Who Gives a Blit? Part II

Authored by: **André Wösten**
Creation date: **May 24, 2004**
Last Edited: **February, 2008.**

Table of Contents

1. About the author

André Wösten is a skilled developer with over nine years of C/C++ experience, as well as considerable knowledge in the area of graphics and GUI programming. Currently, he works for **pulskraft IT & Media Solutions** (http://www.pulskraft.de); an organization that deals with graphics and design, web development, and system integration.

2. Introduction

I wrote this article as a sequel to "Who gives a blit?" by John Fehr, who discussed different blitting methods, including shared and video-memory blit. I recommend that you read this article before continuing.

At first, we will discuss the purpose and advantages of video overlay compared to the other methods, followed by the development of a small engine, directly starting with a "video-overlay-in-a-window" mode.

After reviewing this article, you will be able to extend the engine by a full screen feature and tiny, cute, moving stickmen (I hope so :-)).

Before you start coding your engine, review the QNX documentation for your specific graphics driver (*devg-*.so*), and determine whether it supports video overlay.

Note: Everything written and coded in this article has been done using QNX Neutrino Core 6.2.1. Have fun!

3. What is video overlay?

Video overlay it is a special output mode independent of the normal scope presentation that provides a straightforward rendering, which is assisted by hardware functions.

As the name implies, the video buffer is overlayed by an additional video signal (which can have different formats and features described in the QNX Documentation, **PgScalerProps_t**). Imagine that you have a piece of paper that you place over your monitor - that's a video overlay; it is **not** changing the contents of

the primary surface. You can even draw with the so-called Chroma-key color on your primary surface and it will just overlay pixels with the Chroma-key color.

Currently, QNX supports "Scaler Channels", which means that you are defining the source dimensions of the buffer and the viewport you want to draw in. The scaler will automatically scale your buffer into the viewport using hardware- level functions, which turn out to be very fast.

4. How to use video overlay

At first, we have to create and configure a channel to the scaler hardware, which is made possible by following functions:

```
-   PgVideoChannel_t*   PgCreateVideoChannel (unsigned type, unsigned flags);
-   int  PgConfigScalerChannel (PgVideoChannel_t *channel, PgScalerProps_t *props);
```

The first function will create a channel whose type is specified by 'type'; at the moment, only the scaler channel (**Pg_VIDEO_CHANNEL_SCALER**) is supported.

Now, we have to configure this channel - important properties are the source and destination dimensions, data format, and flags. Since we want to build a window mode, we have to react to events, such as "user-moves-the-window", which requires a reconfigure of the viewport, or "user-hides-the-window", which requires the disabling of the scaler output.

The result is placed in a separated function, which is called with new parameters for all required events. Once the scaler is configured, it provides pointers to off-screen context structures, where we can write our data. All of these steps are defined in **gfx_init**, which calls **config_scaler_channel**.

In the next step, we write a drawing function. Since we're using double buffering, we use **yplane1** and **yplane2** of the video channel structure (**PgVideoChannel_t**), which represent the primary and secondary video buffers. An index determined by **PgNextVideoFrame** chooses the next video buffer to fill. Generally, this is used to support double-buffering.

5. Interaction between the window and overlay

The engine runs two concurrent threads: the first one performs the major tasks, calculating and drawing; while the second one runs the Photon **Mainloop**, which is mainly responsible for window/event-handling, further reconfiguring the scaler channel.

To handle the window messages, we have to assign a callback to the window, which handles following messages:

- **Show overlay**:

  Ph_WM_MOVE, Ph_WM_TOFRONT,
  Ph_WM_RESTORE,
  Ph_WM_FOCUS+Ph_WM_EVSTATE_FOCUS,
  Ph_WM_HIDE+Ph_WM_EVSTATE_UNHIDE

- **Hide overlay**:

