Direct 3D Introduction

This provides introduction and information on what is possibly the most important, and is certainly going to be our most used DirectX component.

This introduction matches with the Direct3D_1.zip file. The .zip file contains a code implementation of this tutorial.

Background

Direct3D is a low level graphics API that is part of DirectX and that enables us to render 3D worlds using 3D hardware acceleration

Direct3D has in recent additions of DirectX replaced DirectDraw, which you can still use due to backwards compatibility but is now a relic of DirectX 7.0.

There are many games that still use 2D graphics and games that don't need 3D graphics, despite this Microsoft still moved it incorporating it in Direct3D. Although Direct3D is primarily a 3D application it can be used for 2D. Some people would argue that it also does it better than DirectDraw.

2D games are also easier to program for. If you spent your whole degree learning 3D graphics programming you would probably still only have an intro to what is essentially an immense subject.
This is something to take into account when completing your assessments; your games do not have to break any graphically boundaries or be immense works of 3D art that utilise many functions.

In Programming 3 you will learn to Program with Direct3D, you will obviously learn other features such as sound and input, but the principle focus will be on Direct3D. Learning will initially focus 2D programming with Direct3D before moving on to 3D programming.

Incorporation of DirectDraw in Direct3D

If you are wondering why DirectDraw was incorporated into Direct3D then here is your answer:
DirectDraw provides the functionality necessary to directly access and manipulate images in memory.
Direct3D must access display memory it requires this functionality of DirectDraw. This means that Direct3D provides the functio0nality necessary to create 3D worlds; DirectDraw provides the link between display and Direct3D.
Windows on its own does not allow direct access, making it on its own a very graphically very slow system.
Libraries such as DirectX and OpenGL that allow this access and add functionality turn Windows machines into good systems for gaming.

Creating a Direct3D Object

The first step in writing a Direct3D program is to create a Direct3D object. As mentioned in the introduction to DirectX this object then provides you with access to the `IDirect3D9` interface.

```
IDirect3D9 *g_pD3D = NULL;

g_pD3D = Direct3DCreate9(D3D_SDK_VERSION);

if(!g_pD3D)
{
	return E_FAIL;
}
```

The code first declares a pointer to `IDirect3D9` interface. It then called g_`pD3D` because it is a global pointer to the `Direct3D` interface.
This is worth noting because in many places you will see the declaration use LPDIRECT3D9. LPDIRECT3D9 is just a typedef, which is a pointer to an IDirect3D9 object. All of the DirectX objects have typedefs which are in all-caps and preceded by "LP". I prefer to use the underlying type for 2 reasons. I find long all-caps declarations to be less legible and I prefer my pointers to look like pointers. So it's easier to read and more obvious what you're doing. Increased readability in your code is always a good thing.

Next `Direct3DCreate9` is called to obtain a pointer to the Direct3D object. Direct3DCreate9 takes only 1 parameter. In almost every case, you'll use D3D_SDK_VERSION. This creates a Direct3D object that is compatible with the version of the SDK you're using. If for some reason you needed to work with an older version of Direct3D9, you could pass that version instead. Likely you will never need to do this, so just pass D3D_SDK_VERSION.

The if statement is to check to see that the call to `Direct3DCreate9()` didn't fail.
If the call failed, the error is handled here. We handle the error by returning a DirectX constant `E_FAIL.`
`E_FAIL` represents a failure of call to `Direct3DCreate9()` as provided by the DirectX SDK.
The DirectX SDK defines many such constants to represent different errors.

At the end of your code you use:

```
if(g_pD3D)
{
	g_pD3D -> Release();
	g_pD3D = NULL;
}
```

When you are done using the `IDirect3D9` object, you call its Release method. The quick short explanation of this method is that it frees the object. The long description is a bit more involved. DirectX uses COM (Component Object Model) and all of its objects inherit from `IUnknown`. When an `IUnknown` object is created an internal reference count is incremented and when Release is called the reference count is decremented. When the reference count reaches 0 the object is freed.

When we call Release, we first check if the pointer is NULL. Dereferencing a NULL pointer will cause an exception which will terminate the application. If the pointer isn't NULL we call Release and then set the pointer to NULL to prevent it from being freed multiple times. A Microsoft-defined MACRO, SAFE_RELEASE, does this as well. SAFE_RELEASE is defined in dxutil.h which is in the Common folder of the SDK.

Often in your programs it helps to declare a function called CleanUpDirect3D() and then use this to release anything that needs releasing.

Creating a Direct3D Device

After you create a Direct3D object you need to create a Direct3D device. The device is an object of the `IDirect3DDevice9` interface. The object provides you with almost a hundred methods for performing various graphical manipulations. Quite a lot of graphical work is performed using the `IDirect3DDevice9` object.

To create the Direct3D device object and obtain a pointer to it you must call the objects `CreateDevice()` method. The DirectX API declares this method like this:

```
HRESULT CreateDevice(
   UINT Adapter,
   D3DDEVTYPE DeviceType,
   HWND hFocusWindow,
   DWORD BehaviorFlags,
   D3DPRESENT_PARAMETERS *pPresentationParameters,
   IDirect3DDevice9** ppReturnedDeviceInterface
);
```

`Adapter`
This specifies which display adapter the device should be associated with. In most cases there is a one-to-one relationship between adapters and video cards. On cards that support multi-head (they can drive multiple monitors from a single card) each "head" may be its own adapter. To get the primary display, use D3DADAPTER_DEFAULT.

`DeviceType`
In most cases you will use D3DDEVTYPE_HAL. HAL stands for Hardware Acceleration Layer and makes the best use of your video card.

D3DDEVTYPE_HAL gives the best results. To use it you will sometimes need to query the device to find out what systems are supported.
D3DDEVTYPE_REF does all of its processing in software. While this is incredibly slow (single digit frames per second are typical) it is a good debugging aid. REF is short for Reference Rasterizer. This is a complete implementation of the Direct3D spec. It supports many operations that your video card may not. While using HAL, if you find something that doesn't work properly you can run it on REF and if it's still wrong, it is likely a bug in your program, otherwise it may be a bug in the video driver. Also, REF is installed with the SDK, so don't count on it being available on your user's PCs.
D3DDEVTYPE_SW exists to support pluggable software implementations, but so far none exist.
You can if you desire check out the MSDN breakdown of DEVTYPE:
http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/directx9_c/directx/graphics/reference/d3d/enums/d3ddevtype.asp

`hFocusWindow`
Set this to the same value as the DeviceWindow in your presentation parameters. NOTE:If running in full-screen, the focus must be a top level window. Child windows and controls are not valid.

