**INITIALIZATION DIRECTX**

Phần này chúng ta sẽ coi thử các bước cần phải làm khi bắt đầu sử dụng DirectX

* Tạo con trỏ tới interface IDIRECT3D9, điều ngày cũng có nghĩa là ta sẽ tạo **LPDIRECT3D9,** con trỏ này sẽ đi tìm thông tin thiết bị đồ họa để có thể tạo ra Device nơi chúng ta sẽ sử dụng để thể hiện đồ họa game.
* Kiểm tra xem thiết bị đồ họa có hỗ trợ Vertex Processing ko (điều này cần thiết vì khi tạo Device ta phải lựa chọn xử lý Vertex bằng CPU hay GPU)
* Tạo D3DPRESENT\_PARAMETTER, structure này chứa một vài tùy chọn để điều chỉnh thể hiện của Device
* Tạo ra Device, tức là LPDIRECT3DDEVICE9, đây sẽ là nơi điều khiển thiết bị đồ họa thể hiện ứng dụng game.

**CREATE DEVICE**

Chúng ta thử xem chi tiết các bước để thực hiện việc tạo Device. Đầu tiên là một vài bước cần phải có

* Chúng ta phải cài DirectX SDK vào máy.
* Chúng ta phải add thư viện d3d9.lib vào project để include nó

Chúng ta bắt đầu làm việc với DirectX, đối tượng đầu tiên ta phải tạo đó là **LPDIRECT3D9**(Số 9 chỉ số phiên bản ta đang sử dụng, LP là long point như vậy LPDIRECT3D9 tức là con trỏ long chỉ về DIRECT3D9). LPDIRECT3D9 là đối tượng quan trọng, là cầu nối khởi tạo khi ta sử dụng DIRECT3D9.

LPDIRECT3D9 d3d = NULL;

Ok, bây giờ ta đã có LPDIRECT3D9, ta sẽ làm gì với nó, tất nhiên ta sẽ phải khởi tạo cho nó, giống như việc gọi hàm constructor cho nó vậy, nhưng với LPDIRECT3D9 thì cách gọi có hơi đặc biệt (tất nhiên là ta phải học thuộc nó). Nếu việc gọi này ko thực thi đc, nó sẽ trả về NULL, ta sử dụng hàm Direct3DCreate9

d3d = Direct3DCreate9(D3D\_SDK\_VERSION);

D3D\_SDK\_VERSION là 1 integer nằm trong d3d9.h (do đó ta nên chắc chắn đã add thư viện này vào)

Ok tới đây ta đã khởi tạo đầy đủ con trỏ tới IDRIECT3D9, chúng ta sẽ nói sơ qua nhiệm vụ của đối tượng này. LPDIRECT3D9 có 2 nhiệm vụ chính:

* Device enumeration : việc này có nghĩa là ta dùng nó để kiểm tra khả năng, định dạng, các thông số của card màn hình để có thể tùy chỉnh thích hợp đối với ứng dụng game của ta. (công việc này rất cần thiết vì ngay sau bước này ta sẽ tiến tới bước tạo Device, lúc đó tất cả các thông tin này phải đc xem xét và khai báo)
* Tạo Device, tức là con trỏ tới IDIRECT3DDEVICE9

Vì công việc xem xét các thông tin của thiết bị đồ họa rất phức tạp và có thể kéo dài nên ta sẽ xem xét nó kỹ hơn sau, ở đây ta chỉ xét 1 trường hợp theo dõi coi thiết bị đồ họa có hỗ trợ Vertex Processing ko

Để kiểm tra card đồ họa, ta phải có **D3DCAPS9** (đã nói ở phần khác). Và như đã nói ở trên, công việc kiểm tra sẽ dính dáng tới IDIRECT3D9, do đó ta phải khởi tạo D3DCAPS9 từ IDRIRECT3D9 (tương tự cho IDRECT3DDEVICE9)

HRESULT IDirect3D9::GetDeviceCaps(

UINT Adapter,

D3DDEVTYPE DeviceType,

D3DCAPS9 \*pCaps

);

Thông số thứ 1 là thiết bị phần cứng nào cần đc kiểm tra, ở đây ta để D3DADAPTER\_DEFAULT để nó kiểm tra thiết bị chính. Thông số thứ 2 là kiểu thiết bị HAL hay REF, tất nhiên ta để HAL. Thông số cuối cùng trả về cho D3DCAP9.

Sau khi lấy đc D3DCAP9, ta có thể thực hiện kiểm tra (đã có nói ở bài khác)

D3DCAPS9 caps;

d3d9->GetDeviceCaps(

D3DADAPTER\_DEFAULT, // Denotes primary display adapter.

deviceType, // Specifies the device type, usually D3DDEVTYPE\_HAL.

&caps); // Return filled D3DCAPS9 structure that contains

// the capabilities of the primary display adapter.

