Graphics Programming

Master Computer Game Development 2013/2014

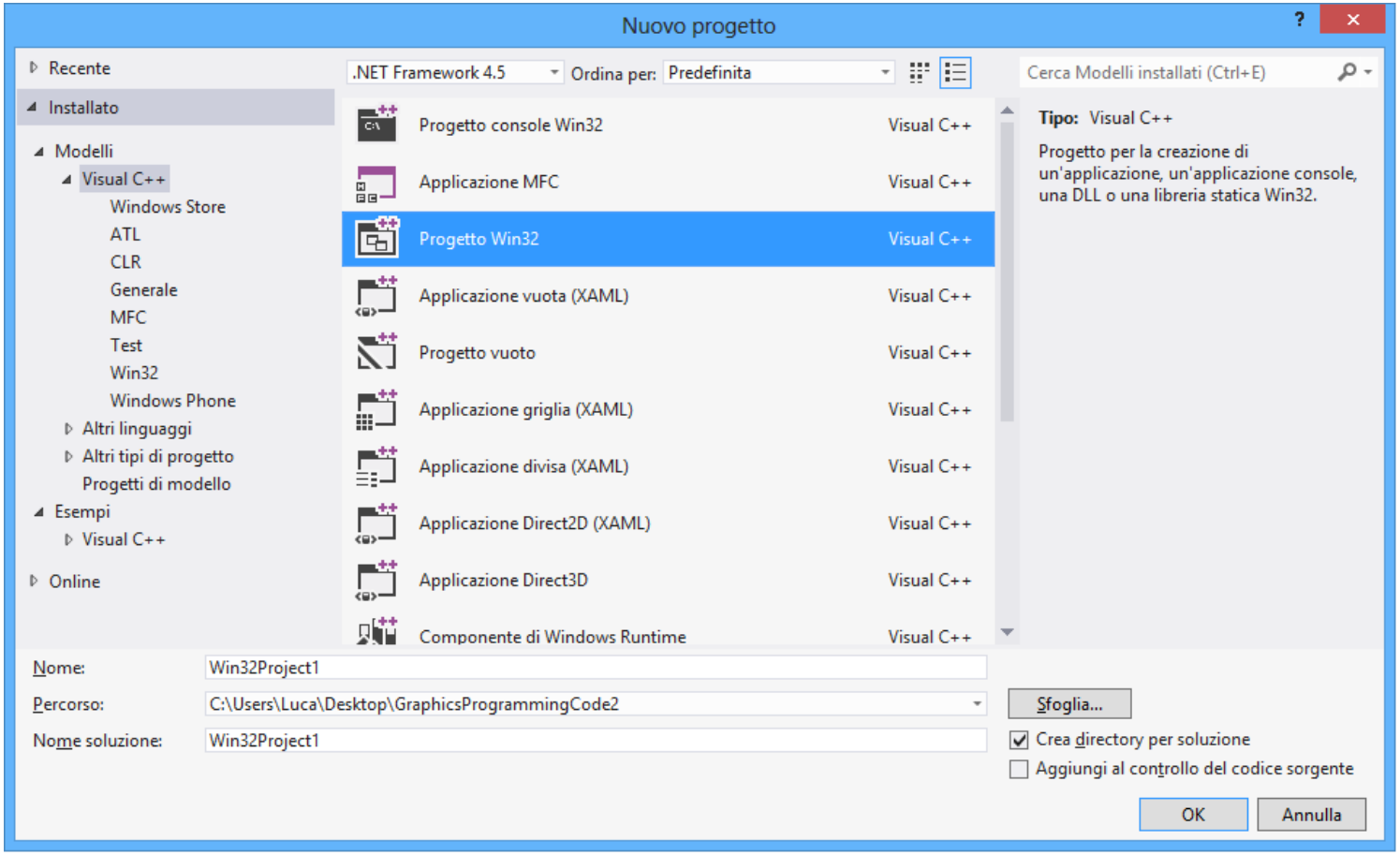
Introduzione a DirectX 11.

