DirectFB Overview (v0.1)

Andreas Hundt <andi@convergence.de>

September 11, 2001

Abstract

This document discusses features and concepts of DirectFB. It is work in progress.

Contents

1 Introduction

DirectFB is a thin library that provides hardware graphics acceleration, input device handling and abstraction, integrated windowing system with support for translucent windows and multiple display layers on top of the Linux Framebuffer Device. It is a complete hardware abstraction layer with software fallbacks for every graphics operation that is not supported by the underlying hardware. DirectFB was designed with embedded systems in mind. It offers maximum hardware accelerated performance at a minimum of resource usage and overhead.

1.1 Goals

- small memory footprint
- maximum possible hardware acceleration
- support of advanced graphics operations incuding multiple alpha blending modes
- no kernel modifications, make use of existing standards
- no library dependencies (except libc)
- meet the requirements of MHP®

1.2 Features

1.2.1 Graphic Operations

DirectFB supports the the following graphics operations. These can be done in hardware if supported by the chipset driver, or as software fallback:

- Rectangle filling/drawing
- Triangle filling/drawing
- Line drawing
- (Stretched) blitting
- Blending with an alphachannel (a.k.a. texture alpha)
- Blending with a constant alpha blend factor (a.k.a. alpha modulation)
- Nine different blending functions respectively for source and destination, so all Porter/Duff rules are supported
- Colorizing (a.k.a. color modulation)
- Source color keying
- Destination color keying

1.2.2 Windowing System

DirectFB has a fast integrated windowing system, that supports translucent windows. Windows using ARGB Surfaces can be blended on per pixel basis. In addition each window has its own global transparency.

1.2.3 Resource Management

DirectFB has its own resource management for video memory. Resources like display layers or input devices can be locked for exclusive access, e.g. for fullscreen games. DirectFB provides abstraction for the different graphics targets like display layers, windows and any general purpose surfaces. The programming effort for switching from windowed to fullscreen and back is minimized to setting the desired cooperative level.

1.2.4 Graphic Drivers

For hardware acceleration DirectFB uses loadable driver modules. Currectly the following chipsets are supported:

- Matrox G200/G400/G450
- ATI Rage 128
- 3dfx Voodoo3/Banshee
- igs CyberPro (not released to the public)
- S3 Savage 3/4 series (not released to the public)
- NeoMagic
- nVidia TNT/Radeon series

Other chipsets will work, but there is no acceleration support.

1.2.5 Input Drivers

DirectFB supports the following input devices:

- standard keyboards
- serial and PS/2 mice
- joysticks
- infrared remote controls (using lirc)

  It is possible to query the hardware directly or to use an event buffer.

1.2.6 Image Loading

DirectFB includes image providers, which allow loading of the following image formats directly into DirectFB surfaces:

- JPEG (using libjpeg)
- PNG (using libpng2)
- GIF

1.2.7 Video Playback

DirectFB includes video providers, which allow rendering of the following video formats into DirectFB surfaces:

- video4linux (/dev/video)
- mpeg1/2 (using libmpeg3)
- AVI (using avifile)
- macromedia flash (using libflash)

1.2.8 Font Rendering

DirectFB supports anti aliased text drawing and includes font providers, which allow loading of the follwing font formats:

- DirectFB bitmap font
- TrueType (using FreeType2)

2 DirectFB architecture

2.1 Access to the graphics hardware by DirectFB

To access the graphics hardware DirectFB relies on the existing kernel interface, which the framebuffer device provides (/dev/fb). That means that DirectFB needs a working framebuffer driver to work. For some chipsets there is a special framebuffer driver in the linux kernel. For unsupported chipsets a vesa framebuffer will also work (although with some limitations). No matter if graphics acceleration is used or not, DirectFB will use the framebuffer device to perform the following tasks:

- setting up the video mode (resolution, color depth, timings)
- memory mapping of the cards the framebuffer
- changing of the viewport of the framebuffer (for doublebuffering)

If a card is supported by DirectFB and a framebuffer driver for a special chipset is present in the linux kernel, DirectFB will use the framebuffer device in addition to the tasks mentioned above to do perform the following tasks:

- memory mapping of the cards memory mapped io ports
- turning off the framebuffer driver's internal acceleration

To execute a specific graphics operation (e.g. blitting of a surface), the DirectFB chipset driver will access the memory mapped io ports of the graphics hardware to submit the command to the card's acceleration engine. That means that the actual hardware acceleration is done entirely from user space.

See figure 1 on page 6.

2.2 Access to input devices by DirectFB

To access input devices DirectFB uses the standard device interfaces that the linux kernel provides. No input hardware is accessed directly by DirectFB.

2.3 Important terms used by DirectFB

2.3.1 Blitting

Blitting refers to the process of copying image data. The simplest example is to Blit a Surface to another while both Surfaces have the same size, depth and pixel format. In this case the memory has to be only copied without being processed (just like copying any other type of data). Advaced types of Blitting include Blitting with an alpha channel, or Bliting from one pixel format to another. Most graphics cards include a hardware blitter that is capable to perform various kinds of Blitting.