// Can we use hardware vertex processing?

int vp = 0;

if( **caps.DevCaps & D3DDEVCAPS\_HWTRANSFORMANDLIGHT** ) // dòng này quan trọng đây

{

// yes, save in ‘vp’ the fact that hardware vertex

// processing is supported.

vp = D3DCREATE\_HARDWARE\_VERTEXPROCESSING;

}

else

{

// no, save in ‘vp’ the fact that we must use software

// vertex processing.  
vp = D3DCREATE\_SOFTWARE\_VERTEXPROCESSING;

}

Bây giờ sau khi ta đã có đối tượng LPDIRECT3D9 rồi, ta sẽ xem xét để làm sao tạo ra đối tượng Device, ở đây đối tượng Device có tên là**LPDIRECT3DDEVICE9**(tương tự như trên ta thấy đây là long point trỏ tới DIRECT3DDEVICE9)

LPDIRECT3DDEVICE9 d3ddev = NULL

Tiếp theo, giống như DIRECT3D9 ta phải khởi tạo cho nó (như construction cho nó). Tuy nhiên đối tượng Device này sẽ được tạo thông qua LPDIRECT3D9 (giống như đối tượng cha sinh ra đối tượng con). Ta thử xét qua hàm khởi tạo LPDIRECT3DDEVICE9 của LPDIRECT3D9

**HRESULT CreateDevice(**

**UINT***Adapter***,**

**D3DDEVTYPE***DeviceType***,**

**HWND***hFocusWindow***,**

**DWORD***BehaviorFlags***,**

**D3DPRESENT\_PARAMETERS \****pPresentationParameters***,**

**IDirect3DDevice9 \*\****ppReturnedDeviceInterface*

**)**;

* Adapter : là nói tới card đồ họa, do ta chỉ sử dụng card đồ họa chính (primary card) cho nên giá trị ở đây là **D3DADAPTER\_DEFAULT** hoặc để số 0.
* D3DDEVTYPE : Có 4 lựa chọn, chủ yếu nằm trong enum \_D3DDEVTYPE là D3DDEVTYPE\_HAL = 1, D3DDEVTYPE\_REF = 2, D3DDEVTYPE\_SW = 3, D3DDEVTYPE\_FORCE\_DWORD = 0xffffffff. Chúng ta sẽ coi thử từng ứng dụng
  + D3DDEVTYPE\_HAL : Sử dụng HAL, tức là các thư viện DirectX sẽ được hỗ trợ bởi phần cứng (coi phần DirectX Graphic Overview). Chú ý là nếu không có xử lý 3D thì không cần HAL hoặc việc thiết lập HAL là không cần thiết. Nếu HAL được tạo thành công thì tức là ứng dụng phải có 3D gì đó. Có nghĩa là quá trình rasteriazation chắc chắn sẽ được làm bởi video card hay cả quá trình transformation và lighting … chúng ta phải check coi card màn hình có hỗ trợ HAL hay không trước khi yêu cầu Device sử dụng HAL.
  + D3DDEVICETYPE\_REF : Nếu HAL không chọn thì HEL là lực chọn của chúng ta (Coi phần DirectX Graphic Overview) và tùy chọn REF này có nghĩa là ta sẽ sử dụng HEL cho device. Tùy chọn HEL thường được sử dụng để kiểm tra xem hardware có làm việc chính xác hay không. Ví dụ video card maker có thể kiểm tra quá trình phát triển xem việc reference rasterizer của card video xem nó render polygon sáng hay tối như thế nào. Chúng ta nên có phần kiểm tra nếu card không có HAL thì ta phải chỉnh tùy chọn HEL cho device.
  + D3DDEVICETYPE\_SW : Tùy chọn này cho ứng dụng của ta chạy trên máy không có card màn hình (hay là card màn hình không hỗ trợ cả HEL). Tất cả sẽ chạy đúng nghĩa trên Cpu.
* HWND **:**là handle của window, ở đây ta gán HWND của window vào (HWND được khởi tạo trong lúc lập trình Window), việc này cho card đồ họa biết đâu là nơi nó sẽ hiển thị hình ảnh. Trong hầu hết trường hợp thì ta chỉ cần truyền tham số HWND vào đây là đủ. Lưu ý nhỏ là trong chế đệ fullscreen mode thì ta nên set flag WS\_EX\_TOPMOST, nếu không thì device có thể fail trong việc tạo chế độ toàn màn hình.
* BehaviourFlag: Ở đây có 3 tùy chọn giúp ta có thể tạo ra device với sự hỗ trợ maximum từ hardware (có nghĩa là sử dụng hết sức mạnh của video card).
  + D3DCREATE\_HARDWARE\_VERTEXPROCESSING : Tạo device sử dụng transformation, lighting và rasterization tất cả trên video card (chủ yếu thêm T&L được xứ lý trên video card). Đây là tùy chọn cao nhất (tức là tận dụng tuyệt đối sức mạnh video card). Tùy chọn này kết hợp với tùy chọn HAL (phần trên). Nếu tùy chọn này thất bại thì ta có thể chọn tùy chọn bên dưới (T&L sử lý trên software, chậm hơn nhưng vẫn bảo đảm các quá trình). Nếu chọn tùy chọn này với HEL thì vẫn successful, tuy nhiên quá trình rasterizer sẽ không được nhanh (trong khi ta có thể chọn HAL, trường hợp này giống như lãng phí bộ nhớ). Cũng có trường hợp video card chỉ hỗ trợ transformation chứ không hỗ trợ light, trong trường hợp này thì ta phải dùng kết hợp cả software và hardware xử lý (xem tùy chọn bên dưới).
  + D3DCREATE\_MIXED\_VERTEXPROCESSING : Nếu video không hỗ trợ một cách đầy đủ capabilities (xem phần trên để hiểu một chút, chúng ta sẽ biết các kiểm tra capabilities ở khúc sau). Lúc này chúng ta phải chọn device vừa chạy trên hardware vừa chạy trên software với tùy chọn này.
  + D3DCRAETE\_SOFTWARE\_VERTEXPROCESSING : Tùy chọn này sử dụng khi HAL không được hỗ trợ (tức là flag D3DDEVICETYPE\_REEF hoặc D3DDEVICETYPE\_SW được chọn). Tất cả các tính toán được dùng trên CPU. Nếu HAL device được chọn nhưng ta không thể thêm được các tính năng nào khác (T&L) thì ta cũng chọn tùy chọn này (tức là video card chỉ hỗ trợ rasterization). Tốt nhất là ta chỉ xài flag này khi video card không hỗ trợ vì thực hiện trên video card vẫn nhanh hơn nhiều.
  + Ngoài các flag thông dụng trên ta còn 1 flag có thể đi kèm với flag D3DCREATE\_HARDWARE\_VERTEXPROCESSING là flag D3DCREATE\_PUREDEVICE. Khi 2 flag này kết hợp có nghĩa là HAL hỗ trợ pure device. Chúng ta có thể nói device object là 1 cái máy tĩnh mà chúng ta có thể can thiệp vào bằng các lệnh như SetRenderState, SetTransform …