`BehaviorFlags`
This controls a lot of behind-the-scenes behaviours. Most are for advanced use so we won't cover them here. The 3 most common flags are: D3DCREATE_SOFTWARE_VERTEXPROCESSING, D3DCREATE_HARDWARE_VERTEXPROCESSING, and D3DCREATE_MIXED_VERTEXPROCESSING. One and only one of these three flags can be used. SOFTWARE sets all vertex processing (lighting, for example) to be done in software. This is supported on all cards and has very relaxed limitations in what it can do. Since it's done in software, all vertex processing is done on the CPU. With HARDWARE, all of the vertex processing is done by the video card. This is faster because video cards are specifically designed for this work and also because it frees up your CPU to work on other things. Most modern cards support HARDWARE vertex processing. Specifying MIXED allows you to switch back and forth between SOFTWARE and HARDWARE. If there are operations that your hardware does not support, you can switch to software for those, and then switch to hardware for everything else.

`*pPresentationParameters`
This is just a pointer to your PRESENTATION_PARAMETERS structure. The DirectX SDK defines the structure like this:

`ppReturnedDeviceInterface`
If the call to CreateDevice succeeds, your new device will be returned here.

So here is how you use your Direct3D object pointer to create a device object for your application:

```
IDirect3DDevice9* g_pDirect3DDevice = NULL;

hResult = g_pDirect3D -> CreateDevice(
    D3DADAPTER_DEFAULT,
  D3DDEVTYPE_HAL, g_hWnd,
  D3DCREATE_SOFTWARE_VERTEXPROCESSING,
    &D3DPresentParams,
    &g_pDirect3DDevice );

if (FAILED(hResult))
{
        return E_FAIL;
}
```

Initially we need to declare a pointer to the `IDirect3DDevice9` interface, `g_pDirect3DDevice` in our example. As with all pointers that aren't set immediately they should be initialised to `NULL`.
We then call the objects `CreateDevice()` method.

Mentioned above is the `D3DPRESENT_PARAMETERS` this is something we have to set some parameters for. It is therefore useful to at understand some parts of it:

```
typedef struct _D3DPRESENT_PARAMETERS_ {
    UINT BackBufferWidth;
    UINT BackBufferHeight;
    D3DFORMAT BackBufferFormat;
    UINT BackBufferCount;
    D3DMULTISAMPLE_TYPE MultiSampleType;
    DWORD MultiSampleQuality;
    D3DSWAPEFFECT SwapEffect;
    HWND hDeviceWindow;
    BOOL Windowed;
    BOOL EnableAutoDepthStencil;
    D3DFORMAT AutoDepthStencilFormat;
    DWORD Flags;
    UINT FullScreen_RefreshRateInHz;
    UINT PresentationInterval;
} D3DPRESENT_PARAMETERS;
```

`BackBufferWidth`, `BackBufferHeight`
Width and height of the new swap chain's back buffers, in pixels. If Windowed is FALSE (the presentation is full-screen), these values must equal the width and height of one of the enumerated display modes found through IDirect3D9::EnumAdapterModes. If Windowed is TRUE and either of these values is zero, the corresponding dimension of the client area of the hDeviceWindow (or the focus window, if hDeviceWindow is NULL) is taken.

`BackBufferFormat`

The back buffer format; For more information about formats, see D3DFORMAT. This value must be one of the render-target formats as validated by IDirect3D9::CheckDeviceType. You can use IDirect3DDevice9::GetDisplayMode to obtain the current format.
In fact, D3DFMT_UNKNOWN can be specified for the BackBufferFormat while in windowed mode. This tells the runtime to use the current display-mode format and eliminates the need to call IDirect3DDevice9::GetDisplayMode.
For windowed applications, the back buffer format no longer needs to match the display-mode format because colour conversion can now be done by the hardware (if the hardware supports colour conversion). The set of possible back buffer formats is constrained, but the runtime will allow any valid back buffer format to be presented to any desktop format. (There is the additional requirement that the device be operable in the desktop mode; devices typically do not operate in 8 bits per pixel modes.)
Full-screen applications cannot do colour conversion.

`BackBufferCount`
This value can be 0 (or 1, 0 is treated as 1), 2, or 3. If the number of back buffers cannot be created, the runtime will fail the method call and fill this value with the number of back buffers that could be created. As a result, an application can call the method twice with the same D3DPRESENT_PARAMETERS structure and expect it to work the second time.
The method fails if one back buffer cannot be created. The value of BackBufferCount influences what set of swap effects are allowed. Specifically, any D3DSWAPEFFECT_COPY swap effect requires that there be exactly one back buffer.

`MultiSampleType`
Member of the D3DMULTISAMPLE_TYPE enumerated type. The value must be D3DMULTISAMPLE_NONE unless SwapEffect has been set to D3DSWAPEFFECT_DISCARD. Multisampling is supported only if the swap effect is D3DSWAPEFFECT_DISCARD.

`MultiSampleQuality`
Quality level. The valid range is between zero and one less than the level returned by pQualityLevels used by IDirect3D9::CheckDeviceMultiSampleType. Passing a larger value returns the error D3DERR_INVALIDCALL. Paired values of render targets or of depth stencil surfaces and D3DMULTISAMPLE_TYPE must match.

`SwapEffect`
Member of the D3DSWAPEFFECT enumerated type. The runtime will guarantee the implied semantics concerning buffer swap behaviour; therefore, if Windowed is TRUE and SwapEffect is set to D3DSWAPEFFECT_FLIP, the runtime will create one extra back buffer and copy whichever becomes the front buffer at presentation time.
D3DSWAPEFFECT_COPY requires that BackBufferCount be set to 1.
D3DSWAPEFFECT_DISCARD will be enforced in the debug runtime by filling any buffer with noise after it is presented.

`hDeviceWindow`
The device window determines the location and size of the back buffer on screen. This is used by Microsoft Direct3D when the back buffer contents are copied to the front buffer during IDirect3DDevice9::Present.
For a full-screen application, this is a handle to the top window (which is the focus window).
For applications that use multiple full-screen devices (such as a multimonitor system), exactly one device can use the focus window as the device window. All other devices must have unique device windows.
For a windowed-mode application, this handle will be the default target window for IDirect3DDevice9::Present. If this handle is NULL, the focus window will be taken.
Note that no attempt is made by the runtime to reflect user changes in window size. The back buffer is not implicitly reset when this window is reset. However, the IDirect3DDevice9::Present method does automatically track window position changes.

