Graphics Programming

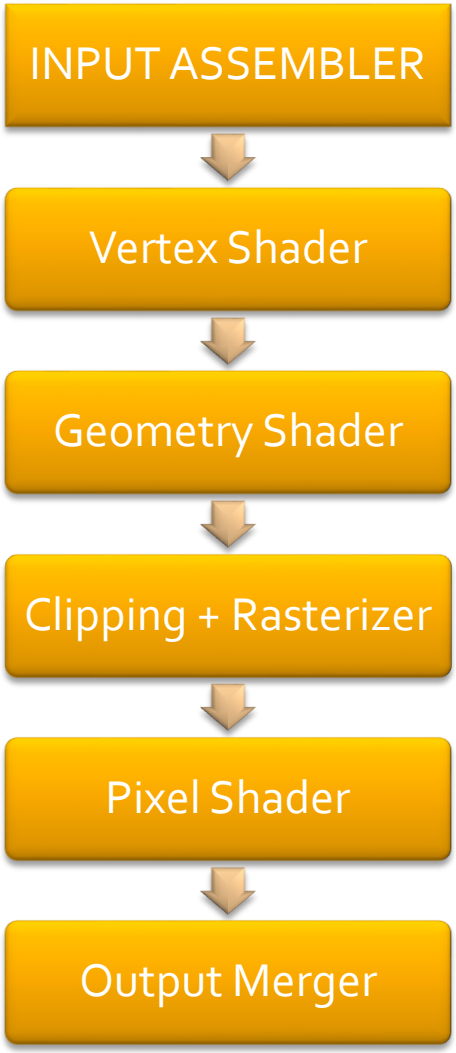
Master Computer Game Development 2013/2014

Risorse in DX11

University of Verona

luca.giona@3dflow.net

Pipeline DirectX 11 (semplificata)



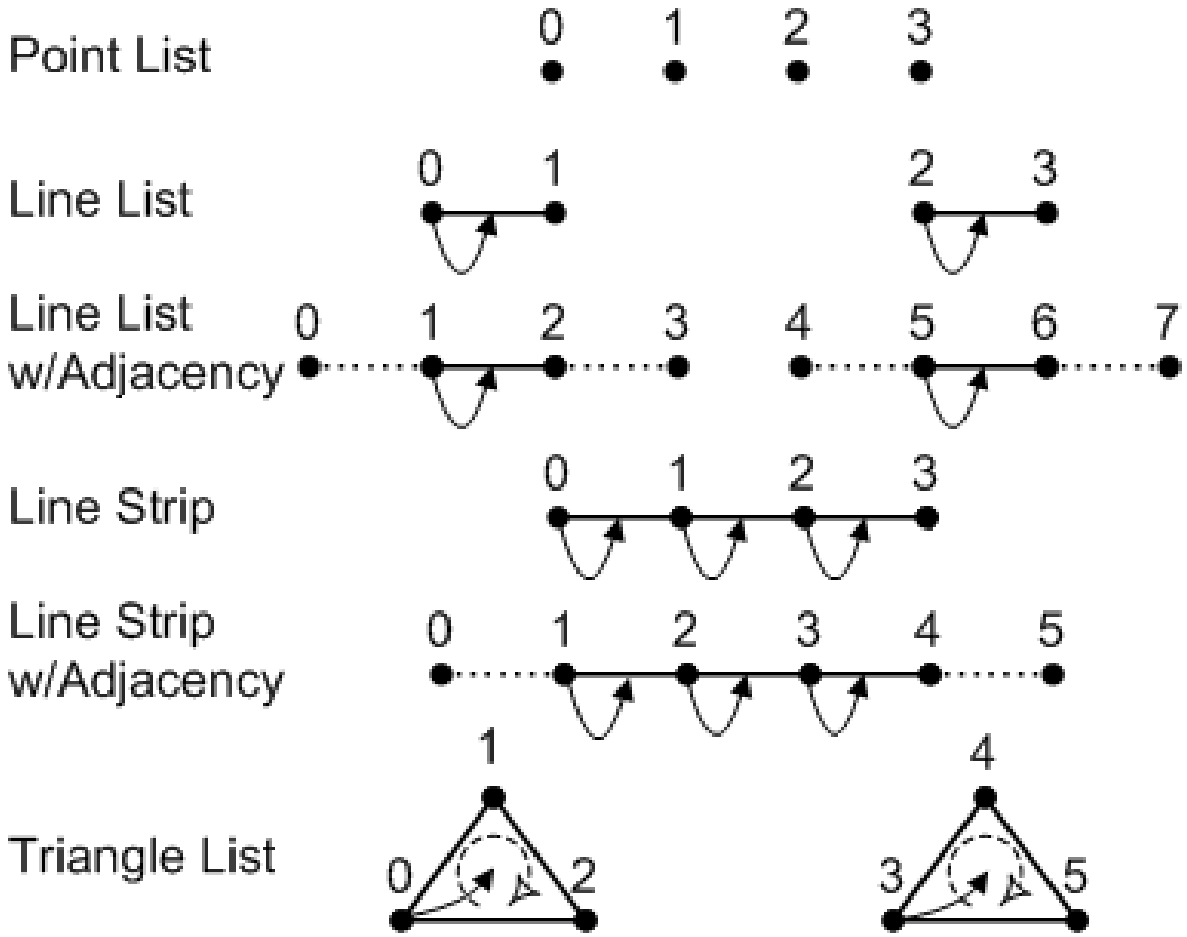
Differenza da DX9

- Con DirectX 9 era possibile utilizzare step prefissati configurabili al posto degli opzionali vertex e pixel shaders.
- Con DirectX 10/11, gli shader devono essere specificati (ad eccezione del Geometry Shader e degli stadi di tessellazione che sono opzionali).
- Anche per fare il rendering di un solo triangolo, dovremo specificare un vertex e un pixel shader associato.

