**DIRECT3D OVERVIEW**

Phần này chúng ta coi qua một số khái niệm cơ bản dễ gặp khi chúng ta lập trình với DirectX

DirectX là thư viện đồ họa cấp thấp (tức là gồm các hàm API) dùng để điều khiển các thiết bị thực hiện các thao tác render đồ họa ra màn hình. Nói như vậy nhưng thật ra nó không điều khiển trực tiếp các thiết bị đồ họa. Có 1 cầu nối trung gian là HAL, DirectX sẽ yêu cầu các thiết bị đồ họa thực thi HAL và thông qua HAL, DirectX có thể điều khiển các thiết bị đồ họa làm theo ý mình.

**D3DDEVTYPE**

Có 2 giá trị loại này, thứ nhất là D3DDEVTYPE\_HAL có nghĩa là ta sẽ thông qua HAL để yêu cầu thiết bị đồ họa giải quyết các thuật toán đồ họa cho ta. Thứ 2 là D3DDEVTYPE\_REF, có lựa chọn này bởi vì đôi khi ta mún test ứng dụng chạy trên CPU, hoặc trong trường hợp vài thuật toán cao cấp mà card đồ họa của ta ko hỗ trợ, ta cũng có thể chạy thử trên CPU, tuy nhiên nó sẽ khá chậm.  
Coi phần HAL để hiểu thêm.

**SURFACE**

Surface là 1 mảng matrix để lưu giữ hình ảnh bitmap 2D. Tất nhiên chiều dài và rộng của Surface đc tính theo đơn vị pixel. Khi code ta có thể khai báo Surface bằng **IDIRECT3DSURFACE9**. Chúng ta coi một vài phương thức chứa đựng trong Interface này

* LockRect – Cho phép ta trỏ con trỏ tới surface, nhờ đó ta có thể thay đổi bitmap trong Surface
* UnLockRect – Sau khi chúng ta xài LockRect và chỉnh sữa ảnh, ta phải gọi hàm này.
* GetDesc – Phương thức này trả về mô tả của Surface bằng structure D3DSURFACE\_DESC

typedef struct \_D3DLOCKED\_RECT {

INT Pitch; // the surface pitch

void \*pBits; // pointer to the start of the surface memory

} D3DLOCKED\_RECT;

Coi thử 1 ví dụ (Ít khi xài, nhưng coi cho hiểu)

// Get the surface description.

D3DSURFACE\_DESC surfaceDesc;

\_surface->GetDesc(&surfaceDesc);

// Get a pointer to the surface pixel data.

D3DLOCKED\_RECT lockedRect;

\_surface->LockRect(

&lockedRect,// pointer to receive locked data

0, // lock entire surface

0); // no lock flags specified

// Iterate through each pixel in the surface and set it to red.

DWORD\* imageData = (DWORD\*)lockedRect.pBits;

for(int i = 0; i < surfaceDesc.Height; i++)

{

for(int j = 0; j < surfaceDesc.Width; j++)

{

// index into texture, note we use the pitch and divide by

// four since the pitch is given in bytes and there are

// 4 bytes per DWORD.

int index = i \* lockedRect.Pitch / 4 + j;

imageData[index] = 0xffff0000; // red

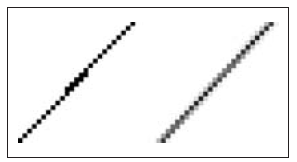
}

}

Chúng ta sẽ biết nhiều hơn về các định dạng của Surface ở phần sau.

**Multisampling**

Đây là từ diễn tả việc làm giảm răng cưa trong game, và tiêu biểu là kỹ thuật antialiasing

 enum **D3DMULTISAMPLE\_TYPE** chứa đựng các giá trị để ta có thể thiết lập việc này

* **D3DMULTISAMPLE\_NONE** : Không sử dụng Multisample
* **D3DMULTISAMPLE\_1\_SAMPLE**…**D3DMULTISAMPLE\_16\_SAMPLE** : thiết lập giá trị từ 1 đến 16

Một chú ý là sử dụng Multisample sẽ khiến ứng dụng chậm đi rất nhiều, tốt nhất là ta nên kiểm tra xem thiết bị đồ họa có hỗ trợ chức năng này không bằng phương thức **IDirect3D9::CheckDeviceMultiSampleType** (tức là kiểm tra bằng **LPDIRECT3D9**)