`Windowed`
TRUE if the application runs windowed; FALSE if the application runs full-screen.

`EnableAutoDepthStencil`
If this value is TRUE, Direct3D will manage depth buffers for the application. The device will create a depth-stencil buffer when it is created. The depth-stencil buffer will be automatically set as the render target of the device. When the device is reset, the depth-stencil buffer will be automatically destroyed and recreated in the new size.
If EnableAutoDepthStencil is TRUE, then AutoDepthStencilFormat must be a valid depth-stencil format.

`AutoDepthStencilFormat`
Member of the D3DFORMAT enumerated type. The format of the automatic depth-stencil surface that the device will create. This member is ignored unless EnableAutoDepthStencil is TRUE.

`Flags`
One of the D3DPRESENTFLAG constants.

`FullScreen_RefreshRateInHz`
The rate at which the display adapter refreshes the screen. The value depends on the mode in which the application is running:
For windowed mode, the refresh rate must be 0.
For full-screen mode, the refresh rate is one of the refresh rates returned by IDirect3D9::EnumAdapterModes.
`PresentationInterval`
The maximum rate at which the swap chain's back buffers can be presented to the front buffer. For a detailed explanation of the modes and the intervals that are supported, see `D3DPRESENT`.

For now all we need to know to incorporate it into our code is:

```
D3DPRESENT_PARAMETERS D3DPresentParams;
ZeroMemory(&D3DPresentParams, sizeof(D3DPRESENT_PARAMETERS));
D3DPresentParams.Windowed = FALSE;
D3DPresentParams.BackBufferCount = 1;
D3DPresentParams.BackBufferWidth = 800;
D3DPresentParams.BackBufferHeight = 600;
D3DPresentParams.BackBufferFormat = D3DFMT_X8R8G8B8;
D3DPresentParams.SwapEffect = D3DSWAPEFFECT_DISCARD;
D3DPresentParams.hDeviceWindow = g_hWnd;
```

Hopefully most of this is self explanatory, if not hopefully it will become clear in time, for now don’t worry.

Cleaning Up

We should from pervious modules and/or personal experience know that when we allocate memory in a program you need to release it before the application terminates.
This applies to Dierect3D objects. When we have finished with them we release them from memory. To release them from memory we call each objects `release()` method, as seen below:

```
void CleanUpDirect3D()
{
    if (g_pDirect3DDevice)
    {
```

```
        g_pDirect3DDevice->Release();
     }

    if (g_pDirect3D)
    {
        g_pDirect3D->Release();
     }
}
```

You should notice a few things about this code
- It checks whether the object pointers are NULL first. Calling release on a bad or NULL pointer as you will crash the program.
- Objects are released in reverse order to when they were initialised.

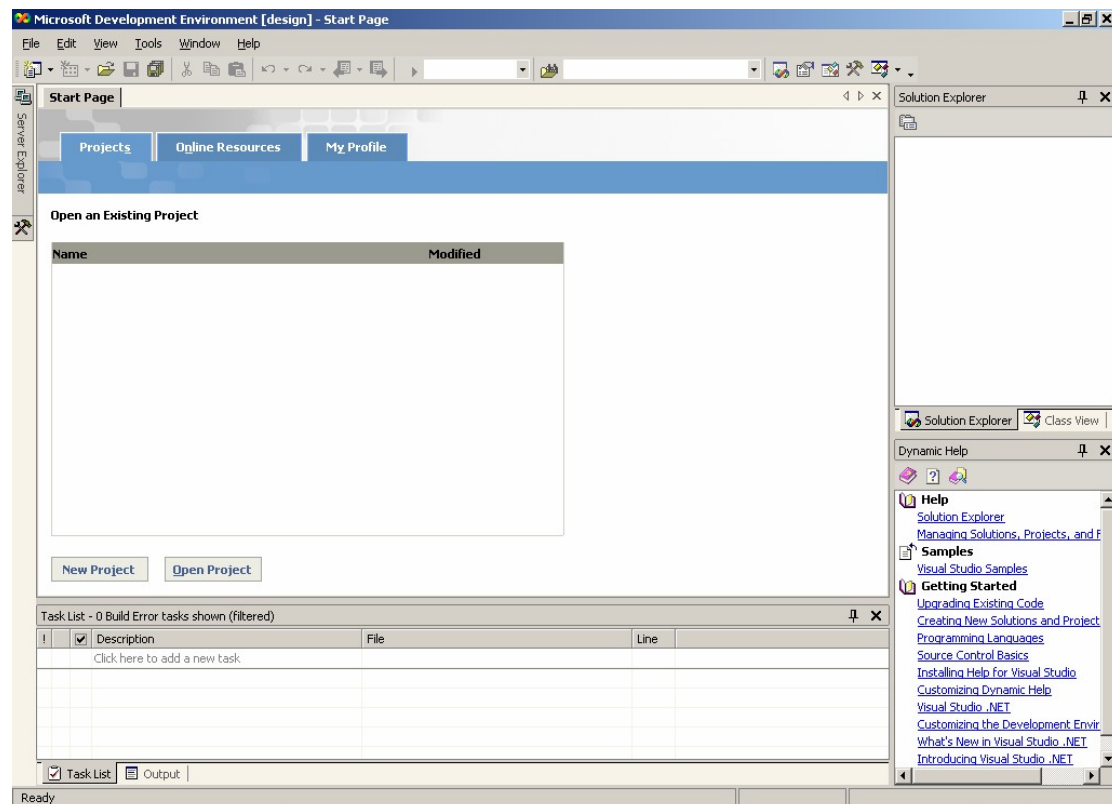
The Example

Now have a look at the Direct3D_1.zip file. This shows the integration of what we have covered along with Win32. This type of code is the starting point for most Direct3D programming.

You can type out the code I have provided or you can copy it across to another c++ file. I know you can just open the project but as I have already set the relevant associations you will not benefit from hands-on experience of setting your associations.

Start Microsoft Visual Studio .NET 2003 using the Start button from Windows or from the desktop shortcut.

