Chapter 22

Debugging

"He that assures himself he never errs
will always err."
Joseph Glanville: *The Vanity of Dogmatizing*, XXIII, 1661
(Cf. Nordern, *ante*, 1607)

"Blue is true, Yellow's jealous,
Green's forsaken, Red's brazen,
White is love, And black is death."
Author unidentified

22.1 Overview

Debugging ordinary programs is often a difficult and frustrating task. Direct3D programs can be even more difficult to debug and lead to even more frustration. This chapter covers techniques and advice for debugging Windows programs in general and Direct3D programs in particular.

The best way to debug a program is to write a program with no bugs. Naturally, this is almost impossible for all but the simplest of programs. However, there are techniques you can use to avoid common mistakes. First, we will discuss techniques for avoiding some of the most common pitfalls of C++ applications.

The Windows operating system also provides some facilities for debugging of applications. Using these facilities can help you obtain information about your program while it is running and can also be used to provide information about the execution context of your program when it encounters a fatal error.

Next, we will discuss the debugging facilities in Visual C++ 6. Detailed information on the debugger is found in the Visual C++ documentation, while we provide suggestions and tips for using the debugger with Direct3D applications.

Finally, we will conclude with debugging techniques specific to Direct3D programs and the unique challenges they present in debugging.

22.2 Version Control Systems

One of the most frustrating mistakes we can make is to accidentally delete the very code we have been struggling to write. This sounds like the sort of thing that you'd never do, until you realize you've already done it. Obviously, keeping backups of your code during development minimizes the chance that you will accidentally delete precious hard work.

A version control system provides for fine-grained control over the changes to the source files in your application on a file by file basis. Using a version control system is even better than a backup because you can associate a log message with each change, you can "fork" your source code into two separate development paths which may later be joined, generate differences between revisions of source code by version number or date and much more.

A full discussion of version control systems is beyond the scope of this book, but if you have never used a version control system before, it is strongly recommended that you do so. The CVS version control system is an open source, well-tested version control system. Command-line and GUI based clients for manipulating the system are available for Windows.

Using a version control system and a frequent schedule of committing changes to the system will help guard you against accidental deletion of your working source code. Of course, should the files used by the version control system for storing your changes be accidentally deleted, you've still got a problem. For this reason, it is recommended that you use a version control system for tracking changes to your source code at a fine-grained level and a regular backup schedule to guard against accidental deletion or HW storage failure.

22.3 C++ Debugging

C++ was designed to be compatible with C. As a result, it is still possible to code all the well-known pitfalls of C within a C++ application.

22.3.1 Dynamic Memory Allocation

The most common sort of nasty C bug results from improper use of the free store: dereferencing `NULL` pointers, failing to properly free dynamically allocated memory, reading memory that has been freed, freeing memory more than once, and so-on. The easiest way to avoid free store bugs is to use the standard container classes in the C++ Standard Library: `std::vector<>`, `std::deque<>`, `std::list<>`, `std::set<>`, `std::multiset<>`, `std::map<>` and `std::multimap<>`.

Another way to avoid these sorts of errors in your C++ application is to only use `operator new`, `operator delete`, `operator new[]` and `operator delete[]`

in the constructor and destructor of a class. Use the lifetime of the instantiated class to control the lifetime of the dynamically allocated memory. Debugging the use of `new` and `delete` once saves you from having to worry about their use throughout the rest of your code.

Even with this technique, it is still possible to have bugs resulting from inappropriate use of the free store. You may find it useful to write your own memory allocator that replaces the global `new` and `delete` operators. Using such an allocator allows you to obtain statistics about your allocations and perform memory leak dumps, enter a debugger breakpoint on a specific allocation or free, etc. There are a variety of open source memory allocators with such debugging instrumentation available on the internet.

For the nastiest of free store problems, a commercial tool such as Bounds Checker or Purify may be required. These industrial strength tools add instrumentation code directly into your executable and perform consistency checks on the free store and other resources during the execution of your program. At the time of this writing, both tools have free trial versions available for download on the internet.

22.3.2 Accidental Assignments

In C and C++, the comparison operator `==` is almost identical to the assignment operator `=`. If the left-hand side of an equality comparison is a valid lvalue, it is easy for typing errors to create unintended assignments.

```
int x;
// ...
if (x = 5)
{
    // ...
}
```

The test clause of the `if` statement contains an unintentional assignment where a comparison was intended. The assignment expression is a valid value for the if clause and this code will compile without error. If the comparison were rewritten with the constant on the left-hand side, the typing mistake will now cause a compilation error:

```
int x;
// ...
if (5 = x)
{
    // ...
}
```

Writing constants (or `const` variables) on the left-hand side of an equality assignment will prevent this mistake from going undetected.

22.3.3 String Handling

Another common source of errors in applications relates to string handling. String handling is closely related to dynamic memory allocation, as strings are not an intrinsic data type in C or C++ and their use has traditionally been treated as an array of characters. The `std::string` and `std::wstring` types in the C++ Standard Library can free you from having to worry about the memory allocations required when manipulating strings. Occasionally, when interacting with Windows or a third party library, you may need to allocate an explicit array of characters. This is most readily done with `std::vector<>`.

22.3.4 Casts

C has only a single casting syntax for use with all manner of casting operations. C++ introduces several casting operators to the language to allow the programmer to specify the intent of the cast. Use the appropriate casting operator to indicate your intention to the compiler; this allows the compiler to identify when the intention of the cast doesn't make sense in context.

The `const_cast<T>(X)` operator should be used when the expression `X` needs the `const` qualifier added or removed. The `static_cast<T>(X)` operator should be used when you need to convert the expression `X` to type `T` and this conversion is allowed but not assumed. The `dynamic_cast<T>(X)` operator can only be used with run-time type information and virtual base classes. It is used to cast the base class pointer `X` to the derived class pointer `T`. The `reinterpret_cast<T>(X)` operator reinterprets the storage for the expression `X` as the type `T` with no other checks by the compiler. For more information on the casting operators in C++ and their appropriate uses, a good reference on C++ should be consulted.

For situations where casting might often be required, consider using a wrapper class that hides the casting inside the implementation of the class. Users of the class can forget about casting requirements and use the class in a type safe manner.

When writing code, consider your actions carefully when you find yourself writing casting expressions. They can lead to difficult bugs because they subvert the ability of the type system in C++ to protect you from mistakes.

22.3.5 C++ Exceptions

As discussed in chapter 1, exceptions provide a way of encapsulating state about an unexpected error and transferring control to an appropriate handler for the unexpected error. Usually the right place to handle an error is at a higher level in the program, while the unexpected condition occurred deep within a call stack. To deal with the problem, routines are usually peppered with error checking logic and shortcut returns and an error code indicating the failure of the called routine. Each caller must take responsibility for checking the returned error code and performing a shortcut return to its caller. The difficulty in ensuring that all such called routines have their error codes checked and a shortcut return

to the caller performed results in spotty error checking at best. Exceptions allow for a much cleaner approach to the problem of unexpected errors. While there is some overhead for using exceptions, this overhead should be minimal when exceptions and `try`, `catch` blocks are used properly.

