Reprinted from the

Proceedings of the Linux Symposium

July 23rd–26th, 2008
Ottawa, Ontario
Canada

Coding Eye-Candy for Portable Linux Devices

Bob Murphy
ACCESS Systems Americas, Inc.
Bob.Murphy@access-company.com

Abstract

Linux is increasingly being used in portable devices with unusual hardware display architectures. For instance, recent OMAP and XScale CPUs support multiple frame-buffer "overlays" whose contents can be combined to form what the user sees on the screen.

As part of the ACCESS Linux Platform, ALP, ACCESS has developed a set of kernel modifications, an X extension and driver, and additions to the GDK/GTK stack to take advantage of the XScale CPU's three-overlay architecture. This paper provides an overview of these modifications and how they have been used to achieve a variety of "eye-candy" features, such as GTK widgets that float translucently over an actively playing video.

1 Introduction

The evolution of display systems for computer graphics has included a wide variety of features and architectures. Some systems, like the vector displays used by Ivan Sutherland in 1963, have largely wound up in the dustbin of history. For the last twenty years, framebuffer-based raster displays have dominated the industry.

However, standardization on raster displays does not mean progress has stood still. Early framebuffer systems used simple video RAM that corresponded to the screen, and programs could modify what was displayed by poking into memory-mapped addresses. Nowadays, nobody would consider a desktop system that didn't include a framebuffer card with enough RAM to permit page flipping, and a floating point graphics processing unit to accelerate 3D rendering using systems like OpenGL.

In embedded devices such as cell phones, framebuffers and related graphics hardware are often built into the CPU or related chips. Cell phone users often don't care much about 3D first-person shooters, but they do want to see photos and videos, and they want eye candy.

Cell phone manufacturers have gone to great lengths to support eye candy, such as adding GPUs and using CPUs with vector integer capabilities. In fact, the iPhone is reputed to use a CPU[1] that has a vector floating point coprocessor, which would allow a straightforward port of the Quartz graphics system used in Mac OS X.

As a response, some embedded CPU vendors have begun to support **video overlays**. These are multiple framebuffers whose contents can be programmatically combined to create the image the user sees. Video overlays can allow developers to provide a wide variety of eye candy.

2 Video Overlays: Hardware Examples

2.1 TI OMAP

Texas Instruments' OMAP 3430 rev. 2 is an ARM CPU with three video overlay framebuffers, as shown in Table 1.[2]

Overlay	Pixel Formats
Graphics	Palette RGB, direct RGB, and RGB with alpha channel
Video 1	Direct RGB and YCbCr
Video 2	Direct RGB and YCbCr

Table 1: OMAP 3430 Overlays

Many common compressed raster formats, such as MPEG, QuickTime, and JPEG, store images or frames

[1]A chip based on the ARM1176JZF design.

[2]The OMAP display architecture has other options and modes not covered in this discussion.

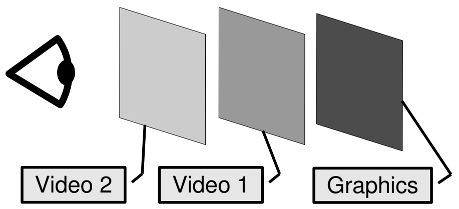

Figure 1: OMAP Normal Mode Overlays

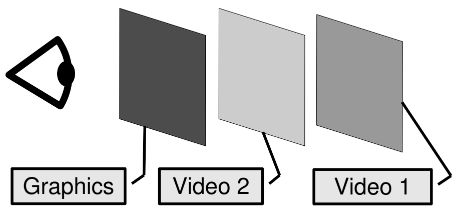

Figure 2: OMAP Alpha Mode Overlays

in a YCbCr color space. YCbCr encodes colors as a combination of *luminance* (brightness) and *chrominance* (hue) values. The human eye is more sensitive to luminance than chrominance, which makes YCbCr suitable for image compression. A codec can compress an image's chrominance and luminance separately, and apply a higher degree of compression to the chrominance values without much perceptual difference.

However, JPEG rendering and MPEG playback usually involve not only image decompression, but pixel color space conversion from YCbCr to RGB. The OMAP chip's direct hardware support for YCbCr formats in the Video 1 and Video 2 overlays allow high frame rates for image and video display with lower CPU effort.