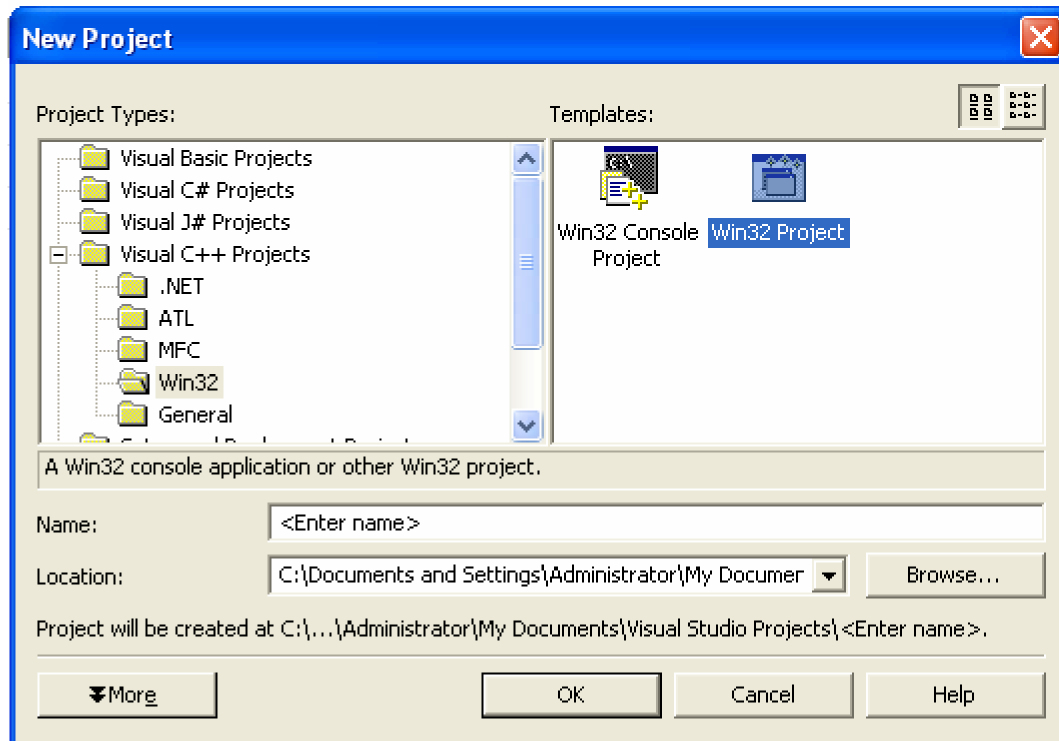
Once you have started Visual Studio you should see a screen similar to the following image:



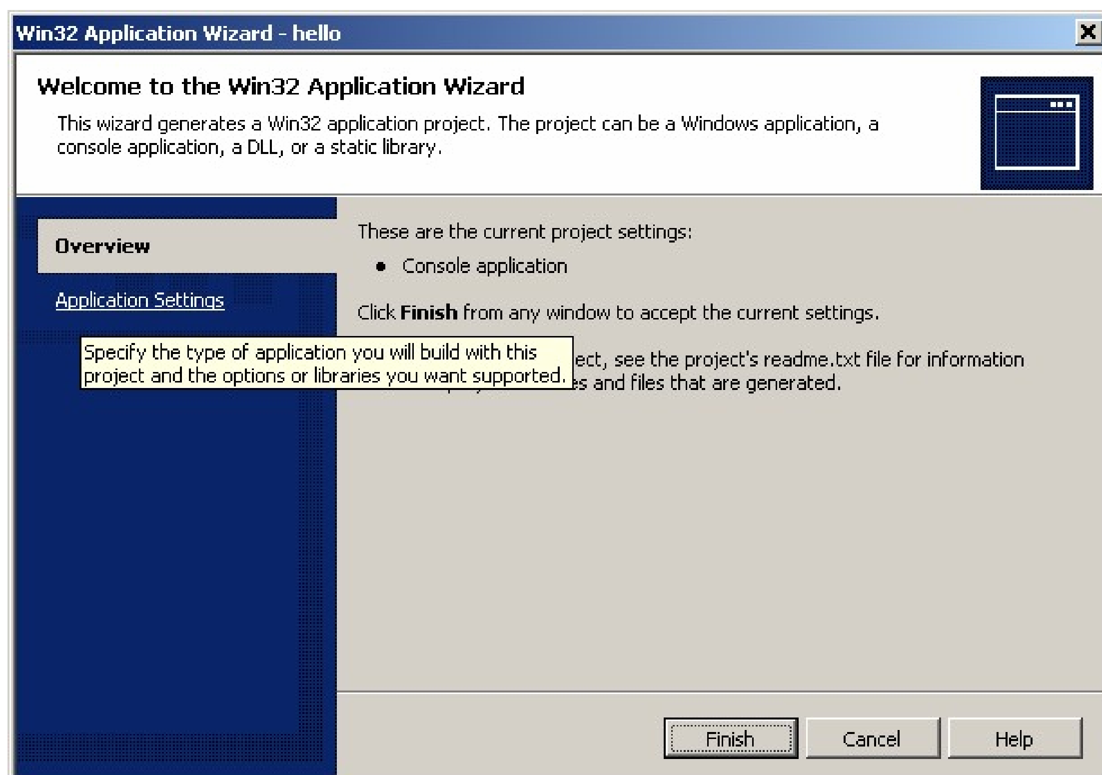
Click the menu item File | New | Project... or press CTRL+SHIFT+N or click the New Project button.

A dialog box is displayed.
In the left pane expand the selection named Visual C++ Projects and then select Win32.
In the right pane, select Win32 Console Project.