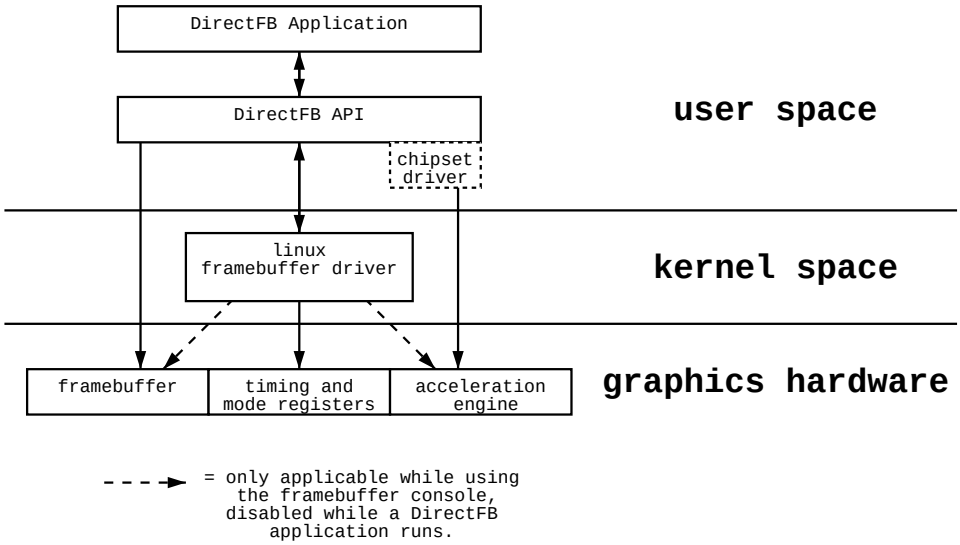

Figure 1: Access to the framebuffer device and the graphics hardware by a DirectFB application

2.3.2 Surface

A Surface is a reserved area in memory where pixel data for one image is stored in a specific pixel format. A Surface can reside in video and/or system memory. It is possible to perform drawing operations on a surface or to Blit the surface to another (see section 1.2.1).

In fullscreen mode, the visible screen is represented by the ”Primary Surface”, so it is possible to perform graphics operations directly on the visible screen.

Every Surface can optionally use double buffering, graphics operations are then first executed on a secondary buffer and become valid after Flip() is called. To prevent flickering it is advised to use Doublebuffering on the primary Surface in most cases.

2.3.3 SubSurface

A SubSurface uses the same interface as a regular Surface. It represents a part of a parent Surface and does not allocate any system or video memory itself.

2.3.4 Layer

Depending on the graphics hardware, there are one or more display Layers. A standard PC graphics card has only one Layer, but sophisticated devices like TV set top boxes may support two or more layers. Layers occupy different areas in video ram, and are usually combined by using alpha blending. This in done automatically

by the display hardware. If the content of the lowest Layer changes, no repaint has to be done and the contents of the above layers remain untouched.

2.3.5 Window / Windowstack

Normally the contents of a layer's surface is controlled by the integrated windowing system that shows the windows belonging to the layer on a configurable background. Each window has its own surface that is used by the windowing system to produce the composed image of overlapping windows. Alternatively applications, especially games, can get exclusive access to the layer. This way the application has direct control over the layer's surface and it's contents, no windows are shown.

2.4 Example of a Layer/Window configuration

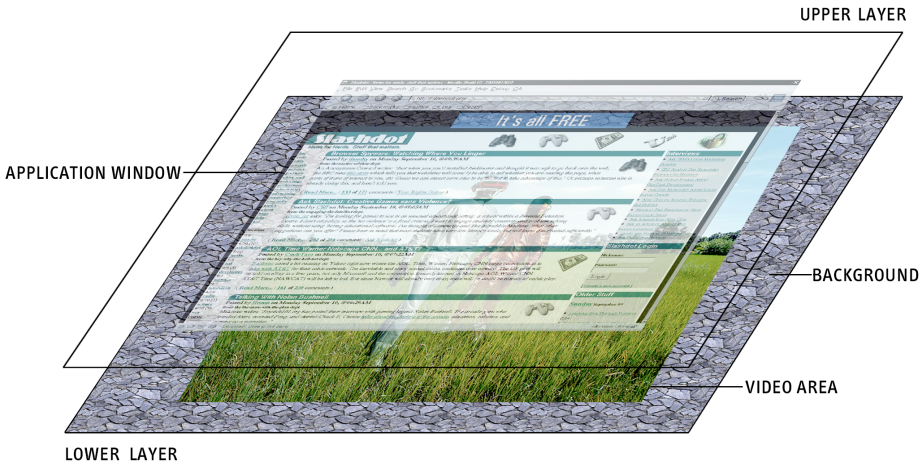

Figure 2: The lower Layer is in fullscreen mode, a background image has been rendered to its primary Surface. The video area is a SubSurface where a live video is displayed. The upper Layer uses a Windowstack, where an application has opened a Window. The video is visible through the application Window, since the upper Layer uses a global alpha value.

