DirectX Introduction

The windows Graphics Device Interface (GDI) is much to slow for creating real-time action games. You can write certain basic programs, including card or action adventure games but that is as far as the GDI is likely to be useful.

** You should remember the GDI from when we covered it last year☺*

Due to the limited use of the GDI DirectX was developed to give programmers the necessary tools to go those extra stages and write professional games that have a performance comparable with DOS32[1] applications.

DirectX is designed as a low-level[2] API[3] that integrates smoothly with Windows and the Win32 API.

DirectX can be used to access video, audio, input devices and networking capabilities without writing one line of GDI or using the standard Win32 libraries.
Using DirectX to work with these systems should not cause conflict with Window, Win32 or the GDI.

For DirectX to work a number of subsystems such as the Component Object Model (COM) and the Win32 API have to communicate with each other.

A Windows Primer

Windows is a shared, cooperative, multitasking operating system. This means that all applications have to share the same resources, resources such as the mouse, graphics card, sound card, and monitor. Video games, by their nature break the rule. Games usually take over everything; they also need high performance so generally only game is run at once. Obviously when we say ‘high performance’ we are talking about most commercial games.

DirectX enables you to access the hardware without going through the normal windows channels.
DirectX is basically a set of Dynamic Link Libraries (.DLLs)[4] and low-level device drivers that have the ability to control aspects of the PC with very little help from either the Win32 or GDI library.
To create a DirectX application all you need are the header files, DirectX libraries and .DLLs on the machine you are working.
DirectX can be viewed as an exceptionally useful add-on.

Dynamically Linked List (.DLL)

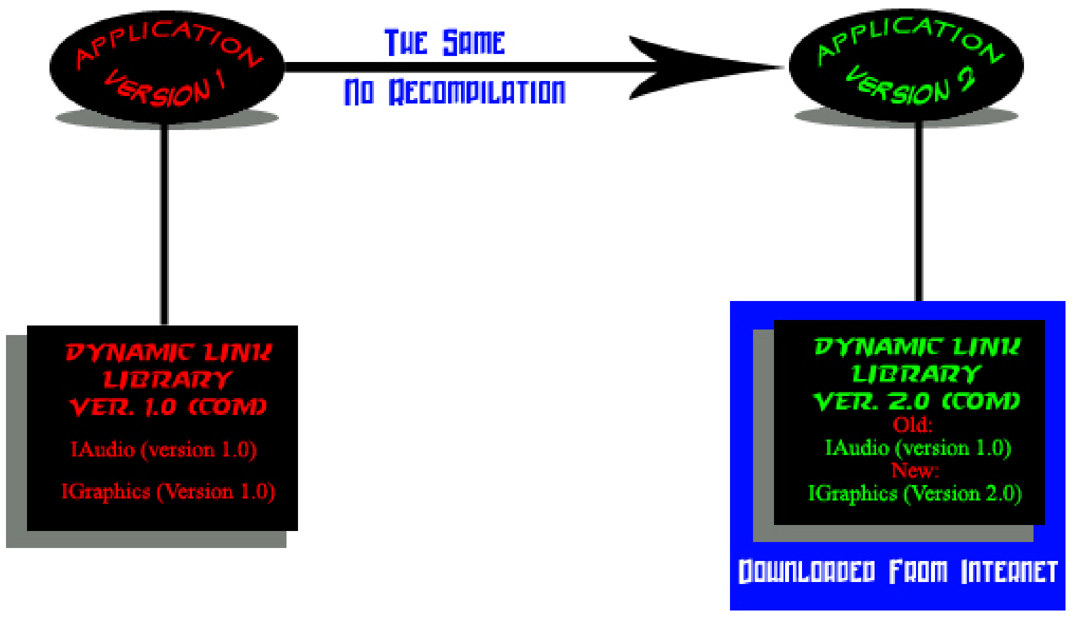
.DLLs are part of the basis that operates much of Windows. They allow software libraries to be loaded on demand and to be shared by other applications. In addition .DLLs can be upgraded or replaced without recompiling or breaking down the applications that use them, as long as the .DLLs continue to implement the functionality of the older versions.
If not you will get errors. If you have used Windows a lot, especially NT or 98 you may have seen the error message: Cannot locate #/&%£%&.DLL (e.g. UsrEnv.dll)