IDirect3D9 cung cấp 1 hàm gọi là IDirect3D9::GetDeviceCaps cho phép nhận thông tin về device capability. Bởi vì đây là thành phần của IDirect3D9 interface, nó có thể gọi để truy vấn thiết bị mà không cần khởi tạo thiết bị trước.

HRESULT GetDeviceCaps(UINT *Adapter*,D3DDEVTYPE *DeviceType*,D3DCAPS9\* *pCaps*);

Chúng ta đưa các thông số đã coi từ phần trước vào

D3DCAPS9 DevCaps;

pD3D9->GetDeviceCaps (D3DADPATER\_DEFAULT, D3DDEVICETYPE\_HAL, &DevCaps);

Sau khi đã có DeviceCap chúng ta sẽ thực hiện kiểm tra khả năng support của hardware (software tất nhiên hỗ trợ tốt) về VertexProcessingCaps thông qua các flag như (coi sdk để thấy chi tiết hơn)

**D3DVTXPCAPS\_DIRECTIONALLIGHTS**

Device supports directional lights.

**D3DVTXPCAPS\_LOCALVIEWER**

Device supports local viewer.

**D3DVTXPCAPS\_MATERIALSOURCE7**

Device supports selectable vertex color sources.

**D3DVTXPCAPS\_POSITIONALLIGHTS**

Device supports positional lights (including point lights and spotlights).

**D3DVTXPCAPS\_TEXGEN**

Device can generate te

**D3DVTXPCAPS\_TWEENING**

xture coordinates.

Device supports vertex tweening.

**D3DVTXPCAPS\_NO\_VSDT\_UBYTE4**

Device does not support the D3DVSDT\_UBYTE4

Các test như sau

if (DevCaps.VertexProcessingFlags & D3DVTXCAPS\_TWEENING)

{

//Tweening is supported

}

Nếu kết quả trên trả về fail tức là ta không thể dùng HAL device.

2 tham số cuối là các biến kiểu **D3DPRESENT\_PARAMETERS** (sẽ đề cập sau) và đối tượng **LPDIRECT3DDEVICE9** sẽ được khởi tạo (cách khởi tạo này khá độc đáo, khởi tạo trong hàm của đối tượng khác).

Tạo ra 1 software device là một công việc phức tạp đòi hỏi ta phải hiểu rõ các quá trình của nó. Hầu hết các game hiện nay đều cần phải chạy rất nhiều trên card đồ họa (HAL) và còn phải hỗ trợ ca3 Vertex transformation và lighting trong hardware.

**D3DPRESENT\_PARAMETERS**

Chúng ta sẽ nghiêm cứu sâu hơn về **D3DPRESENT\_PARAMETERS**như tên gọi của nó, đây là biến điều khiển quá trình hiển thị của Device, vì là biến hỗ trợ cho nên ta không thể sử dụng nó như một đối tượng và không cần phải construction cho nó, chỉ cần tạo biến và đưa vào các giá trị cần thiết.

