Surfaces

With most DirectX programs you will be working something called *surfaces*.

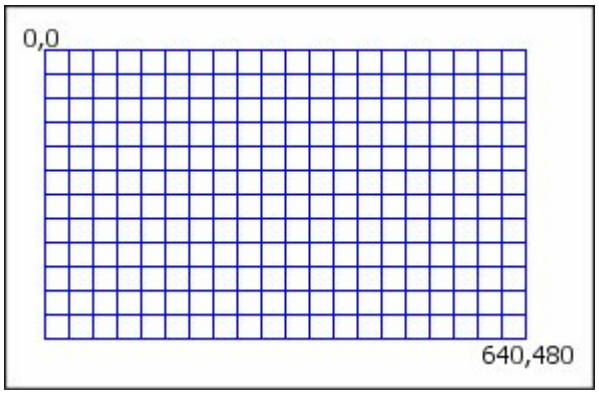
A surface is nothing more than a area of memory in which you can store graphical information. The images and pixels that you see on screen (or displayed in a window) are stored in a surface.
In D3D, all drawing is done to a surface. A surface is simply a section of memory set aside for the purpose of drawing in, which may be a back buffer, a bitmap for texturemapping or a sprite.

Incidentally, when talking about graphics programming the action of copying graphics data from one area in memory to another area is called "blitting". You'll hear this term a lot.

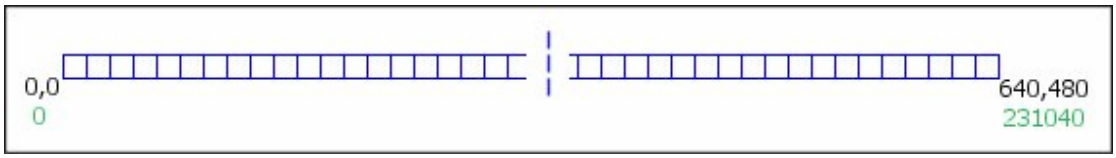
How Surfaces are Represented

A surface (usually screen or window) can be viewed as a matrix of pixels that DirectX uses primarily to store 2D image data.

An example of a surface in resolution 640x480



The actual data storage is in a linear array. This may sound conceptually weird, how do you represent a 2D surface in 1D?
Well, it's much simpler than you may think. Because we're working directly with memory, the entire surface is actually in one long sequence of bytes.
So, conceptually you imagine the above image to be the layout of a surface in memory. In actuality the memory layout is like this:



As you can see, it's just one long strip of bytes. The black numbers represent the x,y conventional graphics locations on screen, whilst the green numbers represent the byte locations.

On screen position (also called Display Offset) = x + (Screen Width* y)
The width and height are measured in pixels. The pitch is measured in bytes.

The pitch maybe wider than the width, depending on the underlying hardware implementation, so we cannot assume:

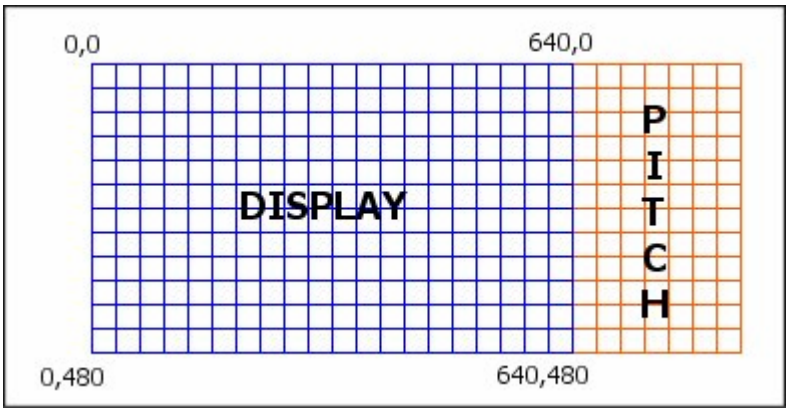
Pitch = Width · sizeof(pixelFormat)

Therefore, if you wanted to set the colour of a pixel at 640, 480 on screen, you'd set the colour of pData[231040], the formula being:

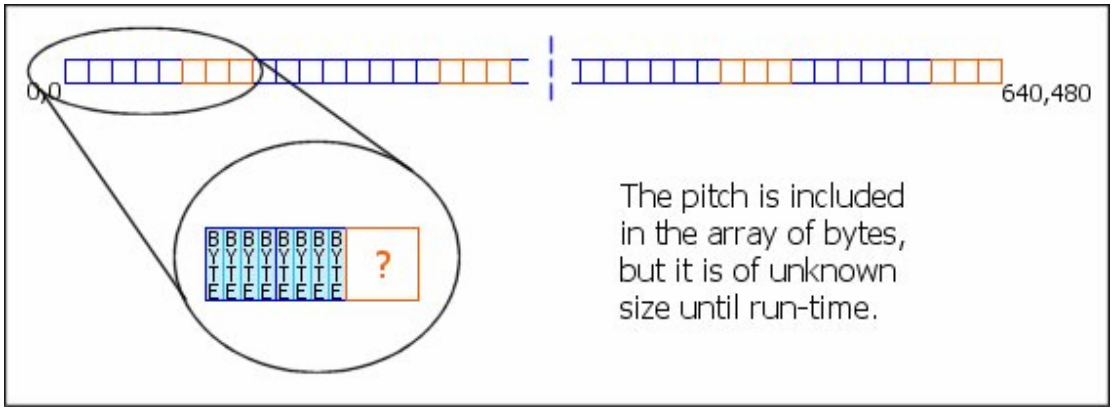
On screen position (also called Display Offset) = (x + (Screen Width y)) * (Number of bytes per pixel)*

Hopefully you will have noticed why we use a 32bit DWORD for a pointer. If we're using a 32bit display, we are therefore using 4 bytes per pixel (8 bits to 1 byte) instead of just the one byte that the above images assume. Because a DWORD is also 32bits, we don't need to make any extra calculations for the offset. For example, let's say we used a BYTE array, in which each element is 1 byte large (unsurprisingly!). If we used an 8 bit display, the calculation would still be the same, as the size of one element of a BYTE array is the same size as one pixel in display memory. But what if we used a 32bit display with an 8bit (2 byte) BYTE pointer? We would have to modify the calculation as follows:

On screen position (also called Display Offset) = (x + (Screen Width y)) * (Number of bytes per pixel)*



But, as we already know, the data is a 1D array of bytes. So, in reality, a 32bit back buffer in memory looks like this:



Because the pitch is integrated into the byte array, the previous equasion for calculating the display offset won't work - we need to take the pitch into account, as well as the screen width. Handily, the D3DLOCKED_RECT.Pitch member is an integer number that represents the entire width of one "line" of the back buffer. So, all we need to do is directly substitute the Screen Width in our last equasion for the Pitch, and we end up with the complete and final formula for indexing into the back buffer:

*On screen position (also called Display Offset) = (x + (Screen Pitch * y) * (Number of bytes per pixel)*

Whilst the type of D3DLOCKED_RECT.Pitch is an int, it represents the entire width in bytes. So, as we are using a 32bit pointer, we must divide the pitch by 4 to turn it into bits (4 * 8 bits per byte = 32bits).

IDirectSurface3D9

Surfaces are generally described using the `IDirectSurface3D9` interface. This interface has several methods for reading and writing directly to a surface as well as for retrieving information about the surface.

`IDirect3DSurface9` Members:

`FreePrivateData`
Frees the specified private data associated with this resource.
`GetContainer`
Provides access to the parent cube texture or texture (mipmap) object, if this surface is a child level of a cube texture or a mipmap
This method can also provide access to the parent swap chain if the surface is a back-buffer child
`GetDC`
Retrieves a device context
`GetDesc`
Retrieves a description of the surface
`GetDevice`

Retrieves the device associated with a resource.
`GetPriority`
Retrieves the priority for this resource
`GetPrivateData`
Copies the private data associated with the resource to a provided buffer.
`GetType`
Returns the type of the resource
`LockRect`
Locks a rectangle on a surface
`PreLoad`
Preloads a managed resource
`ReleaseDC`
Release a device context handle
`SetPriority`
Assigns the resource-management priority for this resource
`SetPrivateData`
Associates data with the resource that is intended for use by the application, not by Microsoft Direct3D
Data is passed by value, and multiple sets of data can be associated with a single resource
`UnlockRect`
Unlocks a rectangle on a surface

The most important methods of `IDirectSurface3D9` are:

`LockRect` – This method enables us to obtain a pointer to the surface memory. Then with some pointer arithmetic we can read and write each pixel to the surface.