Dynamic link libraries provide an important facility within the Windows OS. DLL's provide the ability to plug prewritten modules into the operating system, allowing applications to share common codes modularize commonly used functions and provide extendibility. The Windows OS itself is comprised mostly of DLLs and any application that makes use of the Win32 API and DirectX is making use of the DLL facility. An application can gain the use of procedures located within a DLL by two methods loadtime dynamic linking and runtime dynamic linking.

Runtime linking is performed on run time by specifying a path to the DLL. This method is generally used on advanced procedural designs, where the procedures to be loaded need access time.

Loadtime dynamic linking, on the other side, involves loading the DLL before execution. The application will fail to run if the currently linked library is missing or placed on a different unspecified path. Retrieving procedures and data from a loadtime DLL is a matter of import libraries which must be statically linked on compile time with the application Win32 executable(*.exe).
An application employs loadtime dynamic linking by specifying the names of the DLL procedures directly in the source code. The linker inserts references to these procedures when it locates them in an import library linked with the application by using the IMPORTS section of the module definition file for the application. When the application is executed the Windows loader loads the DLLs into memory and resolves these references.

Dynamic Link Libraries as well as the Win32 API model are the software base for the most popular Multimedia engine on today's technologies known as DirectX.

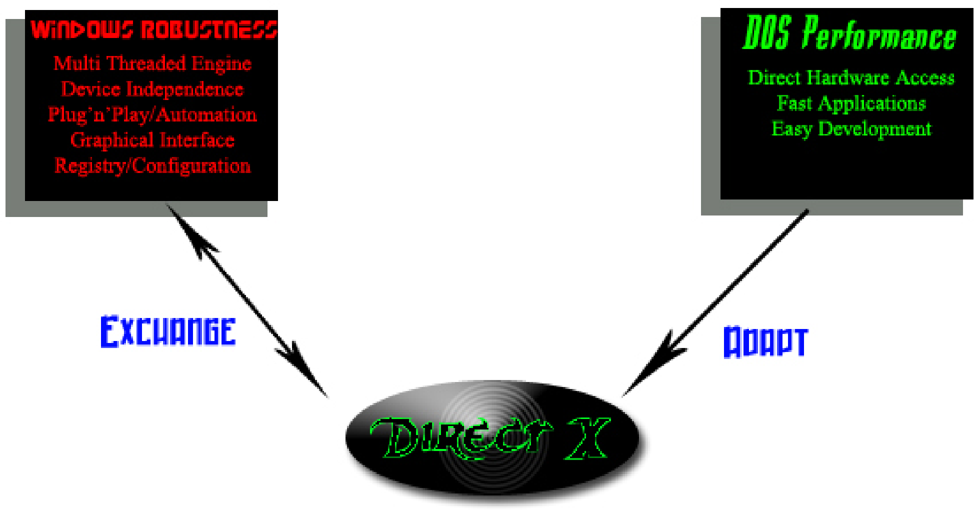


The DirectX infrastructure

DirectX has massive support in that the majority of hardware and software manufacturers have written DirectX applications or Drivers.

This has resulted in a massive infrastructure of support for DirectX.

To make the DirectX technology work Microsoft had to develop new techniques and conventions that would make DirectX very robust.
Later versions of DirectX still contain the functionality of previous models; therefore a game written for DirectX 1.0 should be able to run using any version of DirectX.
Currently we are up to DirectX 9.0c. This new method, calling it 9.0a/b/c instead of 9.1 or 10.0 is because of the way the new revisions (generally minor) have worked.



