Design and implementation of EZoom a zoom utility for Sony Ericsson P800/P900

Peter van Sebille
Revision 1.0, March 2004

Table of contents

Introduction

EZoom is a utility that allows you to zoom in on the entire screen of a Sony Ericsson P800 and P900. It works transparently for all applications apart from the on-board camera viewfinder. Figure 1 shows EZoom running under the emulator zooming in 400%.

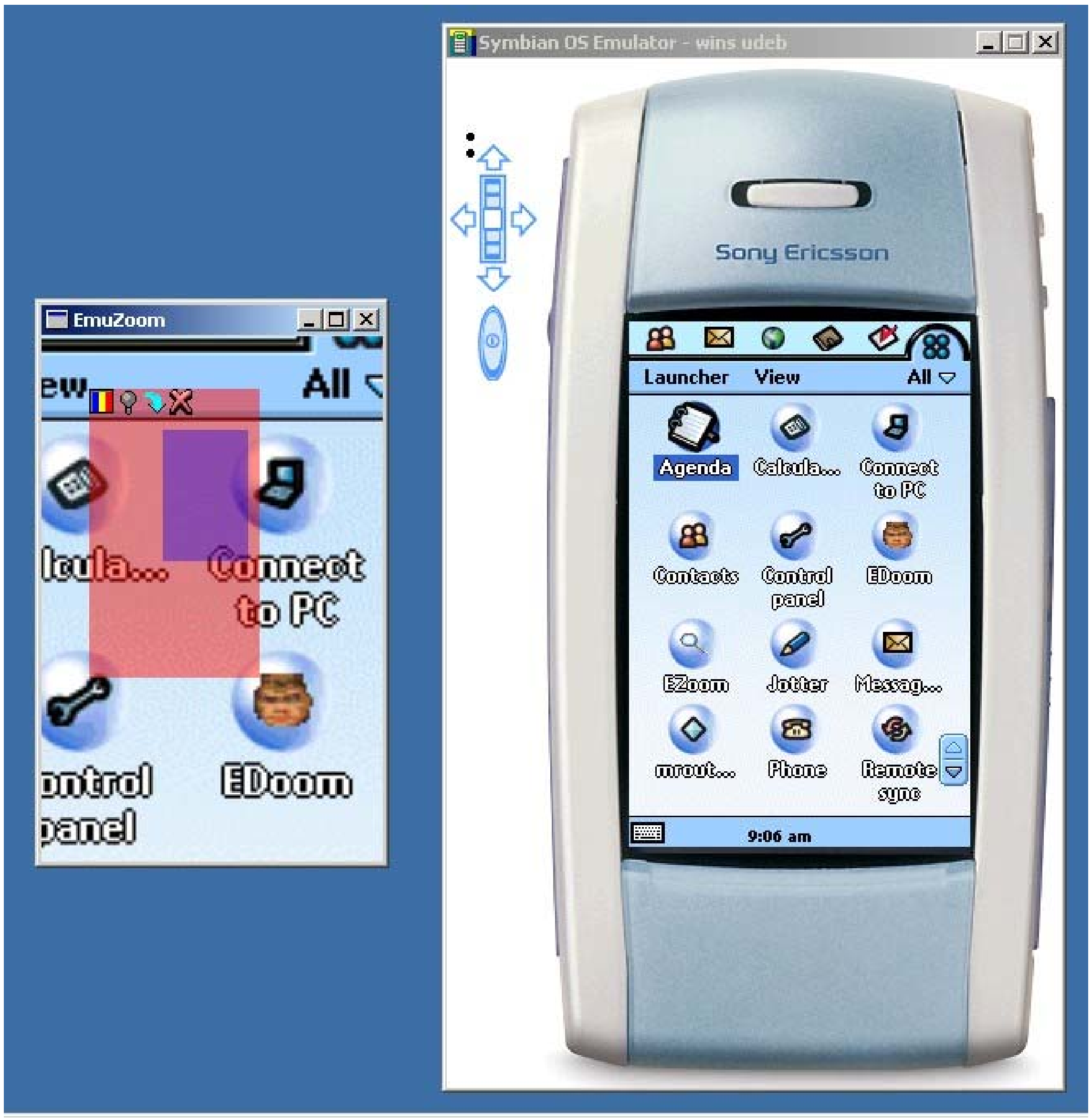

Figure 1: EZoom running under the emulator; zooming 400%.

The window on the right is the normal emulator window, the extra window on the left is EZoom emulating a target device's screen and is what is displayed on a real P800/P900. EZoom boosts a *zoom control*, the red and the blue rectangular areas with the toolbar, which is transparently drawn (i.e. alpha-blended) on top of the normal user interface. The zoom control indicates:

1. Zoom area: the red rectangle represents the original 208x320 pixels screen, the blue rectangle shows which part of the screen is being zoomed into.
2. Zoom factor: the size of the smaller blue rectangle is proportional to the size of the red one, based on the zoom factor. A zoom factor of four (or

zooming 400%) means the size of the blue rectangle is a quarter of that of the red one.

Once EZoom is active you can dynamically increase and decrease the zoom factor using the camera and browser button. You can move the zoom control around by dragging it with the stylus and, similarly, you can move the zoom area around by dragging the smaller blue rectangle. The zoom control has a tool bar with a couple of buttons. One of these buttons is used to rotate the screen 90 degrees clockwise allowing for the entire screen to be displayed in all four orientations. EZoom offers two scaling algorithms: a lower quality next neighbor scaling algorithm and a higher quality bilinear scaling algorithm. One of the tool bar buttons can be used to toggle between the two. Finally, EZoom can operate in two modes: smart update mode and continuous update mode, see the section on notifications of screen updates for an explanation of what they do. Below is another screen shot of EZoom showing off rotation and transparent wallpaper support.

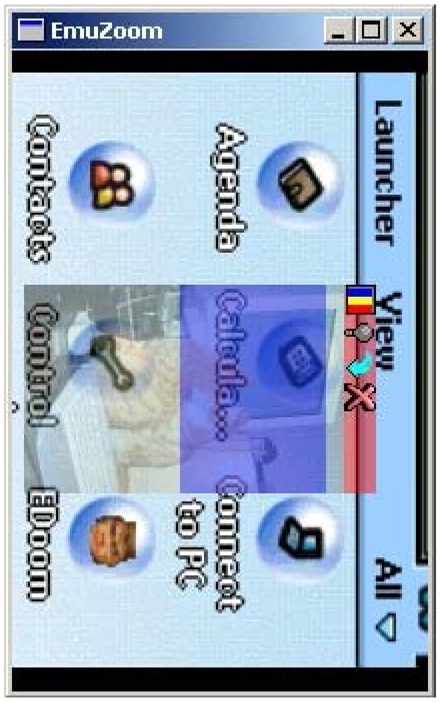

Figure 2: EZoom in 90 degrees rotation mode with a transparent wallpaper in its zoom control.

