

DirectX and Wine

Andrew Lewycky andrew@transgaming.com
and Gavriel State gav@transgaming.com

July 16, 2001

Abstract

The Wine project is an ambitious attempt to create an open implementation of the Win32 API and runtime environment for Linux and other Unix-like systems. This paper covers the issues surrounding the implementation of the DirectX family of multimedia APIs within Wine, with an emphasis on the implementation of Direct3D via OpenGL and recent GL extensions supported by current Linux drivers.

We also provide a background introduction to some of the underlying Wine features that make our work possible. These include the Portable Executable loader, Wine's implementation of COM, and Wine's modular graphics driver architecture.

1 Introduction to Wine

Wine allows Unix systems to run Windows programs. It supports two execution models: the binary loader and the compatibility library.

The binary loader allows the user to run Windows binaries directly. It is capable of loading Portable Executable (PE) formatted EXE and DLL files.

The compatibility library model can be used to compile Windows programs from source to create new native Unix programs. A replica of the Windows SDK is provided for developers.

Both models supply an implementation of the Windows API as a set of ELF format shared libraries, with a mapping from Windows DLLs to shared library.

2 Binary Loader

2.1 Portable Executables (PE)

32-bit Windows executables (including DLLs) are stored in the Portable Executable format, a derivative of COFF. PE has features roughly equivalent to the widely used ELF format.

For more on the PE object file format, see http://msdn.microsoft.com/library/specs/msdn_pecoff.htm or http://msdn.microsoft.com/library/techart/msdn_peeringpe.htm. For more on COFF see <http://www.delorie.com/djgpp/doc/coff/>.

2.1.1 Relocation and PIC

While Windows guarantees that all EXEs are always loaded at the same virtual address it cannot do the same for DLLs.¹ Instead each DLL has a base address, indicating the address that it can be loaded at most efficiently. If it is loaded at another address, it will need to be relocated.

PE DLLs don't use position independent code. Instead they have a fixup table listing all locations in the DLL image that depend on the load address. (For example, addresses of global variables.) The PE loader computes the difference between the the DLL base address and actual load address, and appropriately modifies the executable image at run-time.

Because there isn't a dedicated PIC register, this scheme avoids any run-time penalty. However, there can be a large load-time penalty if the DLL is not loaded at its base address. The usual cause for this

¹ELF is the same in this regard.

is that all DLLs have the the same default base address. Microsoft provides a tool for “rebasng” a DLL and it is widely used.

In Windows the fixup modifications take place when the kernel loads executable pages from disk. In Wine, the modifications take place globally when an image is loaded, making it impossible to use `mmap` to load images that need relocation. A wine-server kernel module is under development that can perform the fixups when a page is faulted in, amongst other features. See <http://cvs.winehq.com/cvsweb/kernel-win32/>.

2.1.2 Dynamic Linking

In PE, symbols are considered as (DLL, name) pairs, so different DLLs can define distinct symbols with the same name. Names that are not explicitly exported or imported from a DLL are considered local and are only resolved from within that DLL.

PE also provides a rarely-used feature for importing symbols by “ordinal” (a numeric index) instead of by name.

Once the imported symbols have been found, their addresses are written into the Import Address Table. The code and data sections are not modified during dynamic linking. Thus it’s impossible to statically initialize pointers with the addresses of imported data. For function pointers, the address of a thunk is used instead.

2.2 ELF

2.2.1 ELF binaries

When a Windows program is compiled against the Wine SDK, it is linked by `gcc` into an ELF shared library, but dynamically loaded by the same ELF executable that contains the PE binary loader. The DLLs created by Wine are similarly compiled into ELF shared libraries.

It should be noted that there is no significant performance benefit to loading ELF binaries instead of PE binaries. The most significant advantages of using ELF are portability to non-x86 CPUs, and the ease of debugging ELF binaries with `gdb`.

2.2.2 .spec files and name resolution

In order to simulate PE DLLs using ELF shared libraries, we need to work around the difference in how names are resolved. Unlike PE, ELF assumes that there is only a single instance of each symbol name across all loaded libraries. If multiple libraries define the same symbol then the first version loaded overrides the others.

