

# Graphics in the Pedigree Operating System

Matthew Iselin

16/08/10

## 1 Introduction

First, please note that this is merely an RFC document and may not directly reflect the final state of this framework within Pedigree. Some features discussed in this document such as the Service mechanism may need explanation; however this is outside the scope of this document.

This document outlines the graphics framework that Pedigree utilises to centralise graphics device management and, more importantly, provide this centralisation in a way that allows applications and kernel modules to access these devices directly. Using this framework it is possible to create anything from complex user interfaces to basic 2D video games without needing to worry about a complex and confusing graphics interface. Additionally, libraries such as SDL could be modified to use this framework to enable a wide range of content-rich applications to be used in Pedigree.

The concept of a graphics framework is not a simple one. Such a framework must cater for applications with varying needs, continue to provide safe and secure access to the device itself, and somehow through all this run very, very fast. An application performing graphics operations at 60 frames per second has only 16 milliseconds per frame to completely render a screenful of information. The framework should not increase latency if possible to increase the amount of time per frame applications have to render objects. Of course, this framework must also be sensible, simplistic, and easy to plug in to existing and new code bases.

In Pedigree, the graphics framework handles both application and kernel requests for graphics devices. This is obviously not all contained within one kernel module; the application side is separate from the kernel side and as such is in its own custom module which can be kept unloaded in environments where application graphics are not required.

Existing applications such as the TUI utilised a (now obsolete and unsupported) graphics framework which can, unfortunately, be quite confusing and difficult to understand at times. The new graphics framework outlined in this document, most of which has been implemented, significantly reduced the amount of code in the TUI and paved the way for future advances in developing a GUI. Navigating graphics operations in the TUI has never been easier:

framebuffer changes are very easy to identify and their behaviour and expected result are both quite well defined.

## 2 Kernel Framework

The kernel side of the framework is the most important component; it is here device/driver selection (a graphics card with a single screen attached, herein referred to as a “rendering interface”) is done and where all central graphics calls eventually end up. The kernel framework is also designed to enforce the same mechanism for obtaining an interface for rendering in order to reduce the impact misbehaving modules may have on the rest of the system. In the kernel, the graphics framework utilises a Pedigree Service named “graphics” which simply implements methods for obtaining access to rendering interfaces, and for notifying the system of new rendering interfaces. See Section 4 for more information on the service itself.

## 3 Application Framework

The most fundamental part of this framework is the application interface to permit applications to make the most of the features provided by the framework. Without this application interface the framework is effectively useless; kernel modules rarely need, or care, about what can be displayed on the screen.

The basic protocol for obtaining a device to perform graphics operations with is as follows:

1. Applications request a device from the kernel using the **native subsystem**. This means at least part of every graphics application will be written in C++. The native subsystem wraps C++ syntax sugar over the **pedigree-c subsystem**, where all graphics system calls are handled. Using the C++ interface is the safest and easiest way to ensure compatibility with future revisions to the framework.
2. The native subsystem uses system calls to obtain information about the current graphics provider. The framebuffer of the provider is tracked locally and an application framebuffer and Display object is created.
3. Applications set screen modes and perform screen operations as needed.
4. Applications "close" the returned handle (effectively a delete operation which cleans up memory on the kernel side - effectively a "delete pHandle;", but applications should not arbitrarily delete handles returned by the native subsystem).

This is obviously quite a basic conceptual view of the framework that is only really suited to application developers. Proper understanding of the actual protocol between the kernel and the application that the pedigree-c subsystem defines should be documented somewhere? TODO?

TODO: Framebuffer chaining programming interface!

## 4 Graphics Service (exposed to applications)

The “graphics” service passes around rendering interfaces and information about them in a single structure. This structure defines a “Graphics Provider” (and is named accordingly), and is available to both the kernel and applications alike<sup>1</sup>. The structure is defined as follows:

- Display pointer
- This points to a Display object which provides information such as available screen modes, the current screen mode (if one is set), and information about the screen itself. This information might be data such as the manufacturer name or monitor model.
- Hardware acceleration capabilities
- Currently merely a Boolean value indicating whether at least one operation can be hardware accelerated on this graphics provider.
- Framebuffer pointer
- Highest level parent framebuffer for all rendering. It is expected that applications will create children of this framebuffer rather than utilise it directly.
- This framebuffer is allowed to point to memory that sits anywhere, or not point to memory at all if all operations can be performed without a known memory location. A raw buffer should **never** be utilised for rendering at any point to ensure code works on all systems.
- Provider maximum resolution and bit depth
- Mainly used for the provider points system when selecting a provider, but can be used in kernel or application code if necessary.

The two features the graphics service provides via the Service interface are the **touch** and **probe** features. The touch feature is used to notify the system of a new graphics provider and request a refresh of the provider points across the system.

---

<sup>1</sup>Applications have a reduced information set in their structure: some pointers are non-typed, for example.