The OMAP chip supports two overlay layering modes: *normal* and *alpha*.

In normal mode, the frame buffers are stacked as shown in Figure 1 . A simple use of this mode would be to treat the RGB-based graphics layer as a normal framebuffer, and map X11 into it. A developer could then display video or pictures "on top" of the X11 layer using one or both of the video overlays.

In alpha mode, the frame buffers are stacked as shown in Figure 2. This places the video layers "under" the graphics layer. Alpha mode is more interesting from an eye candy perspective because any alpha channel information in the graphics overlay is applied as color blending. When X11 is mapped onto the graphics layer, programs can not only "punch holes" in the X11 layer to display video from one of the underlying overlays, but they can have parts of the X11 layer "float translucently" on top of a playing video.

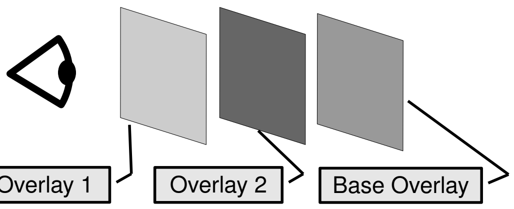

Figure 3: XScale Overlays

2.2 Marvell XScale

Beginning with the PXA270, the XScale ARM CPUs have offered a version of overlays that is generally similar to the OMAP alpha mode, but quite different in detail. These chips provide three overlay frame buffers, stacked as shown in Figure 3, and described in Table 2.[3]

Overlay	Pixel Formats
Overlay 1	Direct RGB, and RGB with T bit
Overlay 2	Direct RGB and YCbCr
Base Overlay	Palette RGB and direct RGB

Table 2: XScale PXA270 and PXA3XX Overlays

A simple use of this architecture would be to disable Overlays 1 and 2, set the base overlay to a direct RGB format, and map X11 to it. This would act like a very standard framebuffer. And as with OMAP, the XScale Overlay 2 is optimized for video and pictures due to its YCbCr pixel format options.

But the T bit in Overlay 1 is rather odd: it is an alpha channel of sorts, but not a normal one. A platform-wide flag in hardware activates or deactivates its effects. And

[3]The XScale display architecture also has other options and modes not covered in this discussion.

T Bit	Pixel RGB	Effect At That Pixel
0	Any	Overlay 1 is opaque: only the Overlay 1 color appears; no contribution comes from either of the other two overlays
1	Non-black	Overlay 1 is color-blended: the RGB color from Overlay 1 for that pixel is color-blended with the colors for that pixel from the topmost active underlying overlay, using a platform-wide alpha value that can be varied from 0.125 to 1.0 in 0.125 unit increments
1	Black	Overlay 1 is transparent: the pixel color from the topmost active underlying overlay is used

Table 3: T Bit Effect On A Pixel

if it is active, this one bit can have three effects on a per-pixel basis, as described in Table 3.

ALP maps X11 to Overlay 1, and displays video and pictures in Overlay 2. This permits GTK widgets that "float translucently" over pictures and playing video. ALP also can disable Overlay 2, and use the T bit in conjunction with the base overlay to create a variety of RGB-based eye candy effects.

3 Eye Candy Implementation in ALP

ACCESS engineers call alpha blending using the topmost hardware overlay **Overlay Transparency**, usually abbreviated **OT**, and distinguish three modes for each pixel, paralleling the XScale modes:

- **OT opaque** means the user sees only the topmost, X11 overlay's color.
- **OT transparent** means the topmost, X11 overlay contributes nothing. The user sees only the color from an underlying overlay, such as one playing video.
- **OT blended** means the user sees a color blended from the color in the X11 overlay and one of the underlying overlays.

Figure 4 summarizes the implementation of OT within the ALP platform. A typical OT-aware application will call into both GTK+ and the ALP OT API. The OT implementation, in turn, affects the behavior and/or implementation of other components in the graphics pipeline, including GTK+, GDK, X11, and the kernel video device drivers.