Hence the ELF dynamic linker cannot be used to resolve imported symbols. Instead Wine ELF DLLs are created through a complicated linking process that essentially disables the ELF dynamic linker and stores enough information in the ELF shared library that Wine can perform PE-style dynamic linking when it is loaded. This also allows Wine to mix and match its own “built-in” ELF format DLLs with “native” Windows DLLs

Each project needs a `.spec` file that contains a list of DLLs that symbols can be imported from. The `winebuild` tool scans the `.o` files to determine what symbols will need to be imported. It then scans the DLLs to determine which DLL provides that symbol.

Based on this information, `winebuild` generates a `.spec.c` file that is compiled and included in the final link. For each imported symbol, the `.spec.c` file has the name of the DLL from which it is taken, an element in the Import Address Table that will be filled in with the symbol’s address at run-time, and a call thunk that redirects the call to the address in the import address table.

The program is then linked with the `-Bsymbolic` flag. This tells the ELF dynamic linker to prefer symbols within the that shared object to those provided by other shared objects, causing any references to an imported symbol to be linked to the call stub.

Finally at runtime, Wine fills in the imported address table, so that the call stubs will use the correct addresses.

3 COM for Wine

3.1 COM Introduction

COM is Microsoft's Component Object Model. It provides a mechanism for making method calls on objects that is independent of language or platform.

Objects are accessed through interfaces that are defined in an "Interface Definition Language" that is roughly programming-language independent. COM defines a binary interface (i.e. the vtable layout and calling convention) that is used when the object is in the same address space as its caller, and a network protocol that is defined in terms of DCE for when the caller and object are in different processes or on different computers.

COM requires that all objects support some basic object lifecycle capabilities. All COM objects can be instantiated by calling `CoCreateInstance` and they must support the `IUnknown` interface. `IUnknown` provides methods that are used to maintain a reference count for the object. Once the reference count reaches 0, the object destroys itself. The other key `IUnknown` feature is `IUnknown::QueryInterface`, which can be used to access other interfaces supported by the object.

Many new APIs developed by Microsoft are defined in terms of COM objects. Major examples are the shell interfaces, ADO and DirectX.

3.2 Wine COM capabilities

Wine support for COM is currently quite limited. Wine supports only the most basic COM functionality: the implementation of interfaces that are binary compatible with the COM vtable standard, and the `CoCreateInstance` object instantiation mechanisms. `CoCreateInstance` makes use of the Windows registry to match a unique interface identifier known as a GUID, or Globally Unique Identifier to the DLL that contains the interface implementation.

Wine only supported so called "in-process" COM. In other words, creating and using objects only within the address space and other context of a single process. Wine does not implement the marshalling of data for method calls to objects in other

processes, or across a network. This is a significant issue for Wine because certain popular Windows installers make use of "out-of-process" COM to separate the underlying installer engine from individual installers. Thus, until Wine implements marshalling, there will be significant issues for users wishing to install certain off-the-shelf applications in Wine.

3.3 COM implementation

The most critical issue for implementing COM interfaces in Wine is the need to have an exact match with the COM vtable format. For a wide variety of reasons, Wine is implemented entirely in C, and does not use C++ at all. As such, Wine mimics the COM vtable format using simple arrays of function pointers. A complex macro system is used in the Wine headers to allow access to COM interfaces from both C code and C++ code.

Use of COM interfaces as though they were standard C++ objects is complicated by a serious issue. The vtable format used by COM interfaces is not the same format that is used by default in g++. g++ vtables include initial pointers to dynamic typing information and exception tables that are located elsewhere in the COM vtable format.

Fortunately, recent versions of g++ have added compatibility support for the COM vtable format, through the use of the `com_interface` attribute in a class or structure definition. Use of this attribute allows WineLib applications to have direct use of COM objects, making it possible for many such programs to compile in g++ with minimal source changes.

The various COM interfaces for objects implemented in Wine are often aggregated into a single underlying implementation object. For example, Wine uses one implementation object to implement four different versions of the `IDirectDraw` series of interfaces. The latest interface - `IDirectDraw7` - is implemented directly, and previous versions are implemented in terms of the latest version of the interface wherever possible. In addition to the `IDirectDraw` interfaces, the same implementation object also implements the `IUnknown` interface, as well as the `IDirect3D` series of interfaces. C macros take the place of C++ style `static_cast` operators within the implementation of the thunks.

4 Wine Graphics and Windowing Driver Architecture

Two of the most important functions of any GUI system such as Windows are low-level graphics, and window management. In Windows these are handled by the GDI and USER subsystems, respectively. Wine provides a flexible back-end driver architecture that allows GDI and USER to be implemented on top of the graphical interface system of choice for the platform. Currently, only the X11 back-end is fully supported, though an experimental TTY back-end has also been implemented. In the future, graphics systems such as Apple's MacOS X Quartz API could be supported as well.