Apart from the usual fun aspect in hobby projects such as EZoom, the reason I developed it was because I was intrigued whether it was technically possible to write an EZoom utility in the first place with just the public SDK for the P800/P900. Well, it turned out that with a little creativity it is :-)

High level solution

Current Symbian OS phones use a *frame buffer* to drive the screen. You can think of the frame buffer as a bitmap in which each pixel one-to-one represents a pixel on the screen. Changing a pixel in the frame buffer effectively updates the according pixel on screen. In Symbian OS, conceptually, the Window Server (*WServ*) owns the screen resource. In most scenarios only WServ updates the frame buffer directly, and thus the screen. It does this on behalf of its clients, typically GUI applications, as a result of their draw requests.

EZoom's high level solution is to create a second frame buffer, associate that frame buffer with the screen and, whenever the screen gets updated by WServ, copy the updated areas of the frame buffer owned by Wserv to the newly created one, scaling and rotating along the way. If done correctly, WServ is unaware that the frame buffer it updates is no longer associated with the screen. To avoid confusion, the two frame buffers are referred to as *WServ's frame buffer* and *EZoom's shadow frame buffer*.

Problems to solve
There are a few basic problems that need solving for the above solution to work:

1. Multiple frame buffers: how to create a second frame buffer and how to switch between the two in software to drive the screen from either of them.
2. Notification of screen updates: how can EZoom figure out which parts of WServ's frame buffer have just been updated.
3. Pointer co-ordinate adjustment: apart from EZoom, no other software component is aware that the entire screen is scaled up and rotated. In order for the end user to interact correctly with the device using the pen, EZoom needs to transparently change pointer events before they are processed by applications.
4. Emulator support: frame buffers are used on real hardware devices. The Symbian OS emulator, however, uses a WIN32 window to emulate a target device's screen. For efficient testing and debugging it would be convenient if EZoom could somehow run in the emulator environment.

The next section describes the detailed solution for each of the above four problems. As part of solving the first problem, a device driver is needed. How to write one with just the public SDK is explained in the next section as well.

Detailed solution

Multiple frame buffers and switching between frame buffers

Just like normal off-screen bitmaps, the frame buffer is nothing more than a piece of memory. Contrary to normal off-screen bitmaps, the frame buffer must be mapped to a piece of contiguous physical memory. The graphics hardware knows nothing about the virtual memory mapping set up by the operating system on the host CPU and therefore requires the frame buffer to be contiguous in physical memory.

Virtual memory is the mechanism by which an operating system can protect software processes from harming each other. It allows for mapping identical virtual memory addresses in different processes to different physical memory addresses. In order for an operating system to use virtual memory, support from a hardware component referred to as the Memory Management Unit (MMU) is needed. The operating system and the MMU work together to manage the virtual-to-physical memory mappings for all processes on the device. The MMU manages physical memory in page-sized quantities (typically 4 KB) and the mapping between virtual and physical memory is stored in so called page tables. These pages tables are maintained by the operating system and are accessed by the MMU when translating virtual addresses to physical ones. Although the ARM CPU (all current Symbian OS phones have an ARM CPU) supports different page sizes, Symbian OS only uses page sizes of 4 KB. A further explanation on virtual memory management is beyond the scope of this paper.

The ideal solution to solve the multiple frame buffer problem would be to allocate a second frame buffer, also contiguous in physical memory, and switch between the two to reprogram the graphics hardware with the physical address of the alternate frame buffer. There are two major obstacles to this approach: 1) you need to write a device driver to be able to access hardware, and 2) reprogramming the graphics hardware requires intimate knowledge of that hardware. Although the first problem of writing a device driver can to some extent be dealt with, as will be discussed later, the second problem is insurmountable because details of the P800/P900's hardware are simply not available.

The approach I've taken in EZoom to solve the multiple frame buffer problem is inspired by one of the "tricks" I've learned in the PsiLinux project (running Linux on Psion devices such as Psion Series 5, 5mx and 7), which is to determine the physical address a virtual address is mapped to by interpreting the page tables directly. Once you have this ability, it's fairly trivial to extend it to modify the page tables to map a virtual address to a different physical address.

In a nutshell, EZoom's solution is this:

1. Allocate a page-sized (i.e. 4 KB) aligned buffer, referred to as the *shadow frame buffer*, of the same size as the WServ's frame buffer. This shadow frame buffer is, as per normal, guaranteed to be contiguous in virtual memory but not necessarily in physical memory.
2. For each 4 KB page in the shadow frame buffer, determine to which physical address the start of the page is mapped to.
3. Determine the physical address of WServ's frame buffer. Remember that that frame buffer is guaranteed to be contiguous in both virtual and physical memory, so it's enough to know the physical address of the first page. For completeness, the allocation of WServ frame buffer's physically contiguous memory is typically done by the device driver for the graphics hardware on startup.

Figure 3 illustrates the memory layouts of the both WServ's frame buffer and the shadow frame buffer.

