 驱动总体结构

如果没有硬件加速，directfb 会使用软件操作，directfb 的加速操作主要包括绘制线段，填充矩形，blit 操作，在 radeon 显卡上如果 3D 功能正常运行的画，还会有绘制三角形的操作。




这里先给出驱动的整体结构图，见图\ref{directfbdriverarch}。

关于 directfb 驱动，我们关心两部分内容，一部分是 system，一部分是 gfxdriver。

DirectFB 的简称为"Direct Framebuffer"，最初依赖 framebuffer 设备提供显示，这里称 framebuffer 为显示后端。

DirectFB 可以使用多种后端显示图形输出结果，在源码 system 目录下面有多个后端：

\begin{description}
\item[devmem] devmem 后端，系统直接操作/dev/mem 节点操作显存
\item[fbdev] 使用 framebuffer 后端
\item[osx] 使用 osx 操作系统作为图形后端
\item[sdl] 使用 sdl 作为图形后端
\item[vnc] 使用 vnc 作为图形后端
\item[x11] 使用 X 图形系统作为后端
\end{description}

根据我们系统环境的需要和平台的特点，我们选择 fbdev 系统。Linux 内核中的 drm 驱动提供了 framebuffer，但是这里的 framebuffer 基本上是 drm 驱动的一个附加产品（在“drm 的

framebuffer 驱动”中我们讨论了相关问题），而且内核只给了这里的 framebuffer 驱动仅供显示的一片内存，而且显卡的显存完全是由内核 drm 驱动管理的，但是 directfb 的 radeon 显卡驱动是不使用 drm 的那一套显存管理方式的（猜测是有 directfb 的时候 drm 还没有出现）。

如果使用 drm 的 framebuffer 会碰到一些问题：

- directfb 驱动会认为 fbdev 映射出来的内存就是可以使用的全部显存，而且在既有的 directfbr100~r300 驱动中，是没有使用第二章我们说的 GTT 内存的，而且没有启用页表基址，做所有加速的时候只能使用 vram 内存，在这种情况下，我们考虑这样一个情形：进行硬件加速的 blit 操作，我们有一个图形要进行多次拷贝操作，要移动的源数据必须存在于 vram，显示区域是最终合成的，源数据需要存储在非显示的区域中，然而单纯使用 drm 的 framebuffer 得到的 vram 只有显示部分的内存，这样硬件加速就没办法做了。[好像说的不是很清楚？？]
- 另外一个问题是我们注意到地址的问题，在第 2 章内存管理章节阶段我们提到过 VM_FB_LOCATION 寄存器，这个寄存器记录了 vram 在 GPU 虚拟地址空间的起始地址（通常起始地址配置为 0x0）和长度，在既有 directfb 的 r100~r300 驱动中，根据这个寄存器读出来的地址作为后续进行硬件加速时使用 vram 的基地址，而且认为这个地址开始处就是用于显示的内存的地址，但是在内核 drm 驱动中，vram 的前面一部分是用于页表之类的用于 framebuffer 的内存是从地址 0xc0000（R580 显卡 GPU 地址）开始的，但是 directfb 在进行硬件操作的时候读出来的地址是 0x0，这样 GPU 进行硬件操作的地址和用于显示的地址是不同的。GPU 认为显示的地址是 0x0，而实际 framebuffer 处在 0xc0000 处。

在使用 drm 的 framebuffer 的情况下，可以使用如下的方法解决上面两个问题：

第一个问题要解决显存不够的问题，在内核中，visiable vram 是通过访问 pci 配置空间获取到的，在核外，可以使用 libpciaccess 来读取 pci 配置空间获取到全部的 vram 和 IO（默认情况下 fbdev 认为通过 mmap 页可以映射 IO，但是 drm 的 framebuffer 是不提供 IO 的映射的）。这样需要修改 system/fbdev 的代码。

第二个问题，可以通过在 directfb 的 radeon 中直接修改 crtc 的基址寄存器，将其内容修改为 0x0，这样实际的显示区域和加速操作认为的显示区域就都是 0x0，但是在进行双缓冲切换的时候也会碰到一些问题，fbdev 的双缓冲是通过 PAN_DISPLAY 来进行的，PAN_DISPLAY 调用后，内核 drm 的 framebuffer 驱动仍然会将 crtc 基地址设为 framebuffer 驱动认为的地址，因而这里我们也需要修改一下。解决第二个问题的另外一个办法就是在进行硬件加速操作的时候地址都加上 0xc0000 这样一个偏移，当然也要考虑双缓冲切换的问题。

上面两个问题上我们是通过修改 fbdev，以修修补补的方式解决这个问题的，实际上，上面的办法还有一个问题无法解决，在进行双缓冲的时候，如果是在场同步的时候切换，则不会产生屏幕闪烁的情况，在“中断机制”一章我们分析过，如果 vblank 中断被开启，则在场同步的时候硬件会产生 vblank 中断通知软件，但是在既有的 directfb 的 radeon 驱动不依赖任何内核驱动，因而就无法获取中断，既有 directfb radeon 驱动使用轮询的方式以比较高的频率（应当保证轮询的时间间隔小于场同步显示处于空白期的时间）轮询中断状态寄存器，如果 vblank 的位被置位，则处于显示空白期，这时候可以发生切换，切换完成后将状态寄存器清除。然而在有内核 drm 驱动的情况下，内核 drm 驱动会处理中断，并在中断处理函数中将状态寄存器清除，这样核外的程序很难等到显示空白的时间。

因此更为直接的方式是将内核的 drm 驱动扔掉，自己写一个驱动并导出一个 framebuffer 驱动，于是就有了我们图\ref{directfbdriverarch}中给出的程序结构。

首先我们的内核驱动模块会完成原有的 drm 驱动完成的所有硬件初始化，包括显存的初

始化以及页表机制，命令环机制等。我们将显示区域调整到 0x0 的位置，framebuffer 驱动将剩余的全部 vram 映射出来，并且使用 framebuffer 驱动映射 IO，这样，directfb 中的 fbdev 不需要进行任何修改，将剩余的 vram 交给 directfb 之后，directfb 会有一个简单的显存管理机制，因而在内核中我们无需担心对这片内存的管理。

由于 r500 有单独 2D 引擎只需要简单读写寄存器就可以，而且部分代码可以参考既有的 R300 代码，所以这部分代码依然按照 MMIO 的方式，因此这里没有命令处理，全部硬件操作都在 VRAM 上进行，没有 GTT，因此显存管理部分的代码也很简单。

而 R600 情况则较为复杂，R600 必须使用 3D 硬件完成 2D 加速操作，能够参考的 R600 EXA 驱动都是使用命令包操作 GPU 核心的寄存器【是必须使用命令包写】，因此这里必须编写完整的显存管理和命令处理等机制。

## 4.3 内核 Framebuffer 驱动

第\verb|\ref{ch1}|中使用了一段简单的代码演示了对 Framebuffer 设备的操作，核内 framebuffer 驱动必须提供相应的接口，核外程序才能正确访问和操作 framebuffer，编写一个实现基本功能的 Framebuffer 驱动还是比较简单的。

```
233 int register_fb(struct radeon_device *rdev)
234 {
235         struct fb_info    *fbinfo;
236         struct radeonfb_info    *info;
......
241         fbinfo = framebuffer_alloc(sizeof(struct radeonfb_info),
242                                 &(rdev->pdev->dev));
243         info = fbinfo->par;
244         info->dev = &(rdev->pdev->dev);
245         info->fb = fbinfo;
......
256         fbinfo->fix = radeonfb_fix;
257         fbinfo->var = radeonfb_var;
258         fbinfo->fbops = &radeonfb_ops;
259         fbinfo->flags = FBINFO_FLAG_DEFAULT;
260         fbinfo->pseudo_palette = &info->pseudo_pal;
261
262         fbinfo->screen_base = rdev->fb_ptr;
263         fbinfo->fix.smem_len = DISPLAY_SIZE;[后面需要修改]
264         fbinfo->fix.smem_start = rdev->fb_phys;
......
271         ret = register_framebuffer(fbinfo);
......
279         return ret;
280 }
```

上面完成了 framebuffer 的注册过程，241-242 行分配了一个 framebuffer 结构，271 行注册 framebuffer 驱动，中间的代码主填充 framebuffer 结构。

256 行就是固定参数结构，257 行是可变参数结构，本节开始使用的应用程序请求的可变和固定参数就是这两个结构体。

258 行是对 framebuffer 驱动的操作函数，本质上 framebuffer 是 linux 内核中的字符驱动，但是 framebuffer 驱动相当于是对字符驱动的封装。这里的操作函数和单纯的字符设备的操作函数是有区别的。在内核 drivers/video/fbmem.c 文件有如下代码：

```
1424 static const struct file_operations fb_fops = {
1425        .owner =        THIS_MODULE,
1426        .read =         fb_read,
1427        .write =        fb_write,
1428        .unlocked_ioctl = fb_ioctl,
1429 #ifdef CONFIG_COMPAT
1430        .compat_ioctl = fb_compat_ioctl,
1431 #endif
1432        .mmap =         fb_mmap,
1433        .open =         fb_open,
1434        .release =    fb_release,
1435 #ifdef HAVE_ARCH_FB_UNMAPPED_AREA
1436        .get_unmapped_area = get_fb_unmapped_area,
1437 #endif
1438 #ifdef CONFIG_FB_DEFERRED_IO
1439        .fsync =        fb_deferred_io_fsync,
1440 #endif
1441 };
```

这些才是作为字符设备的操作函数，选出其中一个 read 函数：

```
 693 static ssize_t
 694 fb_read(struct file *file, char __user *buf, size_t count, loff_t *ppos)
 695 {
......
 711        if (info->fbops->fb_read)
 712                return info->fbops->fb_read(info, buf, count, ppos);
```

只有当用户自己实现了 fb_read 函数的时候才会调用用户自定义的函数，否则使用内核默认的操作。

实际上，大部分 framebuffer 的操作函数都有默认的函，如果用户实现.fb_set_par 和.fb_check_var[mmap 也可以使用内核默认的操作]两个函数就基本上能够让显卡工作了（默认的操作函数大多是运行在有自带显存的显卡上的，如果是使用系统内存作为显存的嵌入式系统，还需要提供 mmap 函数和读写操作函数）。fb_set_par 根据可变结构体中的信息设置硬件，主要是修改分辨率以及改变当前显示区域内存的位置。当然，如果要实现上面应用程序的 PAN_DISPLAY 操作，则还需要提供.pan_display 函数。

关于分辨率的设置通常是 framebuffer 驱动中一个比较重要的问题，但是考虑到是嵌入式系统，在初始化分辨率之后，后面基本上不会修改，因而这里我们不会关心这方面的问题。

260 行设置伪调色板，伪调色板用于 console，linux console 在显示内容的时候最多支持 16 中颜色，但是通常设备都支持的颜色远超过这么多（比如 RGB24 位色深支持 224 种颜色），因而在伪调色板中设置 16 中颜色，console 按照需要从伪颜色表中取颜色值。[需要确认？？]