DirectX had the potential to spiral out of control very quickly, luckily it was written with some foresight. What was needed to keep it from getting out of control was a way of writing software that was object oriented, upgradeable, capable of working with multiple languages and abstracted from the programmer.
This was solved by use of the Component Object Model (COM).

The Component Object Model (COM)

The idea behind COM is abstraction, all you need to do is plug the bits in and make sure they are compatible.

COM is Microsoft's answer in distributed technology. It was invented years back, to support Object Linking and Embedding (OLE) under the Windows OS. It is a standard that defines how objects or components interact with one another.
One of the main aspects of COM technology is that it allows developers to keep their software upgraded without the need for recompilation. For example, a new graphics technology came out to the market that allows faster three-dimensional hardware accessibility. Without COM all of the software created with the older technology would need to be recompiled and redistributed in order to use this new engine. COM technology, on the other side, would just make a simple add on, forcing convention with the existing software without recompilation.
Another simplified method of applying COM convention is the DirectX engine. DirectX uses COM technology for keeping its *temporal identity* accurate and

consistent. This was actually a sophisticated procedure that Microsoft performed in order to apply secure contemporary and future compatibility.
Component Object Modelling is very complex at the low level and designing a COM based software may be challenging for distribution, but using COM objects is very easy.

"A COM object is really a C++ class that contains a number of interface classes. These interface classes are *pure* and *virtual* and must be implemented in a container class. Each interface class contains the functions that the COM object supports, so a COM object can have one or more interfaces that you communicate with and call functions through.

One interface may contain methods to draw objects, another may contain methods to make sounds, and so on. But each interface is a *pure virtual* class that you only create a template for in its definition; you wait to implement the interfaces until defining the component itself.

You have many ways to implement a bunch of interface classes and then contain them all in a container class, but the COM specification dictates exactly how you must do this implementation and containing. In addition, the COM specs explain exactly how a client application can talk to a COM object, how to create the COM object, how to destroy one and so forth. Thus the COM specs explain all the details of the implementation so that all you have to do is create all the interfaces and the code that goes with them.

Finally, because COM objects are dynamically linkable, you must use something like a .DLL to contain them. You don't have to, but that method is the easiest way. Also, because COM objects need to work with any language, they have an exact binary specification, which means you must make sure your compiler creates the exact binary footprint that a standard Windows C++ compiler does when it creates virtual classes. You're saying "What the #$%@ is he talking about?" These concepts become clearer as you actually use COM objects.

Another version of COM, called DCOM (Distributed COM), is even more advanced than COM and enables the use not only of components on your machine but components on other machines over a network. Is that wild or what?"[5]

DirectX components
As should have become apparent now, DirectX is made up of components. These components cover different aspects of video game design:
Graphics
Sound
Input
3D
Networking

There is one problem when looking for a summary of the DirextX components, different places list them differently sometimes missing out components. This is further compounded by the different DirectX releases.

DirectInput allows programmers to make good use of everything from joysticks to steering wheels to Force Feedback mice. It even provides a controller calibration interface. For joysticks the manufacturers write DirectInput drivers for us.

DirectGraphics Is the name of the unified DirectDraw/Direct3D interfaces in DirectX 8.0+. To create 2D graphics we then have to use 3D acceleration and interfaces.
Due to DirectX's backwards compatibility we can use DirectDraw to produce 2D graphics using the DirectX 7.0 interface.

DirectDraw is a library of 2-D graphics functions. It may not be glamorous, but it's vital. DirectDraw enables you to access the video card, hardware acceleration and set up every video mode (true colour, high resolution that sort of thing). It also supports palettes, clipping and animation.

Direct3D is a set of APIs for rendering 3-D graphics. It makes it easy for programmers to pull off sophisticated effects that would have been unimaginable just a few years ago. When a 3-D card is billed as supporting a DirectX version, the manufacturer is mostly referring to Direct3D features such as displacement mapping and pixel shaders.