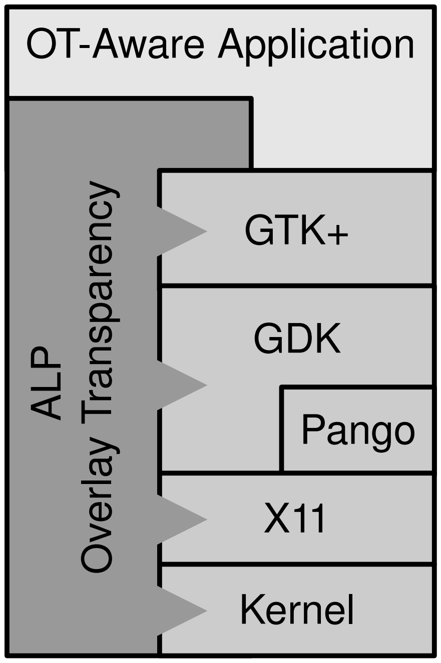

Figure 4: ALP Overlay Transparency Component Stack

3.1 Developer API

Developers can apply OT features using a GTK-based API built into the ALP platform. The API paradigm is that developers can set an 8-bit alpha value for various parts of widgets that is applied as OT, or specify that the alpha channel in a raster image be applied as OT. This makes the API portable while insulating developers from needing to know anything about the underlying hardware. Some hardware, such as OMAP, can support this paradigm directly. Other hardware, such as XScale, do not, but in such cases, the hardware-dependent implementation is written to "do the right thing" and come as close as possible to matching that paradigm.

The API provides several levels of calls:

- **Platform calls** let programs query the system to determine things like whether it supports OT at all.
- **Window calls** let programs enable or disable OT support for an entire GtkWindow and its child widgets. When OT support is disabled for a GtkWindow, it and all its child widgets are OT opaque, no matter what other OT settings have been applied to those widgets. There is also an option to indicate how the window manager should apply OT to the window frame.
- **Widget calls** let a program apply an 8-bit alpha value to an individual widget's background or text. They can also determine whether a widget containing a raster image with an alpha channel will have those alpha values applied to OT.
- **Container calls** parallel the widget calls, and let a program set default values for child widgets of a container. For instance, a program could apply one to a container to set a single OT background alpha value for all child widgets of a container. Programs can override these default values for individual widgets using the widget calls.

Figure 5 shows the widget portion of the API. Using an enum-based approach makes the API readily extensible, and provides backward compatibility since the system ignores enum values it does not recognize.

3.2 GTK/GDK and Platform Support

ACCESS has added OT support to ALP without any changes to GTK or GDK source code; instead, all support at the GDK level and above is written in platform-specific code. ALP applications must call a platform-specific initialization routine, which includes (among other things) a variety of OT-related run-time modifications to GTK and GDK behaviors.

When an application calls one of the API routines on a GTK widget, container, or window, the routine attaches OT-related state information to that GTK object using `g_object_set_data()`. Later, when a widget receives an expose event and draws itself, the OT code applies that state information, climbing the container hierarchy to determine default values if needed. Currently, ALP applies OT alpha values to portions of widgets as follows:

- **Background alpha** is applied to the widget's entire allocation.
- **Text alpha** is applied to the pixels actually rendered as text.
- **Foreground alpha** determines whether a raster image's alpha channel is applied for OT purposes.

3.3 Xtransp Extension

ALP includes a new X extension called **Xtransp**, which provides an interface to the OT features in the X server, as summarized in Table 4. Xtransp provides OT features on a per-Window basis, as well as calls that apply to the entire screen or system, and override any Window-based values to permit effects such as app transitions[4].

Inside the X server, Xtransp uses the devPrivates mechanism to add OT-related information to X Screens and Windows, including state information and an OT alpha array. It also overrides several Screen methods for window handling and drawing. Xtransp always keeps the screen consistent with OT changes to part of a Window or Screen, by adding that area to the Screen's damage region.

In particular, Xtransp head-patches the Screen's `PaintWindowBackground()` method, so that when a Window is painted, its OT alpha information is copied to the Screen.