**Pixel Formats**

Chúng ta phải thiết đặt rõ ràng định dạng pixel khi chuẩn bị xài Texture hay Surface. Các kiểu định dạng đc xác định bằng D3DFORMAT. Chúng ta coi thử các định dạng thông dụng

* **D3DFMT\_R8G8B8** : 24 bit và 8 bit cho mỗi màu
* **D3DFMT\_X8R8G8B8** : 32 bit trong đó có 8 bit đầu ko sử dụng
* **D3DFMT\_A8R8G8B8** : 32 bit trong đó 8 bit cho Alpha (độ trong suốt) còn lại cho 3 màu
* **D3DFMT\_A16B16G16R16F** : 64 bit trong đó mỗi thành phần là 16 bit
* **D3DFMT\_A32B32G32R32F** : 128 bit trong đó mỗi thành phần 32 bit

3 lựa chọn đầu thường xuyên đc sử dụng. (Sử dụng hay không nên phụ thuộc vào thiết bị đồ họa).

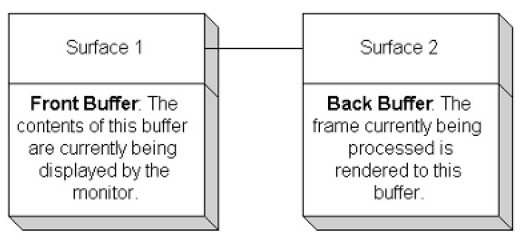
**Memory Pools**

Surface và Resource và tất cả các thành phần khác có thể lưu trữ trong memory pool. Nó được định nghĩa bằng enum D3DPOOL và các trị của nó gồm

* **D3DPOOL\_DEFAULT** : Đây là tùy chọn mặc định cho phép Direct3D lưu trữ ở một nơi hợp lý nhất (do DirectX tự xử lý). Chú ý là memory phải đc giải phóng trước khi IDirect3DDevice9::Reset được gọi và phải nạp lại memory sau đó
* **D3DPOOL\_MANAGED** : Resource đc quản lý bởi Direct3D, có nghĩa là Direct3D có thể chuyển đổi hoặc lựa chọn vùng nhớ nào tùy nó
* **D3DPOOL\_SYSTEMMEM** : Resource lưu trong bộ nhớ của hệ thống
* **D3DPOOL\_SCRATCH** : giống như cái trên, nhưng nó ko hạn chế như cái trên. (chưa hiểu rõ lắm)

**The Swap Chain and Page Flipping**

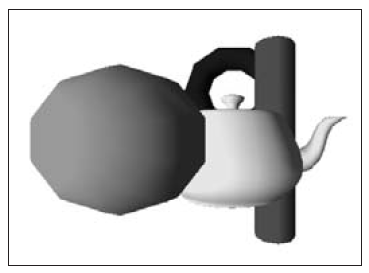
Direct3D duy trì một tập hợp các surface để thực hiện swap chain (nếu chúng ta sử dụng kỹ thuật này), tập hợp surface này đc thể hiện bằng **IDirect3DSwapChain9** nhưng mà chúng ta hiếm khi đụng tới nó (vì dùng double buffering là đủ). Swap chain and Page Fipping là kỹ thuật giúp cho hoạt cảnh di chuyển mượt mà và nhanh hơn



Front Buffer sẽ được hiển thị trên màn hình, việc hiển thị sẽ được dựa vào số frame (ví dụ chuẩn 60 frame một giây). Thực ra ta có thể vẽ front buffer ra màn hình hơn 60 lần 1 giây, nhưng chúng ta muốn cho hình ảnh đc hiển thị đầy đủ để người dùng coi và cũng ko mún số khung hình bị giảm xuống. Do đó chúng ta tận dụng sức mạnh của thiết bị đồ họa để vẽ lên các offscreensurface, sau đó các offscreensurface vẽ lên backbuffer (vì chúng ta ko thao tác front surface). Sau đó BackBuffer sẽ đổi chỗ cho FrontSurface, tức là BackBuffer bây giờ thành FrontSurface và ngược lại cho frame tiếp theo (kỹ thuật flipping coi cụ thể ở dưới).