**typedef struct \_D3DPRESENT\_PARAMETERS\_** {

**UINT***BackBufferWidth, BackBufferHeight;*

**D3DFORMAT***BackBufferFormat;*

**UINT***BackBufferCount;*

**D3DMULTISAMPLE\_TYPE***MultiSampleType;*

**DWORD***MultiSampleQuality;*

**D3DSWAPEFFECT***SwapEffect;*

**HWND***hDeviceWindow;*

**BOOL***Windowed;*

**BOOL***EnableAutoDepthStencil;*

**D3DFORMAT***AutoDepthStencilFormat;*

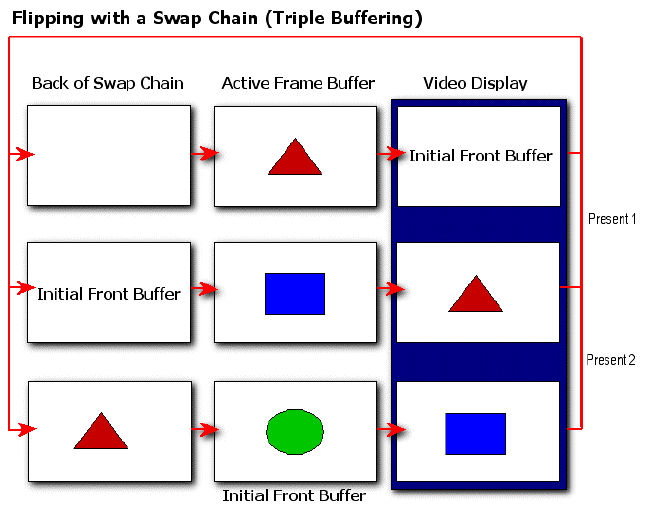
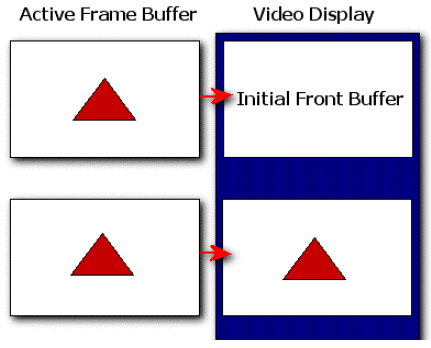
**DWORD***Flags;*

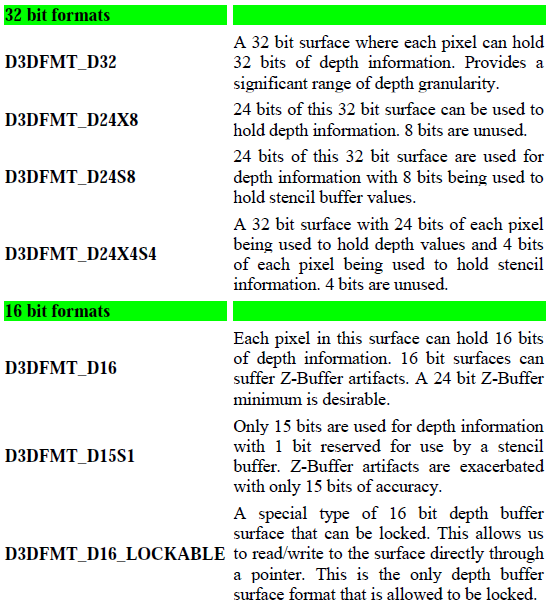
**UINT***FullScreen\_RefreshRateInHz;*

**UINT***PresentationInterval;*

} **D3DPRESENT\_PARAMETERS**;

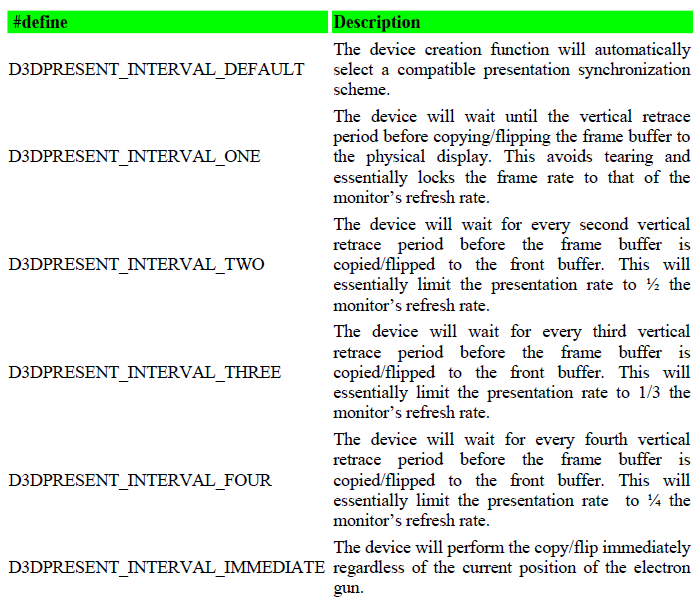
Chúng ta ko cần thông qua hết các giá trị, chỉ cần xem xét vài cái chúng ta cần.