University of Verona

luca.giona@3dflow.net

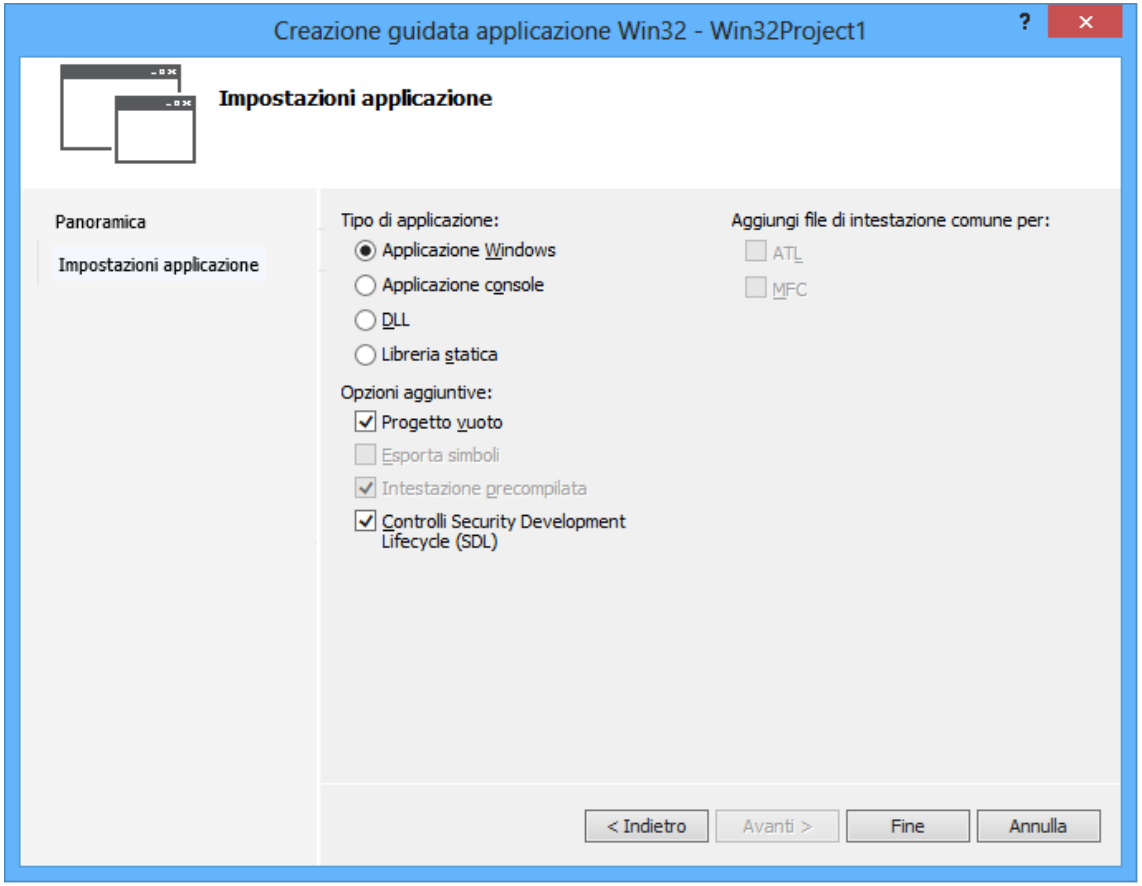
Impostare Visual Studio 2012

- Create un nuovo progetto Win32.