In the Win32 API both low-level objects used to represent graphics context and state information, and high-level objects representing windows, menus, controls, and such are managed through opaque "handles". Applications using the API have no access to the underlying structures used to implement the objects, thus giving the system a certain amount of object-oriented abstraction. This abstraction has proven useful to Microsoft, allowing them to have compatibility across operating systems such as Windows 95 and Windows NT, despite the wide underlying differences between them. This abstraction is similarly useful to Wine.

The most important of these handle-based object are HWNDs and HDCs, which represent windows and graphics contexts, respectively. HWND objects are handled by the window driver, through the USER subsystem, while HDCs and several associated objects such as HBITMAPs, HBRUSHes, and HPENs are handled by one of several back-end graphics drivers through the GDI subsystem.

From the DirectX perspective, both the graphics and windowing drivers are key - DirectX needs to know which window to draw into, and must also make use of key graphics driver-related objects such as HBITMAPs.

4.1 Window Driver

The window driver is used by USER to actually implement windowing, event handling, the clipboard and some miscellaneous other functions.

Microsoft's Windows does not have window drivers. The only implementation is provided directly within the USER DLL, making direct use of GDI for rendering the outward look and feel of its windows. Since in the future, Wine may support windowing systems other than X11, the Wine version of USER implements an indirection layer to pass requests to the window driver.

For example, when the USER function `CreateWindowEx` is called, the Wine USER code initializes the high-level structure behind the opaque HWND object, and then calls the window driver to perform the low-level duties.

Unlike in many other GUI systems, Windows HWNDs are used to represent both top-level user-controllable windows as well as control objects such as buttons, scroll bars, and menus within a top-level window. Since Wine windows must co-exist with the X11 windowing environment, special case code is required in the X11 window driver to force top-level windows (children of the Windows "Desktop" HWND) to be treated differently. This code optionally allows the X11 Window Manager to handle functions such as moving, resizing, and drawing the window border.

Thus, when creating a new window, the X11 window driver must determine whether the window to be created is a top-level window. If it is, the XLib `XCreateWindow` function is called to create the window.

Similarly, when X11 events are received, they are translated into the equivalent Windows messages, and deposited into the Windows event queue.

4.2 Graphics Drivers

Graphics drivers are used by GDI to implement all rendering in the system. There are a wide variety of graphics drivers, for several different purposes - the X11 driver is used to perform drawing to the screen and to off-screen bitmap objects. Wine also supports graphics drivers that output PostScript for printing, as well as WMF and EMF graphics metafiles.

These drivers are also present under windows and have similar responsibilities. In fact, Wine includes a thunking layer that can actually load and use na-

tive 16-bit Windows printer drivers to drive local printers.

As an example, consider `X11DRV_Rectangle` which implements the GDI Rectangle API call in the X11 driver, drawing a rectangle with a border and a filled interior. The API takes an `HDC` identifying the device context that refers to a “pen,” object which stores line-drawing attributes, and a “brush,” object which stores area-filling attributes. In X-Windows, pens and brushes don’t exist, instead the `GC` (“graphics context,” a concept very similar to an `HDC`) possesses these attributes itself. Thus when the pen associated with an `HDC` changes, the `GC` must be updated.

`X11DRV_Rectangle` proceeds roughly as follows:

- the rectangle coordinates are converted from logical coordinates to world coordinates using the viewport transform.
- the coordinates are checked to ensure they are in the correct order, i.e. with `left` less than `right`.
- special computations for the `PS_INSIDEFRAME` pen style are performed. This style is not supported by X but can easily be emulated by adjusting the pen’s geometry.
- if rendering into a `DIBSection`, the `DIBSection` is locked, potentially requiring resynchronization. More information on `DIBSections` can be found below.
- the `HDC`’s brush settings are copied into into the `GC`.
- `XFillRectangle` is called to draw the rectangle’s interior.
- the `HDC`’s pen settings are copied into the `GC`.
- `XDrawRectangle` is called to draw the rectangle’s outline.
- if rendering into a `DIBSection`, the `DIBSection` is unlocked, and marked as having been modified by GDI. More information on `DIBSections` can be found below.