The Windows APIs, COM objects, Direct3D and D3DX interfaces and functions use error codes to indicate success or failure, instead of exceptions. As discussed in chapter 1, a framework for mapping error codes to exceptions will allow your code to be robust in its error handling without all the tedium of checking error codes and managing the shortcut return code paths. Controlling the lifetime of resources with classes works hand in hand with using exceptions for error handling.

22.4 Debugging Windows Applications

Windows provides a variety of mechanisms for assisting in program debugging. A problem with traditional Windows applications is the profusion of `HANDLE` types that have distinct meanings and semantics, but the identical C++ type of pointer to `void`. However, if the preprocessor symbol `STRICT` is defined before the Windows header files are included, then each of the handle types will have their own fictitious type created and a misuse of those types will result in a type mismatch error from the compiler. For this reason, it is recommended that you always compile your code with `STRICT` defined before you include any Windows header files.

Windows 2000 provides more memory protection than Windows 9x. Windows XP is based on the same OS kernel as Windows 2000 and provides the same level of protection. To keep matters simple, we will discuss Windows 2000 with the understanding that all the same statements apply to Windows XP. Similarly, Windows ME is based on the same kernel as Windows 9x and all the statements in this chapter about Windows 9x apply to Windows ME.

The additional protection of Windows 2000 will catch your mistakes earlier with less likelihood of corrupting the operating environment so that you can continue working without rebooting. For this reason, it is recommended that you always compile and test your code on Windows 2000, even if Windows 9x is your target platform. For most Direct3D applications, the differences between the Windows API on the two platforms should hardly ever come up, but it always pays to make sure you check the availability information on Windows API functions in the MSDN documentation.

The FAT file system used in Windows 9x wasn't designed for the same level of robustness as NTFS. Consequently, its possible to lose code changes when using FAT file systems and you encounter an unexpected system hang. For this reason, it is recommended that you use the NTFS file system for development under Windows 2000. If this is not possible, you should at least disable write caching on FAT file system volumes containing your source code.

22.4.1 ANSI vs. Unicode

The native character environment of Windows 9x is ANSI, while the native character environment of Windows 2000 is Unicode. ANSI applications will work on Windows 2000, at the cost of an extra string conversion layer between the application and the OS. If your application doesn't do much string based interaction with the OS, this string conversion probably won't even be noticed. You can avoid the conversion layer by writing your code to use the `TCHAR` character type facilities in the Windows header file `<tchar.h>`. With this facility, you can compile both ANSI and UNICODE images from the same source base.

If you are targetting both ANSI and Unicode builds for your application, be sure to test both build environments every time you test. It is very easy to let an ANSI or Unicode string usage slip into one of the builds where it is not appropriate.

22.4.2 Security Environment

At first glance, it might not seem like Direct3D applications need concern themselves with security. However, Direct3D applications typically access other parts of the system and can run into security related issues during this interaction. The best way to avoid security related bugs is to develop on Windows 2000 as a limited access user. Windows 2000 should be used because Windows 9x does not provide any real security. A limited access user should be used because members of the Power Users group have more access than regular users to files, directories and the system registry.

By using a restricted environment, you will encounter any potential security problems during development. It is much better to find these problems during development then after you ship your application and start receiving support calls from users operating in a restricted environment.

If you have been developing as a member of the Administrator or Power Users group, it is recommended you remove yourself from those groups. Fast user switching in Windows XP will allow you to quickly become the Administrator if necessary. Under Windows 2000, the `RunAs` command can be used to execute a command as another user. This can also be used with shortcuts and Start menu items by holding the shift key down while invoking the context menu on the item and selecting the "Run as..." item.

22.4.3 Special Directories

A good Windows program should never assume the location of special directories on the target system. Not every system has a system drive on volume `C:`, and the names of special directories, such as `Program Files` may be localized to a different string on foreign-language Windows machines. A common source of Windows application bugs is assuming that these special folders have a fixed location or reside on a particular drive. Attempting to store data in the wrong

CSIDL	Contents and Typical Value
CSIDL_APPDATA	Per-user application specific data. C:\Documents and Settings\ *username*\Application Data
CSIDL_COMMON_APPDATA	Application specific data for all users. C:\Documents and Settings\ All Users\Application Data
CSIDL_COMMON_DOCUMENTS	Documents for all users. C:\Documents and Settings\ All Users\Documents
CSIDL_LOCAL_APPDATA	Data files for non-roaming applications. C:\Documents and Settings\ *username*\Local Settings\ Application Data
CSIDL_PERSONAL	Documents for individual users. C:\Documents and Settings\ *username*\My Documents
CSIDL_PROGRAM_FILES	Location for programs. C:\Program Files
CSIDL_PROGRAM_FILES_COMMON	Shared application components. C:\Program Files\Common

Table 22.1: Common CSIDL values.

location can result in failure of your application to run in a secured environment, as well. Restricted users will have very limited write access to the system.

Windows provides a function for obtaining the full path of these special directories. A related issue is the installation directory of the application. Your install program can write the installation location to the system registry for use by the program. Windows Installer also provides an API to obtain installation directories for installed components.

The ::SHGetFolderPath function retrieves the path associated with a special folder. The special folder is identified by its CSIDL value. Common CSIDL values and their uses are given in table 22.1. The CSIDL value should be logically ored with CSIDL_FLAG_CREATE to ensure that the appropriate directories are created if they don't yet exist.

If the user is not meant to load, view or alter the data stored in a file, it should be stored in CSIDL_COMMON_APPDATA if the data is per-machine or CSIDL_-LOCAL_APPDATA if it is per-user. Per-machine data would include items such as system configuration information, while per-user data would include items such as keyboard bindings.

If the data is intended to be manipulated by the user, then a unique file extension should be chosen for the data file and the application should be associated with the file extension during installation of the application. If the data is intended for any user, such as a shared high scores file, then it should be stored

in CSIDL_COMMON_DOCUMENTS. If the data is intended only for the current user, such as a personal high scores list, then it should be stored in CSIDL_PERSONAL.

If an application saves multiple files, it is recommended that they be stored in a subdirectory of the special folder location returned by Windows. The subdirectory should have a name that the user can easily associate with the application. Users may also wish to store their user data in another location; a good application will use the special folder locations as the default location for individual user files and let the user select a different location if desired.

22.4.4 Output Debug Stream

The output debug stream is a facility provided by Windows to allow messages to be sent directly to a debugger. If a process is being executed by a debugger, the debugger will be sent an event whenever a string is written to the output debug stream. If the debug output stream is written by a process not under the control of a debugger and a system debugger is running, Windows sends the string to the system debugger. If no debugger is running, then the string is discarded.