* **BackBufferWidth / BackBufferHeight** : Đây là thông tin của Frame Buffer, chú ý là ở chế độ fullscreen thì framebuffer phải khớp với resolution của front buffer (tức là mode display) và những giá trị phải nằm trong sự hỗ trợ cua DirectX (trong enum IDirect3D9::EnumAdapterModes). Trong windowed mode thì việc này không cần thiết lắm, nếu chúng ta thiết đặt giá trị này là 0 trong windowed mode thì device sẽ tự động tạo frame buffer phù hợp với resolution của vùng window hiển thị (kích thước do phần lập trình window quy định)
* **BackBufferFormat** : Đây là định dạng của frame buffer, trong fullscreen thì format này sẽ là format cho video mode (tức là cũng là front buffer). Ví dụ nếu frame buffer là D3DFMT\_A8R8G8B8 thì front buffer sẽ là D3DFMT\_X8R8G8B8 (vì front buffer không có sử dụng kênh alpha) đồng thời có cùng resolution với Backbufferwidth và Backbufferheight. Chú ý là các định dạng phải nằm trong sự hỗ trợ của hardware, nếu không thì sẽ fail. (Dùng IDirect3D9::CheckDeviceType). Giá trị này có thể từ 0 tới 4, 0 tương đương như 1. Nếu giá trị này quá lớn (hơn 4) thì sẽ fail.
* **BackBufferCount**là số lượng của FrameBuffer. Thường thường ta nên để là 1 vì chỉ cần 1 Backbuffer là đủ (tức là xài double buffering)
* **MultiSampleType / MultiSampleQuality** : ngày nay thì các card đồ họa cũng hỗ trợ chức năng giảm răng cưa. Nếu trong video card có hỗ trợ chức năng này (chúng ta phải kiểm tra nó) thì ta có thể chọn tùy chọn khả năng này. Đặt giá trị 0 nếu ta không sử dụng khả năng khử răng cưa
* **SwapEffect** Kiểu chuyển đổi bộ đệm Buffer (xem phần Direct3D Overview), D3DSWAPEFFECT enum sẽ mô tả cách swap, chúng ta coi thử các kiểu swap giữa frame buffer và front buffer.
  + D3DSWAPEFFECT\_FLIP : Sử dụng khi fullscreen (coi phần Direct3D Overview), lúc này frame buffer và front buffer swapping cho nhau rất nhanh. Khi chúng ta có nhiều hơn 1 frame buffer thì swap chain sẽ xoay vòng theo thứ tự. Chúng ta sẽ thí dụ với triple buffering (sử dụng 2 frame buffer).Chú ý là trong windowed mode không sử dụng kỹ thuật flip (coi phần Direct3D Overview) mà dùng cách khác chậm hơn. Do đó nếu không phải fullscreen thì chúng ta không dùng flip. Nếu chúng ta dùng flip trong windowed mode và sử dụng nhiều buffer thì có thể làm cho overhead và comsume video memory.
  + D3DSWAPEFFECT\_COPY : Cách thực hiện swap này là ta copy những gì trong frame buffer lên fron buffer, trong windowed mode thì cách swap này sẽ được thực hiện (copy tất cả pixel từ frame lên front buffer). Trong fullscreen nếu chọn chế độ này thì hardware có thể thực hiện copy, flip hoặc kết hợp cả 2.Chú ý là khi copy thì dữ liệu ở frame không bị xóa (coi hình trên). Điều này sẽ có ích nếu ta muốn lấy những gì mà scene đang hiển thị (từ frame buffer). Tùy chọn này được sử dụng trong windowed mode, trong fullsceen nếu không bị bắt buộc thì ta nên dùng flipping vì copying chậm hơn và có thể tốn nhiều video memory hơn. Chu ý là D3DSWAPEFFCT\_COPY chỉ có thể dùng với device có 1 frame buffer (BackBufferCount =0 hoặc 1).
  + D3DSWAPEFFECT\_DISCARD : lựa chọn này cho device tực chọn phương thức thích hợp (flipping or copying) dựa trên video và window modes. Điều này có nghĩa là flipping sẽ được dùng nếu ở fullscreen và copying được dùng nếu ở windowed (tuy nhiên điều này không được bảo đảm). Thus our application should not make any assumptions about the state of the frame buffer after the screen presentation. When using D3DSWAPEFFECT\_DISCARD we will always treat the frame buffer as an uninitialized memory buffer requiring that we render over the entire surface. In fact, the DirectX Graphics debug runtime will automatically fill the contents of a presented frame buffer with random data to discourage you from making such assumptions when usingthis swap effect.
* **HWND (Phần này cần nghiêm cứu thêm trong sách)** là handle của window, nếu ta để NULL thì nó sẽ dùng window đang được focus. Tuy nhiên có một vài điều chúng ta cần coi qua khi dùng chế độ fullscreen và windowed mode.
  + Windowed Mode :Trong Windowed Mode thi` HWND là vùng client sử dụng như là font buffer. Có thể minimized và maximized như các window khác. Nếu window di chuyển thì vùng hiển thị di chuyển theo. Vấn đề resizing sẽ được nói sau.
  + Fullscreen Mode : Khi ở chế độ fullscreen thì nó nhận được quyền đặc biệt. toàn bộ màn hình sẽ có kích thước dựa vào BackBufferwidth và BackBufferHeight, destop vẫn hoạt động và nhận message
  + **Ghi chú** Sẽ có lúc chúng ta muốn chuyển qua chuyển lại giữafullscreen và windowed mode. Chúng ta có 2 lựa chọn.
    - Thứ 1 là khi đổi chúng ta sẽ khởi động lại device (trả về trạng thái chuẩn lúc khởi tạo) bằng hàm IDirect3DDevice9::Reset, sau đó nếu là fullscreen thì chúng ta set device cho fullscreen còn windowed thì set device cho windowed mode.
    - Một cách khác là tạo 2 device windowed có border và caption, thứ 2 là device cho fullscreen không có border và caption. Khi dùng cái nào thì ta gọi device đó lên.
* **Windowed** nhận giá trị “TRUE” khi đang ở chế độ windowed, ngược lại là “FALSE” khi ứng dụng fullscreen
* **EnableAutoDepthStencil**Biến kiểu Boolean này hướng dẫn cho hàm khởi tạo của device tạo hoặc không tạo Depth Buffer cho device.
  + Nếu True thì hàm khởi tạo sẽ tạo ra depth buffer (Z-Buffer hoặc W-Buffer) và khởi tạo định dạng trong thuộc tính tiếp theo (coi ở dưới). Nếu depth buffer tạo thành công thì nó tự động kết nối với frame buffer. Bất cứ pixel nào được render sẽ phải kiểm tra độ sâu và ghi nhận trong depth buffer. Nếu Device được reset, nó sẽ tự hủy buffer, mún tạo lại thì khởi tạo buffer mới cho phù hợp với resolution mới.
  + Nếu False thì nó sẽ không tự tạo depth buffer và ứng dụng phải tự tạo nếu cần thiết, ngoài ra khi device reset thì ứng dụng cũng phải tự hủy buffer.
* **AutoDepthStencilFormat**nếu EnalbleAutoDepthStencil là True (biến ở trên) thì biến này sẽ giữ giá trị D3DFORMAT miêu tả định dạng của depth surface. Không giống như các surface sử dụng trong texture và frame buffer, chúng ta có những kiểu D3DFORMAT đặc biệt cho depth buffer. Chúng ta coi danh sách bên dưới