4.2.1 HBITMAPS, DIBs and DIBSections

GDI supports two kinds of bitmaps - device dependent bitmaps, and device independent bitmaps. De-

vice dependent bitmaps are handled entirely in the graphics driver, and are referenced by applications through opaque `HBITMAP` handles. `HBITMAP` objects are very fast to work with, since a driver can theoretically store them in video memory. Applications never have direct access to the bitmap data in an `HBITMAP`. And finally, `HBITMAP` objects can be selected into special `HDCs`, allowing applications to make GDI rendering calls.

By contrast, device independent bitmaps, or `DIBs`, are managed directly by application code. Applications are responsible for allocating and deallocating memory for them, and for ensuring that bitmap data is stored in a format consistent with the standards set out in documentation. There are a variety of different parameters for `DIBs`, allowing applications to specify whether the image data is in top-down or bottom-up format, color-mapped or true-color, or even compressed. Only application code can make modifications to `DIB` data directly.

While these two kinds of bitmaps have close analogues in X - as `XPixmap`s and `XImage`s, respectively - GDI also supports a combination of the two, known as a “`DIBSection`”. A `DIBSection` is an `HBITMAP` that also allows direct application access to image data. While this is similar to a shared memory `XPixmap`, the `DIB` data formats are not directly interchangeable with the `XPixmap` formats supported by most X Servers. This results in some severe complications for Wine’s X11 graphics driver.

In order to support `DIBSections`, the X11 driver represents `DIBSections` with both an `HBITMAP`, represented internally by an `XPixmap`, and with a `DIB` and an associated shared memory `XImage`. The `DIB` data and the `XPixmap` data are kept in sync automatically. This synchronization uses memory protection mechanisms to keep track of when the `DIB` data has been modified by application code. The modified data is copied to the `XPixmap` before any GDI rendering calls are made. Similarly, if the `DIBSection` has been modified by GDI rendering calls, at the next application access to the data, it is copied from the `XPixmap` to the `DIB`.

In the future, it may be possible to improve `DIBSection` rendering performance by introducing a graphics driver that can rasterize directly to a `DIB`, without using any `XLib` calls at all. This will eliminate much of the data format conversion overhead that exists with the current architecture.

DIBSections play a key role in the Wine implementation of DirectX, as we shall see below.

5 DirectX

DirectX is a blanket name for a number of APIs covering a wide range of game and multimedia related functions, including 2D and 3D graphics, sound, input, and networking. Wine implements almost all of these APIs to one degree or another, though currently only up to version 7 of DirectX. DirectX 8, introduced by Microsoft to support the features of their new X-Box game console, is not yet supported.

Discussing the details of each of the DirectX APIs is well beyond the scope of this document. Instead, we shall focus specifically on the 2D and 3D graphics APIs - `DirectDraw` and `Direct3D` - which have recently been extensively rewritten by TransGaming Technologies.

All the DirectX APIs make extensive use of COM. Using the APIs generally involves creating COM objects and requesting a known interface. In this fashion, upgraded versions of the API can be introduced which expose both newer and older interfaces.

5.1 DirectX Graphics Architecture and the X11 HAL

Several key COM interfaces make up the bulk of the DirectX graphics APIs: `IDirectDraw`, `IDirect3D`, `IDirectDrawSurface`, and `IDirect3DDevice`.

Each of these interfaces, and the way in which they are implemented in Wine, are discussed in detail below.

Underpinning all of the implementations of these interfaces is the `DirectDraw Hardware Abstraction Layer`, or HAL. In Windows, the HAL is effectively a driver interface, used to provide a way to implement a common graphics API across a large number of different hardware devices. Wine too, uses the HAL architecture, but to provide an abstract interfaces to lower-level software systems, rather than hardware. The HAL interface is exposed by the X11 graphics driver as a dispatch function that the higher-level `DirectDraw` code makes use of.

Some examples of the functions that are handled in the HAL layer are: Device resolution switching, setting hardware gamma parameters, and glX context management. The HAL architecture allows us to abstract the use of the X11 DGA extensions to change device modes - the core `DirectDraw` code never needs to know anything about what must be done to set up DGA. If DGA is not available, the X11 HAL simply tries to use the `XF86VidMode` extension to re-size the viewable portion of the screen.