You can write a string to the debug output stream with the ::OutputDebug-String function. It takes a single string argument that is written to the debug output stream. No newlines or carraige returns are appended to the string.

```
void OutputDebugString(LPCTSTR string);
```

When your application is run from Visual Studio, strings sent to the output debug stream appear in the Debug tab of the Output window. If your program is run outside of Visual Studio, then the messages sent to the output debug stream will be discarded unless you are running a program designed to intercept these strings. The SDK includes such a tool called dbmon.exe, a console program that captures the output debug stream and writes the strings to the standard output. You can use this to capture the debug output of your program to a file for later examination.

Under Windows 2000, if you are not a member of the administrator group, then dbmon will not receive the debug output stream. However, you can use the RunAs command from the console to run dbmon as the administrator. More sophisticated tools for viewing the debug output stream are freely available on the internet.[1]

When using the debug output stream with C++, it would be handy if we could use it just like the standard streams in the C++ Standard Library. The sample code includes some code that implements a streambuf class for this purpose. The implementation is provided as a header file ods.h for use by any source file that writes to the output debug stream and a source file ods.cpp that provides the implementation of the stream buffer.

To use this class, add the source file to your project and include the header file in any source file that wants to write to the debug output stream. The global

[1] Such as "DebugView" from http://www.sysinternals.com.

variable g_ods represents the debug output stream as a C++ stream and the normal operator<< method of writing output to the stream works as expected. ::OutputDebugString will be called when the stream's internal buffer overflows, when std::endl is written to the stream, or when the flush method is called explicitly on g_ods.

For instance, the following code writes out the contents of the adjacency array of an ID3DXMesh to the debug output stream:

```
std::vector<DWORD> adj(mesh->GetNumFaces()*3);
THR(mesh->GenerateAdjacency(0, &adj[0]));
for (UINT i = 0; i < mesh->GetNumFaces(); i++)
{
    g_ods << i << _T(": ") << adj[3*i + 0]
        << _T(", ") << adj[3*i + 1]
        << _T(", ") << adj[3*i + 2] << std::endl;
}
```

Using the debug output stream in this manner is fine if you're writing standard types with predefined operator<< insert operators. If you're writing Direct3D types to the debug output stream (or any stream), you'll want insert operators that write text representations of those types to the stream. The sample code includes the source files dump.h and dump.cpp containing insert operators for many common Direct3D and D3DX types.

If a very large amount of output is sent to the debug output stream in a short period of time, the debugger may lose part of the output. It can also interfere with execution of the program to have the debugger handling all that text output. The best way to avoid these problems is to limit the output to the debug stream to only the information relevant to the problem you're debugging.

22.4.5 Structured Exception Handling

Structured exception handling (SEH) is a mechanism similar to C++ exceptions provided by the Windows platform. If your program should accidentally dereference a NULL pointer, an access violation results. The access violation will trigger a SEH style exception from Windows. If your program does not handle this exception, Windows will terminate your program.

It is possible to catch an SEH style exception with a catch (...) block after a try block. Since no information about the exception is passed to this style of catch handler, there isn't much you can do about the exception except ensure that the program shuts down gracefully.

However, it is possible to turn these SEH style exceptions into C++ exception classes, including information about the context of where the SEH exception occurred. The Windows include file <eh.h> defines the function _set_se_translator which can be used to map SEH style exceptions into C++ exception classes.

```
typedef
```

```
void (_cdecl *_se_translator_function)(unsigned int code,
        EXCEPTION_POINTERS *pointers);

_se_translator_function
_set_se_translator(_se_translator_function translator);
```

The function `_set_se_translator` is passed a pointer to your translation routine and returns the previously set translation routine. The translation routine will be called when an SEH style exception occurs. The translator will be passed the exception `code` and the `pointers` to the context of the exception. The context pointers contain information about the stack frame and register contents at the time the SEH style exception was thrown. The translator takes the context information and uses it to construct a C++ exception object and then throws a C++ exception. An enclosing catch handler can then extract the context information for use in a stack dump, error report, etc. The following small program illustrates the conversion method:

```
#include <eh.h>

struct s_structured_exception
{
    s_structured_exception(DWORD code,
                           const EXCEPTION_POINTERS *excepts)
        : m_code(code),
        m_excepts(excepts)
    {}
    ~s_structured_exception()
    {}

    DWORD m_code;
    const EXCEPTION_POINTERS *m_excepts;
};

void _cdecl
se_translator(unsigned code, EXCEPTION_POINTERS *excepts)
{
    throw s_structured_exception(code, excepts);
}

int APIENTRY WinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR     lpCmdLine,
                     int       nCmdShow)
{
    try
    {
```

```
        _set_se_translator(&se_translator);
        // force an access violation
        DWORD foo = *reinterpret_cast<DWORD *>(0);
    }
    catch (s_structured_exception &e)
    {
        dump_stack(e.m_code, e.m_excepts);
    }

    return 0;
}
```

22.4.6 Stack Traces

If you're running your application in the debugger you can see the call stack of an unexpected error. If your application is being tested by another user without a debugger or if a bug crept into the customer release, you won't have the luxury of examining the call stack. However, you can obtain a call stack from Windows and record it into a log file or a bug report.

The basic procedure for constructing a stack trace is to use the `::StackWalk` function. The first call to this function is made with the values of the instruction pointer and stack frame pointer at the time the exception was thrown. Each successive call to the function traverses the stack one level deeper. When the stack has been completely traversed, the function returns `FALSE`.

If you're constructing a stack trace from an `EXCEPTION_POINTERS` structure, the `ContextRecord` gives the execution context where the exception occurred as a pointer to a `CONTEXT` structure. The `CONTEXT` structure is CPU architecture dependent and the following discussion assumes an x86 instruction set architecture. The `Eip` member of this structure gives the address of the instruction that caused the exception. The `Ebp` member gives the stack frame address when the exception occurred.

Walking the stack frames just gives you the addresses within your executable. You can map these address to symbolic names using the Debug Help Library in the Platform SDK. Using this library, you can map the addresses in your stack frame to symbols associated with your code, or with Windows. A full explanation of the Debug Help Library can be found in the MSDN documentation.

22.4.7 Mini Dumps

Even better than writing a stack trace to a text file is to collect a "minidump". The minidump is a subset of the information written in a complete crash dump of Windows. The minidump is small enough that it can be sent to technical support for further diagnosis of the problem. The minidump can be read directly by the Visual C++ 7 debugger as well as by debugging tools in the Platform SDK. You can also write your own tools to examine the data in a minidump file.

A mini dump is created with the `::MiniDumpWriteDump` function. The caller supplies information about the context of the dump, obtained from an exception context, and any additional data streams that should be recorded to the dump file. These additional data streams should include any critical context information specific to your application. They will be written to the dump file and can be examined later for further diagnosis. For more information on minidumps and `::MiniDumpWriteDump`, consult the MSDN documentation.