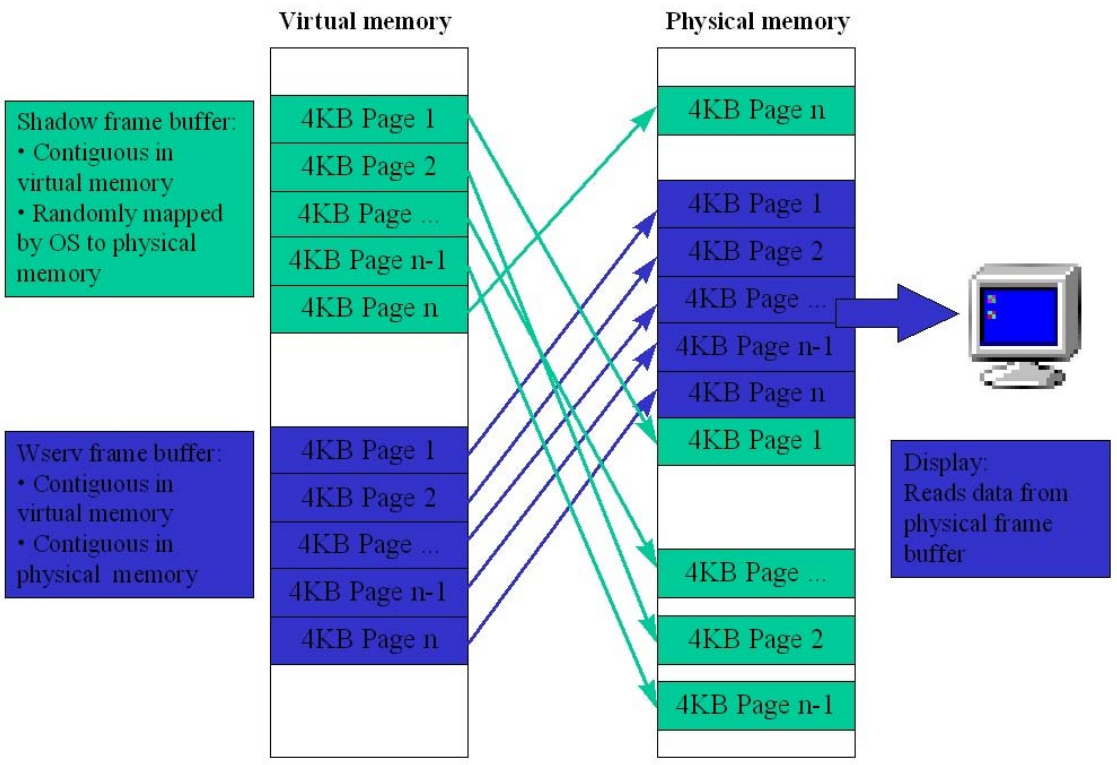
Virtual memory
Physical memory
Shadow frame buffer:
• Contiguous in virtual memory
• Randomly mapped by OS to physical memory
Wserv frame buffer:
• Contiguous in virtual memory
• Contiguous in physical memory
4KB Page 1
4KB Page 2
4KB Page ...
4KB Page n-1
4KB Page n
Display:
Reads data from physical frame buffer

Figure 3: Memory layout of shadow and WServ's frame buffer.

Switching between the two frame buffers goes like this:

1. Remap the virtual address range of WServ's frame buffer to the physical memory of the shadow frame buffer.
2. Remap the virtual address range of the shadow frame buffer to the physical memory of WServ's frame buffer.

Figure 4 shows the memory layouts after switching the two frame buffers.

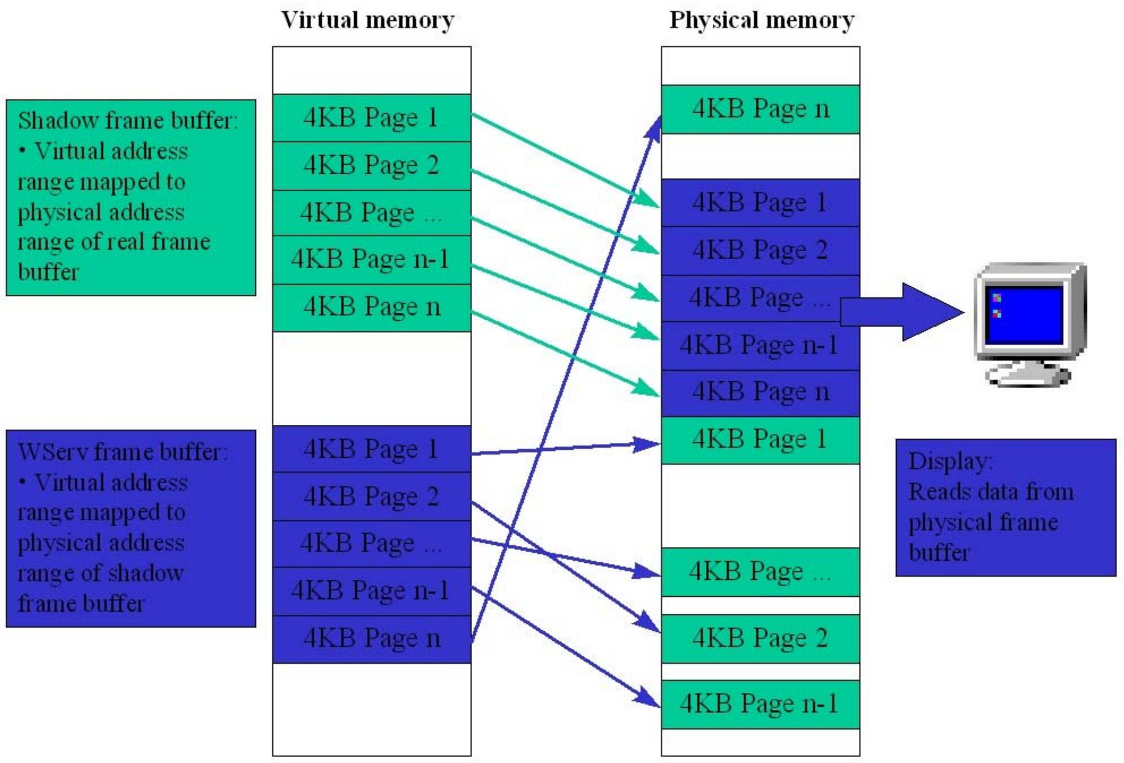
Virtual memory
Physical memory
Shadow frame buffer:
• Virtual address range mapped to physical address range of real frame buffer
WServ frame buffer:
• Virtual address range mapped to physical address range of shadow frame buffer
4KB Page 1
4KB Page 2
4KB Page ...
4KB Page n-1
4KB Page n
Display:
Reads data from physical frame buffer

Figure 4: Memory layouts of shadow and WServ's frame buffer after switching.

As a result, updating the shadow frame buffer now updates the physical frame buffer associated with the screen. WServ's updates to its frame buffer are now stored in what used to be the shadow frame buffer's physical memory. Note that WServ is completely unaware of this remapping; it doesn't know nor care.

The next two basic questions then become: how to determine the physical address given a virtual address and how to re-map a virtual address range to a different physical address range. The bad news is that the Symbian OS kernel does not provide this type of API; the good news is that I've done this sort of thing before in the PsiLinux project (running Linux on Psion devices). A further description on how precisely the page tables are updated is beyond the scope of this paper; the interested reader is referred to the EZoom source code. Suffice to say that updating page tables must be done from kernel mode and thus from a device driver. The section writing a Symbian OS device driver describes how to do that.