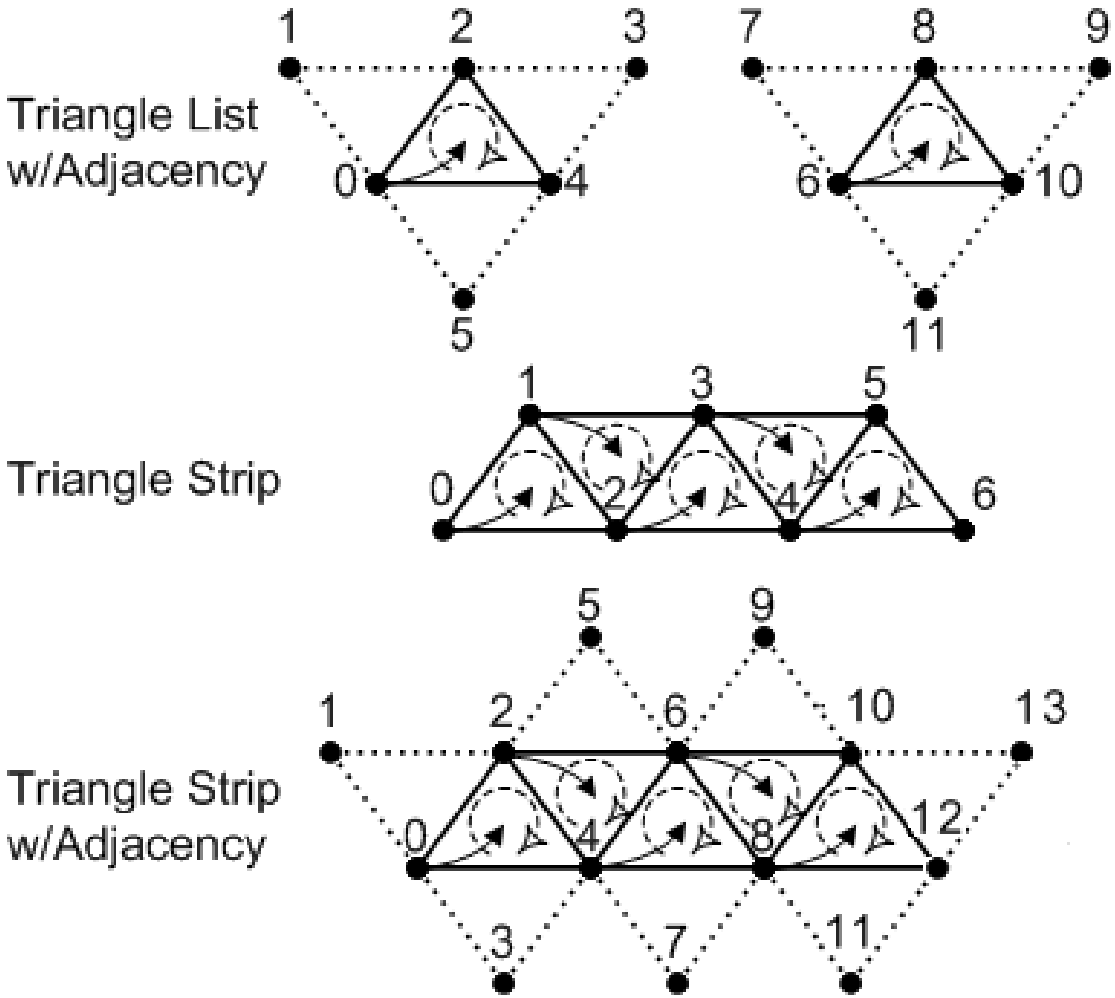
Input Assembler

- E' responsabile del trasferimento dei dati dalla memoria al Vertex Shader. Può avere accesso a 16 buffer per i vertici e uno per gli indici.
- Le regole di trasferimento sono codificate in un oggetto di tipo Input Layout.
- Oltre a buffer per vertici e indici, l'Input Assembler deve conoscere il tipo di primitive in cui sono organizzati i vertici passati.

Primitive disponibili



Primitive disponibili (2)



Primitive disponibili

- Esistono due versioni per ogni primitive (eccetto per i punti), con e senza informazione sui vertici adiacenti.
- L'informazione sui vertici adiacenti può essere utilizzata dal geometry shader.