22.4.8 Debugging Buffer Overruns and Underruns

Knowledge base article `Q264471` in the MSDN Library describes a utility called `PageHeap` that can be used to debug buffer overruns and underruns. A buffer overrun error occurs when the application references memory past the end of an allocated buffer. This type of bug can be particularly difficult to identify as the problem won't always be fatal to the execution of the program, particularly with read overruns. A buffer underrun is similar, except that the program references memory before the start of the allocated buffer.

The PageHeap utility writes to an access controlled portion of the registry, so you will need Administrator rights in order to run the program succesfully. PageHeap works by allocating memory buffers at the end of a page of virtual memory. If the application attempts to write past the end of the buffer, it will either corrupt guard bytes (used to align the allocated memory on an 8 byte boundary), or will cause an immediate access violation if it writes past the end of the page. The knowledge base article describes the use of PageHeap in more detail.

22.4.9 Fast User Switching in XP

Windows XP provides a facility called "fast user switching", which allows multiple users to be logged onto the machine at the same time. This can result in a scenario where your DirectX application thinks it is still running, but the results of its rendering will not be visible. An application can detect fast user switching and change its behavior when the user that launched the application is no longer the user in control of the interactive logon session.

If your application needs to know when the user is switched, you can register to receive the `WM_WTSESSION_CHANGE` events with the `::WTSRegisterSessionNotification` function. Any window that has been registered to receive these events should be unregistered with `::WTSUnRegisterSessionNotification` before it is destroyed.

If you expect your application to be installed on Windows XP, it is recommended that you test your application under a fast user switching scenario to ensure that it works properly. If you need to ensure only a single instance of your application is running in this environment, you can use a global named mutex to see if your application is running in any session. Then, if the named mutex was found, you can determine if your application is running in the current session using `::FindWindow` and switch to the running instance.

For more information on fast user switching, see the Terminal Services documentation in MSDN.

22.5 Debugging With Visual C++ 6

A full discussion of the debugger in Visual C++ is beyond the scope of this section, but there are a few things worth mentioning when developing Direct3D applications. With a few simple customizations to Visual C++ you can increase the effectiveness of the debugger for Direct3D applications.

Some of the customizations involve the files `AutoExp.dat` and `UserType.dat`. These files are located in the `Common\MSDev98\bin` subdirectory of your Visual C++ 6 installation. The installation program in the sample code can be used to integrate the supplied files with Visual C++ 6. The customizations may be incorporated manually for other versions of Visual C++ that are compatible with the data files for Visual C++ 6.

22.5.1 Decoding Direct3D `HRESULT`s

Visual C++ has some tools for helping decode the `HRESULT`s that COM applications are likely to encounter. From the Tools menu in Visual C++, you can select the Error Lookup tool. This tool will decode most common `HRESULT`s into an error string. Unfortunately, it does not understand Direct3D `HRESULT` errors. However, the DirectX Error Lookup Tool provided in the SDK does understand Direct3D `HRESULT` error codes. You can include an item for this program on the Tools menu with the "Customize..." item. In the resulting dialog, select the Tools tab and enter a new item on the list. The executable for the DirectX error lookup tool is located in `Samples\Multimedia\Misc\bin` subdirectory of the SDK installation directory.

22.5.2 Highlighting Direct3D Identifiers

With the `UserType.dat` file, you can categorize specific identifiers as user-defined keywords. The text editor in Visual Studio can assign a different color to user-defined keywords, allowing you to highlight all Direct3D and D3DX identifiers. The sample code contains an `UserType.dat` file that includes all the Direct3D and D3DX defined identifiers.

22.5.3 Expanding Direct3D Structures

The file `AutoExp.dat` contains directives that customize the display of structure types in the debugger watch windows. Each line in the `AutoExpand` section gives a type name, an equal sign, and a format expression for the type's value in the watch window. Structure members can be formatted in the expression by surrounding them with angle brackets (`<>`) and providing an optional format specifier. For example, the following line will cause the vector $\langle 1, 2.2, 3.33 \rangle$ to display as `1, 2.33, 3.33` in the debugger's watch window.

Format	Meaning
c	Single character
d,i	Signed decimal integer
e	Signed scientific notation
f	Signed floating-point
g	General floating-point
hr	HRESULT or Win32 error code
l,s	Long or short integer prefix
m	Memory dump
o	Unsigned octal integer
s	String
st	Unicode or ANSI string
su	Unicode string
u	Unsigned decimal integer
wc	Window class flag
wm	Window message number
x,X	Unsigned hexadecimal integer

Table 22.2: Format specifiers for debugger structure expansion formatting.

```
[AutoExpand]
_D3DVECTOR=<x,g>, <y,g>, <z,g>
```

The allowed format specifiers are summarized in table 22.2 and described in detail in the Visual Studio 6 help under the title "Symbols for Watch Variables". The sample code contains an `AutoExp.dat` file that includes expansions for all the Direct3D and D3DX structure types.

22.5.4 Limiting Step-Into

The `AutoExp.dat` file can also include an `ExecutionControl` section that lists debugger directives for function entry points. Using this section you can prevent the debugger from stepping into functions that are not relevant. For instance, you can prevent the debugger from stepping into all ATL classes and functions. The use of smart pointer classes such as `CComPtr<>` or STL containers then becomes invisible to the debugger.

```
[ExecutionControl]
myfunctionname=NoStepInto
CFoo::*=NoStepInto

; To ignore construction and assignment of MFC CStrings:
; (Notice the extra = in CString::operator=.)
[ExecutionControl]
CString::CString=NoStepInto
CString::operator==NoStepInto
```

```
; To ignore all ATL calls:
[ExecutionControl]
ATL::*=NoStepInto
```

The function name is specified as a fully qualified identifier, with optional wildcards allowing for any identifier within a namespace. The sample code contains an `AutoExp.dat` file that eliminates stepping into ATL classes and functions as well as all `rt` classes and functions.

22.5.5 Remote Debugging

Visual C++ 6 supports the concept of remote debugging. In this scenario, the debugger runs on one machine and the debug target runs on a separate machine. A small monitoring process also runs on the target machine. The monitoring process attaches to the debug target as a debugger and talks to the debugger IDE running on the other machine using TCP/IP. Windows 2000 allows you to connect two machines via TCP/IP over a cable via their serial or parallel ports.

For more information on remote debugging, see the Visual C++ 6 documentation under "Using Visual C++, Visual C++ Programmer's Guide, Debugging, Debugging Specific Types of Applications, Debugging Remote Applications". MSDN knowledge base article `Q241848` may also be helpful in configuring remote debugging.

Remote debugging is the most flexible way to debug exclusive mode problems. The second computer provides a completely independent set of input devices and display. Using the mouse and keyboard with the debugger won't interfere with your application's use of the mouse or keyboard and the display can be kept in exclusive mode while you debug.