**Depth Buffers**

Depth Buffer cũng là 1 surface nhưng nó không chứa dữ liệu hình ảnh mà chứa thông tinh chi tiết về 1 pixel cụ thể. Có một entry (không biết dịch sao) trong depth buffer mà nó mô tả cụ thể điểm ảnh khi được hiển thị ra màn hình. Như vậy thì nếu ảnh ở độ phân giải 640x480 thì depth buffer sẽ có 640x480 extries

 hình này cho thấy một cảnh đơn giản đối tượng bị che bởi đối tượng khác. Để làm điều này thì Direct3D quy định pixel nào sẽ nằm trên, kỹ thuật ở đây gọi là ***depth buffering hay z-buffering*.**

Depth Buffering tính toán độ sâu của mỗi pixel rồi đem so sánh với các kết quả còn lại, kết quả là điểm nào gần camera nhất sẽ được hiển thị (tức là che ảnh phía sau nó).

Định dạng của Depth Buffering sẽ ảnh hưởng tới chất lượng, độ chính xác trong hiển thị chiều sâu. Điều đó có nghĩa 24bit thì sẽ chính xác hơn 16bit. Thường thì người ta hay sử dụng 24bit mặc dù card đồ họa có thể hỗ trợ tới 32bit

* **D3DFMT\_D32** : 32bit
* **D3DFMT\_D24S8** : 24bit và 8bit cho stencil
* **D3DFMT\_D24X8** : 24bit
* **D3DFMT\_D24X4S4** : 24bit và 4bit cho stencil
* **D3DFMT\_D16** : 16 bit

**Vertex Processing**

Chúng ta tạm ko dịch chữ Vertex sang tiếng việt. Xử lý vertex có 2 loại, dùng CPU hoặc GPU, có nghĩa là software hay hardware, hardware còn phải tùy card có hỗ trợ hay ko, tuy nhiên software thì luôn luôn hỗ trợ Vertex Processing. Do đó chọn Software luôn luôn là ổn, tuy nhiên nếu card màn hình có xử lý Vertex thì nên thực hiện trên nó, vì nó sẽ nhanh hơn rất nhiều và tạo thuận lợi cho CPU xử lý các thuật toán khác. (Hầu hết nói tới card màn hình hiện nay thì đều hỗ trợ Vertex Processing)

Lưu ý : Một cách khác để biết card màn hình có hỗ trợ Vertex Processing ko là coi nó có hỗ trợ **transformation and lighting calculations** ko

**Device Capabilities**

Sau khi theo dõi các vấn đề ta đã đặt ra ở trên, ta sẽ hỏi làm các nào để có thể kiểm tra xem thiết bị đồ họa mà ta sử dụng có hỗ trợ các chức năng đó hay ko. Câu trả lời là khả năng của thiết bị đồ họa sẽ được thể hiện trong structure **D3DCAPS9.** Chúng ta nên xem trong SDK xem tên các thành phần cần kiểm tra để có thể làm việc với chúng, sau đây là một ví dụ cách sử dụng D3DCAPS9 để kiểm tra xem card đồ họa có hỗ trợ transformation and lighting calculations ko. Sau khi rà trong SDK ta sẽ thấy rằng bit **D3DDEVCAPS\_HWRASTERIZATION** trong data **DevCaps** xác định xem thiết bị đồ họa có hỗ trợ transformation and lighting (tức là vertex processing) hay ko, ta check như sau

bool supportsHardwareVertexProcessing;

// If the bit is “on” then that implies the hardware device

// supports it.

if( caps.DevCaps & D3DDEVCAPS\_HWTRANSFORMANDLIGHT )

{

// Yes, the bit is on, so it is supported.

supportsHardwareVertexProcessing = true;

}

else

{

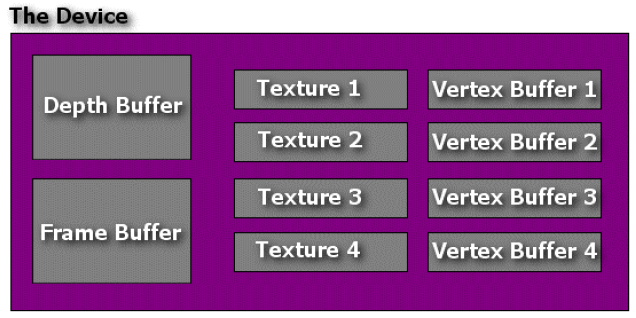
// No, the bit is off, so it is not supported.

