

Programming Windows to Create Palm Games

Carsten Magerkurth, EmperoR Studios

email: carsten@magerkurth.de; www.emperor-studios.de

Abstract

This lecture proposes a platform-independent approach for creating Palm® games. It is described how platform-specific components are integrated into an object oriented framework and then reproduced within a DirectDraw windowed-mode application. This allows for shifting major parts of the development process away from the Palm and its programming tools. Instead, accelerated development by generating native Win32 code is suggested.

Introduction & Motivation

Developers starting for the Palm Computing platform are often faced with a situation that differs from their prior desktop programming experiences in the following respects: There are new forms of interaction like pen-based input with Graffiti® text recognition. There is no windows-metaphor for user interfaces. There are severe constraints in CPU power and memory. And finally there are no heavy-weight frameworks that provide pre-made solutions for typical programming problems (in fact, even the ANSI standard library is omitted for the Palm SDK).

As it will be pointed out in this lecture, not all of these fundamental differences matter much when creating games. The similarity between programming PDA games and adapted desktop counterparts is sufficient to shift the focus away from the PDA and its peculiarities during development allowing for a faster and more comfortable development process.

Writing code that does not natively run on the developer's desktop machine can be a tiresome, if not frustrating experience. Even though the official Palm Emulator POSE (see www.palm.com) reflects the Palm very accurately and is therefore a great help for debugging and testing on the desktop, some serious drawbacks still remain. Apart from missing sound emulation or sporadic connection losses to the IDE, it shares a common problem with other emulators: Source-level debugging is, naturally, much slower than with native code. There are delays in stepping through the source code that reach the dimension of seconds even on modern PC's. Provided that your next bug is really nasty (they always are!) the amount of time and nerves wasted waiting can quickly become significant.

A workable solution is to separate as much back-end code as possible and painlessly debug it in a native Windows test-application. This, however, implies that there are at least two applications to be maintained, so, ideally, the entire code should both be available for native testing on the desktop and for running on a Palm device.

To achieve this, the approach proposed here is to implement the system-specific functionality separated for both platforms, but with the same interface. Thus, there will be two separate projects, one for the Palm and one for Windows, both sharing the same header files with differing implementations of the specific components (see illustration 1). If abstracting Palm/Windows code this is a lot easier than the obvious conceptual differences between PDA's and desktop machines suggest. In the next section the essential platform-dependent parts of typical PDA games are identified and it is shown how they are encapsulated within a thin object-oriented framework. For clarity (and all the other advantages of OOP) C++ is used instead of plain C, even though this implies a certain (but often negligible) performance hit.

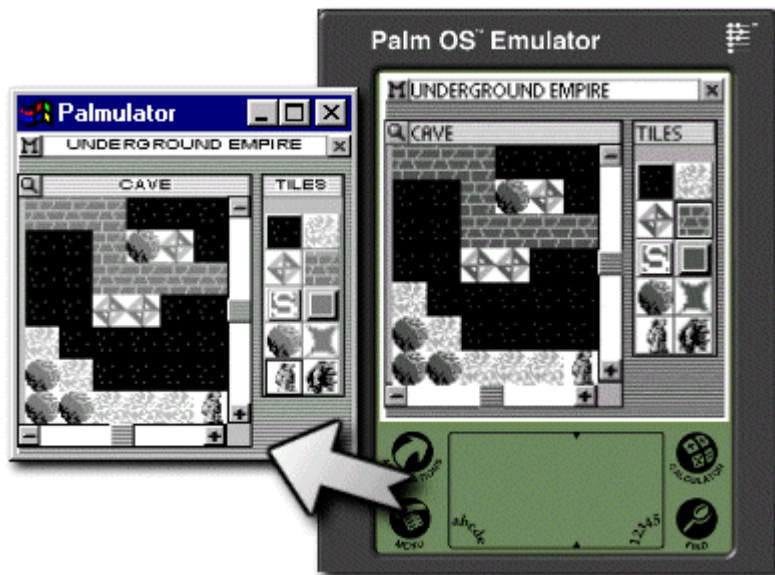


Illustration 1: Shifting the development process from the Palm to Win32

An additional advantage of this approach arises from the implicit free choice of the development environment and tools: Apart from GNU solutions the only officially supported IDE for writing Palm code is the Metrowerks Codewarrior (see www.metrowerks.com). Even though this is a good tool, many game developers coming from the Win32/ DirectX world will be more familiar with more common Windows IDE's like Microsoft Visual Studio (see www.microsoft.com). Since you will be working on the Windows version of your Palm game most of the time, you can still enjoy all the benefits of the IDE you are used to, while the Palm compiler is foremost for building the Palm executables *after* the bulk of programming and testing has been done.