22.5.6 Multiple Monitor Debugging

While remote debugging is the most flexible solution for debugging difficult exclusive mode problems, it does require a second computer. If you can install multiple display adapter cards in your computer, or your computer has a display adapter supporting multiple monitors, you can debug with your Direct3D application running on one monitor while the debugger runs on a different monitor. Of course, you will need to share the input devices between the application and the debugger.

This sounds great in theory; in practice you may find that multiple monitor support is lacking in one or both of the display adapters or that you have difficulty interacting with the debugger while your program has an adapter in exclusive mode. Check with the IHV supplying the drivers for your adapter to determine multiple monitor support under the appropriate version of Windows before installing additional adapters into your system.

22.6 Debugging With Direct3D

C++ and Windows can help you avoid nasty problems lurking within your code, but they are both rather ignorant of Direct3D and its particular pitfalls. There are several important steps that every Direct3D developer should take to diagnose problems:

1. Install the debug runtime.

2. Check all `HRESULT`s for success.

3. Compare results between HAL and the reference rasterizer.

4. Ensure support for all features used in the device capabilities.

In addition to these basic steps, it is recommended that you code into your application the ability to switch to software vertex processing and the reference rasterizer at startup or during execution. Switching to software vertex processing lets you obtain an alternative execution path from the vertex processing in the HAL device. This can be a good way to double check problems related to vertex processing. Similarly, the reference device provides an alternate implementation of the entire pipeline allowing you to double check problems related to rasterization and texture processing. If you are having a problem with a particular piece of Direct3D functionality that is demonstrated in an SDK sample program or in the sample framework, compare your code to that in the SDK.

22.6.1 Debug and Retail Runtimes

Fortunately, the SDK includes two versions of the Direct3D runtime: debug and retail. The debug runtime includes additional input parameter validation and the reference rasterizer. The debug runtime will catch most invalid arguments and cause the resulting API call to fail with `D3DERR_INVALIDCALL`. An explanation describing the cause of the failure will be written to the debug output stream. The debug version of D3DX will also perform additional parameter validation and send explanations of failures to the debug output stream.

Because the retail runtime performs no validation or checking of its parameters, its possible to send bad data to the reatil runtime without ever realizing your mistake. This sort of mistake may succeed by accident during development, only to fail on the customer's machine once the application has been shipped. Developing without the debug runtime is like running a marathon through a minefield while blindfolded with boat anchors tied to your ankles. While its possible to survive such an ordeal, life is so much simpler without the self-imposed handicaps. It is recommended that you always install the debug runtime for development. It is also recommended that testing be performed with the debug runtime, if possible. If necessary, you can test with the retail runtime, but you're less likely to catch problems during testing.

Of course, that extra error checking and validation comes at the cost of some runtime performance. Therefore, if you are performing performance measurement, you should switch back to the retail runtimes before performing those measurements.

22.6.2 Checking HRESULTs

Most methods and functions in the Direct3D interfaces, the D3DX interfaces and functions and the X file interfaces return an `HRESULT` error code. Each and every one of these return codes should be checked for success. Most of these methods and functions are expected to succeed, so its rare that you must insert logic into your main flow of control for handling failures. However, just because the call is not expected to fail doesn't mean it can't fail. Particularly with the debug runtime, checking all `HRESULT`s for unexpected failure is going to save you valuable development time on those "stupid mistakes". The `THR` macro mechanism discussed in chapter 1 is particularly helpful here as you get information about exactly where the failure occurred in your code.

22.6.3 Reference Rasterizer

The reference rasterizer, `D3DDEVTYPE_REF` described in section 2.6 on page 52, is very valuable in determining the source of errors. It is recommended that you always provide a way to run your program on the reference rasterizer, or be able to switch to the reference rasterizer at some point during execution. Switching device types implies a complete destruction of the existing device and creation a new device using the new device type. All the samples in the SDK and those accompanying book support switching to the reference rasterizer.

As the reference rasterizer uses a software implementation of most features, its interaction with the device driver is rather minimal and simplified. When an unexpected rendering result is obtained on both the HAL and reference device types, the fault is usually in the application code. If different rendering results are obtained between the two device types, the likelihood of a driver bug is increased, but the application cannot be completely ruled out. Since the reference device supports all features and usually has a richer set of device capabilities than a HAL device, the differences in rendering output may still be the fault of the application if the device capabailities weren't properly examined.

If the SDK isn't installed on a machine, you won't have access to the reference device. In this case a "null" reference device is supplied to the application if `D3DDEVTYPE_REF` is requested and no rendering will be performed, as described in section 2.6.

22.6.4 DirectX Error Routines

If you encounter failed `HRESULT`s from Direct3D, D3DX or the X file interfaces, you can use the error routines described in section 15.17 to obtain a descriptive string for the `HRESULT`. The DirectX Error Lookup Tool, described in section 22.5

can also be used to obtain a descriptive string from a failed `HRESULT` without changing your code.

22.6.5 Reproducing Problems

In a highly interactive graphics program, such as a CAD application or a game, it can be difficult to reproduce some problems because they may only occur in response to a particular set of user input. The best way to reproduce these kinds of problems is to record a script of the user input while the application is running. Then your application can play back the input script to reproduce the problem again and again until you find the source of the error.

Using a playback script created from user input can also be very helpful if you need to test your program against the reference device. Since the reference device runs much more slowly than a hardware device, the additional time spent rendering may confuse your application during playback. You can guard against this by stopping any simulation timekeeping during the rendering process and resuming the timekeeping once the rendering portion of your main loop is finished.

22.6.6 Exclusive Mode Problems

The easiest way to debug an application is by running it in windowed mode. However, sometimes you encounter problems that only occur in exclusive mode. Since the application has exclusive control of the display, using a debugger can be difficult. The most robust way to debug such a situation is to use remote debugging, described in section 22.5.5. If a second computer is unavailable, you can try using multiple monitors on a single computer as described in section 22.5.6.

If neither additional computers nor additional monitors are available, then the only methods left are those that involve logging internal state of the application to a file for later inspection. The easiest way to do this is to write logging messages to the debug output stream and capture that to a file. The `dbmon` console utility included with the SDK can have its output redirected to a file from the command line. You can start `dbmon` and then start your program to generate the debug output.

22.6.7 Driver Problems

The display driver runs in the "trusted computing base" under Windows 2000, which means that if you manage to confuse the display driver you can crash the whole system. If you are encountering strange hangs or system crashes with your Direct3D program, but otherwise your system is stable, you should examine carefully the rendering data that you are sending to Direct3D. In particular, bad indices can cause drivers to access random portions of memory. Running your application with the reference rasterizer and performing consistency checks on

your data before you write it into index buffers should catch these errors before they become fatal.