DirectAudio is a name for the new unified, tightly coupled sound system interface that includes DirectSound, DirectSound3D and DirectMusic.

DirectSound supports sound input and output, it also supports pure mixing of multiple channels in real time, MIDI and a host of other things.

DirectSound3D is used for advanced spatial effects like 3-D sound placement and real-time echoes.

DirectMusic is a rich library of sound functions specifically for music, allowing games like Asheron's Call 2 to create interactive symphonies and even mix multiple musical themes on the fly. It's also handy for environmental sound effects. It is effectively an API that allows you to play MIDI files and other media through a software synthesizer. It is supposed to sound as good as a wave table or wave guide synthesizer.

DirectShow allows software to easily manipulate video streams, from playing AVIs to managing video capture cards.

DirectPlay is a set of networking tools for multiplayer games and is essential to any frag-fest. Due to DirectPlay enhancements in DirectX 9, Microsoft recommends that all multiplayer aficionados upgrade to this version. It is a rather robust set of functions and systems that allow you to write code for a variety of connections without the hassle of winsock[6] or writing your own code. Unfortunately it takes a long time to learn all that DirectPlay has to offer.

DirectSetup and AutoPlay Not really DirectX objects. DirectSetup is an API that you can use to install and set up DirectX. Autoplay is the standard windows support for automatically loading CDs when they are placed in the machine.

Hardware Abstraction Layer (HAL)

HAL is the totally direct access to an appropriate hardware component. This layer is usually the device driver from the vendor, and communication to it is direct through generic DirectX calls. For example, a memory independent graphics card that supports double buffering will result in faster performance than software emulation double buffering, so HAL must be selected. In this case DirectX is actually making successive function calls to an external driver in the form of DLL or VXD which was supplied by the vendor after purchasing this piece of hardware (Hardware Installation Pack).

Microsoft Direct3D provides device independence through the hardware abstraction layer (HAL). The HAL is a device-specific interface, provided by the device manufacturer, that Direct3D uses to work directly with the display hardware. Applications never interact with the HAL. Rather, with the infrastructure that the HAL provides, Direct3D exposes a consistent set of interfaces and methods that an application uses to display graphics. The device manufacturer implements the HAL using 32-bit code with Microsoft Windows XP, Microsoft Windows NT, and Windows 2000. Windows 98 and Windows Millennium Edition (Windows Me) use a combination of 16-bit and 32-bit code. The HAL can be part of the display driver or a separate dynamic-link library (DLL) that communicates with the display driver through a private interface that the driver's creator defines.

The Direct3DHAL is implemented by the chip manufacturer, board producer, or OEM. The HAL implements only device-dependent code and performs no emulation. If a function is not performed by the hardware, the HAL does not report it as a hardware capability. Additionally, the HAL does not validate parameters; Direct3D does this before the HAL is invoked.

In Microsoft DirectX 9.0, the HAL can have three different vertex processing modes: software vertex processing, hardware vertex processing, and mixed vertex processing on the same device. The pure device mode is a variant of the HAL device. The pure device type supports hardware vertex processing only, and allows only a small subset of the device state to be queried by the application. In addition, the pure device is available only on adapters that have a minimum level of capabilities.

“A layer that consists of hardware and device driver mechanisms that insulate applications from device-specific implementation details. If a capability requested by an application is not implemented by the current hardware, the capability will be emulated by the software.”