5.2 IDirectDraw and IDirect3D

The core of the DirectX API system is the `DirectDraw` object. All other objects that are needed for both 2D and 3D graphics are created through the use of COM interfaces exposed by the `DirectDraw` object. In addition to several different versions of the `IDirectDraw` interface, a `DirectDraw` object can expose other COM interfaces as well, such as `IDirect3D`, which is used to create `Direct3DDevice` objects, which in turn provide the key 3D API for DirectX. The other objects that can be created from an `IDirectDraw` interface include `DirectDrawSurface` objects, used to manage 2D surfaces, texture maps and more, `DirectDrawClipper` objects, which can be used to perform clipping during drawing operations such as blitting, and so on.

In addition to methods used to create other objects, the `DirectDraw` object is also responsible for manipulating the display device. It is used to associate an `HWND` with the display device, to get information about available video modes and the amount of memory remaining on the graphics card, and of course to set the resolution and color-depth of the display. For many of these operations, the Wine `DDRAW` DLL calls downward to the X11 HAL driver, to isolate itself from platform dependencies.

A `DirectDraw` object is created by using the `DirectDrawCreate` or `DirectDrawCreateEx` functions, which are exported from the `DDRAW` DLL. These functions take a `GUID` parameter that specifies which `DirectDraw` object to create - there can be several, representing different drivers that the system has available, or perhaps different modes for a single driver. The result will be a COM interface pointer which can then be queried for further interfaces, such as `IDirect3D`, and used to produce other objects, such as `DirectDrawSurfaces`.

In order to choose the most appropriate kind of `DirectDraw` object to create, an application will generally call `DirectDrawEnumerate` or `DirectDrawEnumerateEx` beforehand. These functions are supplied with a callback, and allow the application to evaluate the suitability of each available `DirectDraw` object. DirectX 7 applications which care about 3D will use the callback to test the feature-set of 3D devices available through the `DirectDraw` object. This is done by creating a temporary `DirectDraw` object from the GUID passed to the callback, querying for an `IDirect3D7` interface, enumerating the available `Direct3DDevice` objects using `IDirect3D7::EnumDevices`, and evaluating the features of each such device object in a further callback.

5.3 IDirectDrawSurface

`IDirectDrawSurface` encapsulates a 2D image and is used to represent the screen (“primary”) surface, back buffers, overlays, off-screen sprites, texture maps and depth buffers. It also is used with two special cases: execute buffers and vertex buffers, which are not images, but store other data in arrays.

Surfaces can be accessed through the `Lock` method for direct access to the surface data using a pointer, and the `GetDC` method which allows GDI APIs to render to the surface. This combination of access methods is already supported by `DIBSections`, so in Wine they are used to provide most of the surface implementation.

Calls to `Lock` and `GetDC` must be balanced with calls to `Unlock` and `ReleaseDC` respectively. This forms the system by which special surfaces such as primary surfaces are kept synchronized with the entity they actually represent. In the case of primary surfaces, when `Unlock` is called, the image contents are copied to the screen - but not immediately.

If the primary surface was copied to the screen immediately, there would be considerable overhead for programs that frequently lock and unlock the surface. Instead, the surface is simply marked as being in need of a refresh, and a background thread copies it to the screen as often as possible. The main thread thus never has to wait for an unlock operation to be completed.

Other types of surfaces are also synchronized simi-

larly - for example, while all 3D applications need a depth-buffer to provide hidden surface removal, few need to directly read or write to the buffer. For those that do, the Wine Z-Buffer surface implementation will automatically synchronize with the OpenGL depth buffer as needed, though at a potentially significant performance cost.

5.4 IDirect3DDevice

`IDirect3DDevice` is the key `Direct3D` interface as it contains the methods that are used to make changes to the 3D graphics state, apply transformations, and ultimately, render polygons. The `IDirectDrawSurface` interface is still important because it is used to access the display surface and backbuffers that `IDirect3DDevice` draws into, as well as the texture data for the polygons being drawn. In fact, `Direct3D` programs often deal with more surfaces than 2D-only programs.

The `IDirect3DDevice` interfaces are implemented in Wine through calls to OpenGL. This allows the Wine implementation of `Direct3D` to take advantage of hardware acceleration, if it exists. In many cases, Wine is able to execute `Direct3D` code on top of OpenGL almost as quickly as Windows `Direct3D` using native `Direct3D` drivers.

5.4.1 IDirect3DDevice::DrawPrimitive

There are a variety of different ways in which to draw using `IDirect3DDevice`, all of which are variants of the simplest such mechanism: `IDirect3DDevice::DrawPrimitive`. This method takes a pointer to an array of vertices and some information to tell it how to interpret the vertex data. The vertex data is stored in a packed format containing all the attributes for each vertex in turn. A set of flags specifies what attributes are present. These flags can also have side-effects. For example: if a vertex normal is present then lighting is enabled, otherwise the vertex is considered to be pre-lit.