On systems with AGP memory, there may be problems with the AGP driver that affect availability of AGP memory or its performance. Check the motherboard manufacturer's web site to ensure that you have the proper AGP driver support for your system. In addition, there were some issues with AGP support in versions of Windows 2000 prior to Service Pack 2. The issue did not result in a crash, but an unexpected low level of performance. If you are experiencing atypical performance with your card and are running Windows 2000 prior to Service Pack 2, you should consider applying the latest service pack release.

In a perfect world, we could assume that the device driver was perfect and free of errors. Unfortunately, in the real world errors in the drivers are encountered. However, the developer is still left with the question of whether or not the error encountered is a result of their application or the driver. If the error can be reproduced on the reference device, then the error is most likely in the application. If the error does not occur on the reference device type, but only on the HAL device type, then it is more likely that the error is in the driver.

If you encounter what you believe is an error in the driver, you should first check to see if an updated version of the driver is available. The hardware vendor may have already fixed the problem you encountere. If the problem persists in the latest driver, you should file a bug report to the vendor of the driver. Independent hardware vendor contact information is given in section 22.9.2. See section 22.10 for the best way to file a bug report.

If you've identified a genuine problem with a particular driver, you may not have the luxury of being able to wait for a fix from the hardware vendor. In this situation, you can use the information returned by `GetAdapterIdentifier` to identify this particular combination of hardware and driver to trigger a workaround in the application. Ideally, the workaround involves the use of a different algorithm or a slightly different set of device state. Unfortunately thre are some operations for which no viable workaround is available. For that situation you will have to consider whether or not you can afford to wait for a fix from the hardware vendor.

22.6.8 Shader Problems

Debugging vertex and pixel shaders can be a difficult task, particularly for a complex shader. NVidia has created a shader debugger that will allow you to single step through a vertex shader and examine the contents of the registers. This debugger is available as a free download on the the developer relations portion of their web site, see section 22.9.2.

Without the shader debugger, you have to resort to divide and conquer techniques for identifying the problems within a shader. In this approach, the portion of the shader containing the problem is located by commenting out portions of the shader and executing it. You will still need a valid result from the shader to test the rendering, so you will probably want to comment out an entire stage of processing as opposed to individual instructions. For instance,

suppose you have a vertex shader that performs skinning followed by fog followed by projection. If the results are totally unexpected, you might try replacing the skinning section with instructions that simply copy the input position. If the results are still not what was expected, the problem may not lie within the skinning section, but elsewhere.

An alternative approach is to start with simple shaders and keep them working at all times. Incrementally adding functionality to a working shader means that if the rendering is incorrect, there is no doubt that the fault lies in the newly added code or as a result of some coupling to the newly added code.

In addition to these techniques, it can be helpful to execute your shaders against the reference rasterizer. The reference rasterizer will do more checking on shader instructions than the hardware. A hardware implementation may allow uninitialized registers or other programming mistakes to go by unnoticed.

22.6.9 Texturing Problems

With the large amount of state that controls how textures are sampled, filtered and rendered, they can be a source of subtle problems. You can use a divide and conquer technique here to identify the source of the rendering error. You can switch to point sampling and disable mipmapping if you suspect that the texture filtering isn't working as you expect. You can load diagnostic patterns into the texture itself to obtain a rendering with your diagnostic texture. For instance, loading checkerboard patterns into a texture, loading different patterns into different mipmap levels, or tinting mipmap levels to identify how the filtering of texels is being performed.

The file `tint.cpp` in the sample code contains a function to tint the mipmap levels of an existing texture. The `rt_Texture` sample uses this function to implement its tinting option for visualizing how mipmap levels are used.

22.6.10 Processor Specific Graphics Processing

Processor specific graphics processing, or **PSGP**, refers to the portion of the software vertex processing pipeline supplied by a CPU vendor for use with Direct3D. PSGP is the portion of Direct3D that uses instruction set extensions[2] from the various CPU vendors to speed up software vertex processing. Sometimes you may encounter a rendering problem that only occurs on specific CPUs with specific software vertex processing operations. You can disable PSGP to validate this hypothesis. If the operation is performed correctly when PSGP is disabled, then the odds are high that the problem is a result of a bug in the PSGP processing for that particular processor.

The registry keys under `HKLM\Software\Microsoft\Direct3D` are used to control PSGP and other CPU instruction set extensions used by Direct3D and D3DX. For each key, a value of one disabled the corresponding CPU specific processing, while a value of zero (or a missing registry key) enables the corresponding CPU specific processing.

[2]Such as Intel's SSE or AMD's 3DNow! instruction set extensions for SIMD processing.

Key	Description
`DisablePSGP`	Disables PSGP in the core runtime
`DisableD3DXPSGP`	Disables PSGP in D3DX
`DisableMMX`	Disables MMX processing

Table 22.3: Registry keys to disable CPU specific software processing.

22.6.11 Interfering Applications

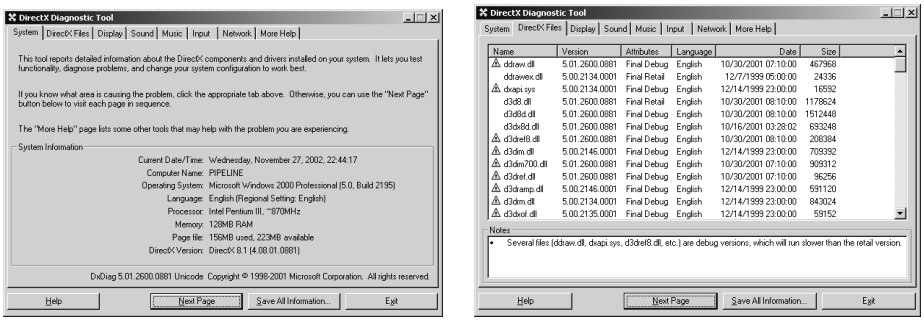
You may experience "hiccups" in the smooth rendering of scenes in your application if other applications are interfering with your Direct3D application. Even a simple resource monitoring tool such as `perfmon` can cause stuttering in the smooth animation of images. The stutter occurs as these other applications wake up and request attention from the CPU. During this time, your application is not running and is not updating its display. A common source of this problem is the `FINDFAST` utility that comes with Microsoft Word (and Office). This utility is intended to operate in the background as an indexing agent in order to build an index of documents on your system so that relevant text within those documents can be located quickly. You can disable this application with a setting in Word's options.

The NetMeeting program from Microsoft is a particular source of problems for DirectX applications. The default configuration of NetMeeting disable Direct3D applications while it is running. To change the default behavior, go to "Tools, Remote Desktop Sharing Settings" and uncheck the appropriate checkbox.

On Windows 98 platforms, the Direct3D runtime itself uses a helper application that can interfere with the proper execution of DirectX applications when things go wrong (as is often the case during development). On Windows 98, the `ddhelp` program may be left in an orphaned state when your Direct3D application crashes and exits. You must kill the `ddhelp` process in order to run your program again.