****

Chúng ta nói một chút về **stencil buffer**. Stencil buffer dùng để che frame buffer những vùng mà nó không cần hiển thị. Nó xử dụng cùng bộ nhớ vật lý với depth buffer. Ứng dụng game của chúng ta phải chắc chắn rằng có định dạng depth buffer mà chúng ta dùng phải được hardware hỗ trợ. Ví dụ nếu card màn hình không hỗ trợ depth buffer 32bit thì ta không thể xài chúng.  
 Ở đây chúng ta có thể dùng hàm **FindDepthStencilFormat** để tự động tìm trị thích hợp.

* **Flags** : Cờ này cho biết ứng dụng nên xử dụng frame buffer và depth buffer như thế nào. Có 2 lựa chọn quan trọng như sau
  + **D3DPRESENTFLAG\_LOCKAVLE\_BACKBUFFER** – Nếu tùy chọn này được chọn, device sẽ tạo ra frame buffer có thể locked hoặc modified. Khi chúng ta lock 1 surface (gọi hàm IDirect3Dsurface9::Lock) thì 1 con trỏ tới surface được trả về. Nó cho phép chúng ta modify frame buffer ở cấp độ pixel và đọc pixel color từ frame buffer. Trong hardware việc tạo frame buffer với flag này có thể phải trả ra 1 vùng bộ nhớ copy của frame buffer mà ứng dụng có thể lấy. Cho dù trường hợp nào thì việc locked frame buffer là toán tử rất lớn cho nên cần phải tránh. Frame buffer được tạo ra với thuộc tính mặc định không locked cho nên flag này cần thiết cho ta trong các trường hợp cần locked (thường không phổ biến lắm)
  + **D3DPRESENT\_DISCARD\_DEPTHSTENCIL** –Nếu device được tạo với depth buffer thì flag này có thể làm tăng performance. Nếu flag này không được thiết đặt thì device sẽ lưu trữ toàn bộ thông tin depth buffer sau khi được vẽ lên front buffer. Nếu ứng dụng không clear depth buffer trước khi vẽ tiếp cảnh tiếp theo thì depth buffer vẫn lưu thông tin của cảnh trước. Đôi khi điều này có lợi và đôi khi cũng bất lợi. Chúng ta sẽ thường thiết đặt flag này bởi vì ứng dụng sẽ clear depth buffer sau khi render mỗi frame. DirectX debug runtime sẽ xóa depth buffer bằng cách điền giá trị xác định đặc biệt vào nó sau khi cảnh được present. Nếu flag này không được thiết đặt thì driver của hadrware sẽ cần một công thức tính toán khá lớn để kiểm tra chắc rằng thông tin trong depth buffer không bị thay đổi.
* **FullScreen\_RefreshRateInHz** : Biến này để thiết đặt refresh rate cho fullscreen device, trong windowed mode thì giá trị này phải là 0 bởi vì chúng ta sử dụng refresh rate chung trên desktop. Thiết đặ D3DPRESENT\_RATE\_DEFAULT sẽ cho phép device chọn refersh rate thích hợp. Đây là kiểu thông dụng thường được dùng.
* **PresentationInterval** :BIến này cho phép ứng dụng quyết định mức độ mà frame buffer được vẽ lên front buffer. Trong fullscreen device chúng ta thường muốn đồng bộ việc presentation với refresh rate của màn hình để tránh tearing artifacts (coi phần Direct3D Overview). Tuy nhiên có một vài tùy chọn.