As a positive side-effect, porting the games to other handheld operating systems like Symbian's EPOC (see www.symbiandevnet.com) or Microsoft's PocketPC/ CE (see www.microsoft.com) is naturally facilitated as well. This, however, is beyond the scope of this lecture.

To begin, the components of Palm game programming that depend on the features of the device will be discussed. A proposed application framework to integrate these components in will then be presented in the concluding sections of this lecture.

Platform-specific components

Obviously, the most important parts of a game that heavily depend on platform-specific features are output and input. Output mainly refers to graphics and sound, while input means “talking” to the program with appropriate input devices like mice, joysticks, or in our case pen and hardware buttons. So dealing with input is about mapping the pen strokes, the Graffiti text recognition, and the hardware keys to the corresponding input devices available on desktop machines.

For each of the above mentioned components the crucial aspects will be identified in the following sections together with proposed wrapper-classes that are appropriate both for Win32/DirectX and for the Palm.

Graphical Output

Due to the very limited display dimensions on a PDA the output of an average application is expected to be form-based. Forms are static templates composed of user interface components like text-fields or buttons that are assembled in resource creators, like Codewarrior’s own Constructor. For typical games that require more dynamic (moving) output the use of forms is awkward and inappropriate. From a Windows game developer’s point of view one would want to make use of surfaces, i.e. portions of memory that represent bitmapped images to be blitted around quickly.

A class representing a surface would, at the very least, need to allocate and free the memory required to hold the bitmap data and provide methods to blit its contents to other surfaces, including the screen. Furthermore, it should provide means to create/ modify its content or let it represent image resources created before compile time. For transparent graphical objects like sprites it may also be necessary to apply logical bit operators to the blit methods in order to allow masking etc. A minimal class definition could therefore look like this:

```
class CSurface
{
public:
    CSurface(); //attach surface to display ("primary surface")
    CSurface(const CSize &Dimensions); //create an empty surface
    CSurface(int nID); //create a surface from a bitmap resource
    virtual ~CSurface(); //get rid of the allocated surface memory
    //blitting
```

```

    void Blit(CSurface *pTarget, const CRect &rcSourceArea, const CPoint
&ptTarget, EBitMode eBitMode = BIT_SET);
    //plus operations on the surface like Clear(), SetPixel() etc.
protected:
    void *m_pBitmap; //memory allocated to hold the surface
};

```

The above example contains references to several classes that hold data structures like points or rectangles (along with their corresponding methods). It is important to implement these in accordance with the types defined in e.g. `Rect.h` in the Palm SDK, so that simple type casts suffice to use them as parameters for PalmOS functions.

Implementing the surfaces on the Palm is not too complicated. Essentially, there are two ways to fill the `CSurface` class. The former is potentially faster but requires more custom code, the latter is very comfortable but may be too slow for very demanding action games.

1. Using BitmapTypes

PalmOS defines a `BitmapType` structure in the `Bitmap.h` header file that refers to bitmap resources. The exact definition of `BitmapType` varies with the different releases of the OS as new features like system supported transparency (which is much too slow to be of any value in games) were added. Therefore, depending on the SDK release not all of the structure's members will be available. The essential leading members are always there, though:

```

typedef struct BitmapType
{
    Int16          width;
    Int16          height;
    UInt16         rowBytes;
    BitmapFlagsType flags;
    UInt8          pixelSize;          // bits/pixel
    /* ... */
} BitmapType;

```

What follows directly after the `BitmapType` structure is a memory area of size `height * rowBytes` bytes that stores the actual image data. On OS 3.5 and up there is also a function to get the address of the image data (`BmpGetBits()`) that should be used for color bitmaps¹ with custom palettes, instead.

¹ Palettes are stored between the `BitmapType` struct and the image bits.

