Graphics Programming

Master Computer Game Development 2013/2014

Texturing

University of Verona

luca.giona@3dflow.net

Texturing

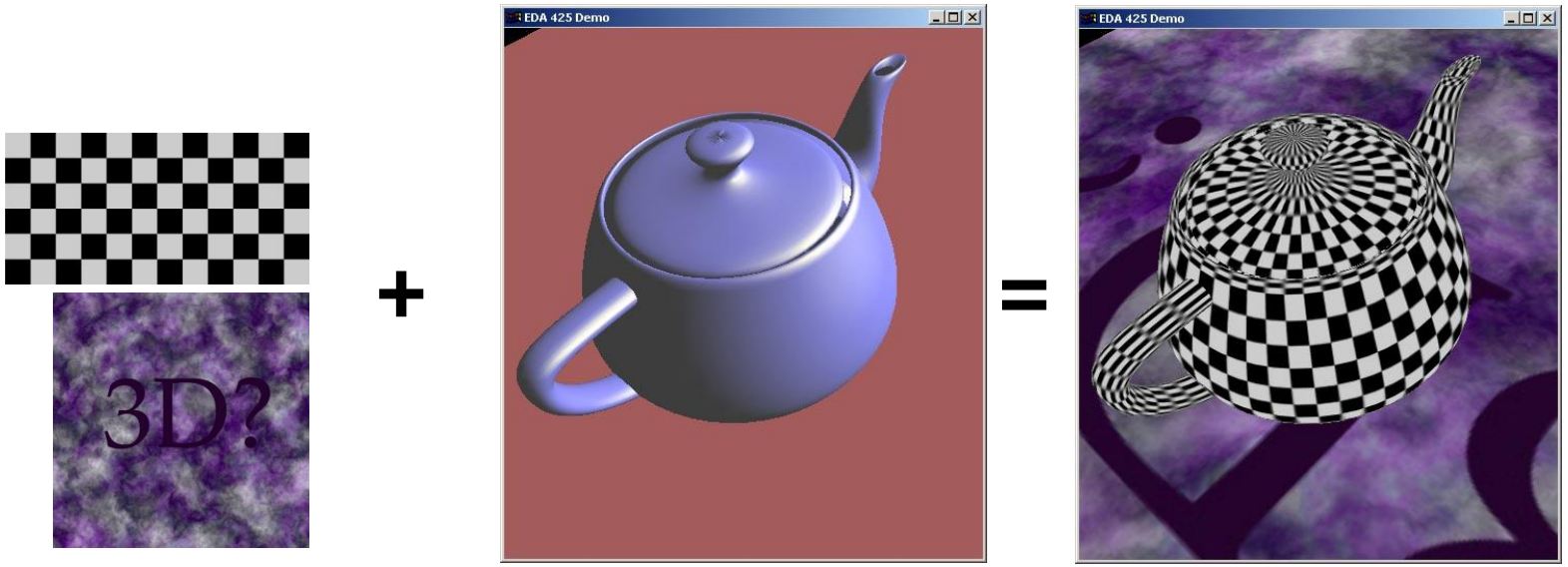
- Nelle lezioni precedenti abbiamo iniziato a vedere alcuni componenti del rendering che aggiungono realismo e tridimensionalità:
  - Proiezioni prospettiche/Trasformazioni
  - Rimozione oggetti occlusi
  - Colore e Illuminazione
- Oggi vedremo una tecnica alla base di molti effetti realistici: texturing.

Texturing

- Abbiamo già visto come possiamo colorare in modo realistico le superfici, definendone le proprietà dei materiali e applicando un modello di illuminazione.
- Le proprietà dei materiali possono essere specificati per ogni vertice: maggiore sarà il dettaglio geometrico, maggiore sarà il realismo.
- Oggi vedremo come aggiungere dettaglio visivo senza aumentare la complessità geometrica di una superficie.

Texturing

- Il processo di texturing – nella sua forma base (color mapping) - consiste nell'"*incollare*" un'immagine su una superficie.



Texturing

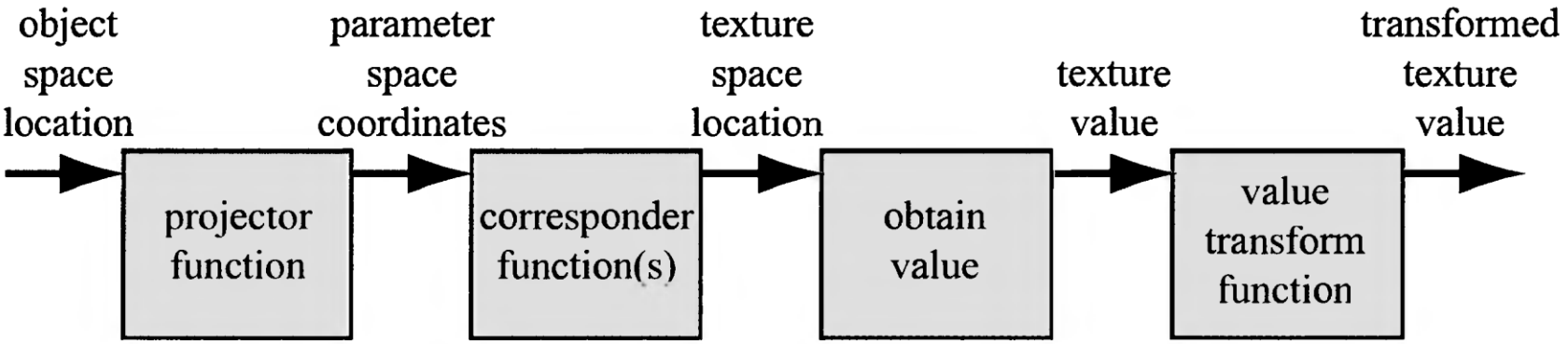
- Tale processo è supportato in modo diretto dalla scheda video e può quindi essere effettuato a basso costo computazionale.
- Attraverso tale processo possiamo aggiungere dettaglio di colore a livello di pixel ad una superficie senza aumentare il numero di vertici.

