The Display and Display Modes

To understand DirectX you need to know about display modes.

Display modes are mainly comprised of screen resolution and colour depth
- e.g. 1024x768 with 32-bit colour

There are dozens of display that are accessible if your graphic hardware supports them.

Most supported display modes are archaic (palletized, monochrome) and no use to a Direct3D programmer.

The most commonly used display modes on modern computers are a mixture of resolution and bit colour.
Resolution generally varies from 640x480 to about 1600x1200, and are either 16, 24 or 32-bit colour.

Resolution
Resolution represents the number of pixels, first expressed vertically and then expressed horizontally.
E.g. 800x600 is 800 pixels vertically and 600 pixels horizontally

Colour Bits
These determine how much video card memory each pixel on the screen uses.
Don't forget that these are in bits and 8-bits are a byte.

Most colours on the screen are determined by having the pixels display these colour variations.
A system generally uses the RGB colour mode where by the colours are created by combining Red, Green and Blue.

Colour values vary between 0 and 255. Combinations of these values make the colour. Examples are: 0, 0, 0 is black - 255, 255, 255 is white – 255, 0, 0 is Red – 0, 255, 0 is Green and 0, 0, 255 is Blue

The display modes can vary, also including a value for Alpha. Alpha specifies a colours transparency.
Generally though it is 32-bit colour modes that fit the inclusion of an Alpha value best, setting aside a byte for each value (ARGB)

Most modern machines also run 32-bit so it is this mode that our programs will use.

Setting the Pixel Format
To set the pixel format we use constants defined by the DirectX SDK.

To see the constants look at this website;
http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/directx9_c/directx/graphics/reference/d3d/enums/d3dformat.asp

The best types to use are:

```
D3DFMT_A8B8G8R8     /* 32-bit ARGB pixel format with
alpha, using 8 bits per channel */
D3DFMT_X8B8G8R8     /* 32-bit RGB pixel format, where 8
bits are reserved for each colour */
```

As these are supported by the majority of hardware as they are common pixel formats. Some of the other are not so common and before using them you should check that they are supported.

Checking for Mode Availability

When your Direct3D application first runs it is considered good programming practice to check the availability of the display mode your application needs.

To perform this check is simple, you just call Direct3D's `CheckDeviceType`. The SDK defines is like this:

```
HRESULT CheckDeviceType(

    UINT Adapter,
    D3DDEVTYPE DeviceType,
    D3DFORMAT DisplayFormat,
    D3DFORMAT BackBufferFormat,
    BOOL Windowed
);
```

Parameters

`Adapter`

[in] Ordinal number denoting the display adapter to enumerate. `D3DADAPTER_DEFAULT` is always the primary display adapter. This method returns `D3DERR_INVALIDCALL` when this value equals or exceeds the number of display adapters in the system.

`DeviceType`

[in] Member of the `D3DDEVTYPE` enumerated type, indicating the device type to check.

`DisplayFormat`

[in] Member of the `D3DFORMAT` enumerated type, indicating the format of the adapter display mode for which the device type is to be checked. For example, some devices will operate only in 16-bits-per-pixel modes.

`BackBufferFormat`

[in] Back buffer format. For more information about formats, see D3DFORMAT. This value must be one of the render target formats. You can use `IDirect3DDevice9::GetDisplayMode` to obtain the current format.

For windowed applications, the back buffer format does not need to match the display mode format 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 because devices typically do not operate in 8 bits per pixel modes.

Full-screen applications cannot do colour conversion.

`D3DFMT_UNKNOWN` is allowed for windowed mode.

Windowed
[in] Value indicating whether the device type will be used in full-screen or windowed mode. If set to TRUE, the query is performed for windowed applications; otherwise, this value should be set FALSE.

Return Value
If the device can be used on this adapter, D3D_OK is returned.

D3DERR_INVALIDCALL is returned if Adapter equals or exceeds the number of display adapters in the system. D3DERR_INVALIDCALL is also returned if IDirect3D9::CheckDeviceType specified a device that does not exist.

D3DERR_NOTAVAILABLE is returned if the requested back buffer format is not supported, or if hardware acceleration is not available for the specified formats.

Our Code

We will generally use the following as our parameters (in parameter order):

```
D3DADAPTER_DEFAULT
D3DDEVTYPE_HAL      // see Direct3D_Intro Tutorial
D3DFMT_A8B8G8R8     /* 32-bit ARGB pixel format with
alpha, using 8 bits per channel */
D3DFMT_X8B8G8R8     /* 32-bit RGB pixel format, where 8
bits are reserved for each colour */
FALSE               /* Here we will use TRUE or False as
needed */
```

The 3rd and 4th parameters are taken from a table that defines the pixel format; see the link above for a wide range of pixel formats.
As specified we will use 32-bit colour.