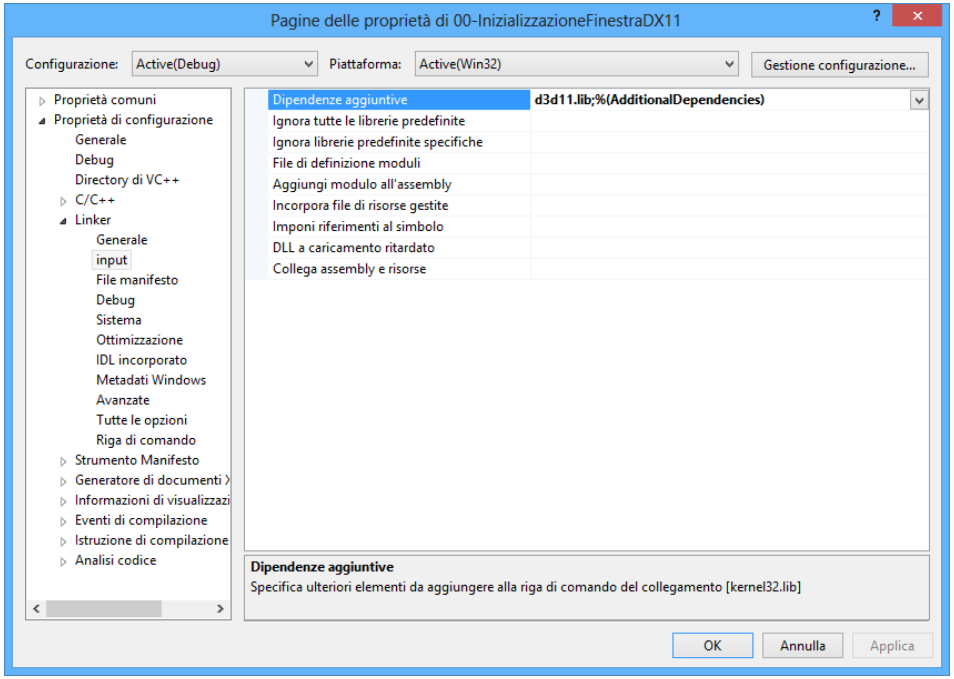
Impostare Visual Studio 2012

- Create un progetto vuoto.



Impostare Visual Studio 2012

- Aggiungete le librerie in Project Properties → Linker → Input → Additional Dependencies.
  - d3d11.lib sia per Debug che per Release



Fasi necessarie per inizializzare un programma DirectX11

1. Creare una finestra Win32:
   - Definire una funzione "*WndProc*" per gestire gli eventi della finestra.
   - Creare una finestra (con la "*WndProc*" come parametro).
   - Entrare nel loop ("infinito") dove si "catturano" I messaggi che arrivano dal sistema operativo e si lancia il redering di un frame.
2. Inizializare DirectX:
   - Creare un **ID3D11Device** e **ID3D11DeviceContext**.
   - Creare una **IDXSGISwapChain**.
   - Creare un *render target view* per il *back buffer*.
   - Creare un depth/stencil buffer e relativa view.
   - Collegare il *render target view* e la *depth/stencil view* alla swap chain.
   - Impostare un viewport.

Header

- Bisogna includere gli header con le definizione delle funzioni utilizzate per le API di windows e di DirectX 11
  - #include <windows.h>
  - #include <d3d11.h>

Programma Win32

- L'Entry point di un programma windows è
  - int WINAPI **WinMain**(HINSTANCE *hInstance*, HINSTANCE *hPrevInstance*, LPSTR *lpCmdLine*, int *nCmdShow*);
    - *hInstance Application Handler*
    - *hPrevInstance Handler padre*
    - *lpCmdLine Stringa linea comando*
    - *Modo di visualizzazione finestra (può essere SW_HIDE, SW_MAXIMAZE, SW_MINIMIZE, SW_RESTORE, SW_SHOW …)*

Finestra Win32: WNDCLASSEX

- Oggetto WNDCLASSEX contiene le informazioni sulla finestra.
- Dobbiamo dichiarare una classe WNDCLASSEX per la finestra, configurarne l'aspetto, e registrarla chiamando RegisterClassEx.