22.7 DirectX Diagnostic Tool

The DirectX Diagnostic Tool is included with the Direct3D runtime and is always installed. A screen shot of the dialog displayed by the diagnostic tool is shown in figure 22.1. This tool will report the exact versions of all files installed with DirectX as well as additional information like the OS version and service pack level and information about the hardware configuration of the machine. It can be started by selecting "Start, Run..., `dxdiag`". The "Save All Information..." button can be used to save all the information to a file. This can be very helpful in identifying problematic configurations during testing or customer support. The DirectX Files page of the dialog will show information about every DirectX DLL file and will flag debug versions of these files as potential performance problems.



(a) (b)

Figure 22.1: DirectX Diagnostic Tool. (a) The System page. (b) The DirectX Files page.

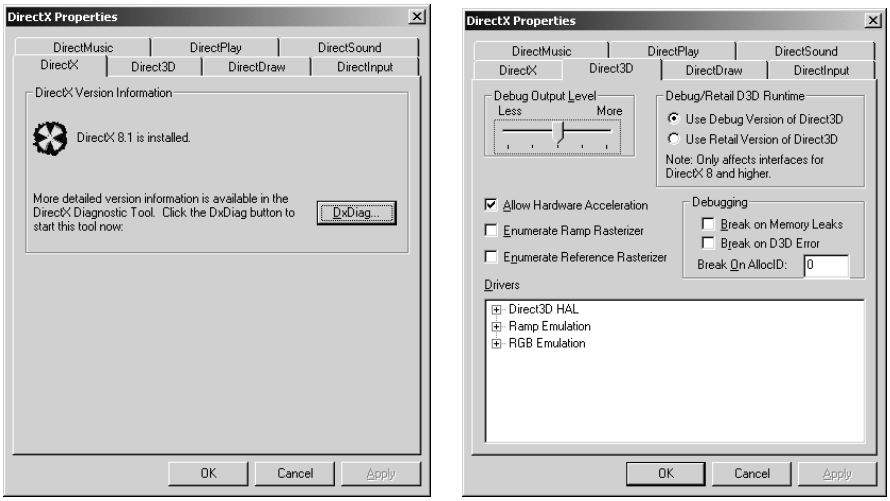
If you should ever need to report a driver problem to a hardware vendor, or a problem with the runtime to Microsoft, a copy of the diagnostic output should always be included to help them reproduce the problem.

22.8 DirectX Control Panel Applet

The DirectX SDK installs a control panel applet that provides you with additional debugging tools. This control panel applet is not present in the end-user installations of the runtime. The main and Direct3D pages of the control panel applet dialog are shown in figure 22.2. The main page displays the version of DirectX installed allows you to invoke the DirectX Diagnostic Tool discussed in the previous section.

There are three groups of controls on the Direct3D page that will help you with your debugging. The "Debug/Retail D3D Runtime" group allows you to choose between the retail and debug runtimes if you installed the debug runtime when you installed the SDK. If you installed the retail runtime with the SDK, then you will not be able to select the debug runtime from the control panel applet. For Windows XP, you should examine the `readme.txt` file in the `SDKDev` directory of the SDK for additional instructions about installing the debug runtime. The control panel affects only "hot switchable" runtime components and will only affect the DirectX graphics components for DirectX 8. The exact files affected will be shown by the diagnostic tool. If you need to restore the retail runtime for components that aren't hot switchable or components from previous versions of DirectX, then you can only change these by uninstalling the SDK to remove the debug version of the components and reinstalling the SDK with the retail runtime selected.

The "Debug Output Level" group allows you to change the amount of messages send to the debug output stream. Moving the slider towards "More" results in additional messages from the debug runtime about the behavior of your program. You can move the slider towards "Less" if the debug output



(a) (b)

Figure 22.2: DirectX Control Panel Applet. (a) The DirectX page. (b) The Direct3D page.

is producing so many messages that the debugger is spending too much time processing the messages. The value of the slider is used by a process once it starts; changes to the slider while the program is running won't take effect until the program is restarted.

The "Debugging" group instructs the runtime to transfer control to the debugger under certain conditions. The Direct3D runtime will check for leaked memory when the DLL is detached from the process, usually at the time the process exits. The runtime only checks for memory that it has allocated that has not yet been released; it does not check for leaks in other memory that may have been allocated by the process. The "Break on Memory Leaks" check box causes the runtime to transfer control to the debugger when the process exits and memory allocated by the runtime has not been properly freed. A message in the debug output will give the allocation ID of the leaked memory.

The allocation ID is a sequential number that is incremented each time the Direct3D runtime allocates memory on behalf of the application. The first allocation performed by the runtime will have an allocation ID of one. You can enter an allocation ID into the "Break On AllocId" edit box to have the runtime transfer control to the debugger when the memory corresponding to the ID is allocated. To debug leaked memory reported by the runtime, you can first check the "Break on Memory Leaks" box and run your program until the breakpoint is encountered at process exit. Then, end debugging of the current process, uncheck the "Break on Memory Leaks" box, enter the allocation ID reported in the debug output into the "Break on AllocId" edit box and run the application again with the same input. Now a breakpoint will be triggered when

the corresponding leaked memory is allocated. You can examine the call stack to find the place in the application where the error occurred.

The "Break on D3D Error" check box causes the runtime to trigger a breakpoint whenever a failed `HRESULT` is returned by the runtime. Note that not all failed `HRESULT`s indicate a bug in your code. For instance, a call to `CheckDeviceFormat` may return a failed `HRESULT` to indicate no support for the supplied arguments. The "Break on D3D Error" checkbox will cause a breakpoint to be triggered in this instance. The checkbox affects only the Direct3D runtime, but if a function or method in D3DX or another library calls a runtime method that fails, a breakpoint will be triggered.

The "Allow Hardware Acceleration" check box will disable HAL support if it is unchecked, leaving only the reference device type available to an application. Both HAL and reference device types are available if it is checked.

The remaining controls on the Direct3D page are for legacy DirectDraw and Direct3D support and have no effect on applications using the 8.1 interfaces.

22.9 Getting Help

Sometimes all we need to solve our problem is someone willing to listen to us explaining them. Many times, we discover the reason for our difficulties once we attempt to explain them to someone else. For more difficult problems, its helpful to talk them over with another programmer experienced with Direct3D. The internet has enabled large communities of physically disparate programmers to join forces and help each other in solving their problems.

Discussion forums generally come in one of three forms on the internet: mailing lists, usenet newsgroups and web forums. A search engine, such as `http://www.google.com`, can be used to find relevant web sites and web-based discussion forums. A mailing list works just like electronic mail between two individuals, except that mail sent to the list is broadcast to all subscribers on the list. Usenet newsgroups are similar to mailing lists, except that messages are delivered by a store and forward technique among a network of servers. Usenet client software connects to the news server to read and post messages in the newsgroups. A sampling of relevant usenet newsgroups and mailing lists is given in table 22.4.