Hardware Emulation Layer (HEL)

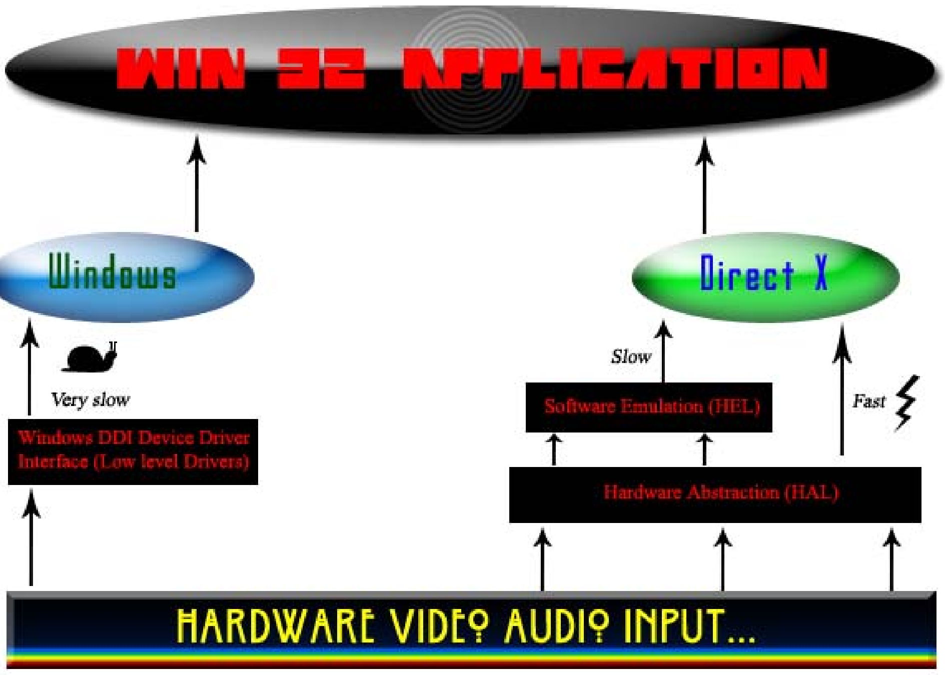
HEL is built on top of HAL and deals with hardware problems. It is used when the hardware doesn’t support the feature requested and it is very slow. A common example is three-dimensional graphics software deceleration when an invalid or old graphics card is attached to the computer-system. DirectX is then responsible for processing all of the three dimensional graphics procedures using the systems processor and this is extremely slow.

To give you a better example if you wrote some graphics code that would scale and rotate a bitmap. You therefore make calls to DirectX to scale and rotate a bitmap. If this code was then run on a machine where the graphics card supported scaling and rotation the code would run at full speed using the hardware. If this code was run on a machine where the graphics card did not support scaling and rotation HEL would then *emulate* the functionality of HAL. This would mean that the code would run more slowly as it is being emulated.

You can in your code query the hardware and then use your own routines if what you want to do isn't supported.

"A middleware layer that provides software-based emulation of features that are not present in hardware."

***Note:** Both HAL and HEL are abstracted from the programmer's eyes. However the programmer is still made responsible for finding, setting and choosing between layers, adapters and modes.*


WIN 32 APPLICATION
Windows
Direct X
Very slow
Slow
Fast
Windows DDI Device Driver Interface (Low level Drivers)
Software Emulation (HEL)
Hardware Abstraction (HAL)
HARDWARE VIDEO AUDIO INPUT...