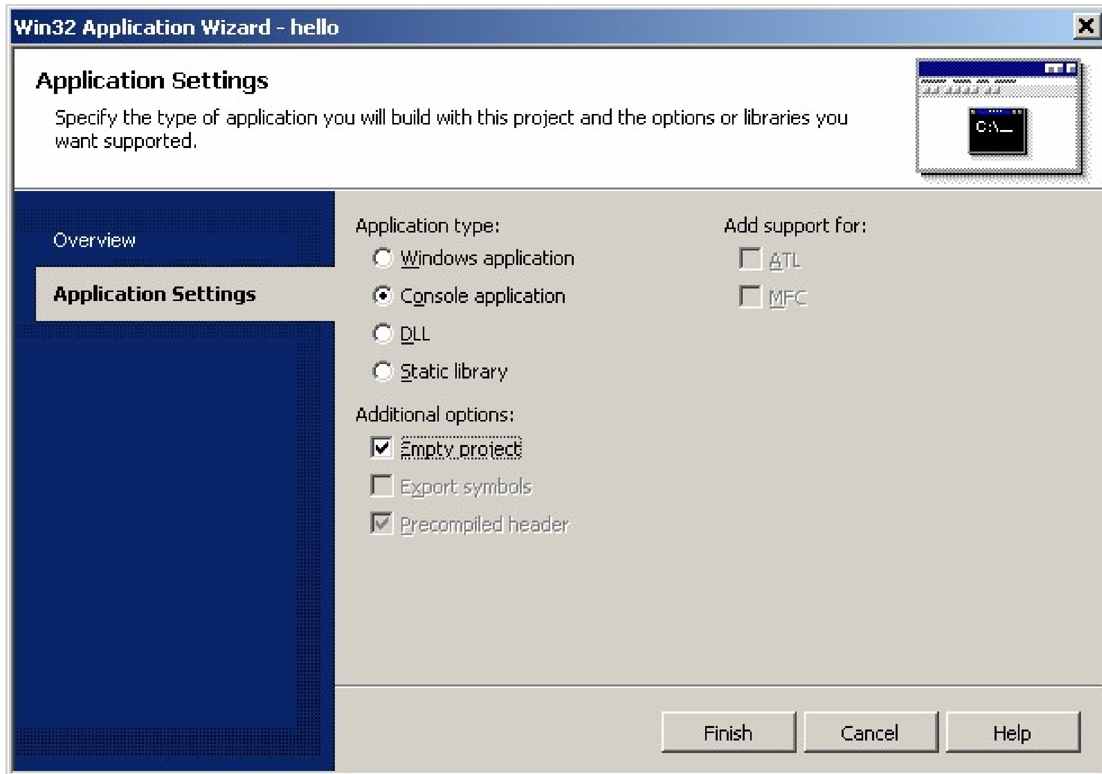
Before you write any code when using Visual Studio .NET, or any other version in fact, you have to choose the type of project. In our case it is a Win32 Project:



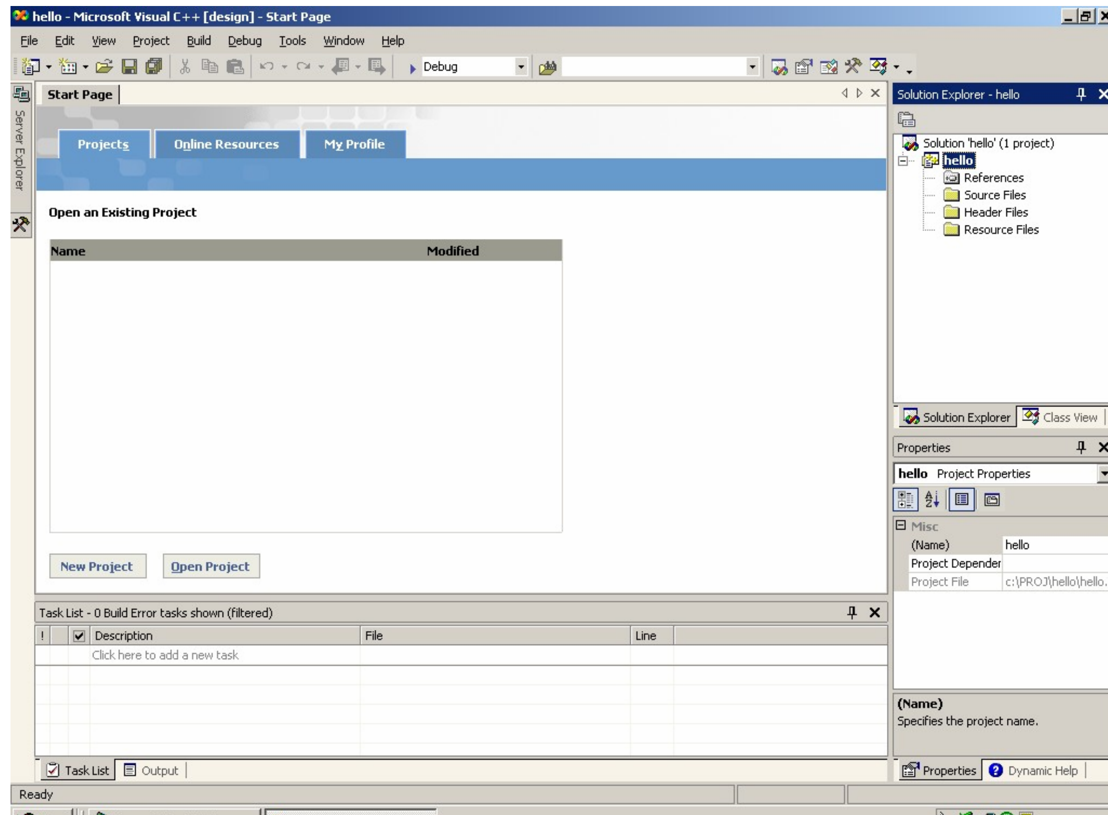
After typing in a name and selecting a suitable directory for your project, click the OK button to continue. You should see the following window:



Click Application Settings on the left side of the window and then place a check in the box marked Empty project. The screen should look like this:

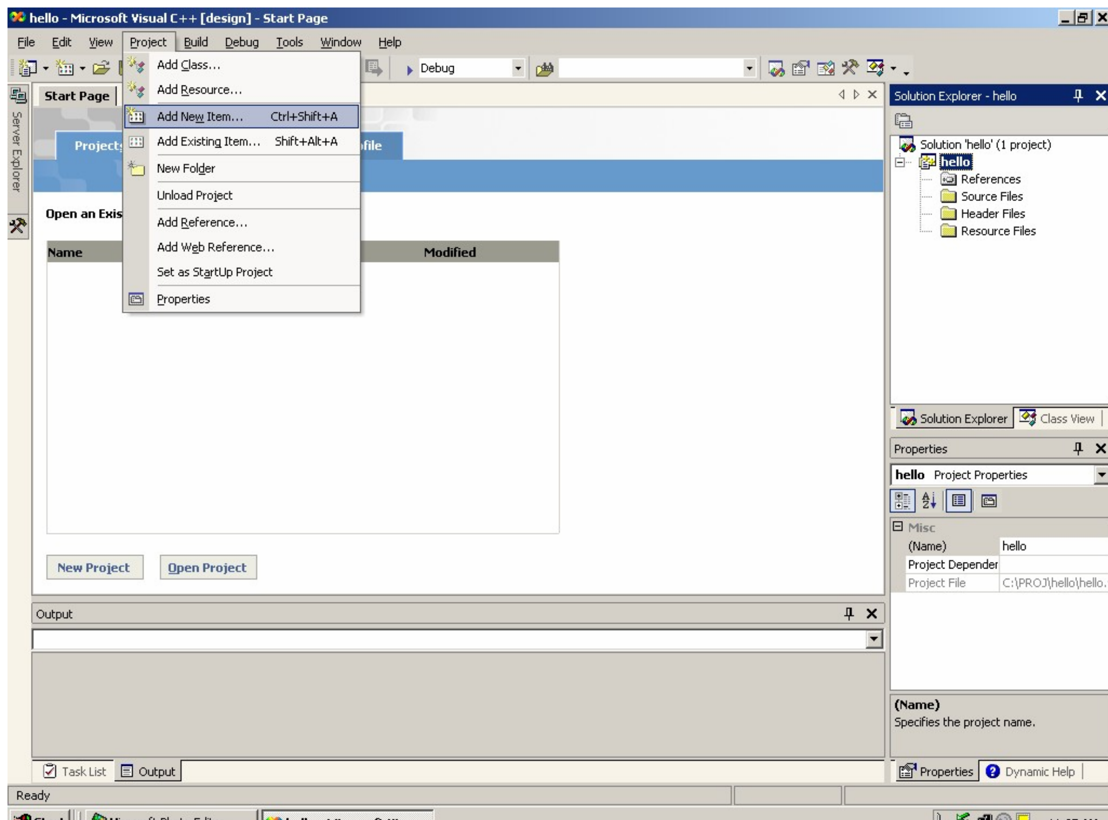


Click the Finish button to create the empty project that looks like this:

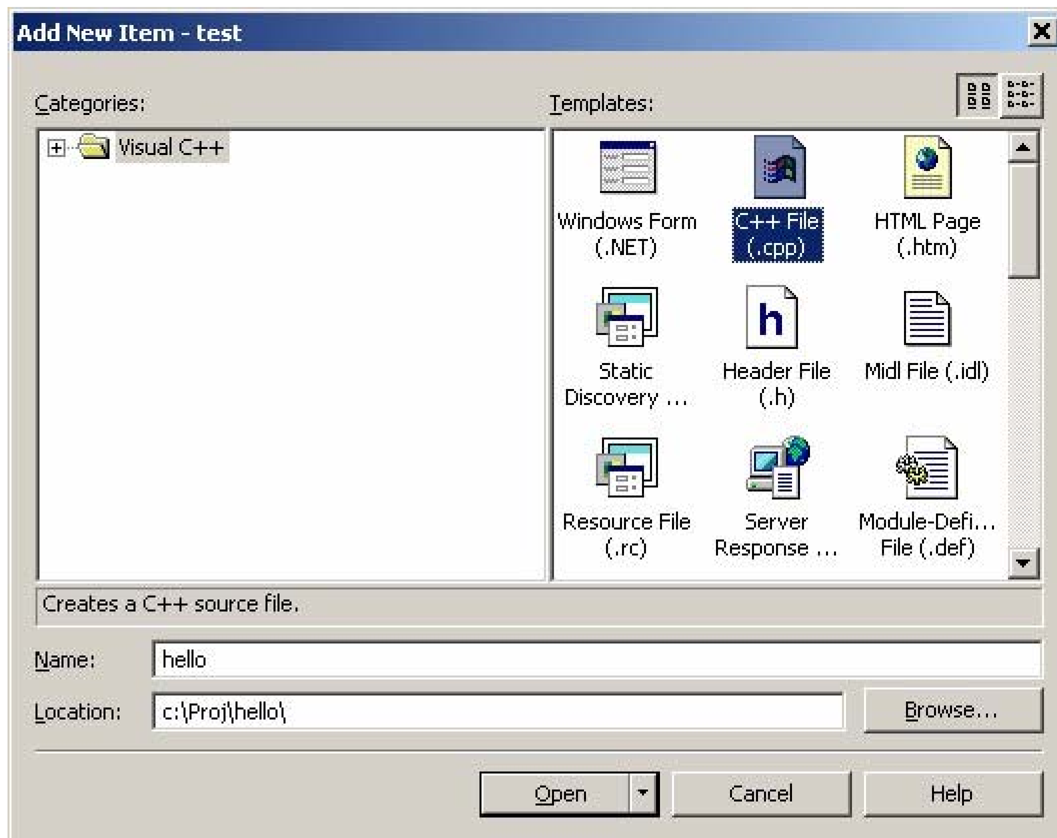


Typing your Program Code

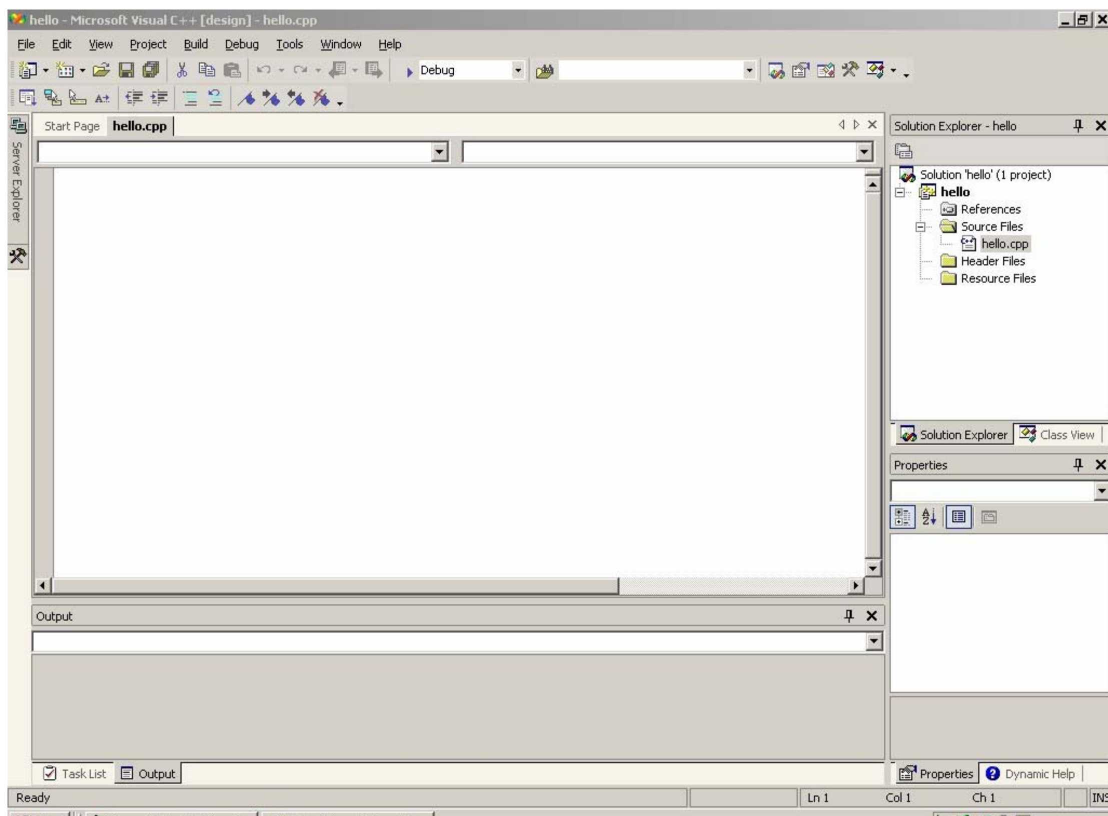
Click the menu item Project | Add New Item... or press CTRL + SHIFT + A or click the Add New Item button.