Notification of screen updates

EZoom has two modes of operation: *continuous update* mode and *smart update* mode. In continuous update mode, EZoom scales the entire screen 15 times a second regardless of whether the screen was updated in the first place. This was my original design idea for EZoom. Using the lower quality next-neighbor scaling algorithm resulted in acceptable performance (but only just), the higher quality bilinear scaling algorithm, however, rendered the system useless because a lot of CPU power and memory bandwidth is consumed by the scaling algorithm. For bilinear scaling to work, a better solution than this unconditional full-screen redraw at 15 Hz was needed. It turns out that with a bit of C++ peekery-pokery you can figure out which part of the screen WServ has updated.

In the Symbian OS graphics framework, an object instantiated from `CFbsScreenDevice` represents the entire screen. Subject to its clipping region, all draw operations on this object unconditionally update pixels on the screen. There aren't many such `CFbsScreenDevice` objects in the system. Typically only WServ has one for its entire lifetime to update the screen, but, as part of the *Direct Screen Access* (DSA) design, each DSA client gets its own client-side `CFbsScreenDevice` object to update the screen directly without issuing WServ client-server draw requests. Conceptually, `CFbsScreenDevice` is the *device independent* representation of the screen. It relies on an object instantiated from `CFbsDrawDevice`, created by the user mode video driver (scdv.dll), to actually draw to screen in a *device dependent* way. All draw operations on `CFbsScreenDevice` call `CFbsScreenDevice::UpdateRegion(const TRect& aRect)` afterwards, where `aRect` is the bounding rectangle of the updated area. This method simply calls down to `CFbsDrawDevice::UpdateRegion(const TRect& aRect)` on its `iDrawDevice` data member.

One of the software components in EZoom is a so called *WServ animation dll*, or *anim dll* for short. This component is also needed to transparently modify pointer events in the system. An anim dll is essentially a plug-in that gets loaded by WServ and thus runs in the context of the WServ process. As part of the API that WServ offers an anim dll plug-in, the anim dll can get a reference to the `CFbsScreenDevice` object owned by WServ, allowing it to directly update the screen in co-ordination with WServ.

The EZoom anim dll implements a "hook" by replacing the `CFbsDrawDevice* iDrawDevice` data member in WServ's `CFbsScreenDevice` object by one of its own: an object instantiated from `CHookedFbsDrawDevice` which inherits from `CFbsDrawDevice`. For all the virtual methods in `CFbsDrawDevice`, `CHookedFbsDrawDevice` simply calls down on the according method of the original `iDrawDevice` data member in WServ's `CFbsScreenDevice` object. By placing this hook, EZoom gets notified via `CHookedFbsDrawDevice::UpdateRegion(const TRect& aRect)` what parts of the screen have been updated by WServ. Needless to say that this is essentially a hack and goes against common software engineering rules. Fortunately, there is an exception to this rule which says that in *war*, *love* and *software engineering* everything is allowed.

In *smart update* mode EZoom uses the hook mentioned in the previous paragraph to selectively redraw parts of the screen as they get updated. If EZoom learns a new area has been updated, it sets a timer for roughly 60 ms. Every redraw that occurs before this time-out gets added to the current redraw area by calculating the bounding rectangle of the current redraw area with the new one. When the timer fires, EZoom redraws the area on the screen corresponding to the accumulated rectangles of all redraws that occurred when the timer was running. The performance in smart update mode is excellent. When using the more CPU intensive, higher quality bilinear scaling algorithm you hardly notice any performance difference. Even in MP4 playback this difference is not really apparent. As a nice bonus, EZoom helps to reduce flickers in the user interface because it updates the screen in a more synchronized manner than the system does itself.

One downside of the smart update mode is that it only detects the screen updates that occur within the WServ process. The types of screen updates not detected are the ones from DSA clients (and clients that scribble directly into video memory). Since the DSA framework creates a client-side `CFbsScreenDevice` object, the EZoom anim dll running in the WServ process not only has no access to it but is also not aware that additional `CFbsScreenDevice` objects exist. It's because of those DSA clients that EZoom still needs to offer the continuous update mode feature.

Continuous update mode has one drawback as well. It doesn't cater for the Camera viewfinder use case. EZoom's solution to swap frame buffers by remapping virtual memory only affects *software* components because they write to the frame buffer via virtual memory. When the camera viewfinder starts, the camera hardware copies its viewfinder image frames directly to the video frame buffer. Because this is done from *hardware* and on the hardware level there is no concept of virtual memory, the viewfinder frames are copied directly in to the frame buffer associated with the screen. EZoom has no means to cater for this use case.

Pointer co-ordinate adjustment

Listed below is the basic pointer input handling in Symbian OS:

1. For each pen down, pen up or pen drag event, the touch screen device driver adds a so called *raw event*, an object from class `TRawEvent`, to the system-wide event queue. This queue is a kernel data structure to which

both kernel mode and user mode clients can add events. From user mode you would use `UserSvr::AddEvent(const TRawEvent& anEvent)`.

2. The kernel allows the first client who asks for it to become the exclusive owner of this queue; only that client can retrieve events from it. On startup, WServ is typically the first such client requesting ownership of this queue. WServ processes these raw events one-by-one, turning *raw pointer events* into *window server pointer events*; objects instantiated from class `TWsEvent` whose event type equal `EEventPointer`.
3. WServ routes pointer events like this:

- It passes the raw event (`TRawEvent`) to all so called WServ *raw event handlers*, asking each of them to consume the event. WServ stops routing the pointer event the moment one of them does.
- If none of the raw event handlers consumed the raw event, WServ passes the window server pointer event (`TWsEvent`) to the window server client who had requested a global pointer capture, if any such client exists.
- If there was no global pointer capture client, WServ passes the window server pointer event (`TWsEvent`) to the window server client with the highest Z-order window containing the co-ordinate of the pointer event.
- If the window server client is a standard application, the pointer event is routed further, client side, by the application framework until it finds its way to the appropriate UI control (objects instantiated from `CCoeControl`-derived classes).