262-264 设置帧缓冲使用的内存，这里记录了两个地址，262 是内核虚拟地址，应用程序直接读取设备节点的时候，内核需要将显存的数据拷贝到用户空间，需要使用虚拟地址访问显存。264 是物理地址，在将显存映射到用户空间的时候，需要使用物理地址（准确的说是物理页帧号）。

在进行双缓冲切换的，核外程序需要做一个 PAN_DISPLAY 的 ioctl 调用，内核 framebuffer 驱动需要实现.pan_dispay。

```
347 static int radeonfb_pan_display(struct fb_var_screeninfo *var,struct fb_info *fb)
348 {
349         struct radeonfb_info *par = fb->par;
350         u32 xoffset, yoffset;
351         u32 offset;
352         u32 xres, yres;
353
354         xres = var->xres;
355         yres = var->yres;
356         xoffset = var->xoffset;
357         yoffset = var->yoffset;
358         if(xoffset+xres > var->xres_virtual || yoffset+yres > var->yres_virtual)
359                 offset = 0;
360         else
361                 offset =(yoffset * xres +    xoffset)*4;
362
363         *(unsigned int *)((( char *)(par->mmio_start))+0x6110)=offset;
364         offset = *(unsigned int *)((( char *)(par->mmio_start))+0x6110);
365         return 0;
366 }
```

[364 和 363 行日后需要修改，对硬件的操作不放在 framebuffer 驱动中]。

这里 pan_display 函数根据传进来的 yoffset 切换显示画面的位置。

由于还需要通过 framebuffer 驱动映射显卡 IO，因此需要提供自己的 mmap 函数，默认操作是不提供 IO 映射的。

```
\begin{verbatim}
296 static int radeonfb_mmap(struct fb_info *info,
297                         struct vm_area_struct *vma)
298 {
299         unsigned long start = vma->vm_start;
300         unsigned long size = vma->vm_end - vma->vm_start;
301         unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
302         unsigned long pos;
303
304         if(offset == info->fix.smem_len){
305                 size_t mmio_size = ((struct radeonfb_info*)info->par)->mmio_size;
306                 pos = ((struct radeonfb_info*)info->par)->mmio_phys;
307                 if(size >((struct radeonfb_info*)info->par)->mmio_size){
```

```
308                     size = mmio_size;    // ok ??
309                 }
310         }
311         else {
312                 if (offset + size > info->fix.smem_len) {
313                     return -EINVAL;
314                 }
315                 pos = info->fix.smem_start + offset;
316         }
317         while (size > 0) {
318                   if  (io_remap_pfn_range(AMD003AMD003vma,  start,  pos  >>  PAGE_SHIFT,
PAGE_SIZE,
319                             PAGE_SHARED)) {
320                     printk("[%s] mmap failed\n", __func__);
321                     return -EAGAIN;
322                 }
323                 start += PAGE_SIZE;
324                 pos += PAGE_SIZE;
325         if (size > PAGE_SIZE)
326                 size -= PAGE_SIZE;
327         else
328                 size = 0;
329         }
330         vma->vm_flags |= VM_RESERVED;
331         return 0;
332 }
\end{verbatim}
```

Framebuffer 驱动从显卡驱动中得到了显存和寄存器的物理起始地址。有了物理地址，然后找到一个空闲的用户空间地址，将他们对应写入页表中，就建立了他们之间的映射关系，从而达到将显存或者寄存器映射到用户空间的目的。如果 mmap 传入的参数 offset 为 fix.smem_len，那就将 IO 地址映射到用户空间，否则就映射显存空间。

**显存的使用**

注意到驱动的主要部分都在核外，显存是映射到用户空间进行操作的。

R500 驱动只使用 VRAM，因此 VRAM 直接通过 Framebuffer 驱动映射出来就可以了。

R600 驱动使用 3D 部件进行操作，至少命令环缓冲区（Ring buffer）、顶点缓冲区（Vertex buffer）需要使用 GTT 内存，GTT 必须是可以进行 DMA 操作的内存，因此还必须在内核中分配并进行 DMA 映射。考虑在 Mesa 环境下，应用程序会根据需要请求 DMA GTT 内存（至少需要 Vertex buffer），但是在当前的 DirectFB R600 驱动中，一次只有一个完整的绘图操作，而且一次绘图操作时间很短，不会被其他绘图操作打断。因此这里不会有动态 DMA 内存分配的需求，一次分配足够的 DMA 内存，然后全部映射出来就可以了。

在 R500 驱动里，映射出来的内容分为两部分：VRAM 和寄存器，fbdev 中有一套自己的显存分配机制，被映射出来的 VRAM 被分成两部分，前面部分作为显示区域（要同内核 framebuffer 驱动约定好），包括当前的显示区域以及用于双环存切换的后备缓存。在硬件加

速操作需要显存的时候，fbdev 的分配器从显存的末端位置开始划分内存。

在 R600 驱动中，还需要映射 GTT 内存，按照我们的需求手动将 GTT 分成 index buffer 和 ring buffer 两部分。

## 3.4 核外 DirectFB 图形系统显卡驱动（驱动运行流程 数据结构等等）

## 3.5 R500 DirectFB 2D 加速驱动

### 3.5.1 实现

这里的 2D 加速全部是使用硬件上 2D 部件做的，2D 部件能够完成基本的 2D 操作，但是不能够处理透明度和三角形绘制。

Radeon 的加速程序位于 directfb 的 gfxdriver/radoen 目录下。编译后会生成一个库文件，当需要加载硬件加速时，dfb 将搜寻并加载这个库。找到库文件后，首先调用 radeon.c 的函数：

```
static int
driver_probe( CoreGraphicsDevice *device )
{
     switch (dfb_gfxcard_get_accelerator( device )) {
          case FB_ACCEL_ATI_RADEON:
               return 1;
          default:
               break;
     }
     return 0;
}
```

只有当函数返回 1 时，这个库才算匹配上，真正的加载，匹配的条件就是 radeon 加速代号 38。dfb_gfxcard_get_accelerator 函数，果有用户配置，返回用户配置的值，否则根据内核提供的 framebuffer 节点读出的信息返回。如果我们不在用户配置里配置，这里就匹配不上，因为从 fb 节点得到的值不是 38，而是 0。

在 dfb 中，硬件加速驱动首先执行两个初始化函数：driver_init_driver 和 driver_init_device。前一个函数主要得到了显卡的 mmio 地址和显存的起始地址：

```
rdrv->mmio_base = (volatile u8*) dfb_gfxcard_map_mmio( device, 0, 0x4000 );
rdrv->fb_base = dfb_gfxcard_memory_virtual( device, 0 );
```

这两个函数都需要内核的 framebuffer 驱动提供支持，关于细节后面讨论。

接下来需要修改的是:

```
if (chip >= CHIP_R300) {
```

```
    funcs->CheckState     = r300CheckState;
    funcs->SetState       = r300SetState;
}
```

这里根据显卡的型号，实现不同的 CheckState 和 SetState 函数。我们的 r500 要有自己的实现，CheckState 用于检查是否支持某种加速操作，SetState 用于硬件加速的准备性工，作设置一些寄存器，如：颜色，绘制方向等。我们在上面的语句前加上：

```
if(1) {
    funcs->CheckState     = r500CheckState;
    funcs->SetState =r500SetState;
}
```

driver_init_device 主要是通过都写显卡的寄存器，得到一些显卡的信息并进行设置，我们将对显卡的设置部分，也就是写寄存器的部分删除了，因为内核已经很好的初始化显卡。另外对于对齐要求有：

```
    device_info->limits.surface_byteoffset_alignment = 32;
    device_info->limits.surface_pixelpitch_alignment = 64;
    device_info->limits.surface_bytepitch_alignment   = 128;
```

将第一行的改为 1024，offset 需要 1k 对齐。对于初始化的改动就完成了。这里还没有具体的画图工作。

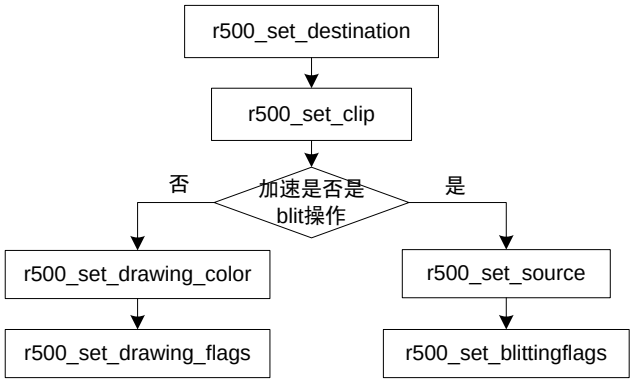
实现 r500CheckState，只需要 copy 一份 r300 的 CheckState，然后改好函数名就可以了。

```
static void r500SetState( void *drv, void *dev, GraphicsDeviceFuncs *funcs, CardState *state, DFBAccelerationMask accel )
{
    RadeonDriverData *rdrv = (RadeonDriverData*) drv;
    RadeonDeviceData *rdev = (RadeonDeviceData*) dev;
    void * mmio=rdrv->mmio_base;

    rdev->set &= ~state->mod_hw;
    if (DFB_BLITTING_FUNCTION( accel )) {
        if ((rdev->accel ^ accel) & DFXL_TEXTRIANGLES)
            rdev->set &= ~SMF_BLITTING_FLAGS;
    }
    rdev->accel = accel;
    r500_set_destination(rdrv,rdev,state);
    r500_set_clip(rdrv,rdev,state);
    switch (accel) {
        case DFXL_FILLRECTANGLE:
        case DFXL_FILLTRIANGLE:
        case DFXL_DRAWRECTANGLE:
        case DFXL_DRAWLINE:
            r500_set_drawing_color( rdrv, rdev, state );
            //if (state->drawingflags & DSDRAW_BLEND)
                //r300_set_blend_function( rdrv, rdev, state );
            r500_set_drawingflags( rdrv, rdev, state );
            funcs->DrawRectangle = r500drawrectangle;
```

```
                funcs->FillRectangle = r500fillrectangle;
                funcs->DrawLine = r500drawline;
                state->set = rdev->drawing_mask;
                break;
            case DFXL_BLIT:
            case DFXL_STRETCHBLIT:
            case DFXL_TEXTRIANGLES:
                r500_set_source( rdrv, rdev, state );
                r500_set_blittingflags( rdrv, rdev, state );
                state->set = (accel & DFXL_TEXTRIANGLES)
                     ? : (rdev->blitting_mask & ~DFXL_TEXTRIANGLES);
                break;
            default:
                D_BUG( "unexpected drawing/blitting function" );
                break;
        }
        funcs->FillTriangle    = NULL;
        funcs->Blit = r500blit;
        funcs->EmitCommands    = NULL;
        state->mod_hw = 0;
}
```

以上代码值得注意的是黑体部分【xxx】，流程如图\ref{setstatepipe}。




首先是设置画图地址 r500_set_destination，然后设置裁剪参数，然后根据不同的画图操作设置画图颜色和标志或者设置源地址和 blit 标志。接下来我们依次展开这些函数。