hardwareSupportsVertexProcessing = false;

}

**Device Memory**

Device chứa và lưu trữ một số lượng dữ liệu quan trọng. The frame buffer (và depth buffer) sẽ được tạo ra cùng với device. Bộ nhớ của device cũng có thể dùng để sử dụng với texture image và mesh với các memory buffer tham chiếu tới device resources. Có những resource còn cho device khả năng tăng tốc trong T&L. Chúng ta coi mô hình của device



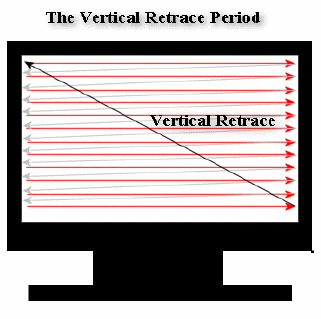
**Frame Buffer**

The Frame Buffer (đề cập ở trên) hay gọi là back buffer là 1 memory buffer nơi mà 3D scene được render trước khi đưa lên màn hình. Điều này tránh được artifact (tác động giả tạo) có thể xảy ra nếu chúng ta render trực tiếp ra màn hình. Giải thích về artifact (có nhiều kiểu lỗi như vậy nhưng ta sẽ nêu vấn đề chủ yếu sinh ra các lỗi này), ta biết là màn hình chính cần có động thái xóa hết image trước khi render 1 hình ảnh khác. Nếu chúng ta render trực tiếp lên màn hình thì sẽ vướng phải giai đoạn xóa màn hình và người dùng sẽ cảm nhận thấy sự flicker (chớp giật) trên màn hình. Frame buffer giải quyết hiện tượng trên bằng cách render image trong buffer lên trực tiếp thay thế cho image đang trên màn hình.

**Refresh Rate**

Refresh Rate đã đề cập trong Display Mode. Tốc độ repaint của màn hình được gọi là refresh rate (đo bằng đơn vị Hertz). Refresh rate 60 Hz có nghĩa là monitor repaint 60 lần 1s. Tốc độ càng cào thì tỉ lệ phản ứng với sự thay đổi hình ảnh càng nhanh và mướt. Thường sẽ có 1 khối lượng bộ nhớ trong video card để lưu trữ hình ảnh được đưa trực tiếp ra màn hình (phần frame buffer ở trên). Khi màn hình repaint bằng các electron … nó lấy thông tin từ display memory và thay đổi trên màn hình cho người xem thấy được hình ảnh mới.

Việc quét lại màn hình sẽ thực hiện từ trái sang phải từ trên xuống dưới cho tới khi toàn bộ màn hình được thay đổi



Nếu chúng ta render frame buffer lên màn hình khi mà monitor mới quét được phân nửa, chúng ta sẽ mất đi 1 phần bên dưới (phần cũ chưa kịp quét) chúng ta gọi hiện tượng này là visual artifact **tearing**.



Như vậy ta thấy là quyết định đưa frame buffer lên màn hình phải là lúc màn hình đã quét xong hết hình ảnh cũ. Khoảng thời gian màn hình quét ảnh gọi là **vertical retrace** (hay là **vertical blank**). Chúng ta phải bảo đảm card màn hình sẽ đẩy frame buffer lên không quá nhanh hơn vertical retrace để tránh bị hiện tượng lỗi trên. Chúng ta sẽ nói cho device để nó có thể đồng bộ với vertical retrace (để tránh lỗi). Điều này gọi là **VSYNC**. device sẽ đợi cho tới khi vertical retrace kêt thúc rồi mới repaint từ frame buffer.