In order for EZoom to alter the pointer events based on the current zoom factor and rotation, it has to somehow hook into the above chain. It does so by implementing a so called *WServ anim dll*. These are dlls that get loaded by WServ in its own process. Typically anim dlls are used for server-side drawing which is faster than normal client-side drawing in, for example, a standard Symbian OS application. An anim dll can have the role of a *raw event handler* as described previously. As a raw event handler, EZoom gets to see raw pointer events very early on in the routing process, as described in the first bullet point in step 3. Changing pointer events is then just a matter of updating the `TRawEvent` which WServ passes to the raw event handler. Although strictly speaking this isn't the correct way to modify pointer events (the object is passed as a `const TRawEvent&` and EZoom simply casts away the `const`-ness), I never had a problem doing this. The correct solution would probably be to consume the raw pointer event (preventing it from further routing) and to generate a new raw pointer event with the modified pointer co-ordinate. A slight complication is that this newly generated raw event is passed by WServ to the anim dll again, so the latter has to have some logic to detect and ignore these. Needless to say that EZoom's current solution (simply update the raw event passed to its anim dll) is much simpler.

Writing a Symbian OS device driver

As part of the solution for multiple frame buffers and switching between frame buffers, EZoom needs to implement functionality to remap virtual memory in a device driver. The bad news is that, strictly speaking, you can't develop device drivers using public Symbian OS SDKs. You will have to sign up as a Symbian partner to get access to development kits that contain the relevant files. The good news is that some limited support for device drivers is possible using just a publicly available SDK.

To develop a device driver you need two things: the kernel include files and the kernel library (`EKERN.LIB`). The first contain APIs that your device driver can use when running in kernel mode; the second is the library you need to link against if you use such API calls. The kernel include files and the WINS version of `EKERN.LIB` are shipped in the public SDKs for the P800/P900 so you can fully develop device drivers for WINS. This is only part of the solution of course, because you'll need a target device's `EKERN.LIB` as well. Unfortunately, that file is not available in the P800/P900 SDKs. The latter means that although you can compile a device driver for a P800/P900 device, you can't link it. As you will see in a minute, a limited work around is available.

Let's take a closer at the EZoom device driver (EZoom.ldd). The API offered by EZoom.ldd is listed below.

```
class RZoomDriver : public RBusLogicalChannel
{
    ...
    TInt LinearToPhysical(TLinAddr aLinAddr, TPhysAddr*
aPhysAddr) ;
    TInt RemapLinearAddr(TLinAddr aLinAddr, TPhysAddr
aPhysAddr, ...) ;
    ...
};
```

It consists of two simple functions; the first one returns the physical address a given virtual address is mapped to; the second changes the physical address a given virtual address is mapped to to some other given physical address. I've ignored some minor details in the above API as they are not relevant to this discussion. As described in the section on multiple frame buffers and switching between frame buffers, the implementation of these two API calls are reused from the PsiLinux project. Very conveniently, the implementation doesn't require any support from the kernel, so no functions are needed from `EKERN.LIB`. The only `EKERN.LIB` functionality needed by EZoom.ldd is the generic device driver infrastructure support to be able to offer an API to user mode clients.

From a software point of view, a Symbian OS device driver is just a dll that exports one function: a factory method which returns an object instantiated from a class derived from `DLogicalDevice` (whereas in user mode, classes whose instantiated objects are allocated on the heap start with a capital C, in kernel mode those classes start with a capital D). In the case of EZoom, that derived class is called DLddZoom and it is defined like this:

```
class DLddZoom : public DLogicalDevice
{
public:
    ~DLddZoom();
    DLddZoom();
    virtual TInt Install();
    virtual void GetCaps(TDes8&) const {}
    virtual DLogicalChannel* CreateL();
};
```

The object instantiated from `DLddZoom` represents the device driver itself. A user mode client which has opened this device driver now needs to create a channel with it. That results in `DLddZoom::CreateL()` being called. This virtual method returns an object instantiated from a class derived from `DLogicalChannel`. EZoom's version is called `DChannelZoom` and is defined like this:

```
class DChannelZoom : public DLogicalChannel
{
public:
    DChannelZoom(DLogicalDevice * aDevice);
    ~DChannelZoom();
protected:
    virtual void DoCancel(TInt){};
    virtual void DoRequest(TInt, TAny*, TAny*){};
    virtual TInt DoControl(TInt aFunction, TAny* a1,
TAny* a2);
    virtual void DoCreateL(TInt /*aUnit*/, CBase*
/*aPdd*/, const TDesC* /*anInfo*/, const TVersion&
/*aVer*/){};
};
```

The virtual method `DoControl(...)` is used in EZoom.ldd to implement the earlier mentioned two API calls.

All of the above is needed as part of the basic implementation of a Symbian OS device driver. Although none of them call functions in `EKERN.LIB` explicitly, exported methods from that library are still needed. For example, when the compiler generates code for `DLddZoom::DLddZoom()`, it needs to generate code to call the constructor of the base class:

`DLogicalDevice::DLogicalDevice()`, a method exported by `EKERN.LIB`. Without doing anything extra, the EZoom device driver will compile correctly for a target build, but will fail linking with a handful (around 10) unresolved methods from `EKERN.LIB` such as `DLogicalDevice::DLogicalDevice()`. The work around is to include dummy implementations for them in the EZoom driver source code. For example, the constructor for `DLogicalDevice` can simply be left empty:

```
DLogicalDevice::DLogicalDevice()
{
}
```

In fact, almost all of the other unresolved method can be left empty as well or be given a trivial implementation such as returning `KErrNone`. After I managed to build EZoom.ldd for a target device I noticed that all worked fine apart from the fact that I couldn't unload the driver. After debugging a bit under the emulator, I figured that `DLogicalChannel::~DLogicalChannel()` needed more than just an empty implementation; changing it to the following solved the problem:

```
DLogicalChannel::~DLogicalChannel()
{
    iDevice->iOpenChannels--;
}
```

As a result, I am now the proud owner of a fully functional Symbian OS device driver that can be built with just the public SDK.

Emulator support

Most of EZoom's functionality is trivially supported in the emulator, including writing a device driver. The biggest issue in the emulator version of EZoom is the emulation of WServ's frame buffer, the shadow frame buffer and switching back and forth between them. All three issues are addressed below:

Emulating WServ's frame buffer

The "trick" I've used in my ports of Doom and Mame to get access to WServ's frame buffer is to do the following:

```
TScreenInfoV01           screenInfo;
TPckg<TScreenInfoV01>    sI(screenInfo);

UserSvr::ScreenInfo(sI);
```

Afterwards, `screenInfo.iScreenAddress` contains the virtual address of WServ's frame buffer. This works fine on a target device, where there is a frame buffer, but doesn't work in the emulator, where the screen is ultimately emulated using a WIN32 window. Although `UserSvr::ScreenInfo()` returns the WIN32 windows handle (`HWND`) of the emulator window and as such you have access to its pixels using normal WIN32 APIs, this isn't a satisfactory solution. The pixels you'll get that way are in the color depth of the WIN32 window rather than the color depth of the emulator (EColor4K for P800 and EColor64K for P900).