Finestra Win32: WNDCLASSEX (esempio)

```
WNDCLASSEX wcex;
wcex.cbSize = sizeof( WNDCLASSEX );
wcex.style = CS_HREDRAW | CS_VREDRAW;
wcex.lpfnWndProc = WndProc;
wcex.cbClsExtra = 0;
wcex.cbWndExtra = 0;
wcex.hInstance = hInstance;
wcex.hIcon = LoadIcon( hInstance, ( LPCTSTR )IDI_APPLICATION );
wcex.hCursor = LoadCursor( NULL, IDC_ARROW );
wcex.hbrBackground = ( HBRUSH )( COLOR_WINDOW);
wcex.lpszMenuName = NULL;
wcex.lpszClassName = L"TutorialWindowClass";
wcex.hIconSm = LoadIcon( wcex.hInstance, ( LPCTSTR )IDI_APPLICATION );

if( !RegisterClassEx( &wcex ) )
      return E_FAIL;
```

Finestra Win32: CreateWindow

- La finestra viene creata con il comando CreateWindow.

```
HWND CreateWindow( LPCTSTR lpClassName,
        LPCTSTR lpWindowName, DWORD dwStyle,
        int x, int y, int nWidth, int nHeight, HWND hWndParent,
        HMENU hMenu, HINSTANCE hInstance,
        LPVOID lpParam );
```

- E viene visualizzata invocando **ShowWindow**.
- *lpClassName* sarà uguale al nodel della classe di finestre appena definita ovvero L"TutorialWindowClass"

Finestra Win32: Message Loop

Il programma entra quindi nel main MessageLoop, dove gli eventi Windows (messaggi) vengono processati.

```
while( WM_QUIT != msg.message )
{
        if( PeekMessage( &msg, NULL, 0, 0, PM_REMOVE ) )
        {
                TranslateMessage( &msg );
                DispatchMessage( &msg );
        }
        else
        {
                Render();
        }
}
```

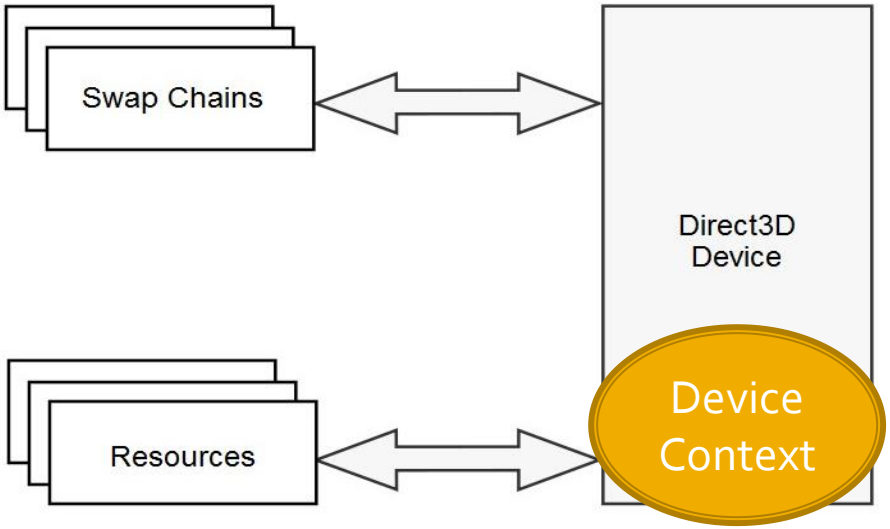
Finestra Win32: WndProc

- La coda dei messaggi viene gestita nella funzione :
  LRESULT CALLBACK **WndProc**(HWND hWnd,
  UINT message, WPARAM wParam, LPARAM lParam)

```
switch( message )
{
        case WM_PAINT:
                hdc = BeginPaint( hWnd, &ps );
                EndPaint( hWnd, &ps );
        break;
        case WM_DESTROY:
                PostQuitMessage( 0 );
        break;
        default:
                return DefWindowProc( hWnd, message, wParam, lParam );
}
```