  Ph_WM_TOBACK, Ph_WM_BACKDROP,
  Ph_WM_FOCUS+Ph_WM_EVSTATE_FOCUSLOST,
  Ph_WM_HIDE+Ph_WM_EVSTATE_HIDE

6. Example

Note: For this example to function properly, you will need to use a layer capable driver.

```
// ------------------------------------------------------------------------
// QNX Graphics Framework using Video Overlay
// coded by André Wösten
// ------------------------------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <math.h>
// ------------------------------------------------------------------------
#include <Pt.h>
#include <sys/neutrino.h>
// ------------------------------------------------------------------------
#define RX     340
#define RY     320
#define VIDEO_FORMAT Pg_VIDEO_FORMAT_RGB565
#define PIXEL_SIZE sizeof(short)
// ------------------------------------------------------------------------
PtWidget_t     *window = NULL;
pthread_t      *thread = NULL;
PgVideoChannel_t *vidchan = NULL;
PhDim_t         dim;

// ------------------------------------------------------------------------
int
config_scaler_channel( PhPoint_t * position, PhDim_t * dimension, int show )
{
        PhRect_t        rect_window, rect_border;
        PgScalerProps_t scalerprops;

        PtCalcCanvas( window, &rect_window );
        PtWindowFrameSize( NULL, window, &rect_border );

        /* Calculate new viewport */
        rect_window.ul.x += position->x + rect_border.ul.x;
        rect_window.ul.y += position->y + rect_border.ul.y;
        rect_window.lr.x += position->x + rect_border.ul.x;
        rect_window.lr.y += position->y + rect_border.ul.y;

        scalerprops.size = sizeof( PgScalerProps_t );
        scalerprops.flags = Pg_SCALER_PROP_DOUBLE_BUFFER |   /* Enable double buffer -
faster and
doesn't flicker - refer to 'Page flipping' */
                ( show ? Pg_SCALER_PROP_SCALER_ENABLE : 0 ); /* En-/Disable scaler */
```

```
        scalerprops.format = VIDEO_FORMAT;
        scalerprops.viewport = rect_window;
        scalerprops.src_dim = *dimension;

        if( ( PgConfigScalerChannel( vidchan, &scalerprops ) ) == -1 ) {
                perror( "Unable to configure scaler channel" );
                return 1;
        }

        return 0;
}

// ------------------------------------------------------------------------
int
wnd_callback( PtWidget_t * widget, void *data, PtCallbackInfo_t * cbinfo )
{
        PhWindowEvent_t *evt = cbinfo->cbdata;
        static PhPoint_t pos;

        switch ( evt->event_f ) {
        case Ph_WM_BACKDROP:
        case Ph_WM_MOVE:
        case Ph_WM_RESTORE:
                pos = evt->pos;
                break;
        }

        switch ( evt->event_f ) {
        case Ph_WM_MOVE:
        case Ph_WM_TOFRONT:
        case Ph_WM_RESTORE:
                config_scaler_channel( &pos, &dim, 1 );
                break;
        case Ph_WM_TOBACK:
        case Ph_WM_BACKDROP:
                config_scaler_channel( &pos, &dim, 0 );
                break;
        case Ph_WM_HIDE:
                config_scaler_channel( &pos, &dim, ( evt->event_state ==
Ph_WM_EVSTATE_HIDE ) ? 0 : 1 );
                break;
        }

        return Pt_CONTINUE;
}

// ------------------------------------------------------------------------

int
gfx_init( PhPoint_t * position, unsigned int width, unsigned int height )
{
        PtArg_t         args[5];
        PtCallback_t    movecb = { wnd_callback, 0 };

        /* Initialize Widget Library */
        if( PtInit( NULL ) == -1 ) {
                perror( "Could not initialize Widget Library" );
                return 1;
        }

        /* Create scaler channel */
        if( ( vidchan = PgCreateVideoChannel( Pg_VIDEO_CHANNEL_SCALER, 0 ) ) == NULL ) {
                perror( "Error while creating video channel" );
                return 1;
        }

        dim.w = width, dim.h = height;

        /* Prepare resources for window */
        PtSetArg( &args[0], Pt_ARG_POS, position, 0 );
        PtSetArg( &args[1], Pt_ARG_DIM, &dim, 0 );
```

```
        PtSetArg( &args[2], Pt_CB_WINDOW, &movecb, 1 );
        PtSetArg( &args[3], Pt_ARG_WINDOW_RENDER_FLAGS, Ph_WM_RENDER_BORDER |
Ph_WM_RENDER_TITLE | Ph_WM_RENDER_CLOSE, ~0 );
        PtSetArg( &args[4], Pt_ARG_WINDOW_NOTIFY_FLAGS,
                        Ph_WM_BACKDROP | Ph_WM_MOVE | Ph_WM_RESTORE | Ph_WM_TOFRONT |
Ph_WM_FOCUS | Ph_WM_HIDE | Ph_WM_CLOSE, ~0 );

        /* Create and realize widget */
        window = PtCreateWidget( PtWindow, Pt_NO_PARENT, 5, args );
        PtRealizeWidget( window );

        if( ( config_scaler_channel( position, &dim, 1 ) ) != 0 ) {
                fprintf( stderr, "Error while configurating scaler channel" );
                return 1;
        }

        return 0;
}

int
gfx_draw( PgColor_t color )
{
        int index = PgNextVideoFrame( vidchan );
        PdOffscreenContext_t *dest = NULL;
        PhDrawContext_t * old;

        if( index == -1 ) {
                perror( "Ouch! Could not determine next frame." );
                return 1;
        }

        dest = ( index ) ? vidchan->yplane1 : vidchan->yplane2;
        old = PhDCSetCurrent(dest);

        PgSetFillColor(color);
        PgDrawIRect(0, 0, dest->dim.w - 1, dest->dim.h - 1, Pg_DRAW_FILL);
        PgFlush();
        PhDCSetCurrent(old);

        return 0;
}

int work_proc(void * data)
{
        PgColor_t val = rand();

        if( gfx_draw( val ) ) {
                fprintf( stderr, "gfx_draw() failed.\n" );
                return Pt_END;
        }

        return Pt_CONTINUE;
}

// ------------------------------------------------------------------------
int
main( int argc, char *argv[] )
{
        int             ret;
        PhPoint_t       pos = { 100, 100 };

        if( ( ret = gfx_init( &pos, RX, RY ) ) != 0 ) {
                fprintf( stderr, "gfx_init(): error (%d)\n", ret );
                return 1;
        }

        PtAppAddWorkProc( NULL, work_proc, NULL );
        PtMainLoop();

        return 0;
}
```

```
// ------------------------------------------------------------------------
```

7. Conclusion

I hope you obtained a short overview about video overlay in conjunction with graphic engines. Since video overlay typically functions at the hardware level, it is the fastest technique for blitting video data to the screen.

If windows overlap each other, you might want to review the QNX documentation and try Chroma keying if you want, for example, to show parts of your overlay on the screen.

While video overlay will continue to function in Photon version 6.4, new users should consider looking at GF because it provides more flexibility and control for YUV overlay surfaces within the GF layer API.

If you have further questions or comments, you can contact me by e-mail at andre.woesten@pulskraft.de