**1） 设置绘图位置**

```
void r500_set_destination( RadeonDriverData *rdrv,    RadeonDeviceData *rdev,
CardState*state )
{
        CoreSurface      *surface = state->destination;
        CoreSurfaceBuffer *buffer    = state->dst.buffer;
        volatile u8      *mmio= rdrv->mmio_base;
        u32offset;
        u32pitch;
```

```
    u32format    = 0;
    bool      dst_422 = false;

    if (RADEON_IS_SET( DESTINATION ))
        return;
    offset=state->dst.offset + rdev->fb_offset;
    pitch = state->dst.pitch;
    if (rdev->dst_offset != offset ||
            rdev->dst_pitch    != pitch ||
            rdev->dst_format != buffer->format) {
        format = R300_COLOR_FORMAT_ARGB8888;
        rdev->gui_master_cntl = GMC_DST_32BPP;
        rdev->gui_master_cntl |= GMC_DP_SRC_SOURCE_MEMORY    |
                GMC_WR_MSK_DIS|
                GMC_SRC_PITCH_OFFSET_CNTL |
                GMC_DST_PITCH_OFFSET_CNTL |
                GMC_DST_CLIPPING;
        radeon_waitfifo( rdrv, rdev, 1 );
        radeon_out32(mmio, RADEON_DST_PITCH_OFFSET,
                                (( pitch/64)<<22) | (offset>>10));
        if (rdev->dst_format != buffer->format) {
            if (dst_422 && !rdev->dst_422) {
                RADEON_UNSET( CLIP );
                RADEON_UNSET( SOURCE );
                rdev->src_format = DSPF_UNKNOWN;
            }
            RADEON_UNSET( COLOR );
            RADEON_UNSET( DST_BLEND );
        }
        rdev->dst_format = buffer->format;
        rdev->dst_offset = offset;
        rdev->dst_pitch    = pitch;
        rdev->dst_422= dst_422;
    }
    RADEON_SET( DESTINATION );
}
```

这个函数十分重要，设置了绘图时主要需要设置的寄存器，如果没有正确设置，图形不会被绘制出来或者绘制到错误的位置上。注意加粗的那部分代码，fb_offset 是 framebuffer 相对显存的偏移，dst.offset dst.pitch 都是通过 state 传来的，暂时不用考虑。看到上面写寄存器的操作，你就明白为什么为什么 offset 必须 1024 对齐了吧，因为它写入寄存器时右移 10 位。图\ref{alignproblem}显示了不对齐的情况下出现的结果，图片被保存到了 fbdev 分配的一块 surface 中，然后通过硬件加速的方式 blit 到了屏幕显示区域，如果 fbdev 分配的内存不是 1024 字节对齐的，那么写入 RADEON_DST_PITCH_OFFSET 寄存器的时候，后面 10 位会被截掉，但是显卡取数据的时候，只会从 1024 字节对齐的地址处开始取数据，这样就

会出现图 4-4 左边出现的情况，右边是对齐情况下 blit 的结果。

**2）设置裁剪参数**

```
void r500_set_clip( RadeonDriverData *rdrv,
                    RadeonDeviceData *rdev,
                         CardState      *state )
{
     DFBRegion      *clip = &state->clip;
     volatile u8 *mmio = rdrv->mmio_base;

     if (RADEON_IS_SET( CLIP ))
          return;
     radeon_out32( mmio, SC_TOP_LEFT,
          (clip->y1 << 16) | (clip->x1 & 0xffff) );
     radeon_out32( mmio, SC_BOTTOM_RIGHT,
          ((clip->y2+1) << 16) | ((clip->x2+1) & 0xffff) );
     rdev->clip = state->clip;
     RADEON_SET( CLIP );
}
```

**3）设置了画图的背景色和画笔色**

```
void r500_set_drawing_color(RadeonDriverData *rdrv, RadeonDeviceData *rdev, CardState
*state)
{
     void *mmio = rdrv->mmio_base;
     unsigned int dp_brush_bkgd_clr = 0x00000000;
     unsigned int dp_src_frgd_clr = 0xffffffff;
     unsigned int dp_src_bkgd_clr = 0x00000000;
     unsigned int dp_brush_frgd_clr = state->color.a << 24 |state->color.r<<16 |state->color.g<<8
| state->color.b;

     OUT_ACCEL_REG(RADEON_DEFAULT_SC_BOTTOM_RIGHT,(0X1fff<<0)|(0x1fff<<1
6));
     OUT_ACCEL_REG(RADEON_DP_BRUSH_FRGD_CLR, dp_brush_frgd_clr);
     OUT_ACCEL_REG(RADEON_DP_BRUSH_BKGD_CLR, dp_brush_bkgd_clr);
     OUT_ACCEL_REG(RADEON_DP_SRC_FRGD_CLR,      dp_src_frgd_clr);
```

```
    OUT_ACCEL_REG(RADEON_DP_SRC_BKGD_CLR,     dp_src_bkgd_clr);
    RADEON_SET( COLOR );
}
```

**4）设置绘图的方向和标志**

```
void r500_set_drawingflags( RadeonDriverData *rdrv,
                            RadeonDeviceData *rdev,
                            CardState       *state )
{
    volatile u8     *mmio          = rdrv->mmio_base;
    u32    master_cntl = rdev->gui_master_cntl     |
                         GMC_SRC_DATATYPE_MONO_FG_LA |
                         GMC_BRUSH_SOLID_COLOR   |
                         GMC_CLR_CMP_CNTL_DIS;
    u32 rb3d_blend;

    if (RADEON_IS_SET( DRAWING_FLAGS ))
        return;
    if (state->drawingflags & DSDRAW_XOR)
        master_cntl |= GMC_ROP3_PATXOR;
    else
        master_cntl |= GMC_ROP3_PATCOPY;
    radeon_waitfifo( rdrv, rdev, 2 );
    radeon_out32( mmio, DP_GUI_MASTER_CNTL, master_cntl );
    radeon_out32(          mmio,          DP_CNTL,          DST_X_LEFT_TO_RIGHT          |
DST_Y_TOP_TO_BOTTOM );
    rdev->drawingflags = state->drawingflags;
    RADEON_SET    ( DRAWING_FLAGS );
    RADEON_UNSET( BLITTING_FLAGS );
}
```

**5）设置 blit 标志**

```
void r500_set_blittingflags( RadeonDriverData *rdrv,
                             RadeonDeviceData *rdev,
                             CardState  *state )
{
    volatile u8     *mmio          = rdrv->mmio_base;
    u32            master_cntl = rdev->gui_master_cntl |
                                 GMC_BRUSH_NONE       |
                                 GMC_SRC_DATATYPE_COLOR;
    u32            txfilter1     = R300_TX_TRI_PERF_0_8;
    u32            cmp_cntl       = 0;
    u32            rb3d_blend;
    if (RADEON_IS_SET( BLITTING_FLAGS ))
        return;
```

```
    if (state->blittingflags & DSBLIT_SRC_COLORKEY) {
        txfilter1 |= R300_CHROMA_KEY_FORCE;
        cmp_cntl      = SRC_CMP_EQ_COLOR | CLR_CMP_SRC_SOURCE;
    }
    else {
        master_cntl |= GMC_CLR_CMP_CNTL_DIS;
    }

    if (state->blittingflags & DSBLIT_XOR)
        master_cntl |= GMC_ROP3_XOR;
    else
        master_cntl |= GMC_ROP3_SRCCOPY;
    radeon_waitfifo( rdrv, rdev, 2 );
    radeon_out32( mmio, CLR_CMP_CNTL, cmp_cntl );
    radeon_out32( mmio, DP_GUI_MASTER_CNTL, master_cntl );
rdev->drawingflags = state->drawingflags;

    RADEON_UNSET    ( DRAWING_FLAGS );
    RADEON_SET( BLITTING_FLAGS );
}
```

接下来介绍各个加速操作。读者读上面这些代码的时候可能已经注意到，上面这些操作写的寄存器的名称和第 4 章 GPU 命令包中命令包含的一些标志位名称相同，实际上后面各个加速操作往寄存器中填写的时俱和那些 3 型命令包的主体部分是类似的。我们选取 blit 过程，读者可参卡代码了解其它过程加速过程所写的寄存器。

```
bool r500blit(CardState *state, void *drv, void *dev,
              DFBRectangle *sr, int dx, int dy )
{
    RadeonDeviceData *rdev = (RadeonDeviceData*) dev;
    RadeonDriverData *rdrv = (RadeonDriverData*) drv;
    unsigned int dir = 0;
    volatile void * mmio=rdrv->mmio_base;
    if (sr->x <= dx) {
        sr->x += sr->w-1;
        dx += sr->w-1;
    }
    else {
        dir |= DST_X_LEFT_TO_RIGHT;
    }
    if(sr->y <= dy) {
        sr->y += sr->h - 1;
        dy += sr->h - 1;
    }
    else {
```

```
        dir |= DST_Y_TOP_TO_BOTTOM;
    }
    OUT_ACCEL_REG(DP_CNTL, dir);
    OUT_ACCEL_REG(SRC_Y_X, (sr->y << 16) | (sr->x));
    OUT_ACCEL_REG(DST_Y_X, (dy << 16) | dx);
    OUT_ACCEL_REG(DST_HEIGHT_WIDTH, (sr->h << 16) | (sr->w) );
    return true;
}
```

Blit 函数中判断源与目标的坐标大小，是为了决定 copy 的方向，blit 是将内存中的一块 framebuffer 信息 copy 到另一个坐标。如果两块区域是重叠的，那么就会在拷贝中出现覆盖的现象。所以 if 语句就是调整拷贝的方和起始坐标。

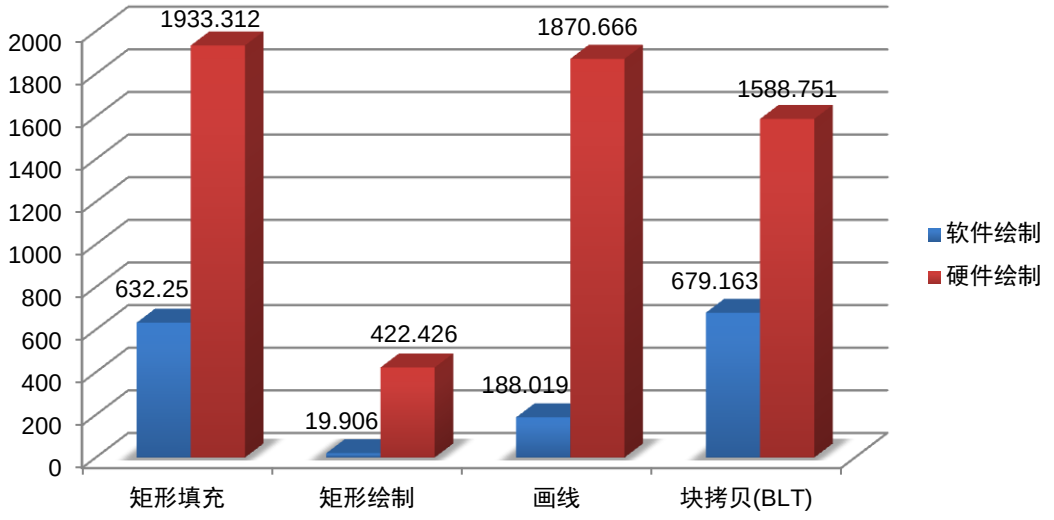
这里共操作了四个寄存器，DP_CNTL 寄存器控制拷贝方向，接下来两个分别是源坐标和目的坐标，最后是拷贝的宽高。