Texturing

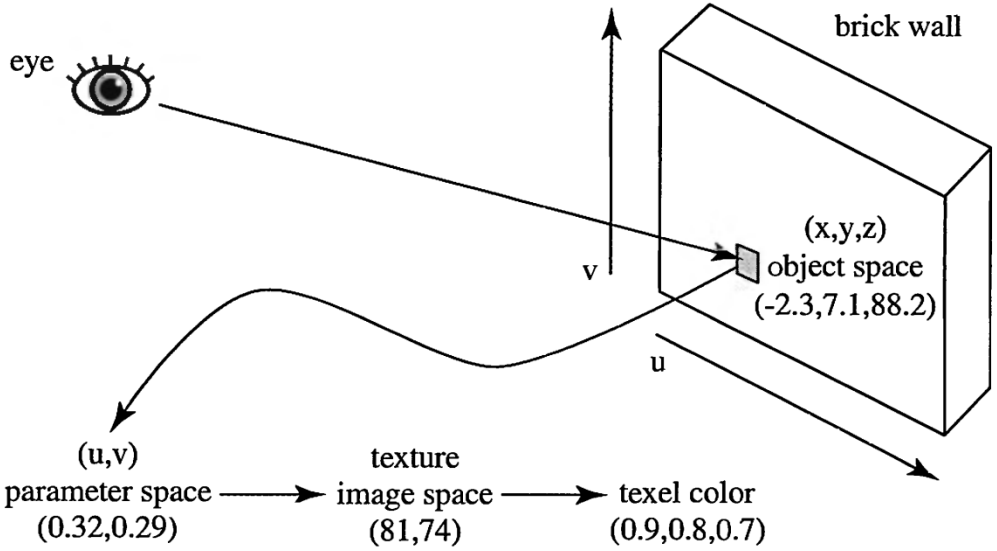
- Le texture interagiscono con la fase di shading.
- Di solito sono applicate a livello di *pixel shader*.
- Nella sua forma più semplice, la texture viene applicata alla superficie rimpiazzandone il colore (color mapping con replace).
- In realtà, come vedremo, può contenere qualsiasi tipo di informazione, non solo colore.

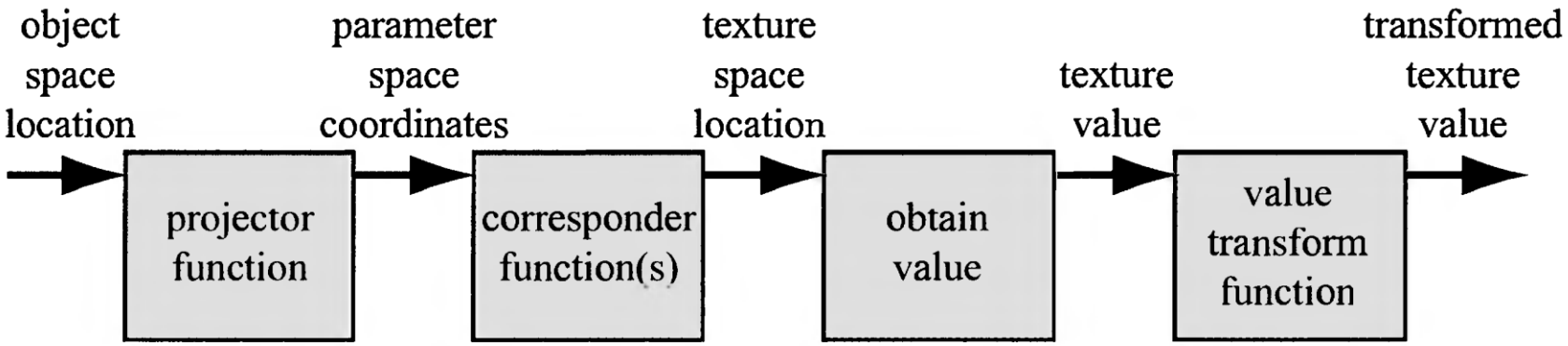
Pipeline texturing

- La micropipeline di texturizzazione all'interno di una gpu è riportata di seguito:



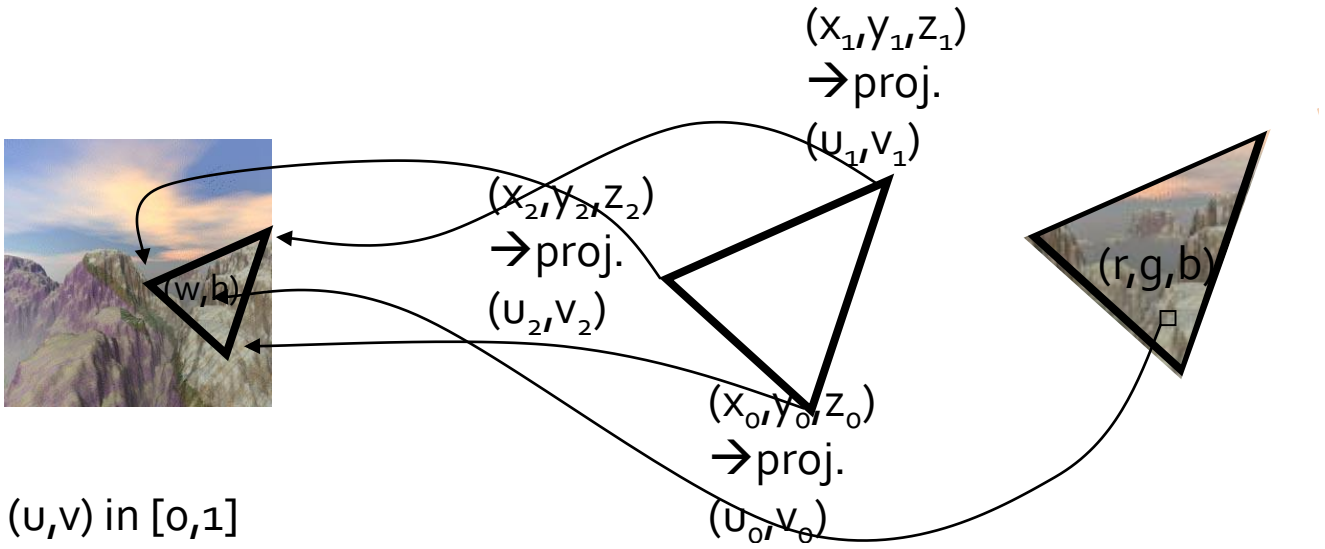
Coordinate texture

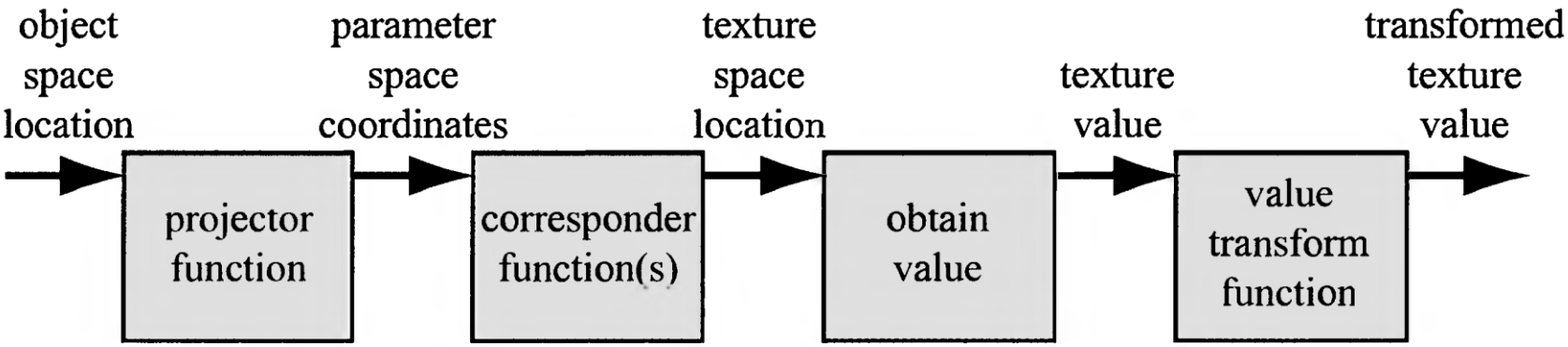




Coordinate texture

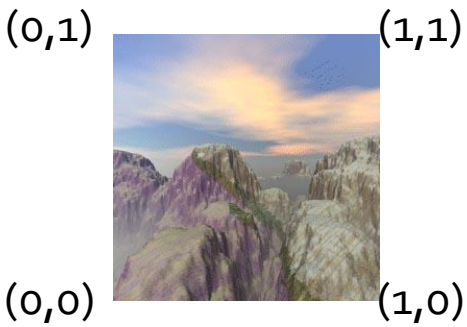
(x,y,z) obj. Sp. loc.
(u,v) par. Sp. Loc.
(w,h) tex. Sp. Loc.
(r,g,b,) trans. tex. value





Projector function

- Il compito della projector function è quello di ottenere una trasformazione dalle coordinate tridimensionali alle coordinate texture.
- Di solito le texture sono bidimensionali (u,v), ma possono anche essere tridimensionali o unidimensionali.
- $0<=u<1$ e $0<=v<1$.
- Projector function:
  - (x,y,z) → (u,v)



Corresponder function

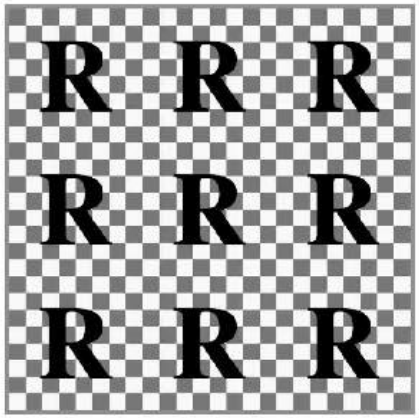
- La Corresponder Function converte le coordinate nel parameter space in coordinate texture space.
- Ad Esempio supponiamo di avere un'immagine di dimensioni 640 x 480.
  - (0,1) → (0, 479)
  - (1,1) → (639, 479).
- In questa fase è possibile includere nella corresponder Function traslazioni,scalatura,rotazione delle coordinate texture rispetto all'immagine originale.