3.4 X Server

3.4.1 Extensions

The OT code in the X server works in conjunction with several X extensions to provide features useful for cell phones and other portable devices:

[4]The routines that set blending values are useful with XScale hardware, which does not support different alpha blending values on different pixels, but not with OMAP, which does.

```
enum _AlpVidOvlWidgetFeatures
{
    // Widget alpha values; values range 0-255
    ALP_VIDOVL_FTR_WIDGET_BG_ALPHA = 0x05000000,    // Background
    ALP_VIDOVL_FTR_WIDGET_FG_ALPHA,                 // Foreground
    ALP_VIDOVL_FTR_WIDGET_TEXT_ALPHA                // Text
};

/*
Get a feature value for a GtkWidget
[in] widget   A GtkWidget
[in] selector A feature code from _AlpVidOvlWidgetFeatures
[in] outValue The value to be retrieved
return        An error code, or ALP_STATUS_OK if no error

If the widget does not have an explicitly-set value, this will return
the corresponding default value from the widget's container stack.
*/
alp_status_t    alp_vidovl_widget_get_feature(GtkWidget *widget,
                    guint32 selector, guint32 *outValue);

/*
Set a feature value for a GtkWidget
[in] widget   A GtkWidget
[in] selector A feature code from _AlpVidOvlWidgetFeatures
[in] inValue  The value to be set for the feature
return        An error code, or ALP_STATUS_OK if no error
*/
alp_status_t    alp_vidovl_widget_set_feature(GtkWidget *widget,
                    guint32 selector, guint32 inValue);
```

Figure 5: Typical ALP Overlay Transparency API

- The **Shape** extension, which supports non-rectanglar Windows, also limits OT alpha value application to Windows. That permits features such as translucent dialogs and menus with rounded or bevelled corners.

- Using the **Shadow** extension simplifies the platform-dependent driver architecture.

- The OT-oriented hardware drivers support the **RandR** extension, so that portable device displays can be rotated. This is common in cell phones, where phone dialing is usually done in portrait mode, but camera features are used in landscape mode.

3.4.2 PXA3XX Video Driver

A new video driver for the PXA3XX series CPUs[5] provides the lowest level of support for OT in the X server. This is a kdrive driver that was developed from Keith Packard's fbdev driver. The changes to fbdev include things one would expect, such as setting up and maintaining hardware-specific states, and limiting pixel formats to those the hardware supports.

The heart of the driver changes are in the shadowbuffer-to-framebuffer blitter, which is where per-pixel OT alpha values in the 0-255 range are converted to the three states the XScale hardware supports. Its behavior is summarized in Table 5.[6]

[5]These are the PXA300, PXA310, and PXA320 XScale ARM CPUs from Marvell, code-named **Monahans**. The driver also sup-

Window Function	**Effect**
XOverlayTransparencySetWindowAlpha	Enable or disable OT support in a given X Window, and set its behavior toward child Windows.
XOverlayTransparencyUpdateWindowAlphaMap	Apply an array of 8-bit alpha values to a rectangle in a Window.
XOverlayTransparencySetBlending	Set a single set of RGB blending values to apply, platform-wide, to all Window-based operations.
Screen or System Function	**Effect**
XOverlayTransparencySetScreenAlpha	Apply a single alpha value to a rectangle on the screen, and disable any Window-based OT features.
XOverlayTransparencyDisableScreenAlpha	Disable the effect of XOverlayTransparency-SetScreenAlpha, and re-enable any Window-based OT features.
XOverlayTransparencySetScreenBlending	Set a single set of RGB blending values for screen-based operations.

Table 4: Xtransp API Summary

Alpha	Effect	32-bit TRGB Result
0	Transparent	0x01000000
1-254	Color-blended	0x01RRGGBB
255	Opaque	0x00RRGGBB

Table 5: Alpha Translation for XScale

3.4.3 Emulation in Xephyr

Embedded systems are notoriously difficult to develop for: programmers require working hardware, and then must use a cross-compiler, and flash or otherwise transfer executables to the device.

To speed development, the ALP Development Suite includes the *ALP Simulator*: an x86-native version of ALP that runs under User Mode Linux and mimics a device display via Xephyr. This lets in-house and third-party developers write and test code quickly on a Linux x86-based host computer. Then, when code works well on the host, they can cross-compile and transfer code for testing on an ARM device.

The ALP Simulator provides limited support for OT via changes to Xephyr, primarily replacing the `shadowUpdateRotatePacked()` blitter with one that is OT-aware. Since this is intended for initial development and testing, no attempt is made to simulate the XScale or OMAP video or base overlays. Instead, pixels that would be OT transparent are rendered as light aqua, and pixels that would be OT blended are rendered as a 50% average with light aqua. This lets developers quickly see whether their use of the GTK-based OT API is generally correct.

ports the PXA270 CPU, which shares the same overlay architecture.