具体到每个操作都不需要写很多寄存器，其实是因为分了很多个函数里面去了。每个画图函数里面只做它特有的操作，其他共性的操作都归类到 SetState 中的几个函数里去了。这样使重复的代码变少了，层次也很分明。

## 3.5.2 加速效果

使用 df_dok 程序进行了测试，测试平台是一台 Intel xx 机器，DirectFB 版本，使用-O3 参数优化。

结果见图\ref{r5002d}，【予以适当说明】




xxxxxx

## 3.6 R600 DirectFB 2D 加速驱动

(本节内容由 xk 编写，请予以适当修改)

### 3.6.1 内核的支持

对于 r600 显卡，内核的模块只需要在 r500 的基础上稍微改动，主要的要点如下：

1）我们在 directfb 中需要给显卡发布命令，让显卡完成我们期望的动作。所以需要让核外的 directfb 中的驱动能够往命令环（显卡所支持的一种与 cpu 交互的机制，一个由 cpu 存放命令，显卡读出执行的环）。所以需要将命令环的存储空间地址映射到用户态可访问的地址（就是通过 mmap，建立页表）。

2）因为我们是在用户态的 directfb 中完成绘图，所以需要在用户态向显卡提交定点数据，还有 shader 程序。所以需要映射一片显卡能访问的存储空间到用户态。

请看下面，内核模块中 mmap 函数，它一共映射了 4 片存储空间到用户态，也就是 r500 中映射的显存和寄存器空间在加上上述两块：

```
static int radeonfb_mmap(struct fb_info *info,
                         struct vm_area_struct *vma)
{
        unsigned long start = vma->vm_start;
        unsigned long size = vma->vm_end - vma->vm_start;
        unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
        unsigned long pos;
        if(offset ==0xc00000){
                size_t mmio_size = ((struct radeonfb_info*)info->par)->mmio_size;
                pos = ((struct radeonfb_info*)info->par)->mmio_phys;
                if(size >((struct radeonfb_info*)info->par)->mmio_size){
                        size = mmio_size;    // ok ??
                }
                // check the size
        }
        else if(offset == ((struct radeonfb_info*)info->par)->ring_offset) {
                pos = ((struct radeonfb_info*)info->par)->ring_phys;
                if(size >((struct radeonfb_info*)info->par)->ring_size){
                        size = ((struct radeonfb_info*)info->par)->ring_size;
                }
        }
        else if(offset == ((struct radeonfb_info*)info->par)->tmp_offset){
                pos = ((struct radeonfb_info*)info->par)->tmp_phys;
                if(size >((struct radeonfb_info*)info->par)->tmp_size){
                        size = ((struct radeonfb_info*)info->par)->tmp_size;
```

```
                }
        }
        else {
                if(offset != 0) {
                        printk(KERN_ERR "[%s %d] page offset is not 0, maybe
error\n",
                                                __func__, __LINE__);
                }
                if (offset + size > 0xc00000) {
                        size = 0xc00000;
                }
                pos = info->fix.smem_start + offset;
        }
        while (size > 0) {
                if (remap_pfn_range(vma, start, pos >> PAGE_SHIFT, PAGE_SIZE,
                                        PAGE_SHARED)) {
                        printk("[%s] mmap failed\n", __func__);
                        return -EAGAIN;
                }
                start += PAGE_SIZE;
                pos += PAGE_SIZE;
        if (size > PAGE_SIZE)
                size -= PAGE_SIZE;
        else
                size = 0;
        }
        vma->vm_flags |= VM_RESERVED;
        return 0;
    }
```

在 mmap 函数中，我们通过 offset 参数判断需要将哪一块存储空间映射到用户空间，得到不同的 pos（就是对应的物理地址），通过 remap_pfn_range 函数建立页表，这样用户态的程序就可以用页表的虚拟地址访问这块存储空间。这几块存储空间是怎么初始化的，请自己阅读内核模块中的初始化过程，这里不做介绍了。

## 3.6.2 directfb 的 fbdev 部分介绍

上面介绍了在内核模块的 mmap 函数，能将一些显卡的存储空间映射到用户态，那么接下来介绍一下怎么在用户空间映射这些空间。

这部分代码位于 system/fbdev/fbdev.c 文件中。

里面有映射 mmio 空间的函数：

```
    static volatile void *
    system_map_mmio( unsigned int       offset,
```

```
                        int                     length )
{
        void *addr;

        if (length <= 0)
                length = dfb_fbdev->shared->fix.mmio_len;

        addr = mmap( NULL, 0x10000, PROT_READ | PROT_WRITE, MAP_SHARED,
                        dfb_fbdev->fd, 0xc00000 );
        if (addr == MAP_FAILED) {
                D_PERROR( "DirectFB/FBDev: Could not mmap MMIO region "
                                "(offset %d, length %d)!\n", offset, length );
                return NULL;
        }
```

Mmio 空间主要是显卡的寄存器的地址，映射得到了 addr 是一个用户态地址，它是显卡寄存器空间的起始地址，有了这个地址，再加上适当的偏移量，就可以在用户态访问显卡的寄存器。

接下来就按照上面的函数，添加映射命令环的函数和映射顶点数组空间。我增加的函数是：

system_map_ring ：映射命令环

system_map_tmp ：映射顶点数组和 shader 程序的存储空间

具体函数这里不再粘贴出来，请自己看源代码，因为跟映射 mmio 的过程很像，只需要修改对 length 的检测和 mmap 函数中最后一个参数（偏移量）的数值。（这里跟内核代码是对应起来的）。

还有一个映射是映射显存（可以看作帧缓冲空间）。

```
dfb_fbdev->framebuffer_base = mmap( NULL, shared->fix.smem_len,
                                    PROT_READ | PROT_WRITE,
MAP_SHARED,
                                    dfb_fbdev->fd, 0 );
```

这条语句就是获得帧缓冲起始地址的。我们需要的 4 种显卡资源现在都映射到用户空间了。

当然我们增加了函数后，还要增加相应的接口函数。这里简单介绍一下：

在 src/core/core_system.h 文件中增加：

```
static volatile void*        system_map_ring( unsigned int offset, int length);
static volatile void* system_map_tmp( unsigned int offset, int length );
```

在 static CoreSystemFuncs system_funcs 初始化中增加两行：

```
static CoreSystemFuncs system_funcs = {
.......
        .MapRing            = system_map_ring,
        .MapTmp              = system_map_tmp,
.......
};
```

在 src/core/system.h 文件的 CoreSystemFuncs 结构体中增加成员

```
volatile void* (*MapRing)( unsigned int offset, int length );
volatile void* (*MapTmp)( unsigned int offset, int length );
```

增加函数：

```
volatile void *
dfb_system_map_tmp( unsigned int       offset,
                    int                length );

volatile void *
dfb_system_map_ring( unsigned int      offset,
                     int               length );
```

在 src/core/system.c 文件中增加函数：

```
volatile void *
dfb_system_map_ring( unsigned int      offset,
                     int               length )
{
     D_ASSERT( system_funcs != NULL );

     return system_funcs->MapRing( offset, length );
}

volatile void *
dfb_system_map_tmp( unsigned int       offset,
                    int                length )
{
     D_ASSERT( system_funcs != NULL );

     return system_funcs->MapTmp( offset, length );
}
```

在./src/core/gfxcard.h 文件中添加函数声明：

```
volatile void *dfb_gfxcard_map_ring( CoreGraphicsDevice *device,
                                     unsigned int       offset,
                                     int                length );

volatile void *dfb_gfxcard_map_tmp( CoreGraphicsDevice *device,
                                    unsigned int       offset,
                                    int                length );
```

在./src/core/gfxcard.c 文件中添加函数：

```
volatile void *
dfb_gfxcard_map_ring( CoreGraphicsDevice *device,
                      unsigned int       offset,
                      int                length )
{
     return dfb_system_map_ring( offset, length );
```

```
}

volatile void *
dfb_gfxcard_map_tmp( CoreGraphicsDevice *device,
                     unsigned int        offset,
                     int                 length )
{
     return dfb_system_map_tmp( offset, length );
}
```

上面说了这么多，都是为了是增加的函数于之前的接口一致，待会我们会调用这些接口。

到此为止，我们已经能够在 directfb 中访问所需要的显卡资源。接下来介绍 r600 的 2d 加速操作。

## 3.6.3 directfb 的 r600 驱动部分

1）r600 的驱动匹配和初始化

在 r500 的加速驱动中已经介绍了，driver_probe 函数完成了驱动的匹配：通过一个加速编号（编号由内核驱动中给出，我们的是在配置文件中配置的），radeon 的加速编号是 38.这里不再重复描述。

接下来，进行一系列的初始化工作，我们只需要稍微修改。基本原则是：所有对显卡的初始化都期望在内核中完成，directfb 的这些操作都去掉。因为 directfb 的初始化代码都是针对 r300 之前的显卡，很多寄存器都变动或废弃了。所以我们去掉 driver_init_device 函数中的有关寄存器的操作。

主要讲一下新增的操作。在 driver_init_driver 中增加了关于帧缓冲和几个映射函数的信息：

```
          rdrv->fb_size = 0xa00000; //0xc00000
          rdrv->fb_w = 1024;
          rdrv->fb_h = 768;

     if(chip == CHIP_RV610){
               rdrv->ring_base = dfb_gfxcard_map_ring( device, 0, 0x100000);
               rdrv->ring_size = 0x100000;
printf("rdrv->ring_base:%p\n",rdrv->ring_base);
               rdrv->tmp_base = dfb_gfxcard_map_tmp( device, 0, 0x400000);
               rdrv->tmp_size = 0x400000;
          }
```

包括了帧缓冲的大小，长，宽。作用后面会讲到。而映射了命令环的基地址和 tmp 的基地址也会在后面介绍作用。

在 driver_init_device 函数中，还有很重要的初始化：

```
        device_info->caps.flags = CCF_CLIPPING | CCF_AUXMEMORY |
CCF_RENDEROPTS;
            if(rdev->chipset == CHIP_RV610){
                    r600_shader_init(rdrv, rdev);
                    rdev->scratch_reg = 0x8500;
                    radeon_ring_init(rdrv, rdev);
                    if(radeon_ring_test(rdrv, rdev))
                            printf("ring_test retrun true\n");
                    radeon_indirect_test(rdrv,rdev);
            }
        if (rdev->chipset >= CHIP_R300) {
```