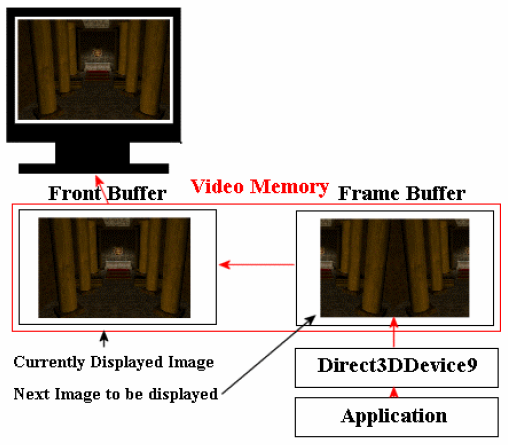
Chú ý là trong 1 vài game có tùy chọn bỏ chế độ VSYNC, điều này cho phép card đồ họa có thể render nhanh nhất có thế được mà không cần chờ vertical retrace (có nguy cơ sẽ xảy ra artifact).

**The Front Buffer**

Bộ nhớ được sử dụng để eletron gun repaint lại monitor được gọi là front buffer (khác với frame buffer). DirectX Graphic sẽ ép chúng ta phải dùng frame buffer và từ chối các lệnh truy cập vào front buffer (như vậy tất cả các thao tác là lên frame buffer hết). Khi mà frame buffer hoàn chỉnh thì húng ta gọi **IDirect3DDevice9::Present** để nói cho device copy và tiến hành đưa frame buffer lên display memory.

Tuy chúng ta không có quyền tác động tới Front Buffer nhưng ta có 1 hàm **IDirect3DDevice9::GetFontBuffer** để trả về 1 bản copy của image đang nằm trên font buffer. Điều này rất có lợi nếu ta mún lấy được hình ảnh đang ở trên màn hình (đây cũng là cách duy nhất để chụp màn hình).

Chúng ta coi hình mô tả mối quan hệ giữa Front Buffer và Frame Buffer với monitor.



**Swap Chain**

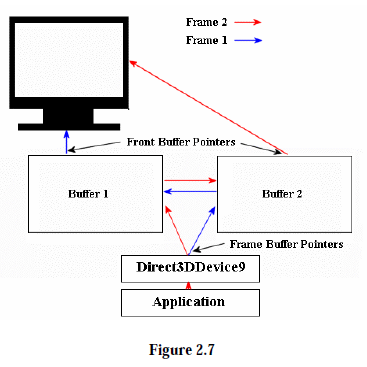
Chúng ta sẽ coi một cách nhìn khác về swap chain (cách nhìn lý thuyết hơn phần trên). Khi có nhiều hơn 1 frame buffer được sử dụng, ta gọi hiện tượng này la swap chain. Tưởng tượng rằng nếu device phải đợi quá trình vertical retrace hoàn thành để có thể đưa frame buffer lên và vẽ cái mới, ta có thể tận dụng thời gian chờ này để tạo thêm frame mới trong dãy swap chain. Điều này có thể làm tăng tốc độ trong những trường hợp nhất định và cho ta cảm giác mịn. Nhưng nó sẽ phải trả giá bằng một phần lớn bộ nhớ trong video card (nơi được sử dụng cho texture hoặc vertex buffer) nhất là với các độ phân giải cao (32 bit color)

Tiến trình sử dụng 2 frame buffer gọi là **triple buffering** (tính luôn cả front buffer) còn sử dụng 1 frame buffer được gọi là **double buffering** (được sử dụng thường xuyên hơn). DirectX cho phép swap chain với 4 frame buffer.

**FullScreen Mode**

Khi chúng ta khởi tạo device, chúng ta phải chọn chế độ fullscreen hay windowmode. Nếu chọn fullscreen, chúng ta phải kiểm tra fullscreen video mode mà video card hỗ trợ. Trong fullscreen mode, frame buffer được tạo ra cho device phải chính xác size và color depth như front buffer. Nếu chúng ta tạo ra device với 640x480 sử dụng 16 bit color thì frame buffer cũng nên được tạo với cùng đặc điểm như vậy (dùng các hàm để kiểm tra mode)