Interfacing

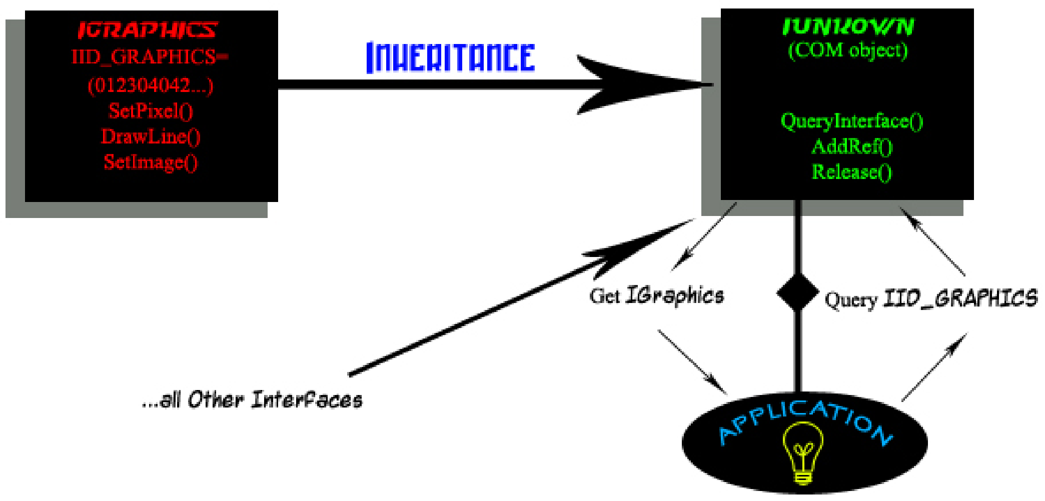
To use DirectX it helps immensely to understand what is occurring with COM and the interfaces. This is because DirectX is implemented as a set of components that each have a number of interfaces….hopefully this was made relatively clear in all the information above.

A component consists of one or more interfaces. Each of the interfaces is like a communications port that you must use to call the functions within the interface.

Each one of these interfaces also has a number of functions that may be accessible from the application after importing the COM object. A single COM object can have one or more interfaces and one or more COM objects can be implemented by an application.
To do this you need to go through a series of stages.

First you need to define your interface classes. Each interface contains a list of methods that you implement later but are methods supported by the interface.

All interfaces are derived from the root interface `IUnknown`. The Microsoft COM specification states that all the COM interfaces must be derived from this special base class.



Seeking COM interfaces is a matter of Globally Unique Identifiers (GUIDs) and Interface IDentifiers (IIDs). Both are 128 bit long numbers assigned to every single interface. They are guaranteed to be unique from each other; this is good as no two IIDs can be the same.
Basically the COM object has a GUID and the interfaces have a IID and they are ways to name them uniquely. Unique identifiers are especially necessary when distributing.
Every interface must have its own IID. These numbers are generated with a Microsoft supplied program surprisingly called `GUIDGEN.EXE`. This should be provided with most compilers. It is this program that guarantees, through mathematics, that it will never produce two identical GUIDs.

Following up is the definition of IUnknown whereby the COM object originates, and COM interfaces are derived.

```
struct IUnknown
{
   virtual HRESULT __stdcall QueryInterface(const IID &iid,(void**)ip) = 0;
   virtual ULONG __stdcall AddRef() = 0;
   virtual ULONG __stdcall Release() = 0;
};
```

***Note:** We are now seeing the keyword `virtual` used a lot more frequently. We covered `virtual` functions and classes in programming 2 (Week 4). Just to recap: the `virtual` keyword identifies to the compiler that a class function may be overridden in a derived class. That is, the derived class may supply a function of the same name and argument list, but with different functionality defined in the function body. It is then the responsibility of the compiler (and/or its runtime libraries) to assure that, given the actual object being referenced, the correct function is called.*

`IUnknown` is the base class that all interfaces must derive from; hence all interfaces must implement at least `QueryInterface(), AddRef()` and `Release()`.

`QueryInterface()`: Requests a pointer to the interface of interest. You must pass an interface IID along with a pointer to hold the interface returned from the function. This is the only way to access an interface. You don't need to use it very often as DirectX generally has a wrapper function[7] or macro[8] to do it for you.

`AddRef()`: Increases the internal reference count of the COM object. All COM objects allocate and deallocate themselves through reference counting. When an object is created its internal reference count is incremented; when a reference to an object is no longer needed its reference count is decremented. By tracking the reference count you can then detect how many other objects are using it. When the reference count is 0 the object is no longer in use and can be deleted. `AddRef()`is generally not used, being called internally by all COM objects.