Inizializzazione Direct3D 11

- Componenti principali:
  - **ID3D11Device** e **ID3D11DeviceContext**
    - Ha il compito di gestire l'applicazione DX11, gestisce tutte le risorse caricate e il (gli) swap chain.
  - **IDXGISwapChain**
    - Gestisce il front buffer e il(i) back buffer
  - Risorse (saranno viste nella prossima lezione)



Swap Chain

- Gestisce front e il (i) back buffer.
- Il front buffer è di sola lettura ed è ciò che viene "spedito" allo schermo.
- I back buffer sono invece scritti e vengono "aggangiati" al merger della pipeline.
- La pipeline scrive in un back buffer; quando ha finito di scrivere il frame corrente, il back buffer diventa front buffer e viene mostrato a schermo (double buffering)
- Il Front e il back buffer sono immagini ovvero matrici bidimensionali rappresentate come risorse *Texture2D*

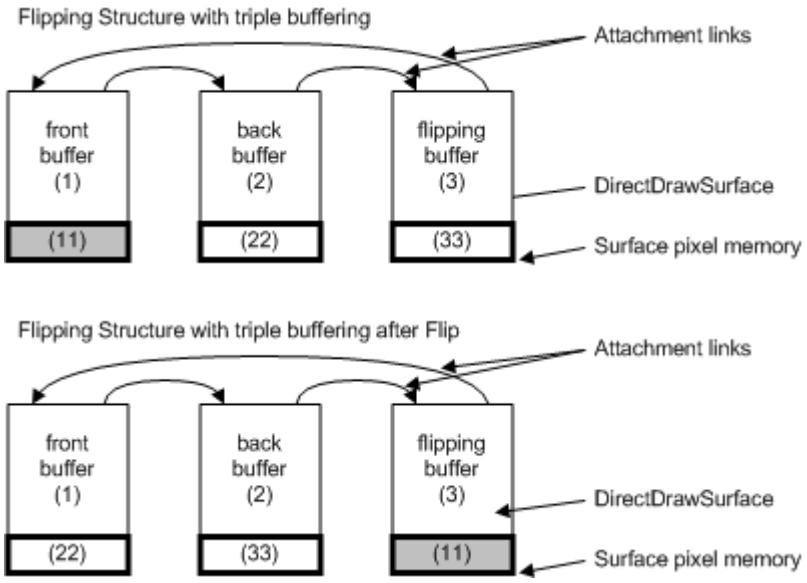
DXGI_SWAP_CHAIN_DESC

- Inizializzazione parametri Swap Chain

```
DXGI_SWAP_CHAIN_DESC sd;
ZeroMemory( &sd, sizeof( sd ) );
sd.BufferCount = 1; // Numero di back buffer
sd.BufferDesc.Width = width;
sd.BufferDesc.Height = height;
sd.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM; // Formato RGBA a 32bit
sd.BufferDesc.RefreshRate.Numerator = 60; // Refresh rate schermo
sd.BufferDesc.RefreshRate.Denominator = 1; // Fattore di divisione refresh rate (intero)
sd.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT; // Utilizzo del backbuffer:
                                                  il rendering viene scritto qui
sd.OutputWindow = g_hWnd; // handler finestra
sd.SampleDesc.Count = 1; // Serve per il multisampling - in questo caso viene disabilitato
sd.SampleDesc.Quality = 0;
sd.Windowed = TRUE; // Modalità fullscreen/windows
```

Triple buffering

- A che cosa potrebbe servire più di un back buffer?
  - Triple buffering: la scheda video non aspetta che i buffer siano swappati e può passare nel frattempo a iniziare il rendering del buffer successivo guadagnando tempo.