[6]When color-blending, black is converted to 0x01000001 (very slightly blue) to avoid accidental transparency.

3.5 Kernel

The kernel sources require very few changes to support OT for XScale. They largely fall into three groups:

- A **new ioctl** lets the X server set the platform-wide color blending level that is used when a pixel has the T bit set and the pixel is not black.
- The video drivers include **pxafb_overlay.c**, an Intel-developed open-source driver for XScale overlays 1 and 2 (`/dev/fb1` and `/dev/fb2`).
- The open-source **pxafb video driver** for the base overlay (`/dev/fb0`) includes support for new pixel formats and other features supported by the XScale PXA270 and PXA3XX chips.

3.6 Video Playback

The current version of ALP includes a GStreamer-based media framework that plays video by activating XScale Overlay 2 as a YCbCr framebuffer, and blitting frames to it. Media playback occurs in a separate thread of execution, so while GTK-based programs can start and stop video playback, they otherwise proceed independently of the media framework.

4 Results

4.1 Transition Effects

One common form of eye candy is transition effects: items sliding on and off screen, zooming around, changing colors, and fading in and out.

The ALP platform achieves some of its transition effects by a combination of OT features in the X11 overlay and raster images in the base overlay. For instance, fade-in at app launch is done in several steps:

1. Fill the bottom (base) overlay with a solid color, typically a light gray.
2. Use the OT screen feature to make the entire X11 (top) overlay transparent. This makes the whole screen light gray.
3. When the app has finished displaying its user interface, gradually increase the opacity of the X11 overlay until it is fully opaque.

4.2 Floating Widgets

Another fun form of eye candy is translucency. ALP's OT implementation allows widgets to float transparently or translucently above playing video. Figures 6 through 11 show this in action. These are screen shots from an XScale development system, showing a sample application which plays a Hawaii travelogue video in the background on Overlay 2.

In Figure 6, the sample application is shown at entry. The main window and all its child widgets have been initialized with a variety of OT alpha values. However, OT support has not been activated on the main window, so it and its child widgets remain opaque. Although the