`Release()`: Decrements the reference count of a COM object. Unlike `AddRef()` this is used all the time, well when you've finished using an interface or component.

The following code illustrates the creation of a COM object. In addition it generates an interface object by passing the appropriate GUID.

```
void main()
{
    // Create the main COM object
    IUnknown* punkown = CoCreateInstance();
    IGraphics* gfx; // pointer to interface
    // from the original COM object query the Interface:
    // passing the appropriate GUID / casting a void** pointer
    punknown->QueryInterface(IID_GRAPHICS,(void**)&gfx);
    // try some methods
    gfx ->SetPixel(10,10,RED);
    // Release the Interface
    gfx ->Release();
    // Release the COM object itself
     punknown ->Release();
}
```

The IID_GRAPHICS is a globally defined GUID, which was assigned to the IGraphics interface in order to identify this interface in the world of COM objects.

After creating a new (IUnkown) COM object the application queries for a new IGraphics object using its globally unique identifier. The COM object finds the interface and dynamical generates a new IGraphics object, which is then assigned to an application pointer. The application may then easily access primitives from IGraphics. After the end of execution all interfaces and COM objects must be released otherwise the program will crash or a memory leak will result in bad later execution.

As you can see from the above snippet of code and the COM discussions as well as getting virtual we are going to be using a lot of function pointers.

***Note:** On my website I have put up a document entitled function pointers.*

Mix and Match

It is possible to combine DirectX, Win32, GDI and COM. They all work together very well.
There are different possible development routes, using the DirectX SDK or by using select header files we can just use parts of DirectX, such as DirectDraw.
We will be focusing in this course on using the SDK.

[1] What is DOS32 – DOS32 is a DOS Extender, what that is, is a really cool program designed to run in protected mode, and it offers programs a flat memory layout, that is, no 64kb segments & 16bit code size, its a pure 32bit environment, with some reflection in 16bit mode.

[2] What do we mean by low-level – In this case interacts directly with the 'machine'

[3] What does API stand for – Abbreviation of **Application Program Interface**, a set of routines, protocols, and tools for building software applications. A good API makes it easier to develop a program by providing all the building blocks. A programmer puts the blocks together.

[4] What is a .DLL – Stands for "Dynamic Link Library." A DLL (.dll) file contains a library of functions and other information that can be accessed by a Windows program. When a program is launched, links to the necessary .dll files are created. If a static link is created, the .dll files will be in use as long as the program is active. If a dynamic link is created, the .dll files will only be used when needed. Dynamic links help programs use resources, such as memory and hard drive space, more efficiently.

DLL files can also be used by more than one program. In fact, they can even be used by multiple programs at the same time. Some DLLs come with the Windows operating system while others are added when new programs are installed. You typically don't want to open a .dll file directly, since the program that uses it will automatically load it if needed. Though DLL filenames usally end in ".dll," they can also end in .exe, .drv, and .fon, just to make things more confusing.

[5] LaMothe, A. Windows Game Programming for Dummies. *Wiley Publishing,* pg 133-135.

[6] Winsock - Short for Windows Socket, an API for developing Windows programs that can communicate with other machines via the TCP/IP protocol. Windows 95 and Windows NT comes with Dynamic Link Library (DLL) called winsock.dll that implements the API and acts as the glue between Windows programs and TCP/IP connections.

In addition to the Microsoft version of winsock.dll, there are other freeware and shareware versions of winsock.dll. However, there is no official standard for the Winsock API, so each implementation differs in minor ways.

[7] Wrapper function - A function that provides a simplified interface to another function, for example by changing the order of some parameters or by interpreting the return code.

[8] Macro - A means of executing a group of instructions within a program. Many programs offer the capability to put together macros so that you don't have to do the same group of repetitive instructions one by one. You can program the macro to do any number of instructions. Many programs allow you to extend the simple macro languages that they provide with other, more complex programming languages.