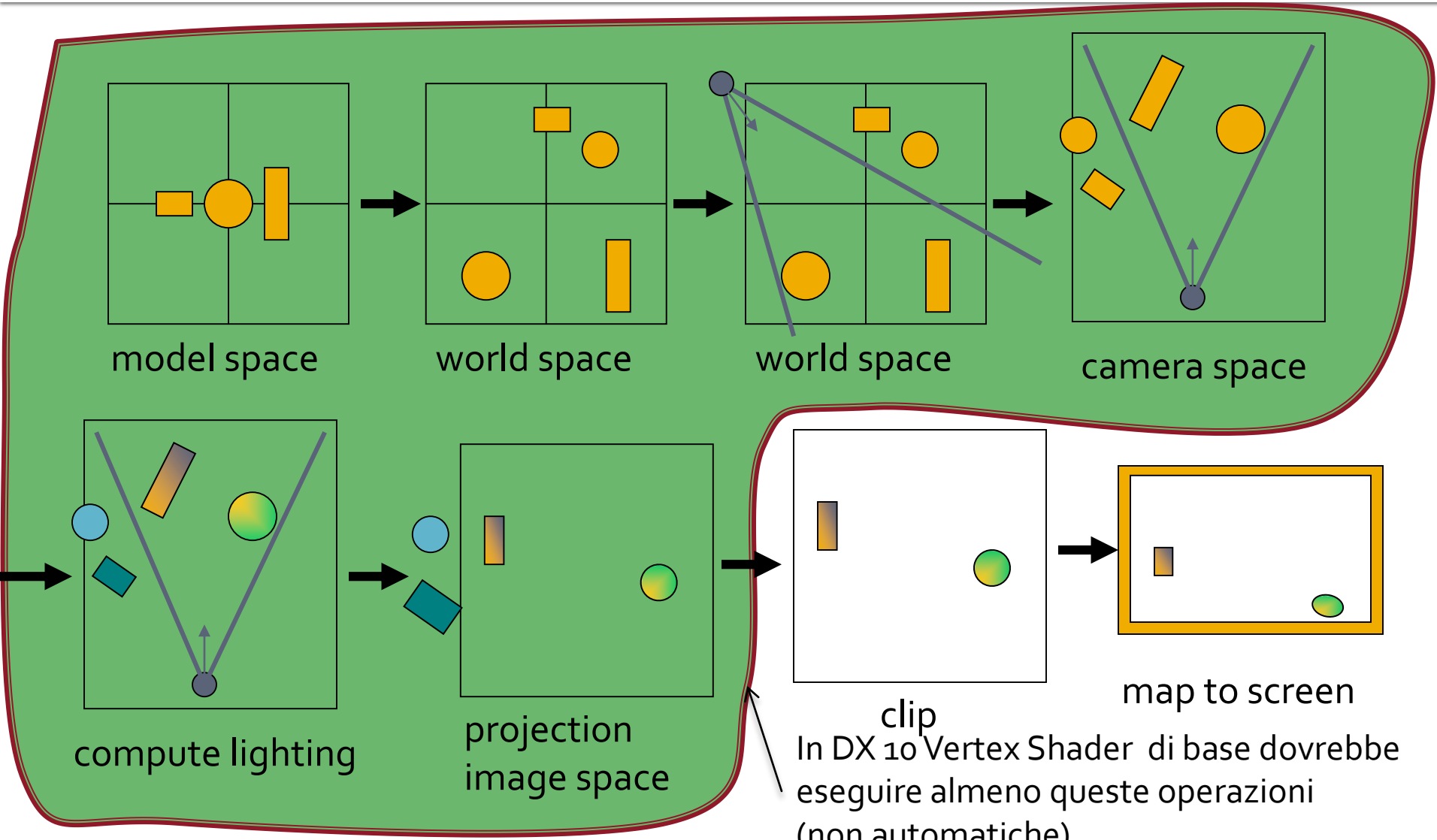
Vertex Shader

- Primo stadio programmabile della pipeline
- Funzione definita in HLSL che prende in input un singolo vertice con relative proprietà.
- Non può accedere ai vertici vicini.
- Possono essere passati parametri:
  - *Varying:* variabili all'interno della draw Call.
    e.g. Array per posizione e array per colore vertici.
  - *Costant*: costanti per tutti i vertici di una DrawCall
  - *Textures*: matrici uni-bi-tridimensionali read only e costanti per tutti i vertici di una DrawCall.
- Come suggerisce il nome, effettua operazioni sui vertici.

Vertex Shader

- Operazioni basilari di un vertex shader:
  - ModelView Transform
  - Vertex Shading
  - Projection
- Ovvero il comportamento almeno di base sarà effettuare le trasformazioni di modeling-viewing-projection sui vertici e calcolare il colore/proprietà per vertice.
- Queste operazioni in DX10/11 non vengono eseguite automaticamente!

Ricordando il Geometry stage



Ouput Vertex shader

- L'output del vertex shader dovrà essere:
  - La posizione del vertice trasformate nel projection image space (obbligatorio)
  - Altre proprietà quali colore/normale/coordinate texture (opzionalmente).

Esecuzione del Vertex Shader

- Dato un set di vertici (*inputVertices)*, l'esecuzione della pipeline del vertex shader (*VertexShader*) puo essere vista come il seguente codice:

```
for( int i = 0; i < inputVertices.size(); ++i )
        outputVerices[ i ] = VertexShader( inputVertices[ i ] );
```

- I vertici in input vengono forniti dall'input assembler, mentre i vertici in outuput vengono passati allo stage successivo.
- Il numero di verici in output è uguale al numero di vertici in input, il vertex shader non può scartarne nessuno.

Geometry Shader

- Il geometry shader viene eseguito subito dopo il vertex shader.
- E' opzionale
- La funzione di shading prende in input una primitiva (triangolo/linea..).
- Input: primitiva → Output: primitive
- Può eventualmente creare nuove primitive o distruggere quelle esistenti.
- Effettua operazioni sulle primitive.

Esecuzione del Geometry Shader

- Analogamente al vertex shader, l'esecuzione può essere vista come:

```
for( int i = 0; i < inputPrimitives.size(); ++i )
        ouputPrimitives.append( GeometryShader( inputPrimitives[i].verices ) );
```

- Il tipo di primitiva in output può essere diversa da quella in input.

Clipping e Rasterizer

- La fase di clipping viene eseguita in automatico. (elimina i vertici che cadono fuori dal cubo di coordinate x = [-1,1], y = [-1 1], z = [0, 1])
- La fase di rasterizer comprende Map to Screen, Triangle Setup e Triangle Traversal.
  - Sono eseguite in automatico e non sono parametrizzabili.

Storia di un vertice

- Ricapitolando:
  - I vertici vengono passati allo shader che produrrà in output i vertici proiettati nel "projection image space"
  - Clipping (se non $-1 < x < 1$, $-1 < y < 1$, $0 < z < 1$).
  - Per ogni triangolo verrà interpolato il valore interno (e.g. Normali, Colore) di ogni frammento proiettato in un pixel del piano immagine a partire dai vertici.

Pixel Shader

- In OpenGL si chiama Fragment Shader. Il nome sarebbe più corretto, perchè stiamo processando frammenti di triangoli proiettati su pixel, non pixel finali (ovvero più frammenti possono proiettare sullo stesso pixel).
- Input: dati interpolati sui vertici.
- Possiamo passare eventuali textures e costanti.
- Output: colore pixel (frammento) attuale.
- Nota: la funzione di pixel shader può restituire il colore finale solo del pixel attuale, non dei vicini.
- Non può nemmeno accedere ai valori dei vicini (eccezione per funzioni predefinite come calcolo gradiente).