Corresponder function

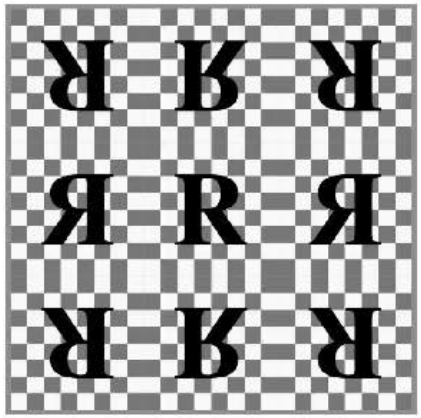
- Abbiamo detto che u,v sono compresi in [0,1].
- Cosa accade se u,v < 0 oppure > 1.
- La correspondance function deve controllare anche il "wrapping mode". Vi sono diverse opzioni:
  - Wrap/Repeat: l'immagine ripete sè stessa periodicamente
  - Mirror: l'immagine ripete sè stessa, ma in modo specchiato (flipping) ad ogni ripetizione.
  - Clamp: per i valori esterni a [0,1] viene effettuato il clamping al valore più vicino. (viene assegnato il valore del bordo più vicino)
  - Border: per i valori esterni a [0,1] viene assegnato un valore di bordo prescelto.

Corresponder function: wrapping mode

Repeat

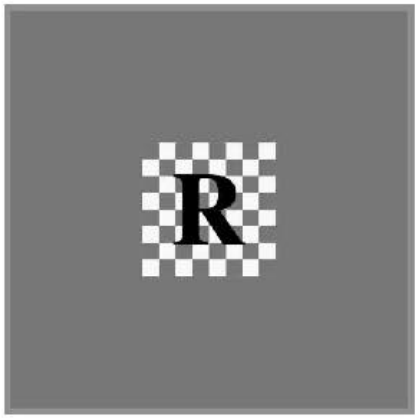


Mirror



Clamp

Border

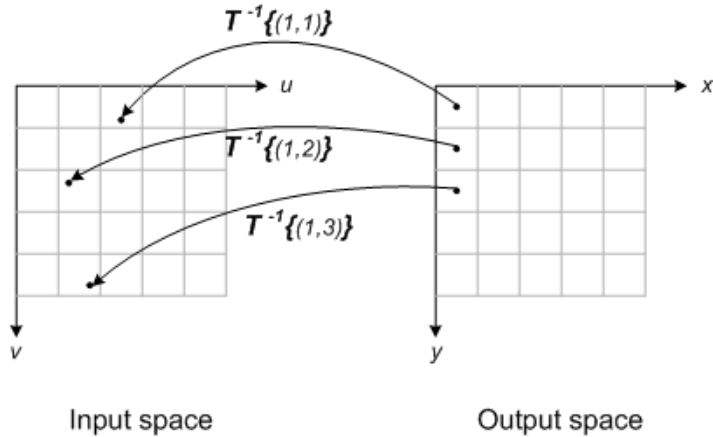


Texture value

- Texel: valore texture corrispondente ad un pixel della texture (introdotto per differenziare tra pixel dello schermo e pixel dell'immagine).
- Il valore del texel viene estratto dalle coordinate texture.
- Esistono vari modi per ottenere tale valore che saranno discussi nel seguito.
- Una volta ottenuto, tale valore di colore può essere eventualmente trasformato prima di essere usato (ad esempio può essere effettuato un cambio tra spazi di colore).

Mappatura inversa

- Per ottenere il valore di un texel si utilizza una mappatura inversa:
  - Per ogni pixel(frammento) si trova la corrispondente coordinata texture.
- Il metodo più semplice e diretto consistente nell'assegnare il colore del pixel al texel corrispondente in cui cade. (nearest neighbor)

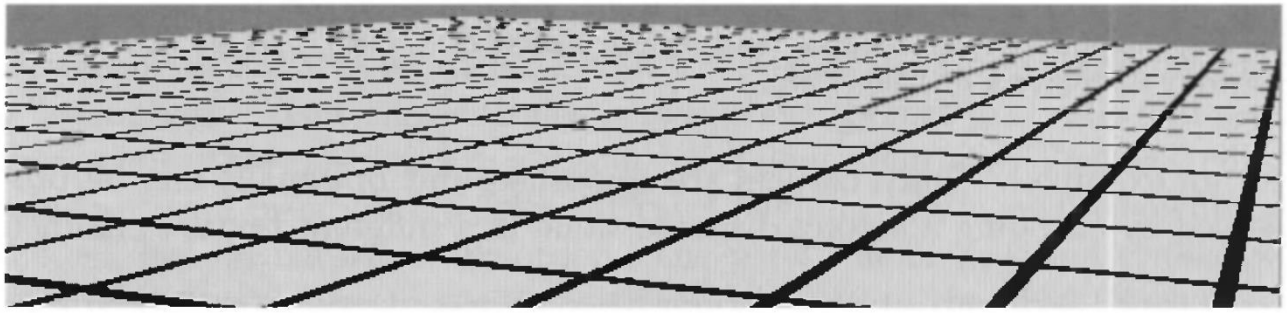


Texture value

- Prima di vedere quali sono le altre funzioni per ottenere il valore texture a partire dalle coordinate texture, è importante capire i problemi a cui possiamo incorrere utilizzando una mappatura di tipo nearest neighbor.
- Aliasing: succede ogni volta che non c'è una mappatura uno a uno tra pixel e texel (praticamente sempre).