It turns out there is another way to get access to the emulated frame buffer in WINS. The Symbian OS 2D graphics framework implements support for 2D hardware acceleration, allowing certain 2D graphics operations to be performed in hardware rather than in software. Although the P800 and P900 don't support 2D graphics acceleration in hardware, the Symbian OS emulator does. For example, the Symbian OS GDI bitblt implementation will use the 2D hardware bitblt operation if it is supported, otherwise it performs the bitblt itself in software. The emulator version of the 2D graphics video driver implements the 2D accelerated bitblt using WIN32's GDI bitblt (after all, the emulated screen is a normal WIN32 window). Furthermore, the 2D graphics framework has the notion of so called *hardware bitmap*s (`RHardwareBitmap`), which are bitmaps that can be accessed by the 2D graphics hardware. Basic properties of a hardware bitmap are: virtual address, physical address, color depth and size. Just like other R-based classes, `RHardwareBitmap` is nothing more than a handle; this allows multiple clients to share a hardware bitmap. A client which knows the handle of an existing hardware bitmap may open it by handle. If there is a 2D graphics hardware accelerator on the device then the 2D graphics framework assumes WServ's frame buffer is represented by `a hardware bitmap` with handle *-1*, and that this well known hardware bitmap is associated with the `CFbsScreenDevice` used by WServ to update the screen.

To get access to the emulated frame buffer in WINS, you can do this:

```
TAcceleratedBitmapInfo  info;
RHardwareBitmap         hardwareBitmap(-1);  // -1 is
the well known handle for WServ's frame buffer

User::LeaveIfError(hardwareBitmap.GetInfo(info));
```

Afterwards, `info.iAddress` refers to the frame buffer, and `info.iDisplayMode` refers to the system's color mode. This is the same value that was used to configure the emulator in `z:\system\data\wsini.ini`.

Emulating switching between the two buffers
I've played with the idea to somehow update the pixels in the emulator's frame buffer as returned by `RHardwareBitmap::GetInfo()`. After a while, I figured it wasn't worthwhile to be that faithful in emulation. A much simpler solution is to create a separate WIN32 window of the same size as the emulator screen (208x320 for P800/P900) and to draw the contents of the shadow frame buffer in

it from a background WIN32 thread. This separate window is what the screen on a target device would look like; the emulator window itself still reflects what's stored in WServ's frame buffer.

It's not too difficult to use WIN32 functionality from within the emulator. Update your .mmp file like this:

- Add .cpp files with WIN32 functionality in the normal way in your .mmp file using a `SOURCE` statement
- Add WIN32 libraries like this:

```
START WINS
WIN32_LIBRARY     gdi32.lib user32.lib
kernel32.lib
END
```

In .cpp files that use both WIN32 and Symbian OS functionality, include header files in the order show below:

```
#define OEMRESOURCE
#include <windows.h>        // Windows first
#include <e32def.h>         // Symbian OS stuff next
#include <e32std.h>
#include <e32base.h>
#include <gdi.h>
```

An example of mixing WIN32 and Symbian OS functionality is shown in the code snippet below. WIN32 pointer events received in the extra WIN32 window are fed back into the Symbian OS environment by adding them to the system-wide event queue.

```
TInt32 CEmuZoomDevice::HandleWin32Event(HWND aHwnd,
TUint aMessage, TUint aWParam, TInt32 aLParam)
{
    switch (aMessage)
    {
        ...
        ...
        case WM_LBUTTONDOWN:
        {
            TRawEvent    event;
            event.Set(TRawEvent::EButton1Down,
LOWORD(aLParam), HIWORD(aLParam));
            UserSvr::AddEvent(event);
            iMouseDown = ETrue;
            return 0;
        }
        ...
        ...
    }

    return ::DefWindowProc(aHwnd, aMessage, aWParam,
aLParam);
}
```

Graphics operations

All of the previous sections describe the functionality needed to allow EZoom to do what it ultimately needs to do: scale and rotate the entire user interface, and draw the Zoom control transparently on top of it (i.e. alpha-blended). Although I could have probably used existing Symbian OS graphics routines to do all of that, it's much more fun to roll out your own. Apart from the fun factor, Symbian OS doesn't provide functionality to do all three (scaling, rotation and alpha-blending) at the same time in one operation. It's therefor probably more efficient to write some specialized routines to combine all three operations into one. The usual downside applies here of course, by rolling out your own low-level graphics routines, you need to implement them for each display mode you wish to support. In P800 and P900 the display modes are respectively `RGB444` and `RGB565`. Since both use 16 bits per pixel, much of EZoom's graphics functionality can be shared between the two color modes but operations on the individual red, green and blue sub-pixel values need to be implemented separately. All three operations are described below. A description on how to combine the three operations into one is beyond the scope of this paper, the interested reader is referred to the EZoom source code.

Scaling

Figure 5 shows a simplistic picture of what 400% zooming in EZoom means. Given the original 208 x 320 pixel image on the left, you select a 104 x 160 subset of that image (a quarter of the surface area of the original one) and blow it up four times to fit a 208 x 320 pixel image again (shown on the right).