D3D11CreateDeviceAndSwapChain

- Device e Swap Chain possono essere inizializzati assieme invocando

```
HRESULT D3D11CreateDeviceAndSwapChain(
    __in IDXGIAdapter *pAdapter,
    __in D3D_DRIVER_TYPE DriverType,
    __in HMODULE Software,
    __in UINT Flags,
    __in const D3D_FEATURE_LEVEL *pFeatureLevels,
    __in UINT FeatureLevels,
    __in UINT SDKVersion,
    __in const DXGI_SWAP_CHAIN_DESC *pSwapChainDesc,
    __out IDXGISwapChain **ppSwapChain,
    __out ID3D11Device **ppDevice,
    __out D3D_FEATURE_LEVEL *pFeatureLevel,
    __out ID3D11DeviceContext **ppImmediateContext );
```

D3D_DRIVER_TYPE

- D3D11_DRIVER_TYPE_HARDWARE -> Utilizza solo le funzionalità supportate direttamente dalla scheda video.
- D3D11_DRIVER_TYPE_REFERENCE -> funzionalità non supportate dalla scheda video, emulate (in modo lento).

Feature Level

- E’ possibile interrogare a tempo di creazione l’adattatore video selezionato per chiedere il set di funzioni supportato ed eventualmente adattare il programma in modo dinamico.

Feature Level

- D3D_FEATURE_LEVEL_9_1
  - Targets features supported by Direct3D 9.1 including shader model 2.
- D3D_FEATURE_LEVEL_9_2
  - Targets features supported by Direct3D 9.2 including shader model 2.
- D3D_FEATURE_LEVEL_9_3
  - Targets features supported by Direct3D 9.3 including shader shader model 3.
- D3D_FEATURE_LEVEL_10_0
  - Targets features supported by Direct3D 10.0 including shader shader model 4.
- D3D_FEATURE_LEVEL_10_1
  - Targets features supported by Direct3D 10.1 including shader shader model 4.
- D3D_FEATURE_LEVEL_11_0
  - Targets features supported by Direct3D 11.0 including shader shader model 5.

Render Target

- Fra il back buffer dello swap chain e la pipeline vi è un altra componente importante: il Render Target.
- Un oggetto Render Target è un contenitore per l'output dello stadio merger della pipeline.
- Dobbiamo:
  - creare un render target con formato e collegarlo al back-buffer dello swap chain.
  - Fare in modo che l'output della nostra pipeline venga scritto su questo Render Target.

Render Target

- Otteniamo il back buffer come texture2D

```
ID3D11Texture2D* pBackBuffer;
hr = g_pSwapChain->GetBuffer( 0,
                              __uuidof( ID3D11Texture2D ), ( LPVOID* )&pBackBuffer );
```

- Creiamo l'oggetto renderview e colleghiamolo (bind) al back buffer, ovvero alla sua texture.

```
hr = g_pd3dDevice->CreateRenderTargetView( pBackBuffer,
                                           NULL, &g_pRenderTargetView );
```