Select C++ File (.cpp) from the list on the right and type a relevant name in the text box labelled Name:



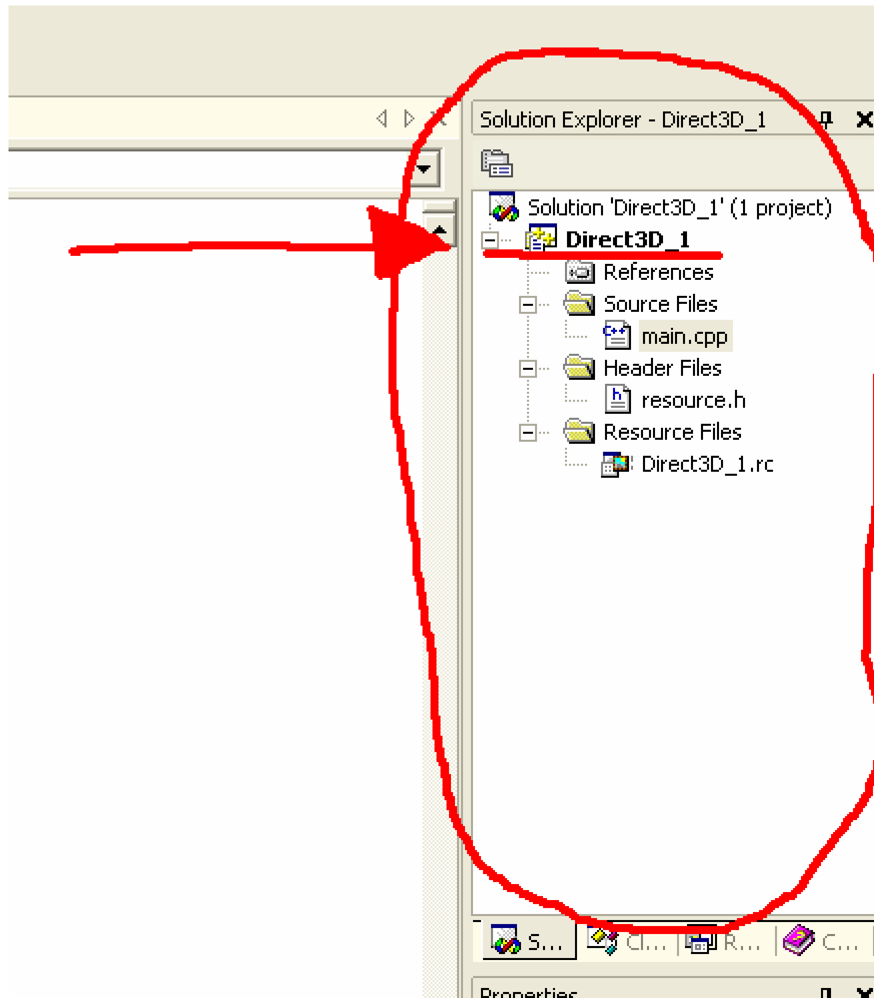
Click the Open button to enter the edit mode. The file will appear in the Source file list in the Solution Explorer pane.
You should see the following screen:



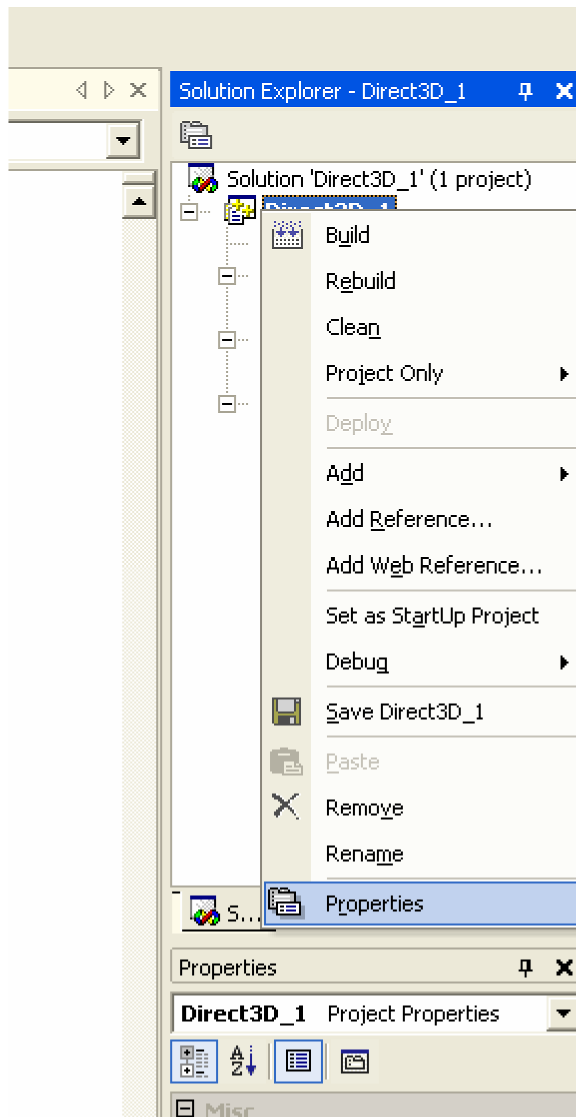
Now enter or copy across the main.cpp code from Direct3D_1.