Aliasing

- Utilizzando un metodo nearest neighbor, l'aliasing dà luogo a due problemi distinti:
  - Magnification: un texel corrisponde a molti pixel.
    - Se si usa nearest neighbor, la texture si vede ingrandita a blocchi uniformi.
  - Minification: un pixel corrisponde a molti texel.
    - Se si usa nearest neighbor, dei texel rimangono fuori e si perde informazione.

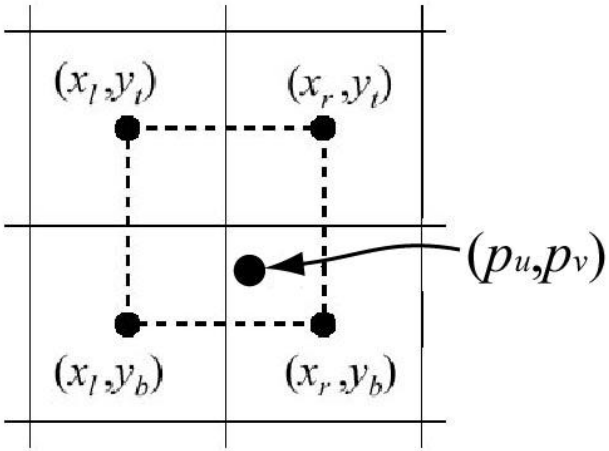


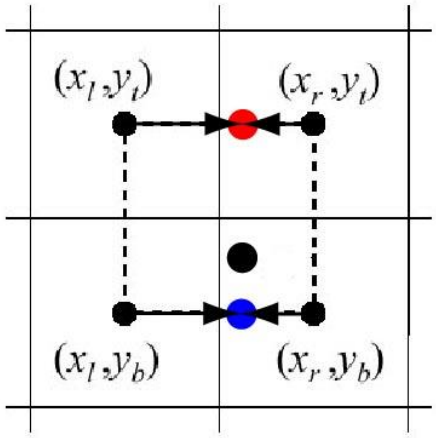
Soluzione magnification

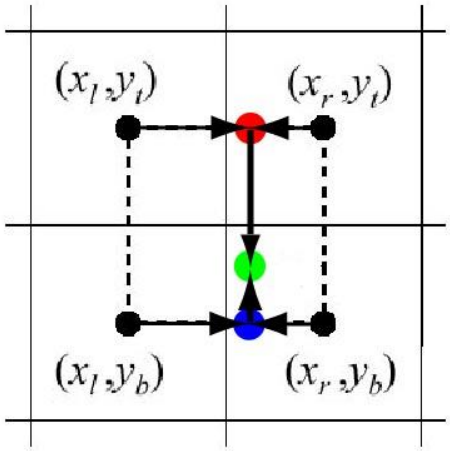
- La soluzione più semplice al problema di magnification è quello di usare un'interpolazione di grado maggiore (nearest neighbor interpolazione di grado o).
- Di solito si utilizza un'interpolazione (bi)lineare, di primo grado.
- Interpolazione lineare, caso unidimensionale:
  - (1-t)*color0+t*color1

Interpolazione bilineare caso bidimensionale

- Supponete che il centro di un pixel cada in (p_u, p_v).
- Nearest neighbor sarebbe $(p_u, p_v) \rightarrow (x_r, y_b)$
- Interpolazione bilineare tiene conto dei valori dei vicini e della posizione rispetto ad essi.

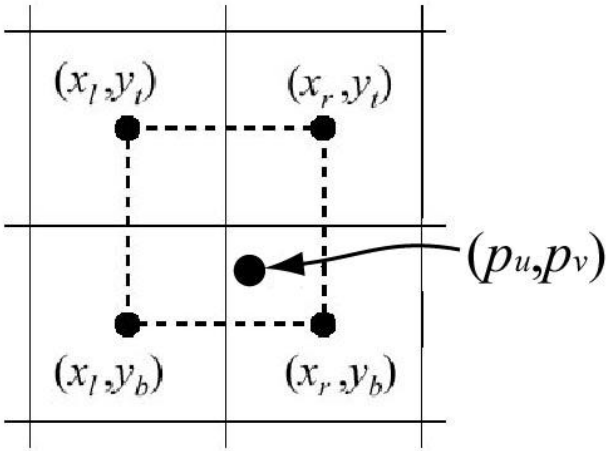


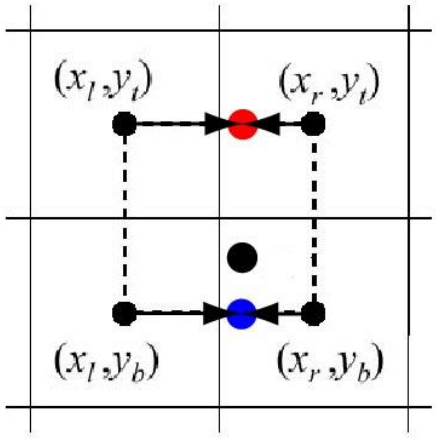


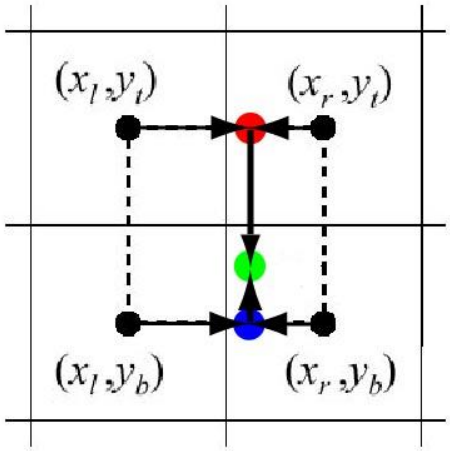


Interpolazione bilineare caso bidimensionale

- Viene effettuata una interpolazione lineare unidimensionale sia nell'asse x che nell'asse y (ecco perchè BILineare)