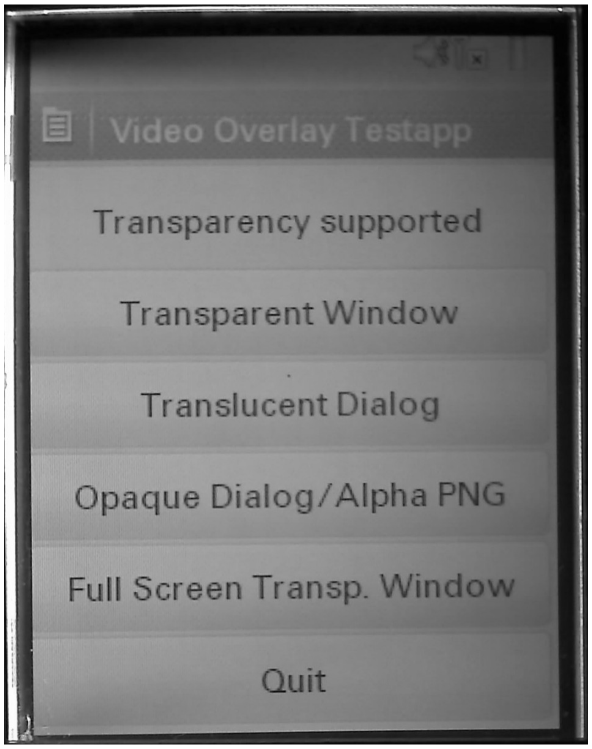

Figure 6: Application on entry

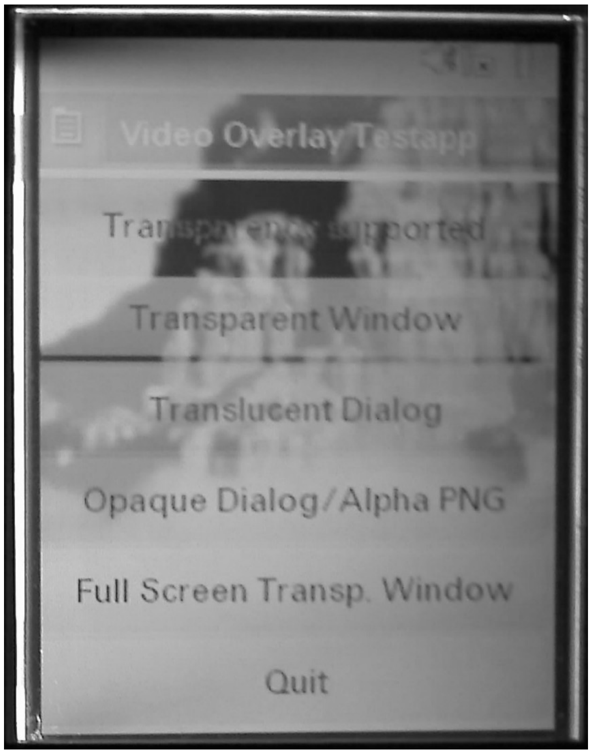

Figure 7: After pressing the *Transparent Window* button

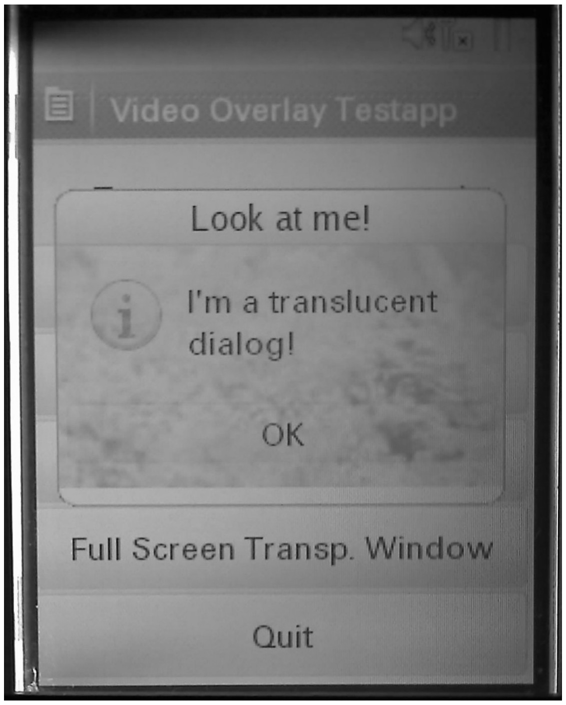

Figure 8: After pressing the *Translucent Dialog* button

video is playing in the background on Overlay 2, none of it is visible.

Pressing the *Transparent Window* button toggles OT support on the main window; Figure 7 shows the result. The playing video is visible through the window's transparent background and the translucent buttons; however, all the text is opaque. Above the main application window, the status bar remains opaque because it belongs to a different process.

Figure 8 shows what happens after toggling the main window's OT support off, and then pressing the *Translucent Dialog* button. The dialog contents, except for the opaque text, are translucent. However, the dialog frame has been set to opaque.

After dismissing the dialog and pressing the *Full Screen Transp. Window* button, Figure 9 shows a full-screen transparent window. In the center, there is a translucent button with opaque text.

Figure 10 shows the same *Transparent Window* result as Figure 7, with an opaque menu on top. The menu's rounded corners show the effect of the Shape extension on OT.[7]

[7]The screenshot shows a menu layout bug; the system was still under development when this paper was written.