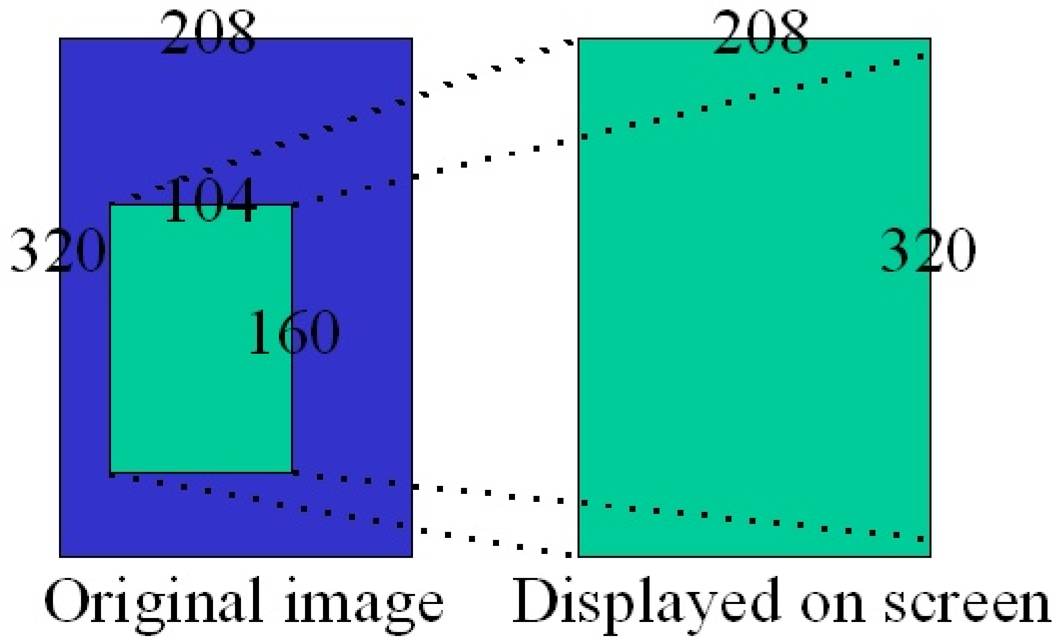

Figure 5: Zooming in 400%.

The problem is that in the 104 x 160 pixel image you don't have enough pixel values to draw a 208 x 320 pixel image. The solution is to *interpolate* the missing pixels, which is the mathematical method to determine a value in-between known discrete values. There are many different interpolation algorithms out there. The two I've chosen to implement are: next neighbor and bilinear interpolation. The first is fast but results in a pixelated image (especially at higher zoom levels because existing pixels are simply duplicated), the second yields better image quality at the expense of CPU and memory bandwidth.

First, some straightforward math:

Call the width and height of the P800 and P900 screen respectively `w` and `h`
Call the width and height of the zoomed in area respectively `w'` and `h'`
Given a zoom factor of *x* % (400 in the above example) then it holds that:

(1) `w' * h' * (x / 100) =  w * h`

Maintaining aspect ratio means:

(2) `w = (208 / 320) * h` and `w' = (208 / 320) * h'`

Combining (1) and (2) results in:

(3) `(h' * (208 / 320)) * h' * ( x / 100) = (h * (208 / 320)) * h`

Which can be reduced to:

(4) `h' = (10 * h) / sqrt (x)`

Substituting (2) in (4) gives the width:

(5) `w' = (10 * w) / sqrt (x)`

Once the user has set EZoom to a given zoom factor, the above formula (4) and

(5) calculate the size of the zoomed in area.
Figure 6 shows how the zoomed in area is interpolated to get the additional pixels. The blue bubbles represent the known pixel values in the zoomed in area; the green ones represent the pixels that we need to calculate.

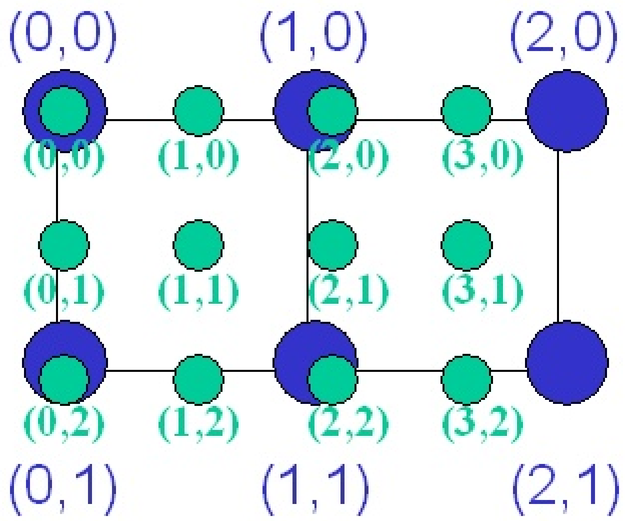

Figure 6: Interpolation.

Some more math:

The distance between the known pixels (the blue ones) is by definition (1,1).
The distance between the interpolated pixels (the green ones) is called (`dw'`, `dh'`)

(6) `dw' = w' / w` and `dh' = h' / h`

Substituting (4) and (5) in (6) gives:

(7) `dw' = 10 / sqrt(x)` and `dw' = sqrt(x) / 10`

Note that when zooming in, `dw'` and `dh'` are values between 0 and 1. The above formula allows us to traverse all interpolated pixels like this:

```
for (float y = 0  ;  y < h  ;  y += dh')
   for (float x = 0  ;  x < w  ;  x += dw')
      pixel = InterpolatePixel(x, y);
```

Given this basic algorithm to traverse all interpolated pixels, Figure 7 shows for a single such pixel how it relates to its surrounding four known pixel values.

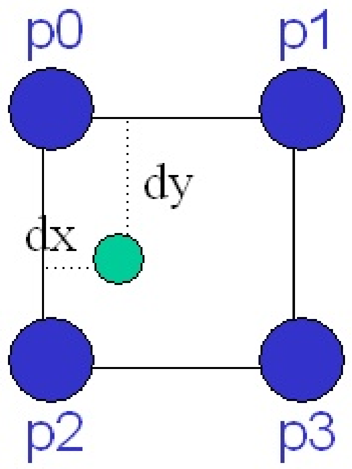

Figure 7: Interpolation of a single pixel.

The question now is, given the four known pixel values `p0`-`p3`, and `dx` and `dy`, what should the pixel value of the interpolated pixel be? Next neighbor and bilinear interpolation are only two examples of how to calculate that pixel value. They are described below.

Next neighbor interpolation
EZoom's implementation for next neighbor scaling looks something like this:

```
pixel =  (dx < 1) ? ((dy < 1) ? p0 : p2) : (dy <
1) ?  p1 : p3);
```

It says that if the position of the green pixel coincides with one of the blue ones, the green pixel's value is that of the blue one, otherwise its value is that of blue pixel p0. This algorithm essentially duplicates existing pixel values and results in a very pixelated image.