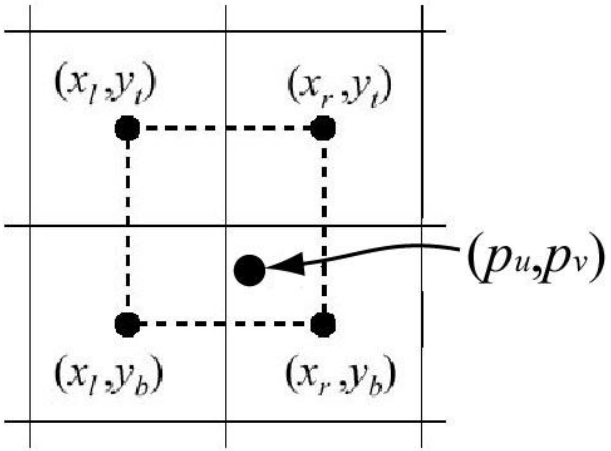


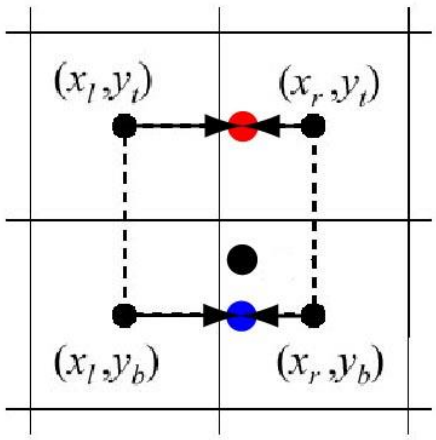


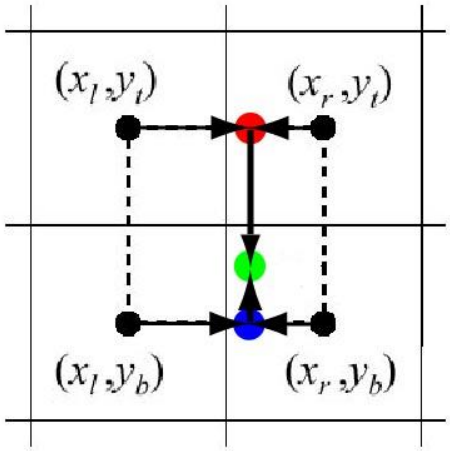
Interpolazione bilineare caso bidimensionale

- B(u,v) valore filtrato ottenuto.
- T(u,v) valore texture in (u,v)

$$\begin{aligned}\mathbf{b}(p_u, p_v) =& (1-u')(1-v')\mathbf{t}(x_l, y_b) + u'(1-v')\mathbf{t}(x_r, y_b) \\ &+ (1-u')v'\mathbf{t}(x_l, y_t) + u'v'\mathbf{t}(x_r, y_t).\end{aligned}$$







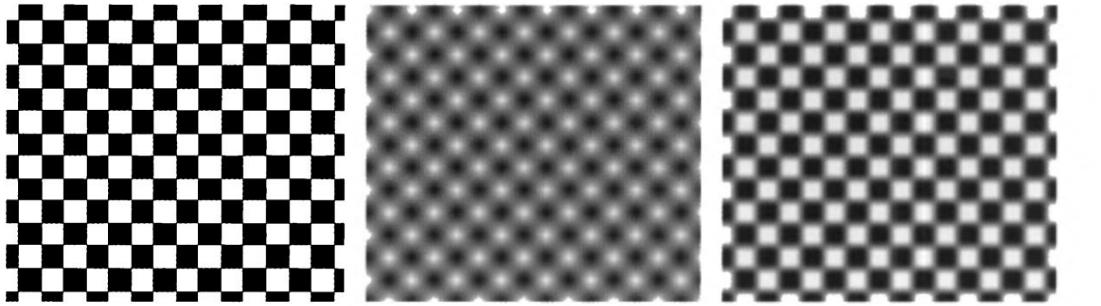
Interpolazioni ordine più elevato

- Si possono ottenere interpolazioni di ordine più elevato(di solito al massimo cubico)
- Nearest neighbor e bilineare sono implementate a livello hardware.

Nearest Neighbor | Bilineare | Cubica

Problemi Magnification + Bilineare

- Si hanno problemi con texture di griglie / testi con bordi ben definiti: con interpolazione lineare si perde informazione ai bordi



Nearest neighbor Bilineare Bilineare + sogliatura

- Soluzione: ad esempio sogliare sui valori intermedi di grigio.