Tuy nhiên chúng ta chỉ nên chọn các tùy chọn như D3DPRESENT\_INTERVAL\_DEFAULT, D3DPRESENT\_INTERVAL\_IMMEDIATE, D3DPRESENT\_INTERVAL\_ONE

Một chú ý quan trọng là để đảm bảo các biến có gái trị 0 (như BackBufferCount nếu ko xài tức là ko có BackBuffer nào) tức là khởi tạo mặc định. Ta phải xài hàm **ZeroMemory(…)**sau khi tạo biến **D3DPRESENT\_PARAMETERS**

D3DPRESENT\_PARAMETERS d3dpp;  
ZeroMemory(&d3dpp, sizeof(d3dpp));

Đến đây ta đã tạo xong Device và có thể bắt đầu xử dụng nó. Chúng ta sẽ xem xét thử vài hoạt động cần thiết phải làm với Device. Đầu tiên ta sẽ xóa trắng màn hình (giống như chuẩn bị màn hình sẵn sàng cho game vậy). Ở đây ta sẽ sử dụng hàm **Clear()**(Lưu ý là việc Clear chỉ là phần chuẩn bị chứ ko phải phần Render)

**HRESULT Clear(**

**DWORD** Count,

**const D3DRECT** \*pRects,

**DWORD** Flags,

**D3DCOLOR** Color,

**float***Z*,

**DWORD** Stencil

**);**

Tham số đầu tiên là số lượng vùng hình chữ nhật được xóa Count, nếu ko có thì chọn 0, cũng đồng nghĩa với xóa hết màn hình.

Tham số thứ 2 là mảng các vùng RECT được xóa, nếu Count=0 thì đồng nghĩa D3DRECT = NULL.

Tham số thứ 3 lại là 1 Flag, như vậy tham số này có nghĩa là hành động Clear như thế nào đối với Surface, có 3 tham số có thể chọn (D3DCLEAR\_TARGET, D3DCLEAR\_STENCIL,D3DCLEAR\_ZBUFFER) nhưng nếu chúng ta code 2D, tức là chỉ có 1 surface chính, thì chúng ta chỉ cần chọn **D3DCLEAR\_TARGET.**(Ta sẽ nghiêm cứu các lựa chọn sau vào phần khác)

Tham số thứ 4 là hệ màu, hay nói đúng hơn là màu bạn muốn Clear màn hình (tham số này nên được thống nhất chung, có nghĩa là toàn chương trình nên có hệ màu giống nhau)

Tham số thứ 5 và thứ 6 tạm thời chưa xài đến.

**Chú ý : Sau khi hoàn thành 2 nhiệm vụ chính thì IDirect3D gần như hết tác dụng và ta phải release cho nó.**

Sau khi chúng ta đã chuẩn bị tất cả đầy đủ, chúng ta sẽ bắt đầu thực hiện việc Render (tức là xuất ra man hình). Ta sử dụng hàm **BeginRender()** của LPDIRECT3DDEVICE9 để thông báo việc render được bắt đầu, sau đó ta khai báo **EndRender()** để kết thúc việc Render này. Như vậy tất cả những gì ta muốn Render ra màn hình đều sẽ được nằm trong khối BeginRender() và EndRender() này

if (d3ddev->BeginScene())

{

// Ta có thể render cái gì đó ở đây

d3ddev->EndScene();

}

Sau khi xây dựng các công việc cần được Render ra màn hình, ta gọi hàm **Present(…)** để bắt đầu việc vẽ ra màn hình (hiện giờ ta tạm nhớ 4 tham số của hàm Present luôn là NULL). Nói 1 cách chính xác hơn hàm Present là thao tác chuyển từ BackBuffer lên FrontSurface (ta sẽ nghiêm cứu trong phần Surface)

d3ddev->Present(NULL, NULL, NULL, NULL);

Chú ý là cuối cùng ta phải giải phóng các đối tượng Device khi kết thúc chương trình

if (d3ddev != NULL)

d3ddev->Release();

**SURVEY GAME APPLICATION**

Chúng ta thử coi qua cái nhìn tổng quát về 1 chương trình game. Đối với 1 chương trình game ta có thể chia ra làm 3 phần chính như sau

* Setup/GameInit/Prepare …. : Đây là bước khởi tạo, nơi ta có thể khởi tạo tất cả các thành phần quan trọng, kiểm tra device, thiết lập các cấu kiện window ….
* Clearup/Releash/GameEnd .... : Đây là bước giải phóng tài nguyên, thực hiện khi chương trình được kết thúc
* Display/Run/GameRun …. : Đây là bước thực thi các thành phần render đồ họa ra màn hình, ta thường đưa vào 1 thông số quan trọng là **timeDelta** để tính toán thời gian game đã trôi qua nhằm phục vụ cho các bước animation hoặc tính toán thời gian render đồ họa thích hợp.