In order to have the maximum performance, a set of self-defined image manipulation routines can directly operate on the image data. The routines provided by PalmOS are general-purpose ones and thus are very flexible, but not too fast. For instance, using a hand-tailored method for drawing horizontal lines is always faster than using the Palm's arbitrary line drawing function. Also, the blit routines for e.g. tile-based games can make certain assumptions about both the characteristics of the tiles, and the characteristics of the target surface, for instance tiles will never be clipped or drawn at uneven horizontal positions so that bit shifting on 2 or 4 bit display modes can be excluded beforehand (see illustration 2).

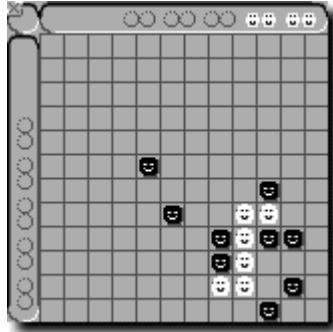


Illustration 2: Characteristics of the graphics afford specialised blit routines.

Depending on the actual constrains in flexibility, the gain of custom code will be significant, if not dramatic. For instance, blitting a surface to a target of equal dimensions is up to 170% slower in PalmOS than it is with a simple while() loop operating directly on the image memory. This is shown below:

```

/* first with custom code */
UInt32 dwStart, dwTicks, nCount;
VoidHand hBitmap = DmGetResource(bitmapRsc, IDB_SplashScreen);
BitmapPtr pBitmap = MemHandleLock(hBitmap);
UInt32 *pDisplay = (UInt32 *) WinGetDisplayWindow()->displayAddrV20;
UInt32 *pBits = (UInt32 *) (pBitmap + 1);
int nDWORDS = (pBitmap->rowBytes * pBitmap->height / 4);
UInt32 *pEnd = pDisplay + nDWORDS;
dwStart = TimGetTicks ();
for(nCount = 0; nCount < 10; nCount++) //draw image ten times
{
    while(pDisplay < pEnd)
        *pDisplay++ = *pBits++;
    pDisplay -= nDWORDS;
    pBits -= nDWORDS;
}
dwTicks = TimGetTicks() - dwStart; //10 ticks for 160*160 pixel

```

```

/* then with PalmOS */
dwStart = TimGetTicks ();
for(nCount = 0; nCount < 10; nCount++) //draw image ten times
    WinDrawBitmap (pBitmap, 0, 0);
dwTicks = TimGetTicks() - dwStart; //27 ticks for 160*160 pixel

```

The good thing about the Palm's memory architecture is that there is no difference between RAM and data media. Therefore, creating surfaces from bitmap resources does not require any memory allocation. As in the example above, you only need to obtain a pointer to the memory area where the bitmap resource resides (provided that the resource is uncompressed). It is important to realize, however, that any resources on the Palm are (fortunately!) in write-protected memory. Thus attempts to modify the contents of a resource surface will fail. To ensure that no unwanted write attempts occur, you could set a flag in the constructor indicating if it is a resource, the display, or a memory surface. It would probably be better, though, to introduce a base class for CSurface, e.g. CBitmapResource, that represents bitmap resources. The blit methods would then be moved to the base class as they are, while any methods modifying the bitmap itself remain in CSurface. In other words: The surface would be a specialized bitmap resource in that it can be modified.

In contrast to resource surfaces, ordinary memory surfaces must allocate `sizeof(BitmapType) + height * width / (8 - color depth)2` bytes and fill the members of the leading `BitmapType` struct in the constructor. For OS 3.5 and up, a simple call to `BmpCreate()` is sufficient.

Surfaces that reference the display memory should set their `m_pBitmap` member to `WinGetDisplayWindow()->WinGetBitmap()` with OS 3.5 and up. Unfortunately, not only `Bitmap.h` has changed, but also `Window.h` was different in previous OS releases in that the bitmap memory of a window was previously not referenced with a `BitmapType` struct (for reasons one can hardly duplicate). Instead, the `WindowType` struct had (and still has) a `displayAddrV20` member that directly points to the actual bits. For the `WindowType` returned by `WinGetDisplayWindow()` this member points to the bits of the display memory. To address this inconsistency, there are several more or less ugly solutions. The display surface could be wrapped into a `CDisplay` class that has its own implementation, a `GetBits()` method could be added to `CSurface`, or `m_pBitmap` could point to `displayAddrV20 - sizeof(BitmapType)` with the content of the `BitmapType` struct stored somewhere else within `CSurface`. No matter which way you choose, you cannot integrate all the three kinds of surfaces without reflecting parts of their differences in the code.