Trong Fullscreen Mode thì device có thể vẻ từ frame buffer ra front buffer nhanh nhờ kỹ thuật **flipping** (trong windowed mode ko hỗ trợ kỹ thuật này). Đây là kỹ thuật được hỗ trợ trong hầu hết các card đồ họa hiện nay. Ta xem trong trường hợp double buffering, video card có 2 pointer, 1 trỏ tới frame buffer và 1 trỏ tới front buffer. Monitor được repaint bởi con trỏ tới front buffer. Nguyên tắc hoạt động như sau, giả vụ như hiện tại front buffer pointer đang được render tren monitor và frame buffer pointer đang được vẽ lại. Sau khi frame buffer đã chuẩn bị xong và front buffer đã sẵn sàng repaint, lúc này front buffer pointer sẽ trỏ sang frame buffer và ngược lại, quá trình này diễn đi diễn lại một cách rất nhanh nơi mà tất cả các pixel được vẽ liên tục trên 2 buffer (có một kiểu khác là frame buffer copy pixel qua front buffer mỗi lần repaint, kỹ thuật này à **blitting** nhưng tốc độ chậm hơn flipping nhiều). Như vậy front buffer và frame buffer đổi vai trò cho nhau liên tục (việc này được DirectX thực hiện, chúng ta chỉ thao tác lên frame buffer)



**Windowed Mode**

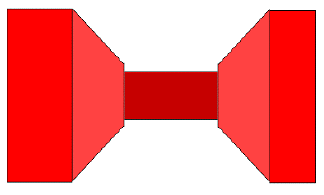
Trong windowed mode thì video mode chia sẽ với các ứng dụng khác. Cho nên video mode không được thay đổi (có nghĩa là nó cố định). Flipping không được sử dụng (because the front buffer is mapped directly to the client area of the application window). Do Flipping không dùng được nên frame buffer copy pixel qua front buffer mỗi lần repaint, tức là xài kỹ thuật Blitting nên chậm hơn so với fullscreen mode.

Việc di chuyển hay resize cửa sổ được DirectX Graphics thực thi, tuy nhiên nó sẽ không linh hoạt (do đó ta thấy nếu éo rộng cửa sổ thì màn hình bị vỡ nhiều)

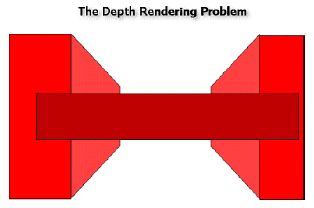
Trong Windowed Mode thì frame buffer không phải cần phải có mode chính xác với front buffer vì DirectX Graphic sẽ tự xử lý kích thước của frame buffer cho phù hợp. Kể cả color mode cũng được DirectX Graphic tự hiễu chỉnh cho phù hợp trong môi trường windowed mode.

**Depth Buffer**

Một vấn đề quan trọng trong 3D game là thể hiện được độ sâu của vật thể, nếu chúng ta render ở chế độ wireframe thì mõi thứ nhìn có vẻ không có vấn đề gì, tuy nhiên nếu chúng ta làm solid các vật thể thì phải có cái nhìn chính xác vật nào chồng lên vật nào, vật nào đặt trước vật nào để có thể hình dung được thế giới 3D của chúng ta. Chúng ta hãy coi 3 đối tượng sau

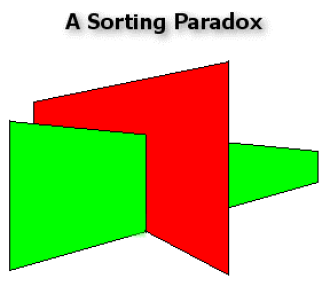


Hình trên mô tả một cách chính xác vị trí của 3 vật thể trong đó vật nằm giữa ở xa camera nhất. Việc thể hiện sẽ sai như sau



Ta thấy lúc này việc thể hiện độ sâu của môi trường 3D đã không còn chính xác nữa. Một kỹ thuật dùng để sữa lại hiện tượng trên gọi là **Painter algorithm**. Cách làm của nó là render vật ở xa camera trước rồi mới tới các vật ở gần, như vậy thì ta sẽ giải quyết được vấn đề ở trên

Tuy nhiên suy nghĩ kỹ thì kỹ thuật này vẫn chưa phải là hoàn chỉnh, nhất là đối với các đối tượng 3D phức tạp hiện nay. Ta coi 1 vấn đề bên dưới



Đối với hình trên thì ta khó lòng xác định được vật nào cần vẽ trước và vật nào vẽ sau. Do đó kỹ thuật Painter Algorithm rõ ràng không tối ưu cho trường hợp này.

Nhìn nhận trên từng pixel ta có thể thấy được các điểm ảnh nằm phía sau sẽ bị che khuất bởi các điểm ảnh nằm phía trước. Như vậy để vẽ được hình trên ta cần phải test các điểm ảnh xem phần nào nằm gần camera hơn thì sẽ được render.