这里包括初始化函数：r600_shader_init，radeon_ring_init，使用了 scratch_reg 寄存器 0x8500.并做了简单的测试。下面详细说明两个初始化函数都做了什么。

r600_shader_init 函数：将 5 段 shader 程序放入到缓冲区 rdev->shader.buffer 中。

```
            rdev->shader.state_offset = 0;
        rdev->shader.state_len = r6xx_default_size;
            dwords = rdev->shader.state_len;
            while (dwords & 0xf) {
                    packet2s[num_packet2s++] = PACKET2(0);
                    dwords++;
            }

            obj_size = dwords * 4;
            obj_size = ALIGN(obj_size, 256);


            rdev->shader.draw_vs_offset = obj_size;
            obj_size += r6xx_vs_size * 4;
            obj_size = ALIGN(obj_size, 256);

            rdev->shader.draw_ps_offset = obj_size;
            obj_size += r6xx_ps_size * 4;
            obj_size = ALIGN(obj_size, 256);

            rdev->shader.blit_vs_offset =obj_size;
            obj_size += r6xx_blit_vs_size * 4;
            obj_size = ALIGN(obj_size, 256);

            rdev->shader.blit_ps_offset = obj_size;
            obj_size += r6xx_blit_ps_size * 4;
            obj_size = ALIGN(obj_size, 256);
            r600_shader_buffer_alloc(rdrv, rdev, obj_size);
```

上面代码的作用是计算出 5 段 shader 程序在缓冲区的偏移值，存放顺序是：

state_offset,draw_vs_offset,draw_ps_offset,blit_vs_offset,blit_ps_offset.注意到 ALIGN 这个宏在每次计算出 size 后出现，它就是使对齐的宏，所有的 shader 都以 256 字节对齐存放在缓冲区中。而中间的 while 循环不难看出是在 PACKET2(0)包补齐 state 以 16 个双字对齐空缺的位置。对齐是由于命令环的要求。

然后用 r600_shader_buffer_alloc 函数申请 shader 的空间，函数得到了预先留给 shader 的显存空间地址，并会简单检测是否可以得到空间。这块空间位于映射出来的 fb 块的偏移为 0xa00000，所以可以得到 buffer 的用户空间地址和 gpu 空间地址。

```
    void r600_shader_buffer_alloc(RadeonDriverData *rdrv, RadeonDeviceData *rdev,unsigned int size)
    {
            if(size < 0x200000){
                    rdev->shader.buffer = (u32 *)(((unsigned char *)rdrv->fb_base)+0xa00000);
                    rdev->shader.shader_gpu_addr = 0xa00000;
            }
            else
                    rdev->shader.buffer = NULL ;
    }
```

接下来将 shader 程序放入缓冲区就很简单了。

radeon_ring_init 函数：初始化命令环。

起始命令环的真正初始化是在内核中进行，在核外我们只是需要使用命令环，所谓初始化不过就是得到命令环的读写指针和基地址。基地址前面已经讲到怎么映射出来了。下面贴出部分代码：

```
            rdev->rb.ring =(u32 *) rdrv->ring_base;
             ......
            rdev->rb.wptr = radeon_in32(rdrv->mmio_base,CP_RB_WPTR);
            rdev->rb.rptr = radeon_in32(rdrv->mmio_base,CP_RB_RPTR);
```

还有要增加 r600 的显卡 dev_table 表，使 radeon_find_chipset 能正确找到显卡的信息。

2）CheckState 和 SetState

这两个很重要的回调函数。在执行完初始化后，每次有了绘图动作，想要调用加速函数时，都要经过这两个函数。

r600CheckState 函数的作用是：检测支持哪些绘图操作。比如来了一个绘制三角形的操作，directfb 会调用 r600CheckState 来检查是否有驱动支持硬件的绘制，如果有，那么就调用相应绘图函数。反之就有软件绘图来完成。

r600SetState 函数的作用是：为接下来的硬件绘制做一些状态设置和准备。比如设置绘制颜色：r600_set_drawing_color;设置 blit 操作源的一些信息：r600_set_source。设置绘制颜色很好理解，就是要用什么颜色绘制图形。而如果知道 blit 操作是把一块图像像素复制到另一块区域，那么设置操作源就很好理解了。操作源就是被复制的像素块，它有的信息是：长，宽，pitch（每行的字节数，

如何换行），offset（偏移地址，如何找到这块像素）。

这里没有给出具体代码分析，因为这两个函数比较简单，而且 r500 中有过比较详细的介绍，所以请对照源码看一看。

3）具体的绘图操作

R600 的绘图操作包括：（目前实现）r600FillRectangle3D，r600FillTriangle3D，r600DrawRectangle3D，r600DrawLine3D，r600Blit3D，r600StretchBlit。前四个是绘制，后面两个属于 blit。我们在两大类中个选取一个详细说明。

首先介绍 draw 类型的操作例子：r600FillRectangle3D

r600FillRectangle3D 函数主要包括 3 个部分：r600_draw_line_prepare，r600_fill_rectangle，r600_draw_done。

```
bool r600FillRectangle3D(void *drv, void *dev, DFBRectangle *rect)
{
        RadeonDriverData * rdrv = (RadeonDriverData *)drv;
        RadeonDeviceData * rdev = (RadeonDeviceData *)dev;
        struct radeon_fence * fence;

        fence = (struct radeon_fence *)malloc(sizeof(struct radeon_fence));
        if(!fence){
                printf(" Failed to alloc fence struct\n");
                return false;
        }
        fence->timeout = false;
        fence->done = false;
        fence->seq = DRAW_LINE;

        r600_draw_line_prepare(rdrv,rdev,DRAW_SHADER);

        r600_fill_rectangle(rdrv, rdev, rect);

        return r600_draw_done(rdrv, rdev, fence);

}
```

首先介绍第一部分：

static int r600_draw_line_prepare(RadeonDriverData *rdrv, RadeonDeviceData *rdev,u32 type)

它主要做了如下几件事：申请顶点缓冲（vb），申请命令环空间，设置 shader 程序。

申请顶点缓冲：r600_vb_buffer_get。顶点缓冲使用的是在初始化是映射出来的 tmp 显存。这里的申请就是以 tmp 的基址为基址，向上划分出 64k 的空间用于存储顶点数据。而我们不仅需要这块空间的用户空间地址，而且需要它的 gpu 空间地址。源代码如下：

```
static int r600_vb_buffer_get(RadeonDriverData *rdrv, RadeonDeviceData *rdev, int type)
{
        unsigned int free;
```

```
        free = rdrv->tmp_size - rdev->tmp_used * 64 * 1024;
        if(free >= 64 *1024){
                rdev->r600_vb.type = type ;
                rdev->r600_vb.base = (unsigned char *)rdrv->tmp_base ;
                rdev->r600_vb.gpu_addr = 0x10000000+1024*1024+(16*16+8)*1024;
                rdev->r600_vb.total = 0x10000 ;
                rdev->r600_vb.used = 0;
                rdev->tmp_used++;
                return 0;
        }
}
```

命令环空间申请：radeon_ring_alloc;命令环的初始化是在内核中进行，这里的申请就是确认是否命令环的空余空间足够完成接下来的命令写入。如果不够，就需要等待命令环处理前面的命令，腾出空间。代码中的 while 语句就是等待和验证是否有足够命令环空间的部分。源代码如下：

```
int radeon_ring_alloc(RadeonDriverData *rdrv, RadeonDeviceData *rdev, unsigned ndw)
{
        int r;
printf("radeon_ring_alloc\n");
        ndw = (ndw +rdev->rb.align_mask) & ~rdev->rb.align_mask;
        while (ndw > (rdev->rb.free_dw -1)) {
                rdev->rb.rptr = radeon_in32(rdrv->mmio_base, CP_RB_RPTR);
                rdev->rb.free_dw = rdev->rb.rptr + rdrv->ring_size / 4;
                rdev->rb.free_dw -= rdev->rb.wptr;
                rdev->rb.free_dw &= rdev->rb.ptr_mask;
                if(!rdev->rb.free_dw){
                        rdev->rb.free_dw = rdrv->ring_size / 4;
                }
                if(ndw < rdev->rb.free_dw){
                        break;
                }
        }
        rdev->rb.count_dw = ndw;
        return 0;
}
```

设置 Shader 程序：r600_set_default_state，r600_set_shaders。前面已经介绍了我们把 5 段 shader 程序存放在一个缓冲区中，这个缓冲区是位于映射出来的 fb 块中的。我们需要知道的这些 shader 存放的 gpu 空间地址也可以通过 buffer 的 gpu 基地址和偏移量相加计算出来。而设置 shader 最重要的就是要让 gpu 知道 shader 程序在哪里。

```
void r600_set_default_state(RadeonDeviceData *rdev)
{
。。。。。。
        gpu_addr = rdev->shader.shader_gpu_addr + rdev->shader.state_offset;
```

```
            radeon_ring_write(rdev,
#ifdef __BIG_ENDIAN
                                    (2 << 0) |
#endif
                                    (gpu_addr & 0xFFFFFFFC));
。。。。。。
}
```

上面是设置 default state 的过程，设置顶点 shader 和像素 shader 也是如此，要交代 shader 程序的 gpu 空间地址，这样 gpu 就能依据地址得到 shader 程序。

接下来介绍绘制举行的第二部分。

void r600_fill_rectangle(void *drv, void *dev, DFBRectangle *rect )

这个函数主要是给出了顶点信息，给出渲染目标（屏幕）信息，裁剪大小，设置顶点的相关参数，绘制图形。

```
void r600_fill_rectangle(void *drv, void *dev, DFBRectangle *rect )
{
。。。。。。
            vb[i++] = i2f(rect->x);
            vb[i++] = i2f(rect->y);
            vb[i++] = i2f(0);
            vb[i++] = i2f(1);
            vb[i++] = f2d(rdev->color[0]);
            vb[i++] = f2d(rdev->color[1]);
            vb[i++] = f2d(rdev->color[2]);
            vb[i++] = f2d(rdev->color[3]);
。。。。。。
            test_r600_set_render_target(rdev, COLOR_8_8_8_8, rdrv->fb_w, rdrv->fb_h,
dst_gpu_addr);
            test_r600_set_scissors(rdev, 0, 0, rdrv->fb_w, rdrv->fb_h);
            test_r600_set_vtx_resource(rdev, vb_gpu_addr,32*4,32);

            test_r600_draw_auto(rdev,DI_PT_QUADLIST,1,4);
。。。。。。
}
```

首先在是在 vb 中装入正确的顶点信息包括位置和颜色信息。

test_r600_set_render_target 函数设置了渲染目标的颜色，宽，高，gpu 地址。

test_r600_set_scissors 函数设置了裁剪窗口大小

test_r600_set_vtx_resource 函数设置了每个顶点的到小和顶点缓冲的总长度。有了这两个信息，gpu 就知道如何从顶点缓冲区取得顶点，到哪截止。上面代码总每个顶点共 32 字节，共 4 个顶点。

test_r600_draw_auto 函数告诉 gpu 我们要绘制那种图元（绘制什么图形），和一共有几个图形实例，以及用到几个顶点。上面的代码中 test_r600_draw_auto(rdev,DI_PT_QUADLIST,1,4);就意味着绘制 1 个矩形串，总共有 4 个顶点。