`UnlockRect` – After you have called and are finished accessing the surface's memory you must unlock the surface by calling this method.

`GetDesc` – This method retrieves a description of the surface by filling out a `D3DSURFACE_DESC` structure.

```
HRESULT LockRect(

    D3DLOCKED_RECT *pLockedRect,
    const RECT *pRect,
    DWORD Flags
);
```

Parameters
`pLockedRect`
[out] Pointer to a `D3DLOCKED_RECT` structure that describes the locked region.
`pRect`
[in] Pointer to a rectangle to lock. Specified by a pointer to a RECT structure. Specifying NULL for this parameter expands the dirty region to cover the entire surface.
`Flags`

[in] Combination of zero or more locking flags that describe the type of lock to perform. For this method, the valid flags are:
`D3DLOCK_DISCARD`
`D3DLOCK_DONOTWAIT`
`D3DLOCK_NO_DIRTY_UPDATE`
`D3DLOCK_NOSYSLOCK`
`D3DLOCK_READONLY`

You may not specify a subrect when using `D3DLOCK_DISCARD`. For a description of the flags, see `D3DLOCK`.

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

If the method fails, the return value can be `D3DERR_INVALIDCALL` or `D3DERR_WASSTILLDRAWING`.

There are four primary types of surface used in a Direct3D application production, they are:

- Primary surface
- Back-buffer surface
- Image surfaces
- Texture surfaces

Primary surface

The primary surface is often called the front buffer. It is what you are looking at on the screen when you run a Direct3D application.
This surface may contain either graphical data for an entire screen, in the case of full screen, or for the applications window.
The primary surface is the only type of surface that every Direct3D application must create and maintain.

You don't have to worry about explicitly creating a primary surface, as it must always exist just creating your Direct3D device automatically creates the primary surface.

Back-buffer surface

The back buffer is an area of memory where we draw a display out of view of the user. After you have drawn the display you call the `Present()` method to make the back buffer the new primary surface. Rendering this way helps prevent flicker and tearing.
We covered the concept of a back buffer briefly in Programming 2. In this we looked at making our own back buffer. Direct3D enables you to automatically create the back buffer at the time you create the applications Direct3D interface.

We looked at D3DPRESENT_PARAMETERS in lesson 1.
To recap follow this link:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/reference/d3d/structures/d3dpresent_parameters.asp
This presents the MSDN break down of the D3DPRESENT_PARAMETERS structure.

So far we have been ignoring the first four parameters. They deal with back buffers.
We are now going to look at these four parameters in more detail:

```
    UINT BackBufferWidth;
    UINT BackBufferHeight;
    D3DFORMAT BackBufferFormat;
    UINT BackBufferCount
```

`UINT BackBufferWidth`
This is the width of the back buffer; it should be set in most circumstances to the same width as the primary buffer.

`UINT BackBufferHeight;`
This is the height of the back buffer; it should be set in most circumstances to the same height as the primary buffer.

`D3DFORMAT BackBufferFormat;`
The pixel format of the back buffer; again this is usually the same as that of the primary buffer.

`UINT BackBufferCount`
The number of back buffers we wish to create, this is usually one.

`BackBufferCount`? Why would we want more than 1? Good question. It actually helps to speed up and smooth the rasterisation process (rasterising is just another word for rendering, or drawing to screen)

Image surfaces
Direct3D applications need a place to store images required by the program. You may, for instance, have a bitmap that is to represent a background scene. Before you can use such an image it has to be loaded into memory. Specifically it needs to be placed in an *image surface*.

You need to explicitly create an image surface in your program. To do this you call the Direct3D device objects `CreateOffscreenPlainSurface()`method.

```
HRESULT CreateOffscreenPlainSurface(

    UINT Width,
    UINT Height,
    D3DFORMAT Format,
    DWORD Pool,
    IDirect3DSurface9** ppSurface,
    HANDLE* pSharedHandle
);
```

Parameters
`Width`

[in] Width of the surface.
`Height`
[in] Height of the surface.
`Format`
[in] Format of the surface. See D3DFORMAT.
`Pool`
[in] Surface pool type. See D3DPOOL.
`ppSurface`
[out, retval] Pointer to the IDirect3DSurface9 interface created.
`pSharedHandle`
[in] Reserved. Set this parameter to NULL.