Pixel shader (funzioni avanzate)

- Addizionalmente il pixel shader può anche modificare il valore dello Z-Buffer (ad un costo!).
- Possibilità di output multiplo (multiple render targets), ovvero non solo nel frame buffer corrente.
  - Perchè? I pixel shader possono svolgere molte operazioni complesse, può risultare utile avere output multipli per diverse operazioni svolte nel pixel shader.
  - Output = al numero di RenderTarget -> gli output andranno a scrivere sui diversi RenderTarget (massimo 8)

Esecuzione del Pixel Shader

- I pixel shader viene eseguito per ogni fragment generato dal Rasterizer.
- Il pixel shader può scartare i fragment e non restituire nulla.

```
for( int i = 0; inputFragments.size(); ++i )
{
        outputFragments.push_back( PixelShader( inputFragment[i] ) );
}
```

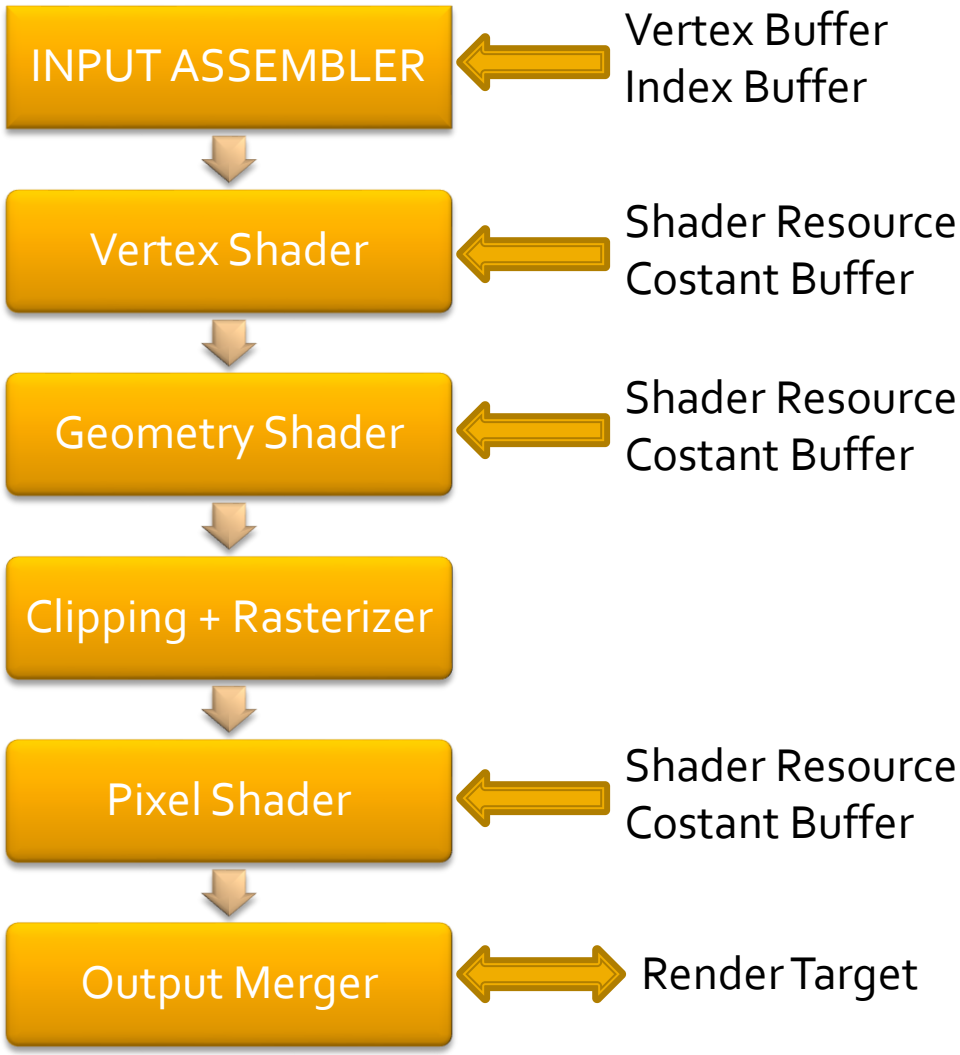
Merging stage

- Come discusso in precedenza, il merging stage è dove vengono applicate tecniche quali Z-Buffer, Stencil Buffer e l'Alpha Blending.
- E' configurabile, ma non programmabile.
- Superati gli eventuali test (z/stencil) il fragment viene scritto nel back buffer (subendo un eventuale alpha blending).

Risorse

- Introducendo la pipeline DX11, abbiamo menzionato svariati dati mandati in output o presi in input nei i vari stadi della pipeline.
- Abbiamo già implicitamente introdotto una risorsa quando abbiamo parlato del Render Target.

Pipeline DX10: Risorse Dove si collegano?



Formato risorse

- Per ogni risorsa dobbiamo definire un formato, ovvero come sono organizzate in memoria i dati in essa contenuti.
- Il formato della risorsa sono dichiarate in maiuscolo con la keyword DXGI_FORMAT_ seguita da simboli che ne descrivono il tipo:
  - Ad esempio un buffer di vertici conterrà un insieme di vertici a 3 canali float da 32 bit: DXGI_FORMAT_R32G32B32_FLOAT.

Prefissi per i canali

- R Canale rosso (o coordinate x)
- G Canale verde (o coordinate y)
- B Canale blu (o coordinate z)
- A Canale Alpha (o coordinate w) – (RGBA classico vettore[0-3])
- D Informazione profondità
- S Stencil buffer Data
- X Inutilizzato (padding)

Formati

- TYPELESS Non specificato.
- FLOAT Floating point.
- UINT Intero non segnato.
- SINT Intero segnato.
- UNORM Spazio normalizzato in [0, 1]
- SNORM Spazio normalizzato in [-1,1]
- UNORM_SRGB Normalizzato in [0, 1], utilizzando uno spazio di colore sRGB non lineare.

Utilizzo della risorsa

- In fase di creazione, dovremo indicare obbligatoriamente il tipo di utilizzo di una risorsa.
- In questo modo DX11 potrà ottimizzarla a seconda dell'utilizzo specificato e inserirla nella memoria più opportuna.