Minification

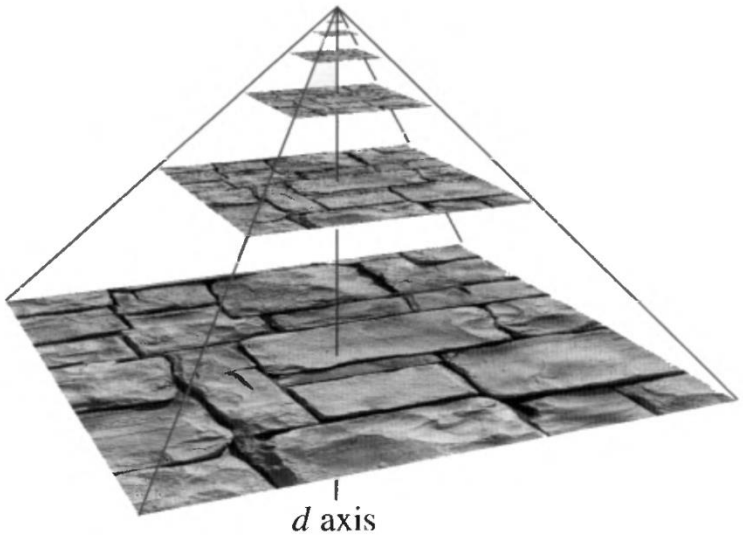
- I problemi maggiori si hanno in fase di minification.
- Si può usare il filtro bilineare, ma li risolve solo nel caso di ordini di rimpicciolimento piccoli (utilizzando sempre i valori di 4 texel).
- Soluzione: bisognerebbe fare la media di tutti i texel che cadono nel pixel -> da evitare in real-time.

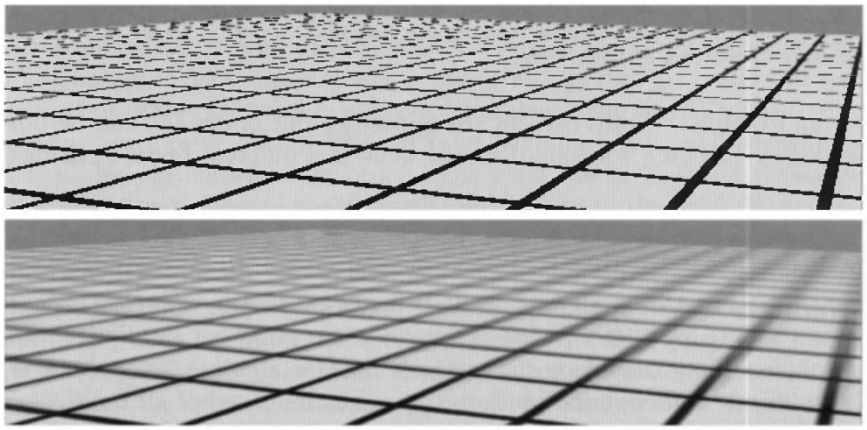
Minification: mipmapping

- La soluzione adottata dall'hardware video è l'uso di *mip*mapping. (Multum In Parvo)
- Una texture viene memorizzata assieme a delle sottotexture in dimensioni ridotte.
- Di solito una texture viene via via sottodimensionata in modo che l'area sia ¼ (dimensione diviso 2), in modo che i texel della texture più piccola siano calcolati ottenendo i valori dei 4 texel corrispondenti nella texture più grande.

Mipmapping

- Si tiene così in memoria una "piramide" di texture e viene selezionata dall'hardware quella di dimensioni più vicine e si utilizza di solito un'interpolazione bilineare.