OMSetRenderTargets

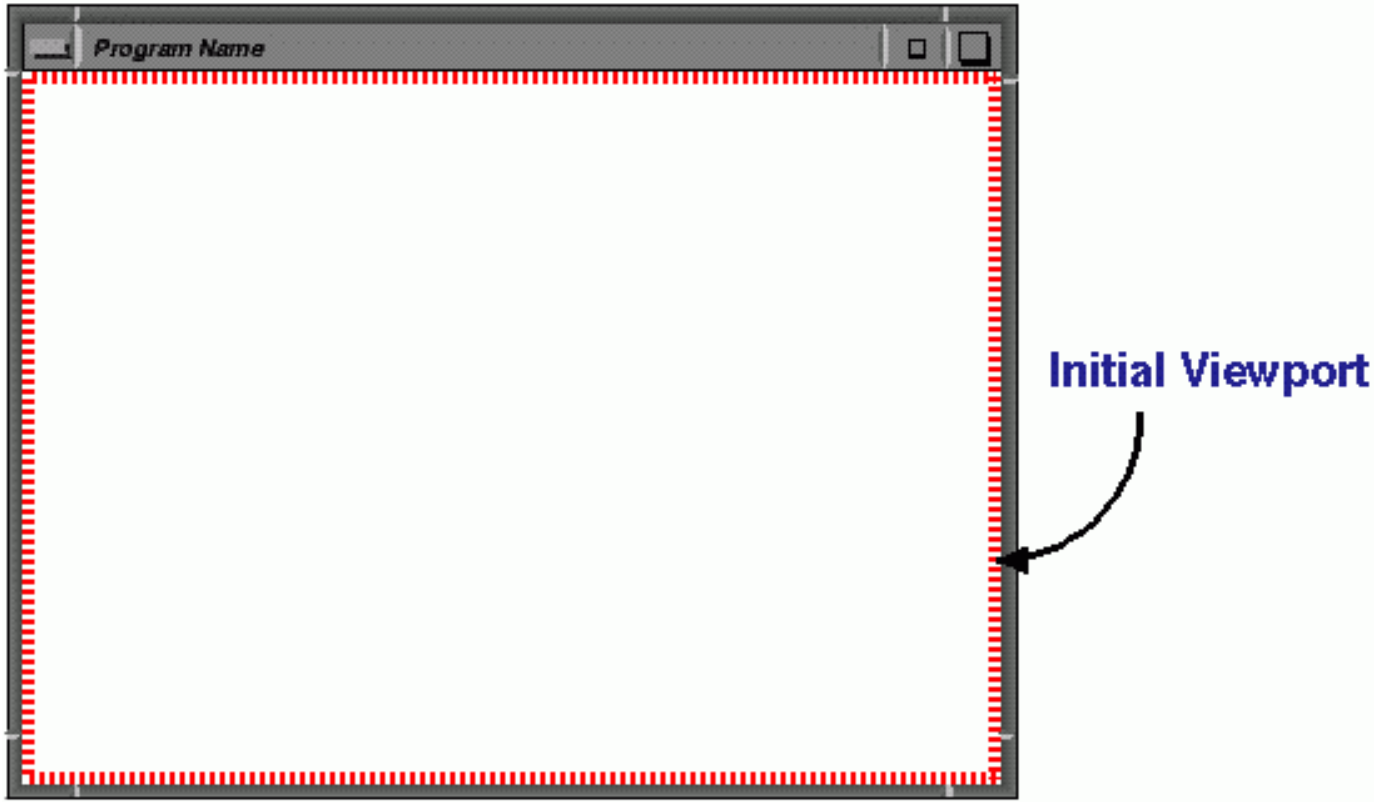
- Infine, facciamo in modo che l'output del merger della pipeline scriva sul rendertarget appena definito.

```
g_pd3dDeviceContext >OMSetRenderTargets( 1, // Numero di render target
                                         &g_pRenderTargetView, // puntatore al back buffer
                                         NULL ); // puntatori al depth stencil buffer
```

- A cosa potrebbe servire definire più di un render target?
  - E' possibile spedire in output qualsiasi tipo di informazione dai pixel shader su più buffer, da usare per esempio per effetti che richiedono più passi di rendering.

Viewport

- Il viewport definisce la porzione di finestra in cui si effettua il rendering.



Viewport

- Tecnicamente, dopo l'applicazione delle trasformazioni di modeling-viewing e projection, il nostro mondo sarà compreso in un cubo unitario compreso tra [-1 e 1].*

  *Precisamente DX mappa x e y in [-1 1] e z in [0 1]

- Dopo il clipping, ciò che resta la porzione di viewport selezionata viene scritta nel rendertarget corrente.

Viewport

- I Viewport sono sempre rettangolari e vengono definiti da una posizione e da una altezza ed ampiezza.
  - La posizione è definita in un sistema di riferimento che ha lo (0,0) nel vertice in alto a sinistra, l'asse x positivo che punta verso destra e l'asse y positivo che punta verso il basso.
  - Attenzione: In DX10/11 a differenza di DX9 e OpenGL, nessun viewport iniziale è settato di default.