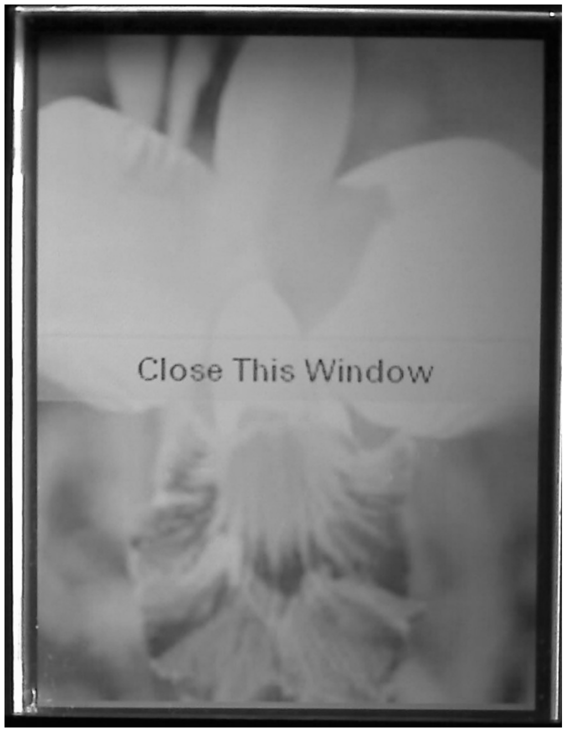

Figure 9: After pressing the *Full Screen Transp. Window* button

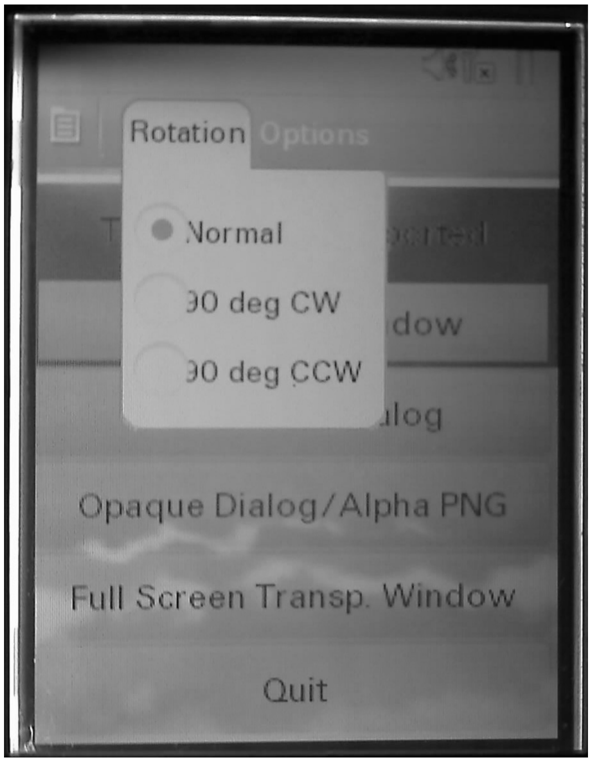

Figure 10: Opaque menu over transparent / translucent widgets

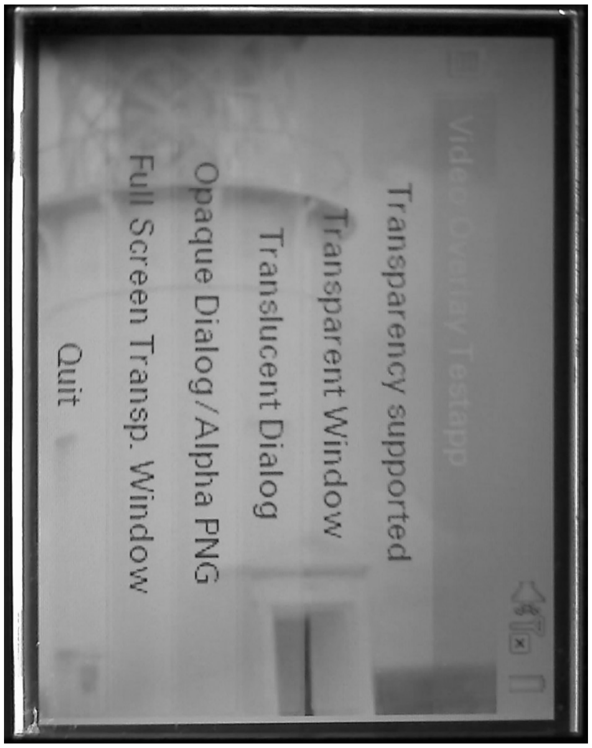

Figure 11: Landscape Mode

After selecting *90 deg CCW* from the menu in Figure 10, Figure 11 shows the result of rotating the entire X layer to landscape mode, while leaving the video playing in portrait mode.

5 References

Texas Instruments, *OMAP3430 Multimedia Device Silicon Revision 2.0 Technical Reference Manual*, 2007.

Marvell, *Monahans L Processor and Monahans LV Processor Developers Manual*, 2006.