第三部分，完成绘图。

static int r600_draw_done( RadeonDriverData *rdrv, RadeonDeviceData* rdev, struct radeon_fence *fence)

它在绘图操作后增加 fence 提交，命令提交和 fence 检测操作。

```
static int r600_draw_done( RadeonDriverData *rdrv, RadeonDeviceData* rdev, struct radeon_fence *fence)
{
        r600_fence_emit(rdev, fence);

        radeon_ring_commit(rdrv,rdev);

        r600_waitfor_fence(rdrv,rdev,fence);    //how to read scratch regs?
        if(fence->done){
                r600_free_vb(rdev);
printf("draw_done finish\n");
                return true;
        }
        else if(fence->timeout){
                printf("waitfor drawing done time out !\n");
                return true;
        }
}
```

r600_fence_emit 函数：fence 就是一种类似与中断的信号，用于提示前面的命令是否执行完。我们利用 scratch 寄存器完成这一操作。

```
static void r600_fence_emit(RadeonDeviceData *rdev, struct radeon_fence *fence)
{
      radeon_ring_write(rdev, PACKET3(PACKET3_EVENT_WRITE, 0));
        radeon_ring_write(rdev, EVENT_TYPE(CACHE_FLUSH_AND_INV_EVENT) | EVENT_INDEX(0));
                /* wait for 3D idle clean */
        radeon_ring_write(rdev, PACKET3(PACKET3_SET_CONFIG_REG, 1));
        radeon_ring_write(rdev, (WAIT_UNTIL - PACKET3_SET_CONFIG_REG_OFFSET) >> 2);
        radeon_ring_write(rdev, WAIT_3D_IDLE_bit | WAIT_3D_IDLECLEAN_bit);
                /* Emit fence sequence & fire IRQ */
        radeon_ring_write(rdev, PACKET3(PACKET3_SET_CONFIG_REG, 1));
        radeon_ring_write(rdev, ((rdev->scratch_reg - PACKET3_SET_CONFIG_REG_OFFSET) >> 2));
        radeon_ring_write(rdev, fence->seq);
        /* CP_INTERRUPT packet 3 no longer exists, use packet 0 */
        radeon_ring_write(rdev, PACKET0(CP_INT_STATUS, 0));
        radeon_ring_write(rdev, RB_INT_STAT);
}
```

加粗部分代码就是利用 scratch 寄存器完成 fence 操作，当前面的命令都正确

执行了，就会在 scratch 寄存器中哦你写入 seq，这样我们就能通过读 scratch 寄存器的内容来判断绘图操作是否正确执行了。

命令提交：补齐空缺命令完成对齐要求，告诉 gpu 开始执行命令。

```
void radeon_ring_commit(RadeonDriverData *rdrv, RadeonDeviceData *rdev)
{
        unsigned count_dw_pad;
        unsigned i;

        /* We pad to match fetch size */
        count_dw_pad = (rdev->rb.align_mask + 1) -
                       (rdev->rb.wptr & rdev->rb.align_mask);
        for (i = 0; i < count_dw_pad; i++) {
                radeon_ring_write(rdev, 2 << 30);
        }
        r600_cp_commit(rdrv,rdev);
}
void r600_cp_commit(RadeonDriverData *rdrv, RadeonDeviceData *rdev)
{
        radeon_out32(rdrv->mmio_base, CP_RB_WPTR, rdev->rb.wptr);
        radeon_in32(rdrv->mmio_base, CP_RB_WPTR);
}
```

检测 fence：读 scratch 寄存器，是否为 seq。

```
static void r600_waitfor_fence(RadeonDriverData *rdrv, RadeonDeviceData *rdev, struct
radeon_fence *fence)
{
        unsigned int val;
        int i;
        for(i=0;i<100000;i++){
                val = radeon_in32(rdrv->mmio_base, rdev->scratch_reg);
                if(val == fence->seq)
                        break;
        }
        if(i >= 100000){
                fence->timeout = true;
                printf("fence->seq:%x",val);
        }
        else fence->done = true;

}
```

绘制图形的操作还有绘制三角形，填充矩形，绘制直线，他们和上面例子的区别很小，几乎只有图元不同和顶点数目不同而已。

接下来介绍 blit 操作，blit 操作是像素拷贝。我们看一下例子：r600StretchBlit

```
bool r600StretchBlit( void *drv, void *dev, DFBRectangle *sr, DFBRectangle *dr )
{
```

```
        RadeonDriverData * rdrv = (RadeonDriverData *)drv;
        RadeonDeviceData * rdev = (RadeonDeviceData *)dev;
        struct radeon_fence * fence;

        fence = (struct radeon_fence *)malloc(sizeof(struct radeon_fence));
        if(!fence){
                printf(" Failed to alloc fence struct\n");
                return false;
        }
        fence->timeout = false;
        fence->done = false;
        fence->seq = FILL_RECT;

        r600_draw_line_prepare(rdrv,rdev,BLIT_SHADER);

        r600_blit(rdrv, rdev, sr, dr);

        return r600_draw_done(rdrv, rdev, fence);
    }
```

跟矩形绘制操作的区别就是粗体部分。一个是 shader 的不一样，另一个是绘制过程不一样。

```
    int r600_blit(RadeonDriverData *rdrv, RadeonDeviceData *rdev, DFBRectangle
*sr,DFBRectangle * dr)
    {
    。。。。。。
        vb[i++] = i2f(dr->x);
        vb[i++] = i2f(dr->y);
        vb[i++] = i2f(sr->x);
        vb[i++] = i2f(sr->y);
    。。。。。。
        test_r600_set_tex_resource(rdev, FMT_8_8_8_8, rdev->src_width,
rdev->src_height,
                                                    rdev->src_pitch/4,
src_gpu_addr);// notice the pitch
        test_r600_set_render_target(rdev, COLOR_8_8_8_8, rdrv->fb_w, rdrv->fb_h,
dst_gpu_addr);
        test_r600_set_scissors(rdev, 0, 0, rdrv->fb_w, rdrv->fb_h);
        test_r600_set_vtx_resource(rdev, vb_gpu_addr,16*4,16);
        test_r600_draw_auto(rdev,DI_PT_RECTLIST,1,4);
    。。。。。。
    }
```

首先是顶点信息不一样，这里没有了颜色信息，而是有两组位置信息组成一个顶点信息，第一对位置信息是 blit 目的的位置信息，第二对位置信息是纹理位置信息（blit 源）。

test_r600_set_tex_resource 函数是告诉显卡关于 blit 源的信息，包括宽，高，pitch，gpu 空间地址。特别注意 pitch，一般的 pitch 都指行字节数，而这里却是以像素为单位的。（如果按照字节为单位写入 pitch，图像会是原来的 4 分之 1，压缩的效果）

这里的顶点与前面不一样，所以 test_r600_set_vtx_resource 的参数也是不一样。

4）shader 程序

接下来说说我们在 r600 中用到的简单的 shader 程序。程序中共有 5 段 shader 程序：r6xx_default_state ，r6xx_vs，r6xx_ps，r6xx_blit_vs，r6xx_blit_ps。

r6xx_default_state 是很多状态量的设置，比如绘制线条的宽度。

Vs 就是顶点 shader 的缩写，顶点处理器就是用来处理顶点信息的程序，每个顶点都会经过它的处理。CF 开头的语句是流程控制语句，

```
    const u32 r6xx_vs[] = {
            CF_DWORD0(ADDR(4)),
            CF_DWORD1(POP_COUNT(0), CF_CONST(0),
COND(SQ_CF_COND_ACTIVE),
                    I_COUNT(2),    CALL_COUNT(0), END_OF_PROGRAM(0),
                    VALID_PIXEL_MODE(0), CF_INST(SQ_CF_INST_VTX),
                    WHOLE_QUAD_MODE(0), BARRIER(1)),
            CF_ALLOC_IMP_EXP_DWORD0(ARRAY_BASE(CF_POS0),
                    TYPE(SQ_EXPORT_POS),
                    RW_GPR(0x1),    // i changed it to 0x8
                    RW_REL(ABSOLUTE),
                    INDEX_GPR(0),
                    ELEM_SIZE(0)),
            CF_ALLOC_IMP_EXP_DWORD1_SWIZ(SRC_SEL_X(SQ_SEL_X),
                    SRC_SEL_Y(SQ_SEL_Y),
                    SRC_SEL_Z(SQ_SEL_0),
                    SRC_SEL_W(SQ_SEL_1),
                    R6xx_ELEM_LOOP(0),
                    BURST_COUNT(0),
                    END_OF_PROGRAM(0),
                    VALID_PIXEL_MODE(0),
                    CF_INST(SQ_CF_INST_EXPORT_DONE),
                    WHOLE_QUAD_MODE(0),
                    BARRIER(1)),
            CF_ALLOC_IMP_EXP_DWORD0(ARRAY_BASE(0),
                    TYPE(SQ_EXPORT_PARAM),
                    RW_GPR(0x2),    // i changed it to 0x8
                    RW_REL(ABSOLUTE),
                    INDEX_GPR(0),
                    ELEM_SIZE(0)),
            CF_ALLOC_IMP_EXP_DWORD1_SWIZ(SRC_SEL_X(SQ_SEL_Z),
                    SRC_SEL_Y(SQ_SEL_W),
```

```
                    SRC_SEL_Z(SQ_SEL_0),
                    SRC_SEL_W(SQ_SEL_1),
                    R6xx_ELEM_LOOP(0),
                    BURST_COUNT(0),
                    END_OF_PROGRAM(1),
                    VALID_PIXEL_MODE(0),
                    CF_INST(SQ_CF_INST_EXPORT_DONE),
                    WHOLE_QUAD_MODE(0),
                    BARRIER(0)),
            VTX_DWORD_PAD,
            VTX_DWORD_PAD,
            VTX_DWORD0(VTX_INST(SQ_VTX_INST_FETCH),
                    FETCH_TYPE(SQ_VTX_FETCH_VERTEX_DATA),
                    FETCH_WHOLE_QUAD(0),
                    BUFFER_ID(0),
                    SRC_GPR(0),
                    SRC_REL(ABSOLUTE),
                    SRC_SEL_X(SQ_SEL_X),
                    MEGA_FETCH_COUNT(16)),
            VTX_DWORD1_GPR(DST_GPR(0x1),    // changed to 1
                    DST_REL(0),
                    DST_SEL_X(SQ_SEL_X),
                    DST_SEL_Y(SQ_SEL_Y),
                    DST_SEL_Z(SQ_SEL_Z),
                    DST_SEL_W(SQ_SEL_W),
                    USE_CONST_FIELDS(0),
                    DATA_FORMAT(FMT_32_32_32_32_FLOAT),
                    NUM_FORMAT_ALL(SQ_NUM_FORMAT_SCALED),
                    FORMAT_COMP_ALL(SQ_FORMAT_COMP_SIGNED),
                    SRF_MODE_ALL(SRF_MODE_ZERO_CLAMP_MINUS_ONE)),
            VTX_DWORD2(OFFSET(0),
#ifdef __BIG_ENDIAN
                    ENDIAN_SWAP(SQ_ENDIAN_8IN32),
#else
                    ENDIAN_SWAP(SQ_ENDIAN_NONE),
#endif
                    CONST_BUF_NO_STRIDE(0),
                    MEGA_FETCH(1)),
            VTX_DWORD_PAD,
            VTX_DWORD0(VTX_INST(SQ_VTX_INST_FETCH),
                    FETCH_TYPE(SQ_VTX_FETCH_VERTEX_DATA),
                    FETCH_WHOLE_QUAD(0),
                    BUFFER_ID(0),
                    SRC_GPR(0),
```

```
                SRC_REL(ABSOLUTE),
                SRC_SEL_X(SQ_SEL_X),
                MEGA_FETCH_COUNT(16)),
        VTX_DWORD1_GPR(DST_GPR(0x2),    // changed to 1
                DST_REL(0),
                DST_SEL_X(SQ_SEL_X),
                DST_SEL_Y(SQ_SEL_Y),
                DST_SEL_Z(SQ_SEL_Z),
                DST_SEL_W(SQ_SEL_W),
                USE_CONST_FIELDS(0),
                DATA_FORMAT(FMT_32_32_32_32_FLOAT),
                NUM_FORMAT_ALL(SQ_NUM_FORMAT_SCALED),
                FORMAT_COMP_ALL(SQ_FORMAT_COMP_SIGNED),
                SRF_MODE_ALL(SRF_MODE_ZERO_CLAMP_MINUS_ONE)),
        VTX_DWORD2(OFFSET(16),
#ifdef __BIG_ENDIAN
                ENDIAN_SWAP(SQ_ENDIAN_8IN32),
#else
                ENDIAN_SWAP(SQ_ENDIAN_NONE),
#endif
                CONST_BUF_NO_STRIDE(0),
                MEGA_FETCH(0)),
        VTX_DWORD_PAD,
};
```