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

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

Remarks
D3DPOOL_SCRATCH will return a surface that has identical characteristics to a surface created by the Microsoft DirectX 8.x method CreateImageSurface.

D3DPOOL_DEFAULT is the appropriate pool for use with the IDirect3DDevice9::StretchRect and IDirect3DDevice9::ColorFill.

D3DPOOL_MANAGED is not allowed when creating an offscreen plain surface. For more information about memory pools, see D3DPOOL.

Off-screen plain surfaces are always lockable, regardless of their pool types.

This code give you an example of how you may implement the `CreateOffscreenPlainSurface` method:

```
rslt=pDevice->CreateOffscreenPlainSurface(Bitmap.bmWidth,
   Bitmap.bmHeight, m_Format, D3DPOOL_SYSTEMMEM,
   &m_pSurface, NULL);

if(FAILED(rslt))
{
    ::OutputDebugString("Unable to create a new surface for the
following bitmap:\n");
    ::OutputDebugString(pathname);
    ::OutputDebugString("\n");

    return E_FAIL;
}
```

To use this code correctly we will need to cover bitmaps and images, something for another time. Consider this a taster. If you can already tell what most of it does, so much the better.

Texture surfaces
These will not be covered now, but basically a texture surface is just a special surface for holding a texture.

Textures are images that are often used to provide greater detail to objects in a 3D world. Textures are also useful for creating images that have transparent portions.

Multisampling
This is a technique used to smooth out blocky looking images that can result when representing images as a matrix of pixels.
One of the common uses for multisampling is for full screen anti-aliasing.

The `D3DMULTISAMPLE_TYPE` is an enumerated type that consists of values that allow us specify the level of multisampling for a surface.

If wish to look at multisampling then look at the MSDN as there is also an associated DWORD that describes quality level.

```
HRESULT CheckDeviceMultiSampleType(

    UINT Adapter,
    D3DDEVTYPE DeviceType,
    D3DFORMAT SurfaceFormat,
    BOOL Windowed,
    D3DMULTISAMPLE_TYPE MultiSampleType,
    DWORD* pQualityLevels
);
```

Parameters
`Adapter`
[in] Ordinal number denoting the display adapter to query. D3DADAPTER_DEFAULT is always the primary display adapter. This method returns FALSE when this value equals or exceeds the number of display adapters in the system. See Remarks.
`DeviceType`
[in] Member of the D3DDEVTYPE enumerated type, identifying the device type.
`SurfaceFormat`
[in] Member of the D3DFORMAT enumerated type that specifies the format of the surface to be multisampled. For more information, see Remarks.
`Windowed`
[in] BOOL value. Specify TRUE to inquire about windowed multisampling, and specify FALSE to inquire about full-screen multisampling.
`MultiSampleType`
[in] Member of the D3DMULTISAMPLE_TYPE enumerated type, identifying the multisampling technique to test.
`pQualityLevels`
[out] The number of quality stops available for a given multisample type. This can be NULL if it is not necessary to return the values.

Return Value
If the device can perform the specified multisampling method, this method returns D3D_OK.

D3DERR_INVALIDCALL is returned if the Adapter or MultiSampleType parameters are invalid. This method returns D3DERR_NOTAVAILABLE if the queried multisampling technique is not supported by this device. D3DERR_INVALIDDEVICE is returned if DeviceType does not apply to this adapter.

If you wish to use multisampling, then also don't forget to check that the device supports it. To do this used `IDirect3D9::CheckDeviceMultiSampleType` method to verify that the

hardware can support the multisampling type and quality level you wish to use.

If you are wondering what anti-aliasing is here is some jargon and definitions:
Antialiasing is a graphical trick to make an object's corners & edges look smooth instead of jagged
Full Scene AntiAliasing (FSAA) is applying Antialiasing to the full screen.