² provided that width is a multiple of (8 - color depth) and color depth is no more than 8 bpp.

2. Using WindowTypes

The direct memory access inevitable for BitmapTypes can be avoided by implementing CSurface with the WindowTypes defined in `Window.h`. There is no obligation to take the conceptual differences between the primary display surface, resource surfaces, and ordinary memory surfaces into account. The constructors just need to call `WinGetDisplayWindow()` or `WinCreateOffscreenWindow()` and save the returned handle. If applicable, bitmap resources can be drawn to the window with `WinDrawBitmap()`, which implies a more expensive creation of resource surfaces than with BitmapTypes. For convenience, PalmOS provides a wealth of drawing routines that work on windows. The `WinCopyRectangle()` function (while not faster than `WinDrawBitmap()`!) even understands various bit modes for blitting.

To modify the WindowTypes, all there is to do is to call `WinSetDrawWindow()` with the stored handle and then apply any appropriate PalmOS functions. Custom surface manipulation can also be implemented, since the `displayAddrV20` member of the WindowType points to the corresponding bits.

The disadvantages of using WindowTypes lie in the potential performance loss due to the slow PalmOS graphics routines, the expensive resource surface creation, and the administration that PalmOS saddles windows with (window lists etc.). Nevertheless, if performance is not of utmost priority, WindowTypes offer a comfortable solution that can even be safe for future OS releases, since no data structures from any of the header files must necessarily be dealt with.

OS 3.5 and later enables both approaches described above to be mingled effectively. For instance, if manipulating the surfaces can take place in a non time-critical part of the application, then it may be a good idea to implement the surfaces using BitmapTypes and create temporary windows around the surfaces to be modified. This enables the graphics routines PalmOS provides for WindowTypes to be applied to BitmapTypes. Wrapping the bitmaps can simply be performed with a call to `WinCreateBitmapWindow()` in perfect analogy to DirectDraw surfaces making use of the slow, but sophisticated GDI functions Win32 provides for device contexts by calling `IDirectDrawSurface3::GetDC()`.

The Win32 implementation of CSurface is trivial, since the surface metaphor is actually borrowed from DirectDraw. Even though you will most probably have your own routines, you might want to dig out the `ddutil.h` & `ddutil.cpp` files from the DirectX SDK, because they provide almost everything you need for the implementation (e.g. `DDCreateSurface()`, `DDLLoadBitmap()`). In addition, there are heaps of other DirectX helper routines, class wrappers, etc. freely available on the net. Since performance is not an issue on the Windows side, there is no need to write tailored code there.

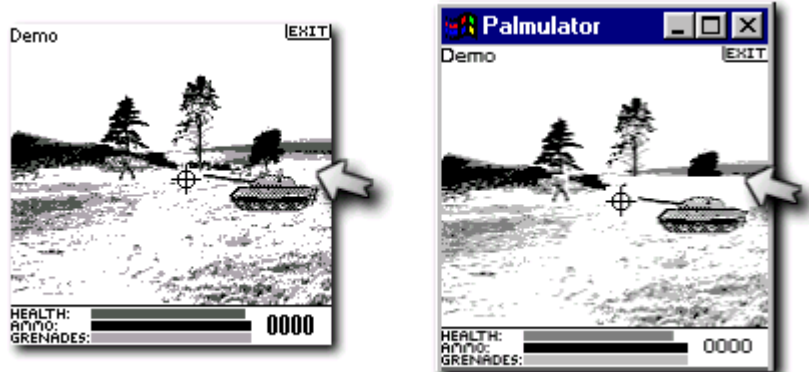


Illustration 3: Transparency does not necessarily have to be transferred.