CF_DWORD0(ADDR(4)),说明第一条语句在偏移为 4 的语句（64 位为一个单位，一个宏是 32 位，就是第 9 个宏开始）。

CF_DWORD1，交代后面有多少条指令，I_COUNT(2)说明一共 2 条指令，一条指令共 16 字节，就是 4 条宏语句。

接下来就跳转到第 9 条宏，在这里一共执行两条指令。第一条：

```
VTX_DWORD0(VTX_INST(SQ_VTX_INST_FETCH),
                FETCH_TYPE(SQ_VTX_FETCH_VERTEX_DATA。。。。
```

说明是获得顶点数据的指令;

```
VTX_DWORD1_GPR(DST_GPR(0x1),    // changed to 1
                DST_REL(0),
                DST_SEL_X(SQ_SEL_X),
                DST_SEL_Y(SQ_SEL_Y),
                DST_SEL_Z(SQ_SEL_Z),
                DST_SEL_W(SQ_SEL_W),
                USE_CONST_FIELDS(0),
                DATA_FORMAT(FMT_32_32_32_32_FLOAT),
                NUM_FORMAT_ALL(SQ_NUM_FORMAT_SCALED),
                FORMAT_COMP_ALL(SQ_FORMAT_COMP_SIGNED),
            SRF_MODE_ALL(SRF_MODE_ZERO_CLAMP_MINUS_ONE)),
```

说明将获得的顶点数据存放到通用寄存器 0x1 中，有四个分量分别对应顶点

的四个分量。顶点的数据格式是 4 个 32 位的浮点数。

```
            VTX_DWORD2(OFFSET(0),
#ifdef __BIG_ENDIAN
                    ENDIAN_SWAP(SQ_ENDIAN_8IN32),
#else
                    ENDIAN_SWAP(SQ_ENDIAN_NONE),
#endif
                    CONST_BUF_NO_STRIDE(0),
                    MEGA_FETCH(1)),
```

这个字段的 offset 是指在一个顶点数据中的偏移，

```
VTX_DWORD_PAD,
```

无实际意义的填充字段。到此就是一条指令，第一条指令的作用就是从顶点数据中取得 4 个 32 位的浮点数，存储在 0x1 的通用寄存器中。接下来是第二条。第二条指令结构和功能上与第一条是一样的，作用是从顶点数据中偏移为 16 字节开始，取得 4 个 32 位的浮点数，存储到 0x2 的通用寄存器中。我们回忆一下绘制矩形的顶点数据的结构，前 4 个浮点数是位置信息，后四个浮点数是颜色信息。

两条顶点指令执行完成，就回到流程控制语句（CF）。接下来的流程控制语句是：

```
        CF_ALLOC_IMP_EXP_DWORD0(ARRAY_BASE(CF_POS0),
                TYPE(SQ_EXPORT_POS),
                RW_GPR(0x1),    // i changed it to 0x8
                RW_REL(ABSOLUTE),
                INDEX_GPR(0),
                ELEM_SIZE(0)),
         CF_ALLOC_IMP_EXP_DWORD1_SWIZ(SRC_SEL_X(SQ_SEL_X),
                SRC_SEL_Y(SQ_SEL_Y),
                SRC_SEL_Z(SQ_SEL_0),
                SRC_SEL_W(SQ_SEL_1),
                R6xx_ELEM_LOOP(0),
                BURST_COUNT(0),
                END_OF_PROGRAM(0),
                VALID_PIXEL_MODE(0),
                CF_INST(SQ_CF_INST_EXPORT_DONE),
                WHOLE_QUAD_MODE(0),
                BARRIER(1)),
```

语句的 EXP 表明是数据导出语句，CF_POS0 表示导出到位置信息寄存器 POS0 中，RW_GPR(0x1)表明从 0x1 通用寄存器中导出，也就是前面存放位置信息的地方。

SRC_SEL_X(SQ_SEL_X),SRC_SEL_Y(SQ_SEL_Y),SRC_SEL_Z(SQ_SEL_0), SRC_SEL_W(SQ_SEL_1)表明位置信息四个分量分别来为 x，y，0, 1.

接下来的两条流程控制语句：

```
        CF_ALLOC_IMP_EXP_DWORD0(ARRAY_BASE(0),
                TYPE(SQ_EXPORT_PARAM),
```

```
                RW_GPR(0x2),    // i changed it to 0x8
                RW_REL(ABSOLUTE),
                INDEX_GPR(0),
                ELEM_SIZE(0)),
        CF_ALLOC_IMP_EXP_DWORD1_SWIZ(SRC_SEL_X(SQ_SEL_Z),
                SRC_SEL_Y(SQ_SEL_W),
                SRC_SEL_Z(SQ_SEL_0),
                SRC_SEL_W(SQ_SEL_1),
                R6xx_ELEM_LOOP(0),
                BURST_COUNT(0),
                END_OF_PROGRAM(1),
                VALID_PIXEL_MODE(0),
                CF_INST(SQ_CF_INST_EXPORT_DONE),
                WHOLE_QUAD_MODE(0),
                BARRIER(0)),
```

这也是导出语句，导出的是参数（SQ_EXPORT_PARAM），源地址是 0x2 通用寄存器。 END_OF_PROGRAM(1),表明这是顶点 shader 程序的结束。

r6xx_ps 程序

```
const u32 r6xx_ps[] =
{
        CF_ALLOC_IMP_EXP_DWORD0(ARRAY_BASE(CF_PIXEL_MRT0),
                TYPE(SQ_EXPORT_PIXEL),
                RW_GPR(0x0),            // i changed it to 0x6
                RW_REL(ABSOLUTE),
                INDEX_GPR(0),
                ELEM_SIZE(1)),
        CF_ALLOC_IMP_EXP_DWORD1_SWIZ(SRC_SEL_X(SQ_SEL_X),
                SRC_SEL_Y(SQ_SEL_Y),
                SRC_SEL_Z(SQ_SEL_Z),
                SRC_SEL_W(SQ_SEL_W),
                R6xx_ELEM_LOOP(0),
                BURST_COUNT(1),
                END_OF_PROGRAM(1),
                VALID_PIXEL_MODE(0),
                CF_INST(SQ_CF_INST_EXPORT_DONE),
                WHOLE_QUAD_MODE(0),
                BARRIER(1)),
};
```

通过 vs 程序的处理，我们已经得到了位置信息和颜色信息。Ps 是处理像素的，所以它只需要将得到的颜色导出到屏幕上就好了。所以 ps 总共只有一条导出指令。CF_PIXEL_MRT0 表明导出的目的是渲染目标（MRT）0,帧缓冲。SQ_EXPORT_PIXEL 表明导出的是像素信息。RW_GPR(0x0)与 vs 中的颜色导出的 ARRAY_BASE(0)对应。

r6xx_blit_vs。

我们先想一想这个顶点处理器应该怎么写。先考虑输入，输入是顶点信息，而 blit 操作的顶点信息是由 2 组位置信息组成的，一组是目的位置，一组是源位置。所以我们需要两条 VTX_DWORD 指令来获得这两组位置信息，把他们存储到通用寄存器中。需要注意的是顶点的格式，每组位置信息的维度。下面给出代码：

```
        VTX_DWORD0(VTX_INST(SQ_VTX_INST_FETCH),
                FETCH_TYPE(SQ_VTX_FETCH_VERTEX_DATA),
                FETCH_WHOLE_QUAD(0),
                BUFFER_ID(0),
                SRC_GPR(0),
                SRC_REL(ABSOLUTE),
                SRC_SEL_X(SQ_SEL_X),
                MEGA_FETCH_COUNT(16)),
        VTX_DWORD1_GPR(DST_GPR(1),
                DST_REL(0),
                DST_SEL_X(SQ_SEL_X),
                DST_SEL_Y(SQ_SEL_Y),
                DST_SEL_Z(SQ_SEL_0),
                DST_SEL_W(SQ_SEL_1),
                USE_CONST_FIELDS(0),
                DATA_FORMAT(FMT_32_32_FLOAT),
                NUM_FORMAT_ALL(SQ_NUM_FORMAT_SCALED),
                FORMAT_COMP_ALL(SQ_FORMAT_COMP_SIGNED),
                SRF_MODE_ALL(SRF_MODE_ZERO_CLAMP_MINUS_ONE)),
        VTX_DWORD2(OFFSET(0),
#ifdef __BIG_ENDIAN
                ENDIAN_SWAP(SQ_ENDIAN_8IN32),
#else
                ENDIAN_SWAP(SQ_ENDIAN_NONE),
#endif
                CONST_BUF_NO_STRIDE(0),
                MEGA_FETCH(1)),
        VTX_DWORD_PAD,

        VTX_DWORD0(VTX_INST(SQ_VTX_INST_FETCH),
                FETCH_TYPE(SQ_VTX_FETCH_VERTEX_DATA),
                FETCH_WHOLE_QUAD(0),
                BUFFER_ID(0),
                SRC_GPR(0),
                SRC_REL(ABSOLUTE),
                SRC_SEL_X(SQ_SEL_X),
                MEGA_FETCH_COUNT(8)),
```

```
        VTX_DWORD1_GPR(DST_GPR(0),
                DST_REL(0),
                DST_SEL_X(SQ_SEL_X),
                DST_SEL_Y(SQ_SEL_Y),
                DST_SEL_Z(SQ_SEL_0),
                DST_SEL_W(SQ_SEL_1),
                USE_CONST_FIELDS(0),
                DATA_FORMAT(FMT_32_32_FLOAT),
                NUM_FORMAT_ALL(SQ_NUM_FORMAT_SCALED),
                FORMAT_COMP_ALL(SQ_FORMAT_COMP_SIGNED),
            SRF_MODE_ALL(SRF_MODE_ZERO_CLAMP_MINUS_ONE)),
        VTX_DWORD2(OFFSET(8),
#ifdef __BIG_ENDIAN
                ENDIAN_SWAP(SQ_ENDIAN_8IN32),
#else
                ENDIAN_SWAP(SQ_ENDIAN_NONE),
#endif
                CONST_BUF_NO_STRIDE(0),
                MEGA_FETCH(0)),
        VTX_DWORD_PAD,
```