Here is some skeleton code for checking and setting MultiSampling

```
// Type of multisampling
if(d3dcurrentsettings.m_bMultiSampling)
{
    if( SUCCEEDED(pD3D ->
CheckDeviceMultiSampleType(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL,
      d3dpp.BackBufferFormat, d3dcurrentsettings.m_bWindowed,
D3DMULTISAMPLE_2_SAMPLES, NULL)))
    {
        d3dpp.MultiSampleType = D3DMULTISAMPLE_2_SAMPLES;
    }
    else
    {
        d3dpp.MultiSampleType=D3DMULTISAMPLE_NONE;
      }
}
else
{
    d3dpp.MultiSampleType=D3DMULTISAMPLE_NONE;
}
```

Memory Pools

Surfaces and other Direct3D resources can be placed in a variety of memory pools. The memory pool is specified by one of the members of the D3DPOOL enumerated type.
The possible memory pools, as defined by the SDK, are:

```
typedef enum _D3DPOOL {
   D3DPOOL_DEFAULT = 0,
   D3DPOOL_MANAGED = 1,
   D3DPOOL_SYSTEMMEM = 2,
   D3DPOOL_SCRATCH = 3,
   D3DPOOL_FORCE_DWORD = 0x7fffffff
} D3DPOOL;
```

Constants

`D3DPOOL_DEFAULT`

Resources are placed in the memory pool most appropriate for the set of usages requested for the given resource. This is usually video memory, including both local video memory and accelerated graphics port (AGP) memory. The D3DPOOL_DEFAULT pool is separate from D3DPOOL_MANAGED and D3DPOOL_SYTEMMEM, and it specifies that the resource is placed in the preferred memory for device access. Note that D3DPOOL_DEFAULT never indicates that either D3DPOOL_MANAGED or D3DPOOL_SYSTEMMEM should be chosen as the memory pool type for this resource. Textures placed in the D3DPOOL_DEFAULT pool cannot be locked unless they are dynamic textures or they are private, four-character code

(FOURCC), driver formats. To access unlockable textures, you must use functions such as IDirect3DDevice9::UpdateSurface, IDirect3DDevice9::UpdateTexture, IDirect3DDevice9::GetFrontBufferData, and IDirect3DDevice9::GetRenderTargetData. D3DPOOL_MANAGED is probably a better choice than D3DPOOL_DEFAULT for most applications. Note that some textures created in driver-proprietary pixel formats, unknown to the Microsoft Direct3D runtime, can be locked. Also note that—unlike textures—swap chain back buffers, render targets, vertex buffers, and index buffers can be locked. When a device is lost, resources created using D3DPOOL_DEFAULT must be released before calling IDirect3DDevice9::Reset. For more information, see Lost Devices.

When creating resources with D3DPOOL_DEFAULT, if video card memory is already committed, managed resources will be evicted to free enough memory to satisfy the request.

`D3DPOOL_MANAGED`
Resources are copied automatically to device-accessible memory as needed. Managed resources are backed by system memory and do not need to be re-created when a device is lost. See Managing Resources for more information. Managed resources can be locked. Only the system-memory copy is directly modified. Direct3D copies your changes to driver-accessible memory as needed.

`D3DPOOL_SYSTEMMEM`
Resources are placed in memory that is not typically accessible by the Direct3D device. This memory allocation consumes system RAM but does not reduce pageable RAM. These resources do not need to be re-created when a device is lost. Resources in this pool can be locked and can be used as the source for a IDirect3DDevice9::UpdateSurface or IDirect3DDevice9::UpdateTexture operation to a memory resource created with D3DPOOL_DEFAULT.

`D3DPOOL_SCRATCH`
Resources are placed in system RAM and do not need to be re-created when a device is lost. These resources are not bound by device size or format restrictions. Because of this, these resources cannot be accessed by the Direct3D device nor set as textures or render targets. However, these resources can always be created, locked, and copied.

`D3DPOOL_FORCE_DWORD`
Forces this enumeration to compile to 32 bits in size. This value is not used.

Remarks
All pool types are valid with all resources. This includes: vertex buffers, index buffers, textures, and surfaces.

Example Code

There is a code example of writing and basic surface manipulation at www.activehelix.com/prog3.html -> D3DSurface_Intro.cpp