## 4.1 Provider Selection Algorithm

In order to select the best provider for the rest of the system to use, an algorithm is used to assign “points” to a provider. The provider with the most points after all have been checked is selected as the “provider of choice” and is used for all future probe calls until a new “best choice” is found.

The algorithm for calculating points is as follows:

$$p = (w \times h \times b) + (16384 \times 16384 \times 32 \times a)$$

Here,  $w$  and  $h$  refer to the maximum horizontal and vertical resolution, respectively, the provider is capable of. The maximum bit depth the provider is capable of is referred to as  $b$ . Variable  $a$  is a one when hardware acceleration of any sort is available from this provider, and zero otherwise.

An example output of the algorithm in use in the graphics service with two graphics cards in a system is shown below:

```
(DD) GraphicsService: provider with display name
'Generic Display' got 0x1200000 points [new best choice]
(DD) GraphicsService: provider with display name
'vmware-gfx' got 0x2009088000 points [new best choice]
```

## 5 Framebuffer Chaining

There is one important concept that this framework in its current state does not address: chaining of framebuffers. It is crucial to adequately define the concept of “framebuffer chaining” in order to lay a foundation to build further framebuffer concepts on. Framebuffer chaining, in the context of this framework, is the concept of linking multiple framebuffers together in a parent/child relationship, where child framebuffers ask their parent framebuffer to perform an operation. This still creates a speedy routine for hardware-accelerated blits, as these can be directed up the chain with mere modifications to X and Y coordinates. Per the design of this framework, all framebuffers share a common pixel format, and therefore there is little need for a conversion to be done at any point.

The following table shows how a graphical user interface might utilise framebuffer chaining:

Layer	Category at this Layer
Highest (1)	Graphics providers. Never have parents.
2	Management: window managers or full user interface parents
3	Global parent rendering: “desktop” rendering, parent terminal rendering, full-screen applications
4	Global child rendering: subdivision of global parent into window areas (menus, controls, terminals)
Lowest (5)	Subwindowing: subdivision of window areas (multiple document interfaces)

Using the Windows user interface as an example, these layers can be easily visualised like so:

1. Graphics drivers
2. Window manager. Allocating screen space for child windows.
3. Parent rendering. Desktop, Start Menu, taskbar.
4. Child rendering. Individual windows, controls (text boxes, buttons), menus and icons.
5. Subwindowing. Multiple document interfaces, controls as children of other controls. Group boxes containing controls.

The following list shows the logic process for drawing an image from an application and how the draw is passed through multiple layers within the chain:

1. Perform a blit operation on the application framebuffer.
2. This blit is passed to the parent framebuffer. Here, its destination X and Y co-ordinates are adjusted before passing the buffer on to its parent.
3. Until the highest level is reached, repeat step 2.
4. At the highest level (device), perform the blit. If possible, it will be accelerated here. To this point, the only operations that have needed to happen have been additions and subtractions; the buffer to blit has been able to remain untouched through every layer (note however this is a “perfect” concept; in reality at least one copy will be done to update a local buffer - see below).

As this diagram clearly shows, the actual rendering procedure is still performed by the highest layer: the device. In this design costly operations are performed at the layer where they can be accelerated.

One of the most appealing bonuses to framebuffer chaining is the ability to subdivide parent framebuffers when creating children, and to also enforce a strict order when displaying content.

This means applications can be given a heavily-restricted framebuffer to do graphics operations within without directly affecting the entire screen. Furthermore, applications such as the terminals which start applications that draw to the screen can regain control of the area they release to children by simply performing a redraw; these are generally not performed while an application is running within a terminal. The redraw determines a child framebuffer is, or was, using an area within the redraw rectangle and gives the parent priority over any child framebuffer. This typically involves at least one copy operation in order to perform a draw to the parent framebuffer, which could end up being a high-latency operation if the child is more than few levels down the framebuffer chain. As such, this kind of operation should be used as little as possible - for example, a terminal which runs applications should only redraw once an application no longer has input focus.

Without actually visualising code to do this it can be very difficult to actually perceive how this works. The following Python code showing the algorithm should clear up any confusion:

```
def redraw(x, y, w, h):
    redraw_rect = [x, y, x + w, y + h]

    for region in child_regions:
        # Overlap detection
        if overlap(redraw_rect, region):

            # Overlap found, copy our own framebuffer to
            # the parent's. This will propagate up the
            # chain, but will overwrite the child's
            # framebuffer data.
            parent.draw(ourBuffer, region)
```

Subdivision is enforced by using a completely different allocated buffer per defined framebuffer. This does mean that each operation on a framebuffer will directly cause a modification to memory as well as a graphics operation passed up the chain; unfortunately this is a side-effect of the parent redraw algorithm and cannot be avoided. Using a different memory area per framebuffer also allows framebuffers to be defined in application memory, on the application heap, keeping framebuffers well away from kernel data structures. This safety benefit is considered significant enough to outweigh the potential performance loss.

The actual programming interface for framebuffer chaining is defined in Section 3.