**MORE INFORMATION ABOUT IDIRECT3D9**

Ở phần này chúng ta coi qua một số hàm thông dụng trong IDirect3D9 chưa được đề cập đến

**UINT GetAdapterCount** (VOID);

**HRESULT GetDeviceCaps** (UINT *Adapter*, D3DDEVTYPE *DeviceType*, D3DCAPS9\* *pCaps*);

**UINT GetAdapterModeCount** (UINT *Adapter , D3DFORMAT Fomat* );

**HRESULT GetAdapterDisplayMode** (UINT *Adapter*, D3DDISPLAYMODE\* *pMode* );

**HRESULT CheckDeviceType** (UINT *Adapter*, D3DDEVTYPE *CheckType*, D3DFORMAT *DisplayFormat*,D3DFORMAT *BackBufferFormat*, BOOL *Windowed* );

**HRESULT EnumAdapterModes** (UINT *Adapter*, D3DFORMAT *Format*, UINT *Mode*,

D3DDISPLAYMODE\* *pMode*);

**HRESULT CreateDevice** (UINT *Adapter*, D3DDEVTYPE *DeviceType*, HWND *hFocusWindow*,

DWORD *BehaviorFlags*, D3DPRESENT\_PARAMETERS\* *pPresentationParameters*,

IDirect3DDevice9\*\* *ppReturnedDeviceInterface*);

GetAdapterCount : hàm này trả về số lượng các display adapter có trong hệ thống hiện tại. Hàm này luôn trả về 1, thể hiện chỉ 1 display adapter được installed và graphic crad đầu tiên được installed sẽ là **primary display adapter**. Chúng ta sẽ nghiêm cứu vấn đề chuyển đổi adapter về sau.

GetAdapterDisplayMode : Xem phần Display Mode

EnumAdapterModes :Trong DirectX Graphic có 1 kiểu định dạng cho ta biết có bao nhiêu image pixel được hiển thị trong bộ nhớ. Enum D3DFORMAT chứa tất cả các kiểu format hỗ trợ bởi DirectX. Khi chúng ta chạy game, chúng ta sẽ yêu cầu nhiều độ phân giải khác nhau từ 640x480 cho hệ thống yếu tới 1600x1200 cho các hệ thống hiện đại ngày nay. Hàm EnumAdapterModes cho phép ta yêu cầu 1 danh sách các video resolutions các pixel format

Chúng ta làm 1 ví dụ như sau, chúng ta có 1 adapter (card màn hình) trong hệ thống. Chúng ta quyết định sử dụng D3DFORMAT với định dạng D3DFMT\_R5G6B5. Bây giờ chúng ta sẽ xem đoạn code tìm các định dạng hỗ trợ bới adapter

D3DDISPLAYMODE Mode;

UINT AdapterOrdinal = 0;

D3DFORMAT Format = D3DFMT\_R5G6B5;

LPDIRECT3D9 pD3D;

pD3D = Direct3DCreate9( D3D\_SDK\_VERSION );

if (!pD3D) return FALSE;

UINT NumberOfModes = pD3D->GetAdapterModeCount (Adapter, Format);

if (!NumberOfModes) return FALSE;

for (UINT I=0; I < NumberOfModes; I++)

{

pD3D->EnumAdapterModes( Adapter, Format , I , &Mode);

FormatModeList->push\_back(Mode);

}

Chúng ta thấy trong chương trình này có sử dụng 1 hàm mới là GetAdapterModeCounttrả về số lượng các kiểu định dạng có thể có đối với định dạng D3DFMT\_R5G6B5. Sau đó ta cho 1 vòng lặp và áp dụng hàm EnumAdapterModes để trả về các kiểu định dạng (rất đa dạng và tùy sức mạnh của card màn hình).

Chúng ta nên thường kiểm tra D3DDISPLAYMODEđể nhận được các kiểu mode thích hợp, chúng ta nên sử dụng 2 kiểu thông dụng là D3DFMT\_R5G6B5 và D3DFMT\_X1R6G5B5, 2 kiểu này đều hỗ trợ 16bit

**MORE INFORMATION ABOUT IDIRECT3DDEVICE9**

Device object sẽ quản lý tất cả các quá trình transformation pipeline, rendering to the frame buffer, pixel blending, depth testing và texture mapping thông qua hardware khi vần thiết.

Có thể nhận thấy Device như một 3D engine. Ở cấp độ cơ bản nhất thì chúng ta có thể nói device render 1 polygon khi truyền dữ liệu vertices cho IDirect3DDevice9::DrawPrimitive. Polygon mà chúng ta yêu cầu device vẽ ra có thể thêm các hiệu ứng như lighting, multiple texture blended, color blending, transparency ….

Device giống như một cỗ máy có thể điều khiển bằng các hàm của chính nó (ví dụ IDirect3DDevice9::SetRenderState). Những điều khiển này lệnh cho device thực hiện việc transforms, color blend các polygon trên màn hình.

Sử dụng các thành phần cơ bản để điều khiển việc transformation and lingting của vertices được gọi là fixed function pipeline. Từ DirectX8, Microsoft đã mở rộng rendering pipeline cho developer sử dụng programmable shader. Shader cho phép ta tạo ra đoạn code nhỏ để thực hiện transforming and lingting vertices and coloring pixels theo ý của chúng ta thay vì sử dụng các hàm có sẵn. Chúng ta sẽ ngâm cứu shader rất nhiều trong các phần sau.

Hình sau mô tả quá trình vẽ của device object và những software module trong nó. Màu blue là vertex processor và màu tím là pixel processor