The fundamental aim of the Windows side is to provide a platform for testing. Therefore, the functionality of the Palm side does not have to be re-implemented in its entirety. Usually, it is sufficient to implement just as much to get the big picture. Transparency, for instance, can be achieved easily with color keying. However, this does not seem necessary, because even with visible bounding boxes a fast moving action game can save the impact of its Palm counterpart (see illustration 3).

According to the above, most of the CSurface methods can thus be implemented with simple one-liners like the ones shown below:

```
extern LPDIRECTDRAW lpDD; // DirectDraw object.
```

```
CSurface(const CSize &Dimensions);
```

```
{
    m_pBitmap = DDCreateSurface(lpDD, Dimensions.x, Dimensions.y);
}
```

```
void CSurface::Blit(CSurface *pTarget, EBitMode eBitMode)
```

```
{
    ((IDirectDrawSurface *) (pTarget->m_pBitmap))->Blit(NULL, ((IDirectDrawSurface *) m_pBitmap), NULL, DDBLT_WAIT, NULL);
}
```

Sound output

Sound programming is not a big topic on the Palm. The currently used Motorola Dragonball CPU inside the Palm devices is capable of producing nothing more than cheap and annoying beeps. Even though there are approaches that allow for rudimentary music with multiple virtual sound channels (for instance our own Palm Tracker:-)), the quality of the sound output is unbearable. Since switching between sound channels eats up too much CPU cycles to do

anything more than message processing in parallel, music is only applicable for static game sections like the title or end screen. For in-game action you are practically limited to describing the beeps with the parameters PalmOS understands, i.e. frequency, duration, amplitude and then playing them asynchronously with a call to `SndDoCmd()`.

Therefore, a wrapper class could just be made from the Palm's `SndCmdIDType` struct with the appropriate methods coming behind the struct. Fortunately, on the Windows side you can skip the implementation completely as long as sound is not a central part of your game. To summarize, you will probably not bother much with sound programming, and it is not surprising that the majority of Palm games does not really make use of music or sound f/x at all.

Input

Due to the very different interaction styles between using a PDA and a desktop PC it might at first seem difficult to transfer the Palm's input handling to a Windows machine. However, quite on the contrary, there is little to do in order to have the Windows side deal with Palm UI events. On the Palm side there are three kinds of actions that must be processed:

1. Tapping or moving on the display with the pen.
2. Entering text with Grafitti.
3. Pressing any of the six hardware buttons.

The former two actions generate events in perfect analogy to the corresponding Windows events. Thus for the pen the Palm's `penDownEvent` matches a `WM_LBUTTONDOWN`, `penUpEvent` is `WM_LBUTTONUP` and a `penMoveEvent` is a `WM_MOUSEMOVE` (with `wParam | MK_LBUTTON`). Also, for entering text the `keyDownEvent` on PalmOS matches a `WM_KEYDOWN` on Windows.

Since the `EventType` struct defined in the PalmOS `Event.h` header is very appropriate for storing UI events, a wrapper class is not really necessary or sensible. The Palm's event handling forms the basis for the event handling on the PC.

Consequently, the Windows side has to translate parts of the messages received by the application into the corresponding `EventTypes` and then send them to the framework for interpretation. One suitable place to intercept and translate system messages is the `WindowProc()` function. By simply pasting the relevant parts of the `Event.h` header into a corresponding header on the Windows side, the Windows application becomes enabled to fill the necessary members of the `EventType` struct, as is exemplarily shown below for a `WM_LBUTTONDOWN` that is translated into a `penDownEvent`.

```

long FAR PASCAL WindowProc( HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam )
{
    switch ( message )
    {
        case WM_LBUTTONDOWN:
            POINT ptCoords;
            ptCoords.x = LOWORD(lParam);
            ptCoords.y = HIWORD(lParam);
            ConvertToPalmDisplay(hWnd, &ptCoords);
            EventType Event;
            Event.eType = penDownEvent;
            Event.screenX = ptCoords.x;
            Event.screenY = ptCoords.y;
            Event.penDown = true;
            CApplication::GetCurrentAppState()->Input(&Event);
            break;
        ...
    }
}