Microsoft runs the `LISTSERV` mailing list software on the host `discuss.microsoft.com`. This machine hosts a variety of developer mailing lists. These are not official support channels and the information is provided purely by the grace of the participants. Microsoft staff may participate, but they are under no obligation to answer your questions. That being said, they are generally a friendly bunch and there are many other subscribers on the mailing lists that are willing to help out their fellow programmer.

To subscribe to a mailing list through `LISTSERV`, send a mail message to address listed in the table with the phrase "`subscribe` *list name*" in the body of the message. The `LISTSERV` software will respond with a welcome message describing the charter of the mailing list and the behavior expected of all list

General C++ Programming
`comp.lang.c++.moderated`
`comp.lang.c++`

General Graphics Programming
`comp.graphics.algorithms`
`comp.graphics.visualization`

General Windows Programming
`microsoft.public.win32.programmer.*` hierarchy

DirectX Programming
`microsoft.public.win32.programmer.directx.*` hierarchy
`microsoft.public.win32.programmer.directx.graphics`
`microsoft.public.win32.programmer.directx.graphics.shaders`
`microsoft.public.win32.programmer.directx.video`
`microsoft.public.vb.directx`
DirectXDev at `listserv@discussms.hosting.lsoft.com`
DirectXAV at `listserv@discussms.hosting.lsoft.com`

Table 22.4: Community newsgroup and mailing list resources.

members. For instance, sending HTML formatted email or attachments to a mailing list is typically not allowed.

Usenet newsgroups are organized in a hierarchy by subject. Hierarchies are named by using a shell style wildcard, such as the `comp.lang.*` hierarchy for the discussion of programming in a variety of languages. The `comp.*` hierarchy is for the discussion of computing in general and contains thousands of newsgroups. This hierarchy is typically carried by the news servers of an internet service provider. Check with your provider to find out about the availability of newsgroups in the `comp.` hierarchy.

Microsoft operates a usenet newsgroup server on the host `news.microsoft.com`. This server provides the `microsoft.*` hierarchy of newsgroups. Microsoft's server exchanges messages with other usenet servers, so you may be able to access these newsgroups through the server operated by your ISP. However, you can depend on the `microsoft.*` newsgroups appearing on `news.microsoft.com`.

22.9.1 Asking Smart Questions

One very important thing to remember when asking for help from the community is that the people who can help you aren't paid to help you. They help others because they are the sort of person that likes helping others. When you ask a question of the community, you want to ask a smart question that shows you've tried to figure out your problem on your own, but you just need a little help.

First, you should try not to antagonize people who can help you. This means sticking to accepted norms of posting behavior, such as using plain text encoding on messages posted to mailing lists and usenet and avoiding HTML encodings of your message. You should pick one forum that you think is most appropriate for your question and ask it there. If you post a copies of your question to multiple forums, you give people the impression that you didn't take the time to find an appropriate forum. This is considered rude and antagonizes people who might help you.

Having found the appropriate forum, make sure that you include all the relevant information in your question. Posting a long code excerpt with the question "Why doesn't my code work?" just makes you appear lazy expecting other people to debug your code for you. Posts that make the author appear lazy or demanding of others are generally ignored. Remember that you are talking to volunteers, not slaves. They are not obligated to help you. If you're posting code, remove any code that is not relevant to the problem at hand so that the reader can focus on what is wrong without having to sift through many lines of source code. Include information about what version of DirectX you are using, what operating system you use for development and testing, and what graphics card you are using to test your code. You should look in the documentation for the answer you need before posting. If you haven't found what you need in the documentation, then mention that in your post so that the reader will know you've done your homework.

I have only touched on the basics of asking a smart question. More specific and detailed suggestions on how to ask smart questions are given by Eric Raymond in his web essay "How To Ask Questions The Smart Way".[3]

22.9.2 IHV Developer Relations

Sometimes the information you need is specific to a particular video card, particularly when investigating problems that could be related to the driver. The contact information for some **IHV**s is listed in table 22.5. If your hardware vendor isn't listed, then consult their web site for developer related information. Hardware manufacturers want to help developers get the most out of their cards and generally have a staff of engineers to assist developers. The developer areas of their web sites often have white papers, "how to" articles and additional documentation specific to individual cards, such as vendor-specific four character surface formats, performance tips, and special texture stage setups for achieving special effects.

22.10 Filing Bug Reports

After investigating a problem, you may come to believe that the bug is not in your code, but in the driver or the runtime. At this point you should file a bug report on the problem. After filing the bug report, you can decide if you want to

[3] `http://www.catb.org/~esr/faqs/smart-questions.html`

Company	Contact Information
3DLabs	developer.relations@3dlabs.com
	http://www.3dlabs.com/support/developer/index.htm
ATI	devrel@ati.com
	http://mirror.ati.com/developer/index.html
NVidia	DeveloperRelations@nvidia.com
	http://developer.nvidia.com
Matrox	devrel@matrox.com
	http://developer.matrox.com

Table 22.5: IHV Contact Information

wait for action from the vendor or if you want to attempt to code a workaround for the bug.

The most important thing to remember about bug reports is that the recipient of a bug report is a person just like yourself. Remember to be courteous and as informative as possible in the bug report. If you had the job of assigning priorities to bug reports, would you assign a high priority to bug reports that were insulting or provided little useful information on how to reproduce the bug?

The best way to file a bug report is to provide a simple program in source code form that reproduces the bug. When you encounter a bug, first try to write the simplest program that reproduces the problem. You can use the AppWizard from the SDK, or the `rt_Apprentice` sample application to generate the program. Along the way you may discover that the problem was really in your application.

Screen captures of improperly rendered output also help, but remember to keep the file size to a minimum. Rarely do screen shots need to be pixel accurate in order to portray the problem; converting them to JPEG format will reduce the filesize without compromising their ability to portray the issue.

You should always include the following items in your bug report:

- OS version and service pack level.
- DirectX version.
- DirectX Diagnostic Tool output.
- An accurate, step-by-step set of instructions for reproducing the problem.
- A sample application that reproduces the problem, preferably as source code.

The DirectX Diagnostic Tool includes detailed OS and DirectX version information, but including these in the body of your bug report helps the recipient see the relevant versions at a glance.

If you can't reproduce the problem in a small sample program that you can provide as source, then at least provide a copy of your program that does

reproduce the problem. If the size of the program and its associated data (best packaged as a ZIP file) is large, consider uploading the relevant package to a web site and provide a URL for it. If a URL is provided, make sure that the package will be available on the web site until you get feedback that the bug has been addressed.

If you believe the bug is within the Direct3D runtime, or within the reference rasterizer, you can submit the bug to Microsoft at the address `directx@microsoft.com`. For driver bugs, you can report them to the contact address for the hardware vendor. See section 22.9.2 for a list of contact addresses for some popular hardware vendors. It is a good idea to send a copy of driver bug reports to Microsoft as well, so that they can incorporate the feedback into the Windows Hardware Quality Labs tests that validate drivers.