Utilizzo della risorsa

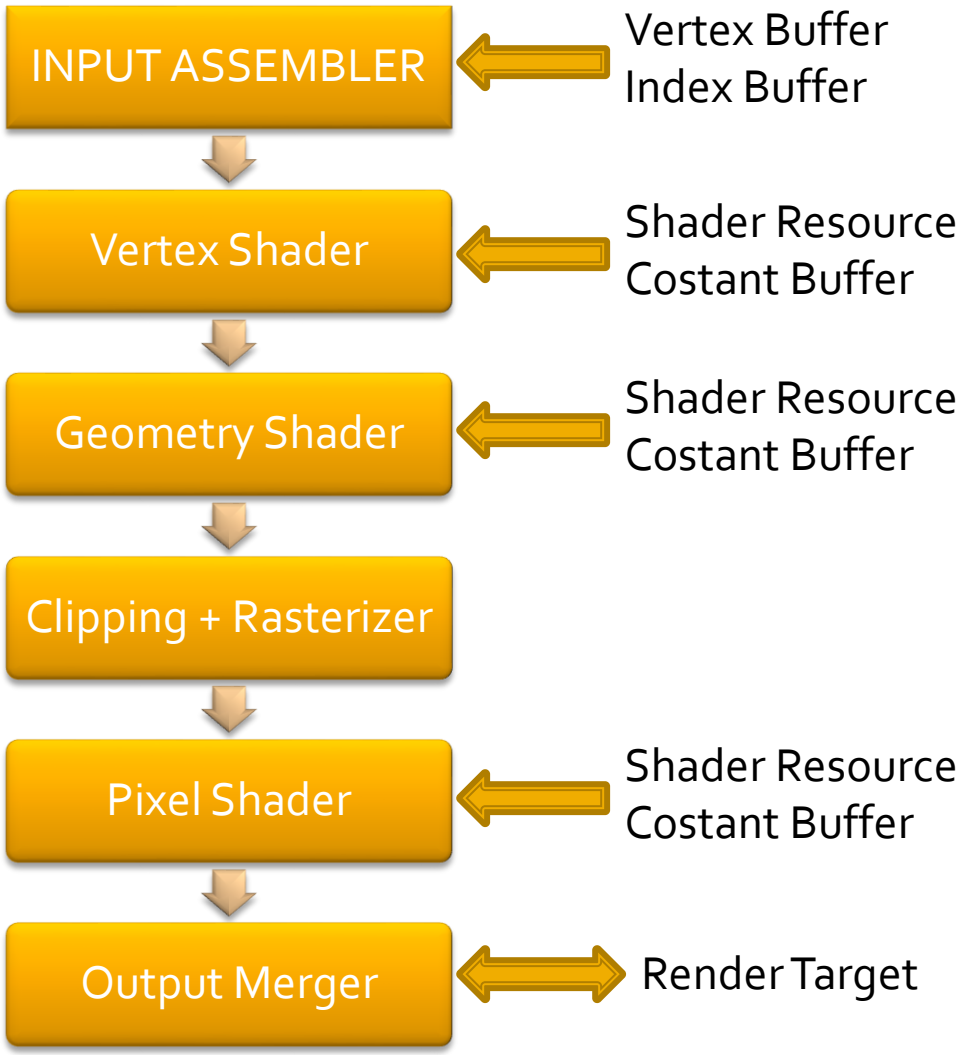
- D3D11_USAGE_IMMUTABLE
  - Risorsa mai modificata dopo la creazione.
    - Accesso CPU: Nessuno
    - Accesso GPU: Lettura
- D3D11_USAGE_DEFAULT
  - Una risorsa che cambia il valore al più una volta per frame
    - Accesso CPU: Nessuno
    - Accesso GPU: Lettura / Scrittura

Utilizzo della risorsa

- D3D11_USAGE_DYNAMIC
  - Risorsa modificata varie volte all'interno dello stesso frame.
    - Accesso CPU: Scrittura
    - Accesso GPU: Lettura
- D3D11_USAGE_STAGING
  - Risorsa utilizzata per trasferire dati da e verso la GPU
    - Accesso CPU: Lettura / Scrittura
    - Accesso GPU: Solo copia

Tornando alla pipeline

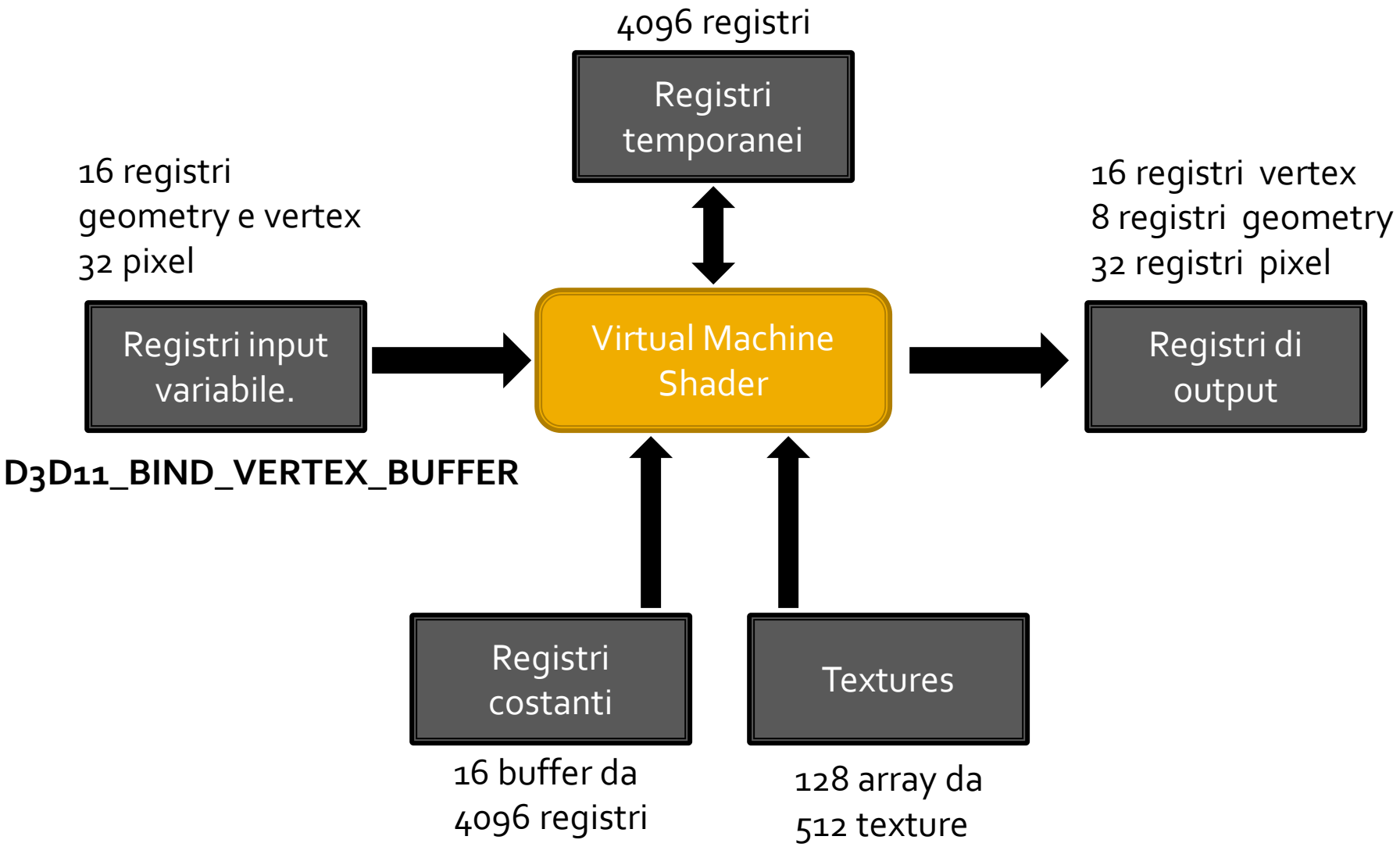
- Abbiamo detto che possiamo collegare le varie risorse ai vari stadi della pipeline -> Va specificato in fase di creazione della risorsa.
- Tali collegamenti prendono il nome di BIND.