```

A proposed CApplication class to which the `penDownEvent` can be sent will be defined below. Handling the remaining messages is similar to the `WM_LBUTTONDOWN` example. Messages relating to mouse positions should call a transformation function like the `ConvertToPalmDisplay()` in the example. Such a transformation routine allows for arbitrary application window sizes to which the Palm's tiny 160*160 pixel display is mapped, thus avoiding the inadequacy of aiming for single pixels on a PC monitor. A straightforward implementation of this function could be this :

```

void ConvertToPalmDisplay(HWND hWnd, POINT *ptPoint)
{
    //PALM_X & PALM_Y are 160 pixel each
    RECT rcClient;
    GetClientRect( hWnd, &rcClient );
    ptPoint->x = PALM_X * ptPoint->x / (rcClient.right - rcClient.left);
    ptPoint->y = PALM_Y * ptPoint->y / (rcClient.bottom - rcClient.top);
}

```

Pressing any of the six hardware buttons at the bottom of the Palm device generates `keyDownEvents` similar to entering Graffiti text in the silkscreen area. For games utilizing the hardware buttons, this, in most cases, is inadequate, because the events generated might easily pile up in the event queue. Instead, keys should be polled, which is supported on the Palm platform with the `KeyCurrentState()` function. The corresponding `keyDownEvents` can

then safely be eaten. Additionally, `KeySetMask()` allows to mask off certain keys resulting in a (minor) performance gain.

On the Windows side, polling is best implemented using `DirectInput`. However, it is usually sufficient to handle key presses just like ordinary `WM_KEYDOWN` events. This means that the Palm side polls keys and eventually generates `keyDownEvents`, whereas the Windows side does not poll, but generates the same `keyDownEvents` after receiving `WM_KEYDOWNs` from those keys that map the Palm's hardware buttons. Thus, no specialized code is necessary here, as well.

Additional requirements

The major platform-specific components related to game programming were identified above. Further issues of specific interest to Palm games will now be investigated.

A game for the Palm Computing Platform must be able to react on the system's request to stop the application at any time. This does not only include the user's wish to switch to another application, but due to the Palm's single process design also implies reacting properly on system generated events that e.g. fire an alarm. Furthermore, especially for games it is essential to continue the application at the exact point where it was left before. The game does not only have to take care of serializing the data necessary to restore its current state, but also has to make sure that the display is properly restored. The amount of data serialized at game startup/ termination should be small enough to create the impression of instant switching between applications. Therefore, dumping the video memory as part of the application termination is usually a bad idea.

In contrast to most serious applications, games tend to have several sections or states that might differ radically from each other in terms of user interaction and functionality. For instance, a typical space invaders clone would at least consist of a title screen, a high-score table, and, of course, the shooting action itself. Each of these application states will most probably have different demands when it comes to serializing its necessary data or reconstructing its display information. For instance, quitting and re-entering the title screen will probably not require any serialization and repainting will thus be trivial. The shooting state, on the other hand, will need to gather information on restart about e.g. enemy positions and build up the display accordingly. Furthermore, regarding the serialization it may matter much, if the application state to be saved is "active" or not: The high-score table will always need to serialize the scores, but when it is active, it may additionally have to save positions of decorating spaceships that float in the background.

To summarize, Palm applications are required to terminate anytime and they must do it fast, i.e. the amount of data to be saved for an appropriate restart must be minimal. For a game, an appropriate restart includes restoring the display information. Thus, it may be wise to wrap all

the different application states into distinct classes that are derived from an abstract CAppState class with the following requirements:

1. Serializing related data in dependence of being the active appstate
2. Updating the display in its entirety as well as incrementally in the course of the game (e.g. moving sprites).

Translated into a class definition the CAppState class might then look like this:

```
class CAppState
{
public:
    CAppState();
    virtual ~CAppState();

    virtual void Draw(bool bRedrawAll) = 0;
    virtual void Serialize(CFile *pFile, bool bActive, bool bSave);
    //the remainder is addressed in the next section
    virtual void Input(EventType *pEvent);
    virtual void Process();
    virtual void Sound();
    virtual void Initialize(CAppState *pSwitchedFrom);
};
```

Please note that the class definition contains a pointer to a CFile object that holds the serialized data. PalmOS, however, only offers a database approach as a substitute for a real file-system. Since most programmers will be more familiar with the file metaphor than with the Palm's database Creator IDs and Appl-Type ushorts, it is more intuitive to create a CFile class (with real file names etc.) and again hide the Palm specific details in the implementation.