Try compiling it. You should generate a link error. If you first generate a link error that indicates <d3d9.h> being at fault, follow the procedures set out in the installing the DirectX SDK file (found under week 1).
Despite the correct associations between the DirectX SDK and the Comiler you should still generate a link error – this should either be focused around a unknown function call or something similar.
This error can be solved by correctly setting the properties of your project. To do this follow the steps set out below.

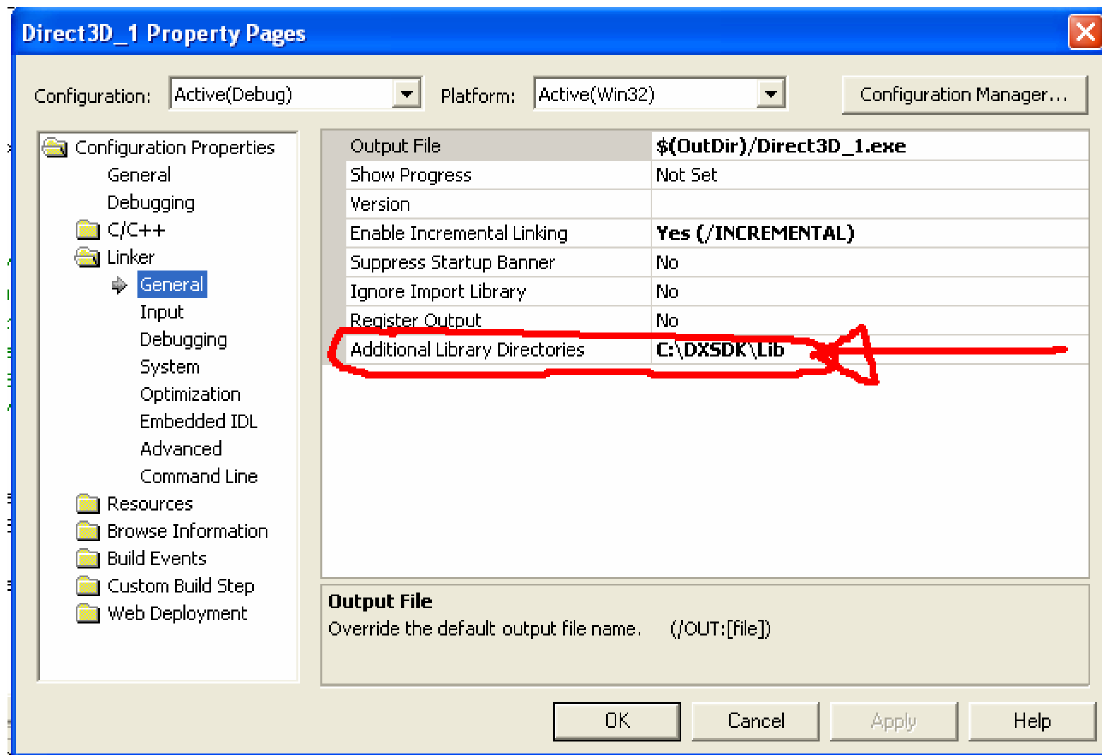
Right click your file project name in the solution explorer.



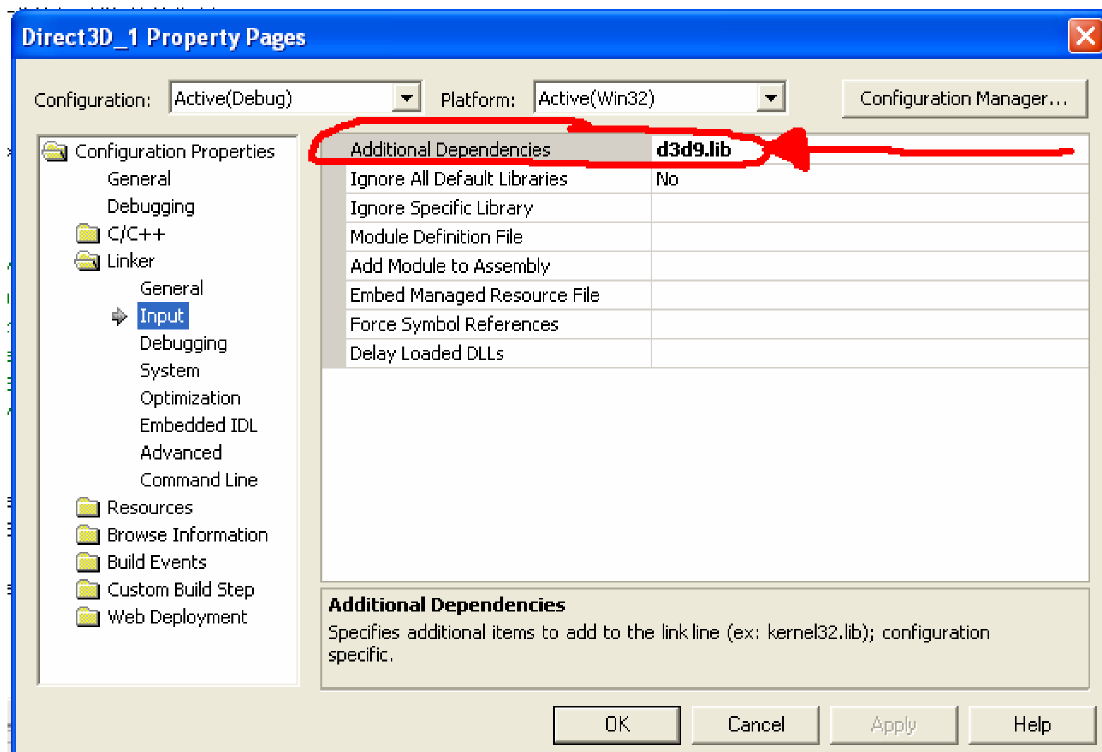
And then click on the properties item.



After you have done this you are faced with the screen as seen below. Click on Linker | General and add the path to your DirectXSDK Library folder. You can always sue my computer to browse to it and then copy the path directory, before pasting it in the relevant part.



Next, do the same again but adding the line d3d9.lib to the Linker | Input | Additional Dependencies part



Your code should now compile and work wonderfully.
You will need to set these dependencies for all your DirectX projects.

The Additional Dependencies may vary depending on what you are doing, for instance we may need to `#include <dxerr9.h>` in this case we add `dxerr9.h` to the list of Additional Dependencies.