Resource Binding (Input assembler)

- D3D11_BIND_VERTEX_BUFFER
  - Slot: 16
  - Accesso: sola lettura
  - Set-Method: ID3D11DeviceContext::**IASetVertexBuffers**
  - Get- Method: ID3D11DeviceContext::**IAGetVertexBuffers**
- D3D11_BIND_INDEX_BUFFER
  - Slot: 1
  - Accesso: sola lettura
  - Set-Method: ID3D11DeviceContext::**IASetIndexBuffer**
  - Get- Method: ID3D11DeviceContext::**IAGetIndexBuffer**

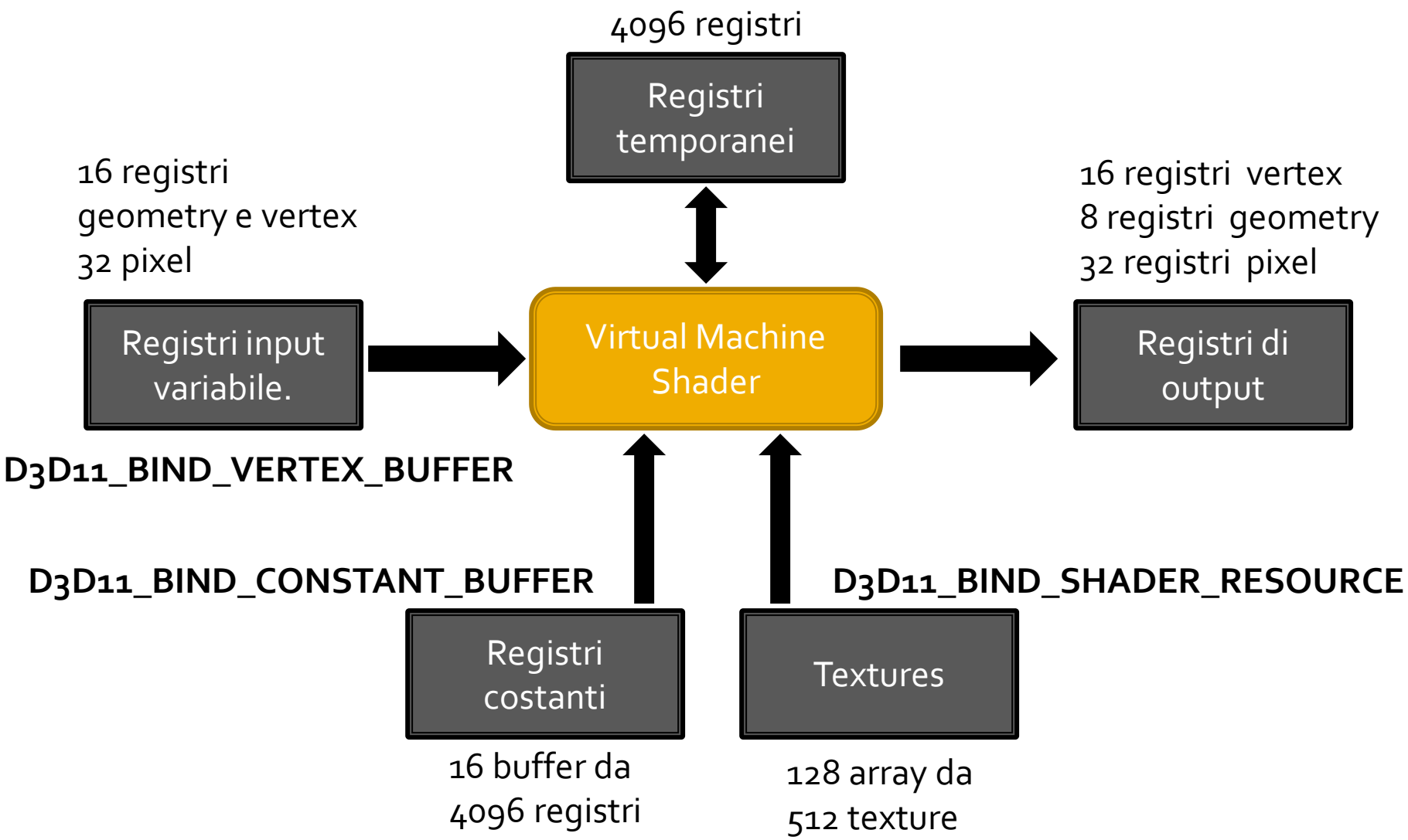
Unified Architecture



Resource Binding (Shaders)

- D3D11_BIND_CONSTANT_BUFFER
  - Slot: 16
  - Accesso: sola lettura
  - Set-Method: ID3D11DeviceContext ::**XXSetConstantBuffers**
  - Get- Method: ID3D11DeviceContext ::**XXGetConstantBuffers**
  - XX = VS, vertex shader; GS, geometry shader; PS, pixel shader
- D3D11_BIND_SHADER_RESOURCE
  - Slot: 128
  - Accesso: sola lettura
  - Set-Method: ID3D11DeviceContext ::**XXSetShaderResources**
  - Get- Method: ID3D11DeviceContext ::**XXGetShaderResources**
  - XX = VS, vertex shader; GS, geometry shader; PS, pixel shader