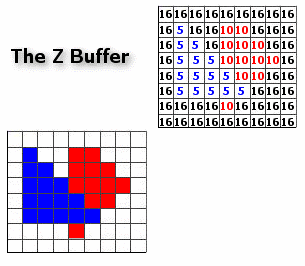
**The Z-Buffer**

Giải pháp phổ biến nhất để giải quyết vấn đề chiều sâu (depth) là tạo ra 1 memory buffer có cùng kích thước với frame buffer gọi là z-buffer. Thay vì mỗi vị trí của buffer lưu trữ các điểm ảnh (mà nhiệm vụ này đã có frame buffer làm), z-buffer nó sẽ lưu trữ độ sâu tương ứng của mỗi điểm ảnh trong frame buffer (cùng một vị trí nhất định)

**(Thiếu 1 phần nội dung liên quan tới math, cập nhật sau)**

Trước khi chúng ta render, chúng ta phải clear Z-buffer (còn với frame buffer thì ta clear target). Z-Buffer là một Byte array, do đó mỗi phần tử có thể giữ số từ 0-255, như vậy khi clear chúng ta sẽ cho mỗi phần tử giá trị 255

Tiếp theo khi chúng ta render polygon, sau hi polygon được transformed vào không gian, chúng ta tính toán thành phần Z lưu trong mỗi vertex của polygon. Sau khi chúng ta có được giá trị này, chúng ta so sánh nó với giá trị đang có trong z-buffer. Nếu nó gần camera hơn thì ta ưu tiên lưu trữ giá trị nó vào z-buffer (xóa giá trị cũ tại điểm đó đi), ngược lại thì ta ko cần lưu giá trị z. HÌnh sau mô tả z-buffer với 2 polygon



Ta thấy rõ ràng với hàng ngàn điểm ảnh cùng với rất nhiều các vật thể thì phải mất rất nhiều thời gian để tính toán cho z-buffer. May mắn là các card đồ họa đều có hỗ trợ tính toán Z-Buffer và DirectX Graphic sẽ hỗ trợ việc tính toán này tự động co chúng ta. Chúng ta chỉ việc khởi động Z-Buffer, bảo đảm rằng vertex có giá trị z … và mọi việc sẽ được DirectX làm với tốc độ cao.

**Z-Buffer Inaccuracy**

Card đồ họa có thể hỗ trợ 16, 24, 32 bit Z-Buffer (thường là 16 vì các định dạng khác khá nặng đối với card thông thường). Chúng ta sẽ nói qua một vài vấn đề có thể xảy ra với z-buffer định dạng 16 bit (còn 24, 32 thì chất lượng khá tốt).

Giá trị z cho mỗi vertex sẽ là kết quả của phép tính toán projection matrix multifly và chia cho w (coi phần math). Nó đưa cho vertex 1 độ sâu (depth) trong khoảng 0.0 đến 1.0. Trong code thì giá trị này là 32 bit. Nhưng khi sử dụng 16 bit z-buffer thì 2 byte float phải được giảm đi, do đó giá trị có thể sẽ không chính xác tuyệt đối như ta muốn (tất nhiên với 32 bit z-buffer thì rất chính xác).

Chúng ta coi ví dụ mô tả như sau

32 bit depth values:

Pixel A = 0.1025

Pixel B = 0.1029

16 bit truncated depth values:

Pixel A = 0.10

Pixel B = 0.10

Ta thấy là lúc này giá trị của pixel A và B là bằng nhau (nhưng thực tế nó khác nhau) Vì thế đôi khi 2 vật thể nằm ở độ sâu khác nhau nhưng lại trùng nhau khiến việc render ko chính xác. Tiếc rằng nếu card màn hình chỉ hỗ trợ Z-Buffer 16 bit thì ta ko có cách nào khắc phục lỗi này.