To include this in your code you do the following:

```
HRESULT hResult = g_pDirect3D -> CheckDeviceType(D3DADAPTER_DEFAULT,
D3DDEVTYPE_HAL, D3DFMT_A8B8G8R8, D3DFMT_X8B8G8R8, FALSE);

if(hResult != D3D_OK)
{
      return E_FAIL;
}
```

Windowed vs. Full-screen Applications

Checking for a specific pixel format is something done usually when dealing with full screen Direct3D applications. This is because with full screen you can use any display mode you want, as long as the hardware supports it.

With windowed applications you have to share the display any other running applications and are therefore stuck to whatever mode the user has specified.

This means that your application must either adapt to the display mode or ask the user to change it. It is because of this most games run full screen.

However with quick and easy type games, image a Direct3D version of free cell for instance, having full screen is pointless and stops the user from switching between the game and current applications.

Checking the Current Display Mode
With windowed Direct3D applications it therefore makes more sense to check the current mode than to check for a specific one.

To check the current display mode you all another Direct3D object method just like for `CheckDeviceType` the call this time is to `GetAdapterDisplayMode`.

Syntax

```
HRESULT GetAdapterDisplayMode(

    UINT Adapter,
    D3DDISPLAYMODE *pMode
);
```

Parameters
`Adapter`
[in] Ordinal number that denotes the display adapter to query. `D3DADAPTER_DEFAULT` is always the primary display adapter.
`pMode`
[in, out] Pointer to a `D3DDISPLAYMODE` structure, to be filled with information describing the current adapter's mode.

Return Value
If the method succeeds, the return value is `D3D_OK`.

If Adapter is out of range or `pMode` is invalid, this method returns `D3DERR_INVALIDCALL`.

Remarks
`IDirect3D9::GetAdapterDisplayMode` will not return the correct format when the display is in an extended format, such as 2:10:10:10. Instead, it returns the format X8R8G8B8.

Display mode is defined as below:

```
typedef struct _D3DDISPLAYMODE {
    UINT Width;
    UINT Height;
    UINT RefreshRate;
    D3DFORMAT Format;
```

```
} D3DDISPLAYMODE;
```

`Width`
Screen width, in pixels.
`Height`
Screen height, in pixels.
`RefreshRate`
Refresh rate. The value of 0 indicates an adapter default.
`Format`
Member of the `D3DFORMAT` enumerated type, describing the surface format of the display mode.

You use the `GetAdapterDisplayMode` in the following way:

```
D3DDISPLAYMODE d3ddisplaymode;
HRESULT hResult = g_pDirect3D ->
   GetAdapterDisplayMode(D3DADAPTER_DEFAULT, &d3ddisplaymode)

if(hResult != D3D_OK)
{
      return E_FAIL;
}
```

Once you have the display mode information you can use it to create the device for your application:

```
D3DPRESENT_PARAMETERS D3DPresentParams;
ZeroMemory(&D3DPresentParams, sizeof(D3DPRESENT_PARAMETERS));
D3DPresentParams.Windowed = TRUE;
D3DPresentParams.BackBufferCount = d3ddisplaymode.format;
D3DPresentParams.SwapEffect = D3DSWAPEFFECT_DISCARD;
D3DPresentParams.hDeviceWindow = g_hWnd;

hResult = g_pDirect3D->CreateDevice(D3DADAPTER_DEFAULT,
   D3DDEVTYPE_HAL, g_hWnd, 3DCREATE_SOFTWARE_VERTEXPROCESSING,
   &D3DPresentParams, &g_pDirect3DDevice);

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

Ok, that's how you define a windowed application. You can do this if you wish however full screen generally looks more professional as a game, unless its cards, so we will generally code for full screen. The choice with your own applications is always yours though!
The code to what we did above for a full screen application is below:

```
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;
```

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

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

Drawing to the Display

Here we will just go over a couple of Direct3D methods that enable you to control drawing to the screen. Pretty handy methods really!

The `Clear` Method:

```
HRESULT Clear(

    DWORD Count,
    const D3DRECT *pRects,
    DWORD Flags,
    D3DCOLOR Color,
    float Z,
    DWORD Stencil
);
```

Parameters
`Count`
[in] Number of rectangles in the array at `pRects`. Must be set to 0 if `pRects` is NULL. May not be 0 if `pRects` is a valid pointer.
`pRects`
[in] Pointer to an array of `D3DRECT` structures that describe the rectangles to clear. Set a rectangle to the dimensions of the rendering target to clear the entire surface. Each rectangle uses screen coordinates that correspond to points on the render target. Coordinates are clipped to the bounds of the viewport rectangle. To indicate that the entire viewport rectangle is to be cleared, set this parameter to NULL and Count to 0.
`Flags`
[in] Combination of one or more `D3DCLEAR` flags that specify the surface(s) that will be cleared.
`Color`
[in] Clear a render target to this ARGB color.
`Z`
[in] Clear the depth buffer to this new z value which ranges from 0 to 1. See remarks.
`Stencil`
[in] Clear the stencil buffer to this new value which ranges from 0 to 2n–1 (n is the bit depth of the stencil buffer). See remarks.

Return Value
If the method succeeds, the return value is `D3D_OK`.

If the method fails, the return value can be:
`D3DERR_INVALIDCALL`
The method call is invalid. For example, a method's parameter may have an invalid value.

Remarks

Use this method to clear a surface including: a render target, all render targets in an MRT, a stencil buffer, or a depth buffer. Flags determines how many surfaces are cleared. Use pRects to clear a subset of a surface defined by an array of rectangles.

`IDirect3DDevice9::Clear` will fail if you:
Try to clear either the depth buffer or the stencil buffer of a render target that does not have an attached depth buffer.
Try to clear the stencil buffer when the depth buffer does not contain stencil data.

Although this looks quite complicated we don't have to concern ourselves, or understand, most of it. This is because most of our calls to Clear will look like this:

```
Clear(0, 0, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,0,255), 0, 0);
```

The 3rd argument `D3DCLEAR_TARGET` tells Direct3D to clear the surface to the colour specified in the 4th argument.
The 4th argument is a `D3DCOLOR` value, covered above. You can create this using the XRGB macro. As seen this macro takes the Red, Green and Blue colour intensities as arguments.

The `Present` Method:

Once the surface has been cleared you need to display it.
This uses the Direct3D object method `Present`.

```
HRESULT Present(

    CONST RECT *pSourceRect,
    CONST RECT *pDestRect,
    HWND hDestWindowOverride,
    CONST RGNDATA *pDirtyRegion
);
```

Parameters
`pSourceRect`
[in] Pointer to a value that must be NULL unless the swap chain was created with `D3DSWAPEFFECT_COPY`. pSourceRect is a pointer to a RECT structure containing the source rectangle. If NULL, the entire source surface is presented. If the rectangle exceeds the source surface, the rectangle is clipped to the source surface.
`pDestRect`
[in] Pointer to a value that must be NULL unless the swap chain was created with `D3DSWAPEFFECT_COPY`. `pDestRect` is a pointer to a `RECT` structure containing the destination rectangle, in window client coordinates. If NULL, the entire client area is filled. If the rectangle exceeds the destination client area, the rectangle is clipped to the destination client area.
`hDestWindowOverride`
[in] Pointer to a destination window whose client area is taken as the target for this presentation. If this value is NULL, then the `hWndDeviceWindow` member of `D3DPRESENT_PARAMETERS` is taken.
`pDirtyRegion`
[in] Value must be NULL unless the swap chain was created with `D3DSWAPEFFECT_COPY`. For more information about swap chains, see Flipping Surfaces and `D3DSWAPEFFECT`.

If this value is non-NULL, the contained region is expressed in back buffer coordinates. The rectangles within the region are the minimal set of pixels that need to be updated. This method takes these rectangles into account when optimizing the presentation by copying only the pixels within the region, or some suitably expanded set of rectangles. This is an aid to optimization only, and the application should not rely on the region being copied exactly. The implementation can choose to copy the whole source rectangle.

Return Value
If the method succeeds, the return value is D3D_OK.

If the method fails, the return value can be one of the following values.
`D3DERR_DEVICELOST`
The device has been lost but cannot be reset at this time. Therefore, rendering is not possible.
`D3DERR_DRIVERINTERNALERROR`

Internal driver error. Applications should generally shut down when receiving this error. For more information, see the MSDN for Driver Internal Errors.
`D3DERR_INVALIDCALL` The method call is invalid. For example, a method's parameter may have an invalid value.

Remarks
If necessary, a stretch operation is applied to transfer the pixels within the source rectangle to the destination rectangle in the client area of the target window.

Present will fail, returning `D3DERR_INVALIDCALL`, if called between BeginScene and EndScene pairs unless the render target is not the current render target (such as the back buffer you get from creating an additional swap chain). This is a new behaviour for Microsoft DirectX 9.0.

You generally use `Present` in the following way:

```
Present( NULL, NULL, NULL, NULL );
```

This basically displays the entire surface overwriting what used to be on the screen.

The End

This is the final part of the Direct3D introduction. You should now be able to code a basic Direct3D app. That displays a blue screen (or another colour if you change the values).
Not particularly exciting yet, but these are the basic building blocks for more in-depth applications.

To see what this code looks like when added together see:
www.activehelix.com/prog3.html -> InitialisingD3D.cpp