Bilinear interpolation
In bilinear interpolation, three interpolation steps are performed. First, horizontally, the pixel values in-between p0 and p1, and p2 and p3 are interpolated. Then these two temporary pixel values are interpolated vertically. The linear interpolation between two pixels is done on the individual sub-pixel values and is therefore color mode dependent. The code snippet below shows the one for `RGB565` which is used on the P900.

```
pixel = InterpolatePixelRgb565
( InterpolatePixelRgb565(p0, p1, dx * 100),
InterpolatePixelRgb565(p2, p3, dx * 100), dy * 100);

TUint16 InterpolatePixelRgb565(TUint16 aPixel0, TUint16
aPixel1, TInt aScale)
```

```
{
        // linear interpolation of pixel aPixel0 and
aPixel1
    TInt scale0 = iModel.iScaleTab[100-aScale];
    TInt scale1 = iModel.iScaleTab[aScale];
    TUint    b = ((((aPixel0 & 0x001f) * scale0) +
((aPixel1 & 0x001f) * scale1)) >> 10) & 0x001f;
    TUint    g = ((((aPixel0 & 0x07e0) * scale0) +
((aPixel1 & 0x07e0) * scale1)) >> 10) & 0x07e0;
    TUint    r = ((((aPixel0 & 0xf800) * scale0) +
((aPixel1 & 0xf800) * scale1)) >> 10) & 0xf800;
    return (TUint16) (r | g | b);
}
```

EZoom uses some simple fixed point arithmetic to prevent floating point multiplications for each sub-pixel value. Symbian OS phones don't have hardware floating support so these operations are CPU intensive. As can be seen by comparing the code snippets of the two scaling algorithms, bilinear scaling requires much more operations to calculate an interpolated pixel. The benefit, though, is that the image quality is much higher than using next neighbor scaling.
For completeness, the source code above isn't a strait copy of the EZoom implementation; I've made some changes to make it more in line with the concepts and definitions described in this paper.

Rotating

EZoom supports all four rotation modes, internally referred to as `rot0`, `rot90`, `rot180` and `rot270`. Given the screen size `TSize iScreenSize` and the rotation `iRotation`, the following method shows how to rotate a single point.

```
TPoint TZoomModel::Rotate(const TPoint& aSrc) const
{
    switch (iRotation)
    {
        default:
        case ERot0:
            return aSrc;
            break;
        case ERot90:
            return TPoint((iScreenSize.iWidth - 1) -
aSrc.iY, aSrc.iX);
            break;
        case ERot180:
            return TPoint((iScreenSize.iWidth - 1) -
aSrc.iX, (iScreenSize.iHeight - 1) - aSrc.iY);
```

```
            break;
        case ERot270:
            return TPoint(aSrc.iY, (iScreenSize.iHeight
- 1) - aSrc.iX);
    }
    return TPoint(0, 0);
}
```

Alpha-blending

Alpha-blending is no longer a novelty, most systems provide support for it. For EZoom I have reused the alpha-blending routines from my EDoom project, which in turn had borrowed it from the WINE project. Alpha-blending operates on the sub pixel values as well, below is the implementation for `RGB565`:

```
TUint16 CZoomDrawer::AlphaBlendRgb565(TUint16 aPixel0,
TUint16 aPixel1, TUint8 aAlpha) const
{
    TUint    t = aPixel1 & 0xf800;
    TUint    m1 = (((((aPixel0 & 0xf800) - t) *
aAlpha)>>8) & 0xf800) + t;
    t = aPixel1 & 0x07e0;
    TUint    m2 = (((((aPixel0 & 0x07e0) - t) *
aAlpha)>>8) & 0x07e0) + t;
    t = aPixel1 & 0x001f;
    TUint    m3 = (((((aPixel0 & 0x001f) - t) *
aAlpha)>>8) & 0x001f) + t;
    return (TUint16) (m1 | m2 | m3);
}
```

EZoom software components

Listed below are EZoom's software components and, for each for them, a short description of the functionality it provides.

EZoom.app

- Provides the user interface for EZoom's user configurable options and saves them to an .ini file.
- Provides *start* and *stop* menu options to respectively start and stop the zooming operation. Starting the zooming operation is done by loading the ZoomAnim.dll, sending it a *configure* command followed by a *start* command.

- Dynamically reconfigure the zooming operation by sending ZoomAnim.dll a *configure* command.
- Provides the help user interface.

ZoomAnim.dll

- Figures out which part of the screen WServ has just updated by placing a *hook* in-between WServ's `CFbsScreenDevice` and its `CFbsDrawDevice`.
- Implements the actual zooming by:
  - Creating a second frame buffer called shadow frame buffer.
  - Remapping WServ's frame buffer and the shadow frame buffer so that they refer to the other frame buffer's physical address range (using the EZoom.ldd API below for each 4 KB page in both buffers).
  - Scales the entire screen 15 times a second in *continuous update mode*.
  - Scales only those parts of the screen as they get updated by WServ in *smart update mode* (using the above mentioned hook).
- Provides the user interface for the zoom control which is transparently drawn on top of the screen.
- Controls the zooming process: rotation, change between smart update and continuous mode and change image quality.
- Implements bitmap routines for scaling (bilinear and next neighbor), rotation and alpha-blending (for transparency).
- Transparently changes pointer events based on the current zoom level and visible area on screen so that system is not aware of the scaling and rotation.
- Zooms in and out when respectively camera and browser button are pressed; also implements key repetition.
- For WINS, emulates the shadow frame buffer by creating a separate WIN32 window controlled in a separate WIN32 helper thread.
- Implements *start*, *stop* and *configure* commands which are issued by EZoom.app.

EZoom.ldd

- Provides an API to return the physical address given a virtual address.
- Provides an API to change the physical address a given virtual address is mapped to to some other physical address.

Conclusion

EZoom is an utility that zooms in and rotate the entire user interface of a P800/P900, and draws a zoom control transparently on top. Although not the most useful utilities out there, its implementation uses some interesting techniques that stretch the boundaries of what people think is possible an a Symbian OS phone. Examples are the ability to create a second frame buffer and associate that with the display, to transparently change pointer events in the phone, to filter WServ's screen updates and to write (very simplistic) device drivers using the public P800/P900 SDK.

I potentially could have used Symbian OS' built-in graphics routines in EZoom, but decided not to. Apart from the fun factor to write your own, it also allowed me to combine the scaling, rotation and alpha-blending operations into one operation, which never hurts from a performance point of view.

Tinkering with EZoom's design concepts, implementing it and even writing this paper have all contributed to the fun and satisfaction factor of this hobby project. If you have any suggestions for challenges similar to the ones in EZoom then feel free to drop me a line.