After the file is created, data must be exchanged. Fortunately, I/O streaming is supported on both platforms, so methods like `read(byte *pData, word nLength)` or `write(byte *pData, word nLength)` are trivial to flesh out. As long as the game's data is suitable for simple streaming, a CFile class with nothing more than the methods defined above plus constructor, destructor, and possibly a static method checking for file existence, works well enough.

For large amounts of data to be serialized (something you should avoid anyway!), it may be sensible to implement the file class with the Palm's traditional Data Manager instead of the file streaming functionality due to a certain performance penalty involved with streaming in PalmOS.

Application structure

The typical flow of a game is a repetitive cycle of processing computer AI, evaluating user input, displaying graphical changes and possibly playing a sound effect. Therefore, a game's main function would perform these actions perpetually:

```
void CApplication::Run()
{
    for(;;)
    {
        if(Input() == QUIT_APP)
            break;
        Process();
        Draw();
        Sound();
    }
}
```

As stated in the previous section, the concrete steps performed when processing or drawing depends on the state or section a game is in. Accordingly, CApplication could maintain a list of its appstates and handle the commands through to the active appstate. Therefore, e.g. the `Process()` method could be implemented like this:

```
void CApplication::Process()
{
    if(m_pCurrentAppState)
        m_pCurrentAppState ->Process();
}
```

Only the `Input()` method needs to have a specific implementation for each of the platforms, as was shown before.

For transitions between appstates it may be sensible to add a `SwitchTo()` method that sets the active appstate and calls the corresponding `Initialize()` in CAppState, so that the activated appstate can specifically react depending on which appstate was active before.

Concluding, the CApplication class could be defined as below:

```

class CApplication
{
    friend class CAppState;
public:
    CApplication(); //in derived classes the appstates can be defined here
    virtual ~CApplication();
    void Run(); //called in the main() function
protected:
    CAppState **m_pAppStates;
    CAppState m_pCurrentAppState;
    void Draw();
    word Input();
    void Process();
    void Sound();
};

```

An instance of the CApplication class (resp. a derived class tailored to the demands of the game) would then be placed in the appropriate main functions like `WinMain()` on Win32 or `PilotMain()` on the Palm:

```

unsigned long PilotMain (Word cmd, Ptr cmdPBP, Word launchFlags)
{
if (cmd == sysAppLaunchCmdNormalLaunch)
    {
        CMyApp theApp;
        theApp.Run();
    }
    return 0;
}

```

Further issues

For the Win32 side you will probably want to use an existing windowed-mode application and plug the additional code required for re-building the Palm game into it. We found that a slightly modified sample from the DirectX SDK works fine. We grabbed the 'Inawin' example from the 'Inside DirectX' book (ISBN: 1572316969) which is a derivate of the infamous donut application found in the SDK (see illustration 4).



Illustration 4: Re-inventing the wheel (resp. donut) is often unnecessary.

Also, even complex GUI's can be built from the atomic user interface components proposed in this lecture. For instance, the (yet unfinished) project shown in illustration 1 features two draggable windows with buttons and slider controls. However, it is questionable, if creating more advanced GUI's without utilizing what the Palm already provides contradicts with the resource constraints in mobile devices. On the other hand, there is nothing stopping you from implementing sophisticated controls like edit fields with platform specific code, so it comes down to a question of proper balancing.

After all, the methods presented here are a compromise between cost of implementation, speed and workability. For your own projects you might have different demands, so you would want to realize things quite differently. For typical, i.e. simple Palm games with moderate demand on resources, an application framework similar to the one proposed here seems to be adequate enough, as we have experienced during the development of our own titles.

About the author

Carsten Magerkurth is lead programmer at EmperoR Studios (www.emperor-studios.de). Previous work include 'Palm Tracker', the Palm's first and only three-channel Soundtracker/MOD-Tracker, the 'Palm Tracker SDK' that allows Palm developers to add music to their creations, 'ARCHON', the classic Free Fall game revived for the Palm, and recently 'Stormfront', the game of shooting people dead. When Carsten is not writing Palm games, he develops creativity support software at the German National Research Center for Information Technology (<http://www.ipsi.gmd.de/ambiente>).