A simple example with an untextured triangle that will be transformed and lit by `Direct3D` illustrates this in listing 1.

Parameters to `DrawPrimitive` indicate that each vertex has position, normal and diffuse colour attributes. The data is stored in a user-defined struc-

```

static const struct
{
    D3DVALUE x, y, z;    // position
    D3DVALUE nx, ny, nz; // normal
    D3DCOLOR colour;    // diffuse colour
} vertices[3] =
{
    { { 0, 5, 0 }, { 0, 0, 1 },
      RGBA_MAKE(255, 0, 0, 255) },
    /* ... remaining two vertices */
};

pd3dDevice->DrawPrimitive(
    D3DPT_TRIANGLES,
    D3DFVF_XYZ | D3DFVF_NORMAL |
    D3DFVF_DIFFUSE,
    vertices, 3, 0);

```

Listing 1: Pseudocode to render an untransformed, un-lit triangle using Direct3D.

ture, with the order and C type of the attributes defined by the Direct3D API.

Internally, a “strided” representation is used in which each attribute is considered to be stored in a separate array. The array elements are not contiguous, instead they are separated by a constant distance known as the stride. This representation can be computed from the flags indicating which vertex attributes are present.

If the `D3DFVF_XYZRHW` flag is used instead of `D3DFVF_XYZ | D3DFVF_NORMAL`, then the vertices have already been transformed by the application, so the transform must be disabled. OpenGL always performs some transform, so in this case Wine temporarily loads an identity transform.

When vertices including colour data are used, a serious problem is encountered. OpenGL requires that the colour components are in red, green, blue, alpha order, while Direct3D requires them in blue, green, red, alpha order. To circumvent this problem, Wine copies the colour data to a shadow buffer and switches the components; Wine then passes the shadow buffer to OpenGL instead of the original colour data.

The strided representation is then used to set up the other OpenGL attribute arrays, containing vertex data, texture coordinates, etc. `glDrawArrays` is then called to render the vertices.

If the `D3DFVF_NORMAL` flag is used and lighting is

on, then the vertices must be lit by OpenGL. Because Direct3D allows a more flexible mapping from vertex colours to material colours, the colour data for each vertex has to be processed individually, thus making it impossible for Wine to use the `glDrawArrays` call. The other attribute arrays are still used since OpenGL permits immediate and array data to be mixed.

In the future, Wine will check for certain special cases that can be exploited to allow for the use of `glDrawArrays` even with vertices requiring lighting in the near future. TransGaming Technologies is also exploring the possibility of creating a simple OpenGL extension that would make material colors as flexible as those in Direct3D.

5.4.2 Texture State States and Render States

Texturing in Direct3D is performed in sequential stages. Each stage is split into colour and alpha components. Each component has two inputs, each chosen from one of: the output of the previous stage, the pixel’s diffuse colour, the pixel’s specular colour or a constant colour. The component then applies a function to its inputs to produce the output from that stage. The functions and inputs for the colour and alpha components are independent of one another.

This differs widely from the multitexture support in OpenGL 1.2, in which each stage must use a texture as one input and the previous stage’s output as the other, and the colour and alpha stages are linked. Fortunately, the `GL_ARB_texture_env_combine` extension provides for texture stages that are as flexible as Direct3D. Other extensions exist to provide some missing blend functions.

In some cases, texture settings that are not related to blend modes require special handling. OpenGL breaks all texture settings into two classes, texture “parameters” that are applied to the texture object and texture “environment” settings that do not change when the selected texture changes. In OpenGL, filtering modes are texture parameters, while in Direct3D all texture settings are similar to environment settings. Thus Wine must store the filtering mode for each texture stage so that when it calls OpenGL to load a new texture, it can set the filtering mode again. The current implementation

does not track the last parameters for each texture, so they are sometimes changed unnecessarily.

Finally, Direct3D also has a wide array of render states that map to nearly identical OpenGL render states. A good example of this is the Direct3D fill mode render state - `D3DRENDERSTATE_FILLMODE` - which controls whether polygons are to be drawn filled, simply outlined, or with individual vertices showing only as points. OpenGL has a polygon mode state with the same settings, set through a simple call to `glPolygonMode`.