Unified Architecture



Resource Binding (Output merger)

- D3D11_BIND_RENDER_TARGET
  - Slot: 8
  - Accesso: lettura-scrittura
  - Set-Method: ID3D11DeviceContext::**OMSetRenderTargets**
  - Get- Method: ID3D11DeviceContext::**OMSetRenderTargets**
- D3D11_BIND_DEPTH_STENCIL
  - Slot: 1
  - Accesso: lettura-scrittura
  - Set-Method: ID3D11DeviceContext::**OMSetRenderTargets**
  - Get- Method: ID3D11DeviceContext::**OMSetRenderTargets**

Limitazione binding per utilizzo di risorsa

	Default	Dynamic	Immutable	Staging
Index Buffer	OK	OK	OK	**ERROR**
Vertex Buffer	OK	OK	OK	**ERROR**
Constant Buffer	OK	OK	OK	**ERROR**
Shader Resource	OK	OK	OK	**ERROR**
Depth Stencil	OK	**ERROR**	**ERROR**	**ERROR**
Render Target	OK	**ERROR**	**ERROR**	**ERROR**

Tipi di risorse

- Fondamentalmente le risorse passate alla pipeline si dividono in due categorie:
  - Buffer (Vertex-Index-Costant Buffers)
  - Textures (Shader Resources) – le tratteremo più avanti

Buffer

- I buffer sono dei semplici blocchi di memoria senza sottorisorse.
- In fase di creazione bisogna definire la dimensione totale.
- Inizializzato con la funzione CreateBuffer che prende in input un oggetto di tipo D3D11_BUFFER_DESC con la descrizione su tipo di risorsa e eventualmente il valore iniziale (obbligatorio se di tipo immutable)

Buffer

- L'eventuale inizializzazione del buffer, va passata come secondo argomento di CreateBuffer che prende in input D3D11_SUBRESOURCE_DATA.

```
typedef struct D3D11_SUBRESOURCE_DATA {
        const void *pSysMem;  // Puntatore ai dati
        UINT SysMemPitch;  // Usati per texture 2D e 3D
        UINT SysMemSlicePitch; // Usato solo per texture 3D.
} D3D11_SUBRESOURCE_DATA;
```

Buffer

- Il buffer sarà memorizzato nel programma in un oggetto di tipo ID3D11Buffer che va passato come ultimo argomento di CreateBuffer .
- Se passiamo NULL come ultimo argomento di CreateBuffer non darà alcun errore ma non creerà alcuna risorsa.

```
HRESULT CreateBuffer( [in] const D3D11_BUFFER_DESC *pDesc,
    [in] const D3D11_SUBRESOURCE_DATA *pInitialData,
    [out] ID3D11Buffer **ppBuffer );
```

Buffer (esempio)

- Supponiamo di aver definito:

```
struct SimpleVertex
{
        XMFLOAT3 Pos;
};

SimpleVertex vertices[] =
{
        XMFLOAT3( 0.5f, 0.0f, 0.5f ),
        XMFLOAT3( -0.5f, 0.0f, 0.5f ),
        XMFLOAT3( 0.0f, 0.5f, 0.5f ),
};
```

XMFLOAT3 è un semplice array di [3] valori

Buffer (esempio)

- Codice per creare un buffer:

```
D3D11_BUFFER_DESC bd;
bd.Usage = D3D11_USAGE_DEFAULT;
bd.ByteWidth = sizeof( SimpleVertex ) * 3; // Dimensione
bd.BindFlags = D3D11_BIND_VERTEX_BUFFER;  // Binding all'INPUTASSEMBLER
bd.CPUAccessFlags = 0; // La cpu non ha accesso in scrittura.
bd.MiscFlags = 0; // nessun flag addizionale

ID3D11Buffer*        pVertexBuffer = NULL;

D3D11_SUBRESOURCE_DATA InitData;
InitData.pSysMem = vertices;

HRESULT hr = pd3dDevice->CreateBuffer( &bd, &InitData, &pVertexBuffer );
```

Buffer

- Con il codice precedente possiamo creare un vertex buffer nel device, ma non abbiamo ancora effettuato alcun binding, ovvero non lo stiamo collegando direttamente alla pipeline per essere utilizzato.
  - Per effettuare il binding bisogna richiamare **IASetVertexBuffers** ma prima…
  - **IASetVertexBuffers** richiede che sia specificato un layout, ovvero deve sapere come sono organizzati i dati in input e come deve passarli al vertex shader.

Layout

- Un oggetto di tipo layout mi descrive i tipi di dati sono presenti nel vertex buffer passato all'input assembler.
- L'input assembler sarà deputato a prendere tali dati e passarli al vertex shader.
- Con l'oggetto layout viene effettuato anche un controllo: i dati che passiamo in input devono essere compatibili con quelli ad input variabile che richiede il vertex shader.

Elementi Layout

- L'oggetto layout viene descritto da un D3D11_INPUT_ELEMENT_DESC che è un array di strutture. Ciascuna struttura (riga) contiene una descrizione per una proprietà passata ai vertici dal vertex buffer.
- Quali valori contiene ogni riga?
  - SemanticName
    - Nome (semantico) dell'input. Libertà di scelta, ma si consigliano nomi significativi ad esempio POSITION se stiamo dando in input le posizioni dei vertici, COLOR il colore, TEXCOORDS per le coordinate texture.
  - SemanticIndex
    - Utilizzato se abbiamo due input con lo stesso nome. Ad esempio due colori per ogni vertice, al posto di nominare le variabili con COLOR0 e COLOR1, diamo lo stesso SemanticName e le differenziamo con il SemanticIndex a 0 o a 1.

Elementi Layout

- Format
  - Il formato dell'input come descritto in precedenza.
- InputSlot
  - Abbiamo detto che vi sono vari slot per il vertex buffer (16 per la precisione). Possiamo decidere in quale slot mettere l'input attuale settando InputSlot.
- AlignedByteOffset
  - Posizione iniziale dell'input corrente
  - Notate che è possibile "impacchettare" più di un attributo in uno slot. Settando AlignedByteOffset > o possiamo indirizzare alla posizione corretta.