**(Còn phần math sẽ nghiêm cứu sau)**

**Surface Format**

1 surface là 1 object lưu trữ dữ liệu ảnh. Ví dụ cả frame buffer và depth buffer đều lưu trữ ở dạng surface. Texture cũng là dạng surface. Chúng ta có thể thao tác tới surface qua interface IDirect3Dsurface9. Surface có rất nhiều kích thước và color depths khác nhau và tất cả định dạng surface đều phụ thuộc vào sự hỗ trợ của hardware. Một trong những việc phải quan tâm khi khởi tạo liên quan tới surface là

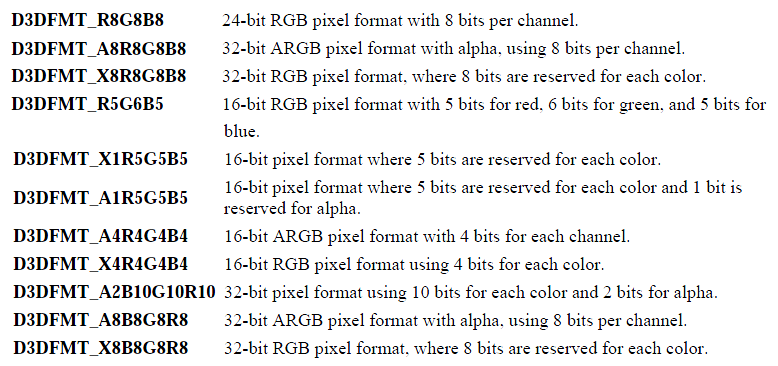
* Frame Buffer surface phải được tạo bởi device trong định dạng mà hardware hỗ trợ
* Chúng ta tạo 1 z-buffer surface mà hardware hỗ trợ
* Chúng ta load texture của chúng ta vào surface chứa định dạng và kiểu mà hardware support.

Như vậy tất cả các định dạng của surface phải được hardware hỗ trợ nếu chúng ta muốn sử dụng.

Các định dạng (format) mà hardware hỗ trợ nằm trong enum D3DFORMAT. Nhiều card hỗ trợ rất nhiều định dạng trong khi một số khác hỗ trợ rất ít.

Khi chúng ta tạo device lúc khởi động chướng trình, chúng ta phải nói kiểu format của frame buffer mà chúng ta muốn xây dựng. Kiểu format phải là 1 trong số kiểu mà hardware hỗ trợ và chúng ta phải bảo đảm frame buffer, depth buffer và texture được tạo ra phải nằm trong số các format được hỗ trợ.

Những kiểu format thông dụng.



**Adapter Format (Front Buffer)**

Khi chúng ta tạo ra fullscreen device chúng ta phải chọn một trong số định dạng (format) sau, tức là front buffer chỉ có thể là 1 trong số format sau

D3DFMT\_X1R5G5B5

D3DFMT\_R5G6B5

D3DFMT\_X8R8G8B8

Sẽ rất hữu ích khi chúng ta biết rằng tất cả các video card sẽ ít nhất hỗ trợ 1 trong 3 định dạng này. Chú ý là front buffer không thể sử dụng định dạng với alpha channel.

**Frame Buffer Format**

Khi ở chế độ Windowed mode thì application chia sẽ với desptop, do đó front buffer phải sử dụng format mà adapter đang dùng. Frame buffer format lúc này không cần thiết. Trong windowed mode thì format và resolution của frame buffer không cần phải phù hợp với adapter mode (front buffer). Lúc này thì device sẽ tự xử lý giữa frame buffer và front buffer.

Một vài kiểu định dạng mà ta sử dụng tron frame buffer

D3DFMT\_X1R5G5B5

D3DFMT\_R5G6B5

D3DFMT\_X8R8G8B8

D3DFMT\_A8R8G8B8

D3DFMT\_A1R5G5B5

D3DFMT\_A2R10G10B10

Chú ý là nếu sử dụng định dạng nào ở trên thì hardware phải hỗ trợ cái đó mới xài. Chúng ta sẽ không thường cần alpha chanel trong frame buffer. Chúng ta chỉ cần chọn cho front buffer và frame buffer phù hợp với nhau. (Chế độ format cuối hơi đặc biệt)

Ở chế độ fullscreen device, format và resolution của frame buffer và front buffer phải phù hợp. Alpha channel (“A” value) có thể phù hợp với “X” value. Ví dụ khi ta đặt định dạng fron buffer là D3DFMT\_X8R8G8B8 thì frame buffer có thể là D3DFMT\_X8R8G8B8 hoặc D3DFMT\_A8R8G8B8.