将两组位置信息存储在通用寄存器 0 和 1 中，其中 0 存储的是源位置信息，1 存储的是目的位置信息，接下来就是导出了。目的位置信息肯定是作为顶点的位置信息导出的，这一点应该不用质疑什么。那么源位置信息呢，通过图像学知识，纹理信息也跟颜色信息一样是顶点信息的一部分。所以就是作为参数导出。

```
        CF_ALLOC_IMP_EXP_DWORD0(ARRAY_BASE(CF_POS0),
                TYPE(SQ_EXPORT_POS),
                RW_GPR(1),
                RW_REL(ABSOLUTE),
                INDEX_GPR(0),
                ELEM_SIZE(0)),
        CF_ALLOC_IMP_EXP_DWORD1_SWIZ(SRC_SEL_X(SQ_SEL_X),
                SRC_SEL_Y(SQ_SEL_Y),
                SRC_SEL_Z(SQ_SEL_Z),
                SRC_SEL_W(SQ_SEL_W),
                R6xx_ELEM_LOOP(0),
                BURST_COUNT(0),
                END_OF_PROGRAM(0),
                VALID_PIXEL_MODE(0),
                CF_INST(SQ_CF_INST_EXPORT_DONE),
                WHOLE_QUAD_MODE(0),
                BARRIER(1)),

        CF_ALLOC_IMP_EXP_DWORD0(ARRAY_BASE(0),
                TYPE(SQ_EXPORT_PARAM),
```

```
                RW_GPR(0),
                RW_REL(ABSOLUTE),
                INDEX_GPR(0),
                ELEM_SIZE(0)),
        CF_ALLOC_IMP_EXP_DWORD1_SWIZ(SRC_SEL_X(SQ_SEL_X),
                SRC_SEL_Y(SQ_SEL_Y),
                SRC_SEL_Z(SQ_SEL_Z),
                SRC_SEL_W(SQ_SEL_W),
                R6xx_ELEM_LOOP(0),
                BURST_COUNT(0),
                END_OF_PROGRAM(1),
                VALID_PIXEL_MODE(0),
                CF_INST(SQ_CF_INST_EXPORT_DONE),
                WHOLE_QUAD_MODE(0),
                BARRIER(0)),
```

r6xx_blit_ps

像素处理器应该如何工作呢，我们将纹理坐标作为参数传给了像素处理器，该如何使用呢？像素处理的输出无疑是像素信息，就是颜色。而输入是纹理坐标，那么像素处理器就应该依据纹理坐标取的颜色信息。const u32 r6xx_blit_ps[] = {

```
          CF_DWORD0(ADDR(2)),
        CF_DWORD1(POP_COUNT(0),
                CF_CONST(0),
                COND(SQ_CF_COND_ACTIVE),
                I_COUNT(1),
                CALL_COUNT(0),
                END_OF_PROGRAM(0),
                VALID_PIXEL_MODE(0),
                CF_INST(SQ_CF_INST_TEX),
                WHOLE_QUAD_MODE(0),
                BARRIER(1)),

        CF_ALLOC_IMP_EXP_DWORD0(ARRAY_BASE(CF_PIXEL_MRT0),
                TYPE(SQ_EXPORT_PIXEL),
                RW_GPR(0x6),            // i changed it to 0x6
                RW_REL(ABSOLUTE),
                INDEX_GPR(0),
                ELEM_SIZE(1)),
        CF_ALLOC_IMP_EXP_DWORD1_SWIZ(SRC_SEL_X(SQ_SEL_X),
                SRC_SEL_Y(SQ_SEL_Y),
                SRC_SEL_Z(SQ_SEL_Z),
                SRC_SEL_W(SQ_SEL_W),
                R6xx_ELEM_LOOP(0),
                BURST_COUNT(1),
```

```
                END_OF_PROGRAM(1),
                VALID_PIXEL_MODE(0),
                CF_INST(SQ_CF_INST_EXPORT_DONE),
                WHOLE_QUAD_MODE(0),
                BARRIER(1)),

        TEX_DWORD0(TEX_INST(SQ_TEX_INST_SAMPLE),
                BC_FRAC_MODE(0),
                FETCH_WHOLE_QUAD(0),
                RESOURCE_ID(0),
                SRC_GPR(0),
                SRC_REL(ABSOLUTE),
                R7xx_ALT_CONST(0)),
        TEX_DWORD1(DST_GPR(0x6),    //    i changed it to 0x6
                DST_REL(ABSOLUTE),
                DST_SEL_X(SQ_SEL_X), /* R */
                DST_SEL_Y(SQ_SEL_Y), /* G */
                DST_SEL_Z(SQ_SEL_Z), /* B */
                DST_SEL_W(SQ_SEL_W), /* A */
                LOD_BIAS(0),
                COORD_TYPE_X(TEX_UNNORMALIZED),
                COORD_TYPE_Y(TEX_UNNORMALIZED),
                COORD_TYPE_Z(TEX_UNNORMALIZED),
                COORD_TYPE_W(TEX_UNNORMALIZED)),
        TEX_DWORD2(OFFSET_X(0),
                OFFSET_Y(0),
                OFFSET_Z(0),
                SAMPLER_ID(0),
                SRC_SEL_X(SQ_SEL_X),
                SRC_SEL_Y(SQ_SEL_Y),
                SRC_SEL_Z(SQ_SEL_0),
                SRC_SEL_W(SQ_SEL_1)),
        TEX_DWORD_PAD,

    };
```

前面的流程控制语句我们现在已经很好看懂了，先是执行跳转，跳转结束后导出像素信息。那么重点是 TEX_DWORD 语句。

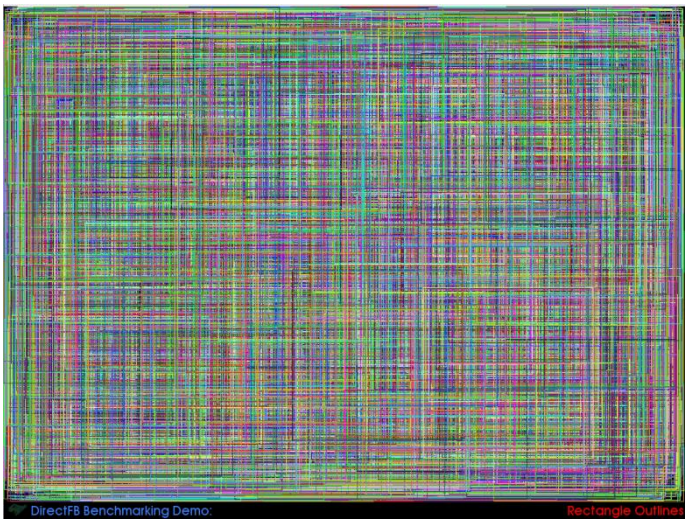
TEX_DWORD0 表明操作的是哪个纹理对象，从中取得像素信息

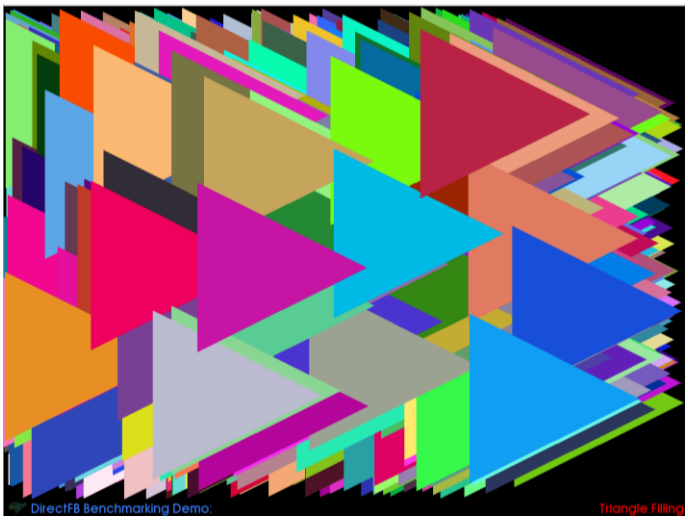
TEX_DWORD1 中 DST_GPR(0x6)表明取得的中间结果存放在 0x6 通用寄存器中，

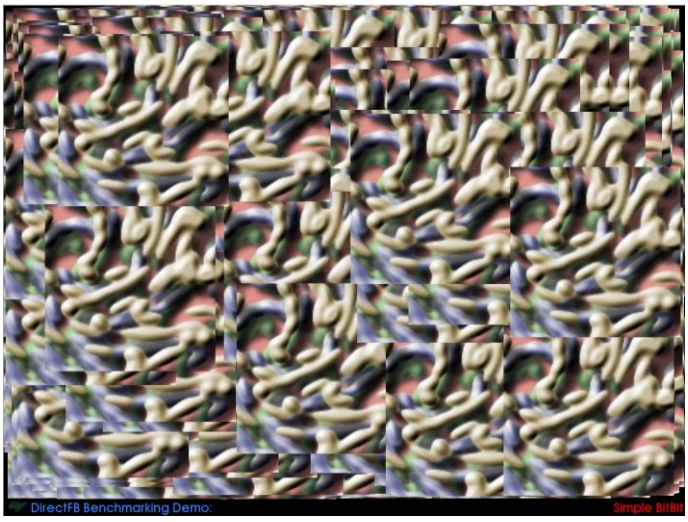
TEX_DWORD2 表明在纹理对象的坐标是 x，y，0,1.

合起来看的意思是，我们在纹理对象 0 中，坐标为 x，y 的点上，取得这个点的颜色，存放在 0x6 的通用寄存器中。联系接下来的导出指令，整个 blit 的 ps 就是将纹理的颜色作为绘制物体的颜色。

添加结果图片和说明


DirectFB Benchmarking Demo:
Rectangle Outlines


DirectFB Benchmarking Demo:
Triangle Filling


DirectFB Benchmarking Demo:
Simple BitBlt