3 DirectFB API Concept

The DirectFB API is structured using Interfaces. An Interface is a C structure that contains function pointers. Depending on the implementation of the Interface these pointers point to different core functions. A good example is IDirectFBSurface, which can represent the whole screen, the contents of a Window or a SubSurface. The application uses the same API and the same code can be used in each case.

3.1 The DirectFB Super Interface

IDirectFB is the Super Interface, which is the only one that can be created by a global function (DirectFBCreate()). All other Interfaces are reachable through this Interface. New Interfaces can be created by calling a function of the IDirectFB interface (e.g. IDirectFBSurface). Existing interfaces (like interfaces to existing input devices) can also be enumerated and accessed though IDirectFB.

3.2 Interface Diagram

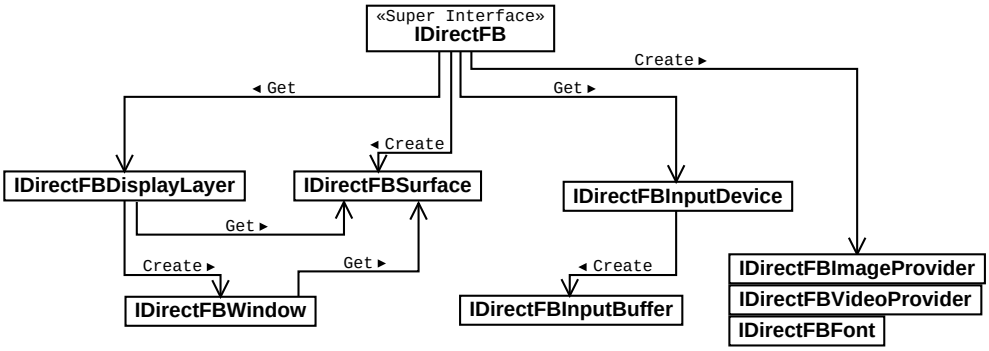

Figure 3: The relationship between DirectFB Interfaces

3.3 Loadable Modules

The following DirectFB Interfaces are implemented as loadable modules:

- IDirectFBImageProvider (see 1.2.6 for current implemtentations)
- IDirectFBVideoProvider (see 1.2.7 for current implementations)
- IDirectFBFont (see 1.2.8 for current implementations)

A specific implementation of these Interfaces is only loaded at run time if needed. For example if an image is loaded, the availible implementations of IDirectFBImageProvider will be probed. If the image is a PNG image the Probe() function of

the PNG module will notice that it can handle the file. Finding a suitable implementation for video formats works the same way.

- IDirectFBInputDevice (see 1.2.5 for current implementations)
- GfxCard (internal interface only, not accessible by applications, see 1.2.4 for supported cards)

Input and and graphics card drivers are loaded during initialization of DirectFB depending on the hardware configuration of the system DirectFB runs on.

It is possible to implement and use new graphics card drivers, input drivers and media providers without changing or recompiling DirectFB.

3.4 Example code

The following listing is a small complete DirectFB program that shows the usage of DirectFB Interfaces. It loads an image and displays it.

```
#include <directfb.h>

/* DirectFB interfaces */
IDirectFB               *dfb;
IDirectFBSurface        *primary;
IDirectFBImageProvider  *provider;

/* describes size, pixel format and capabilities of a surface*/
DFBSurfaceDescription dsc;

int main( int argc, char *argv[] )
{
     /* initalize DirectFB and pass arguments */
     DirectFBInit( &argc, &argv );

     /* create the super interface */
     DirectFBCreate( &dfb );

     /* set cooperative level to DFSCL_FULLSCREEN for exclusive access to the
        primary layer */
     dfb->SetCooperativeLevel( dfb, DFSCL_FULLSCREEN );

     /* get the primary surface, i.e. the surface of the primary layer we have
        exclusive access to */
     memset( &dsc, 0, sizeof(DFBSurfaceDescription) );
     dsc.flags = DSDESC_CAPS; /* only the caps field of the description is valid.
                                 height, width and pixelformat are not set, since
                                 the resolution and pixel are determined by the
                                 current video mode */
     dsc.caps = DSCAPS_PRIMARY; /* we want the primary surface (the whole screen) */

     /* Since we are in fullscreen mode no new surface will be created,
        the whole screen will be made accessible through a IDirectFBSurface    */
     dfb->CreateSurface( dfb, &dsc, &primary );

     /* create the image provider that will load the image */
     dfb->CreateImageProvider( dfb, "tux.png", &provider );

     /* render the image to the screen */
```

```
     provider->RenderTo( provider, primary );

     /* release the image provider, it is no longer necessary */
     provider->Release( provider );

     sleep (5);

     /* clean up */
     primary->Release( primary );
     dfb->Release( dfb );

}
```