Elementi Layout

- InputSlotClass
  - Può essere D3D11_INPUT_PER_VERTEX_DATA o D3D11_INPUT_PER_INSTANCE_DATA . Di solito è D3D11_INPUT_PER_VERTEX_DATA se non utilizziamo l'instanziamento.
- InstanceDataStepRate
  - Usato per l'instancing, per ora settiamo sempre a 0.

Layout (esempio)

- Quindi se vogliamo passare solo un array di posizione dei vertici al vertex shader, dovremo scrivere un layout del tipo:

```
D3D11_INPUT_ELEMENT_DESC layout[] = {
        {
                L"POSITION",
                0,
                DXGI_FORMAT_R32G32B32_FLOAT,
                0,
                0,
                D3D11_INPUT_PER_VERTEX_DATA,
                0
        },
};
```

Layout

- Cosa ci manca?
- In DX11 dobbiamo specificare almeno un vertex e un pixel shader anche per un semplice programma.

Shaders (esempio base)

- Nessuna variabile (vedremo più avanti)
- Vertex Shader:

```
float4 SimpleVertexShader( float3 pos:POSITION ) : SV_POSITION
{
        return float4(pos, 1.0f);
}
```

- Pixel Shader:

```
float4 SimpleVertexShader( float4 pos : SV_POSITION ) : SV_TARGET
{
        return float4( 1.0f, 1.0f, 0.0f, 1.0f );    //Yellow, with Alpha = 1
}
```

Shaders

- Vedremo gli shader più in dettaglio in seguito, vi dovrebbe risultare comunque facile capire la semplice funzione svolta da quelli appena definiti.
- Un Pixel Shader (di solito) non ha senso di esistere senza un Vertex Shader associato e viceversa.

Creazione Layout

- Il layout viene memorizzato in un oggetto ID3D11nputLayout e viene creato invocando da device:

```
HRESULT CreateInputLayout(
        const D3D11_INPUT_ELEMENT_DESC *pInputElementDescs,  // Array di descrizione del Layout
        UINT NumElements,  // Numero elementi di input
        const void *pShaderBytecodeWithInputSignature,  // Input Signature dello shader
        SIZE_T BytecodeLength,  // Lunghezza input signature
        ID3D11InputLayout **ppInputLayout  // Oggetto input-Layout
        );
```

Creazione Layout (esempio)

- Il codice diventa:

```
hr = pd3dDevice->CreateInputLayout(  layout, numElements,
                                     PassDesc.pIAInputSignature, PassDesc.IAInputSignatureSize,
                                     &g_pVertexLayout );
```

- Come risultato, il nostro layout sarà stato creato e caricato nell'oggetto **ID3D11InputLayout *g_pVertexLayout .**

Back-face Culling

- I triangoli/poligoni vengono definiti con una faccia davanti e una dietro.
- Di default in DirectX la faccia davanti è quella per cui i vertici sono visibili in senso *orario.*
  - In OpenGL il verso della facce davanti è *antiorario*
- Alle facce dietro, viene applicato il culling, ovvero non viene effettuato il rendering.
- Tale processo prende il nome di Back-Face Culling.
- Vedremo come impostare arbitrariamente il Back-Face Culling più avanti, per ora ricordatevi che di default vengono renderizzate a schermo solo le facce visibili in senso orario.

Ricapitolando

- Dobbiamo creare in oggetti appropriati:
  - Un vertex buffer contenente in un'unico slot le posizioni dei vertici (ID3D11Buffer).
  - Un vertex e pixel shader (ID3D11VertexShader, ID3D11PixelShader)
  - Un oggetto Layout che mi descrive il tipo di input passato all'input assembler (ID3D11InputLayout).
- Attenzione: Tutte queste fasi devono avvenire una sola volta all'inizio, non ad ogni frame!

Rendering

- Ora che tutto ciò che ci occorre è stato creato. Dobbiamo effettuare il rendering collegando(bind) il buffer all'input assembler, associando gli shaders correnti al device e descrivendo l'input con il layout opportuno.
- Importante: Il device può avere un solo vertex buffer, un solo layout, e una sola coppia di vertex/pixel shader associata alla volta. Il binding deve essere effettutato ogni frame, per ogni oggetto / shaders utilizzata.

Binding e Rendering (collegamento layout)

- Il layout è associato richiamando IASetInputLayout dal device:

```
pd3dDevice->IASetInputLayout( pVertexLayout );
```

Binding e rendering (collegamento Vertex Buffer)

- Il (I) Vertex Buffer corrente viene collegato nella pipeline richiamando IASetVertexBuffers da device context:

```
void IASetVertexBuffers( UINT StartSlot,  // Elemento iniziale array buffer
                         UINT NumBuffers, // Numero di elementi buffer da caricare
                         ID3D11Buffer *const *ppVertexBuffers,  // Puntatore buffer
                         const UINT *pStrides, // Dimensione elementi array buffer
                         const UINT *pOffsets  // Offset iniziale elementi array
                         );
```

Binding e Rendering (collegamento Vertex Buffer)

- Nel nostro esempio diventa:

```
UINT stride = sizeof( SimpleVertex );
UINT offset = 0;
pd3dDevice->IASetVertexBuffers( 0, 1, &g_pVertexBuffer, &stride, &offset );
```

Binding e Rendering (definizione primitiva geometrica)

- Dobbiamo definire quale tipo di primitiva geometrica stiamo disegnando. Nel nostro caso disegneremo un triangolo di 3 vertici, la primitiva sarà TRIANGLELIST:

```
pd3dDeviceContext->IASetPrimitiveTopology(
        D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST );
```

Rendering

- Ora che abbiamo effettuato il binding del buffer e associato shaders, layout e primitive da utilizzare possiamo ordinare al device di effettuare il rendering della scena con il comando Draw: **void Draw(UINT** *VertexCount*, **UINT** *StartVertexLocation*);
- Nel nostro esempio: pd3dDeviceContext->Draw(3, 0);
- Il comando Draw disegna vertici assumendo un vertex buffer senza Index Buffer associato. (con vertici nell'ordine canonico 0,1,2,3…)

Esempio 01: rendering di un triangolo

- Esercizio 01: Il programma creato eseguirà il semplice rendering di un triangolo. Provate ad aggiungere nuovi triangoli, nuove figure geometriche o sfruttare altri tipi di primitive.