D3D11_VIEWPORT

- Parametri di un viewport settabili con D3D11_VIEWPORT .

```
// Setta il viewport
  D3D11_VIEWPORT vp;
  vp.Width = width;
  vp.Height = height;
  vp.MinDepth = 0.0f;
  vp.MaxDepth = 1.0f;
  vp.TopLeftX = 0;
  vp.TopLeftY = 0;
```

- Possiamo specificare anche oggetti a profondità specifica da visualizzare. Con MinDepth < MaxDepth e in [0 1].
- Infine dobbiamo associare il viewport al rendertarget corrente chiamando RSSetViewports dal device selezionato.

```
g_pd3dDeviceContext ->RSSetViewports( 1, &vp)
```

Rendering

- Dopo aver inizializzato con successo la finestra d3d11, possiamo effettuare il rendering di scene ad ogni frame.

```
// Main message loop
MSG msg = {0};
while( WM_QUIT != msg.message )
{
        if( PeekMessage( &msg, NULL, 0, 0, PM_REMOVE ) )
        {
                TranslateMessage( &msg );
                DispatchMessage( &msg );
        }
        else
        {
                Render();
        }
}
```

Render()

- Render() verrà eseguita ad ogni frame.
- Nella nostra funzione render ci andranno tutte le operazioni per fare il rendering della scena.
- In generale dovremo:
  - Pulire i buffer (per ora solo il back buffer) ad ogni ciclo
  - …. Modeling & rendering ….
  - Fare lo swap fra back e front buffer alla fine di ogni frame

Pulizia frame buffer

- Per ora l'unico buffer presente sarà il frame buffer.
- Dobbiamo ripristinare il colore dello sfondo del rendertarget corrente.
  - Chiamata a ClearRenderTargetView del device context corrente
  - Il colore in input è di tipo RGBA.

```
void ClearRenderTargetView(  [in] ID3D11RenderTargetView *pRenderTargetView,
                             [in] const FLOAT ColorRGBA[4] );
```

Swap buffer

- Alla fine di ogni frame dovremo scambiare il back buffer con il front buffer in modo da renderlo visibile.
- Ricordiamo che l'output viene scritto nel back buffer. Al termine dello swap quello appena renderizzato sarà visibile, mentre nel back buffer effettueremo il rendering del prossimo frame
- Per effettuare lo swap si deve chiamare Present dal swap chain corrente:

```
HRESULT Present( [in] UINT SyncInterval, [in] UINT Flags );
```

Pulizia generale

- Tutti gli oggetti DirectX (risorse, device, swap buffers…) sono oggetti COM (Component Object Model).
- Gestione semiautomatica della memoria con Smart Pointers.
  - Ogni oggetto contiene il numero di reference (puntatori che si riferiscono a lui).
  - Ogni volta che un puntatore viene distrutto il numero viene decrementato.
  - Quando tale numero è uguale a 0, l'oggetto viene distrutto.
  - NON bisogna distruggere (free/delete) MAI un oggetto direttamente, solo rilasciare il puntatore.

Pulizia Generale

- Come rilasciare un oggetto puntato?
  - Tramite il comando **Release()**.
  - Ogni risorsa dovrebbe essere rilasciata quando non più utilizzata da un puntatore. Esempio:

```
if( g_pRenderTargetView )
        g_pRenderTargetView->Release();
if( g_pSwapChain )
        g_pSwapChain->Release();
if( g_pd3dDevice )
        g_pd3dDevice->Release();
```

Esempio 00

- Esempio 00: Mettendo assieme quanto detto possiamo far partire il nostro primo programma DirectX 11!*

  *(ma quante operazioni!)

Prossimamente

- Nei prossimi esempi utilizzeremo una semplice libreria scritta appositamente per il corso che nasconde la gestione della finestra.
  - Per creare i vostri progetti potete aggiungere un progetto alla solution degli esempi.
- Questo ci eviterà di addentrarci nelle Windows API (complicate) e di concentrarci sulle DirectX.