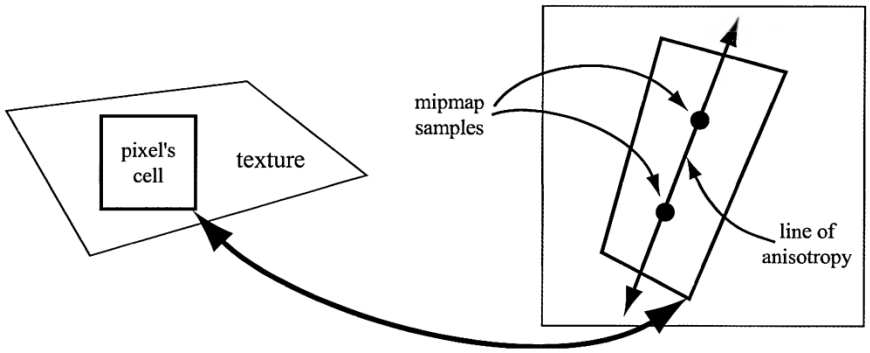


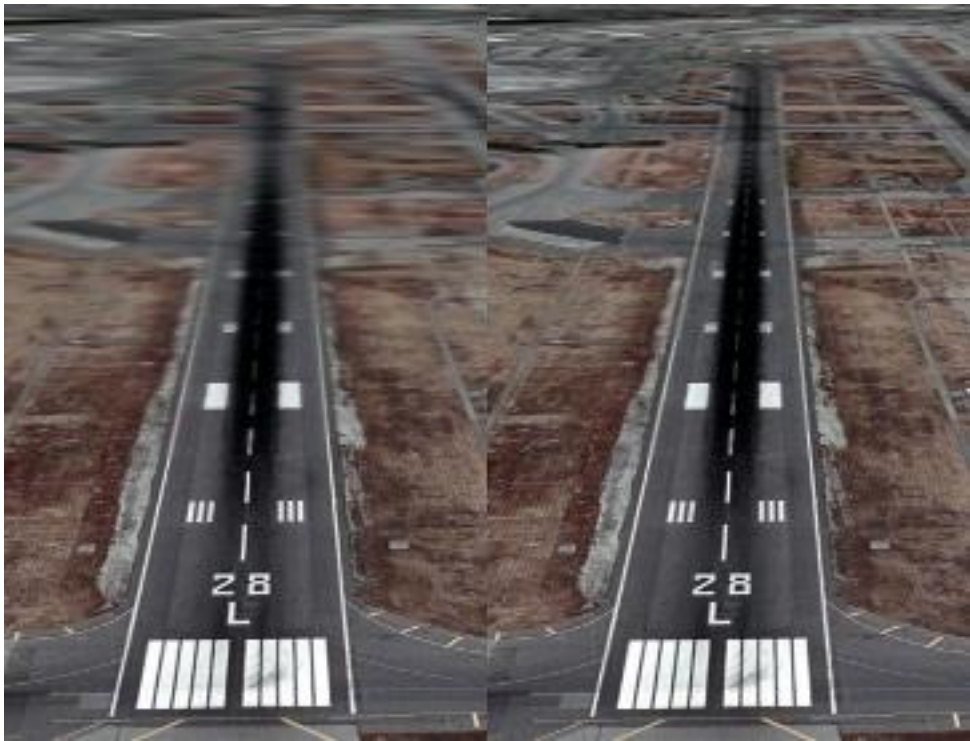


Problema mipmapping

- Le texture non tengono conto delle trasformazioni prospettiche anisotropiche (non mantengono proporzioni tra gli assi).

-Soluzione: filtering anisotropico
- Idea, prendi più sample di texel nella piramide mipmap lungo la linea di anisotropia



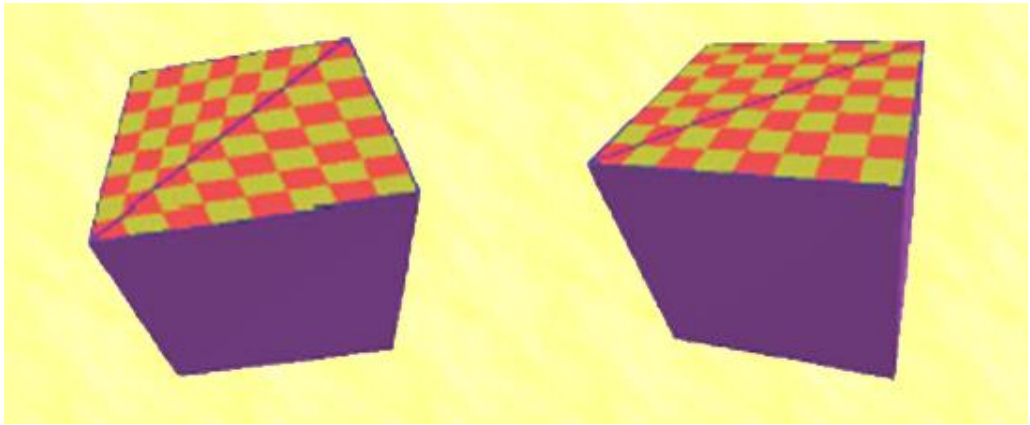


Texture nella pipeline DX

- Riassumiamo i passaggi fondamentali per la fase di texturizzazione all'interno della gpu:
  - A ciascun vertice viene associata una coordinata texture.
  - Durante lo scan conversion si determinano le coordinate texture di ciascun pixel/fragment
  - Il campionamento della texture verrà effettuato (di solito) nel pixel shader.

Distorsione prospettica scan conversion

- L'interpolazione scan-line lineare di (s,t) crea delle distorsioni prospettiche.
- Bisogna utilizzare l'interpolazione iperbolica:
  - In pratica si interpolano le coordinate omogenee (s/z,t/z,1/z) con z profondità pixel e poi si divide per l'ultima componente.
  - E' utilizzata automaticamente e supportata da tutte le moderne gpu.



Pixel shader

- Al pixel shader arriveranno le coordinate texture interpolate per il frammento attuale da cui si può ottenere il valore corrispondente nella texture.
- Dovremmo impostare il tipo di sampling (nearest neighbor, bilineare, mipmapping, anisotropo) tramite uno stato.
- L'utilizzo di mipmapping con relativo filtraggio deve essere invece specificato in fase di caricamento della texture in memoria.
- Il pixel shader decide anche come utilizzare il valore ottenuto.

Applicazioni texture

- Vedremo ora alcune delle applicazioni / effetti più comuni nelle texture.
- Se la texture rappresenta il colore parliamo di *Color Mapping*.

Color Mapping

- Replace: se non abbiamo/vogliamo illuminazione, la texture può rappresentare direttamente il colore del pixel.
- Nel caso più comune invece, il colore contenuto nella texture è *modulato* con il colore calcolato dallo shading.

Gourad shading e color mapping

- L'opzione di default(fixed pipeline) in DX9 e OpenGL è utilizzare lo shading di Gourad.
- Il calcolo dell'illuminazione avviene "per vertice".
- Il calcolo della texture avviene invece "per pixel".
- In questo caso non si può utilizzare la texture come colore/coefficiente diffusivo/ambientale/emissivo.
- La soluzione comune è quella di moltiplicare il colore finale dell'illuminazione con il colore della texture:
- ColorePixel = I_{out} * tx

Texture mapping e Phong Shading

- Se il calcolo dell'illuminazione avviene per pixel (phong shading), la texture può trasportare informazione sul materiale (Es. coefficienti Ka-Kd).

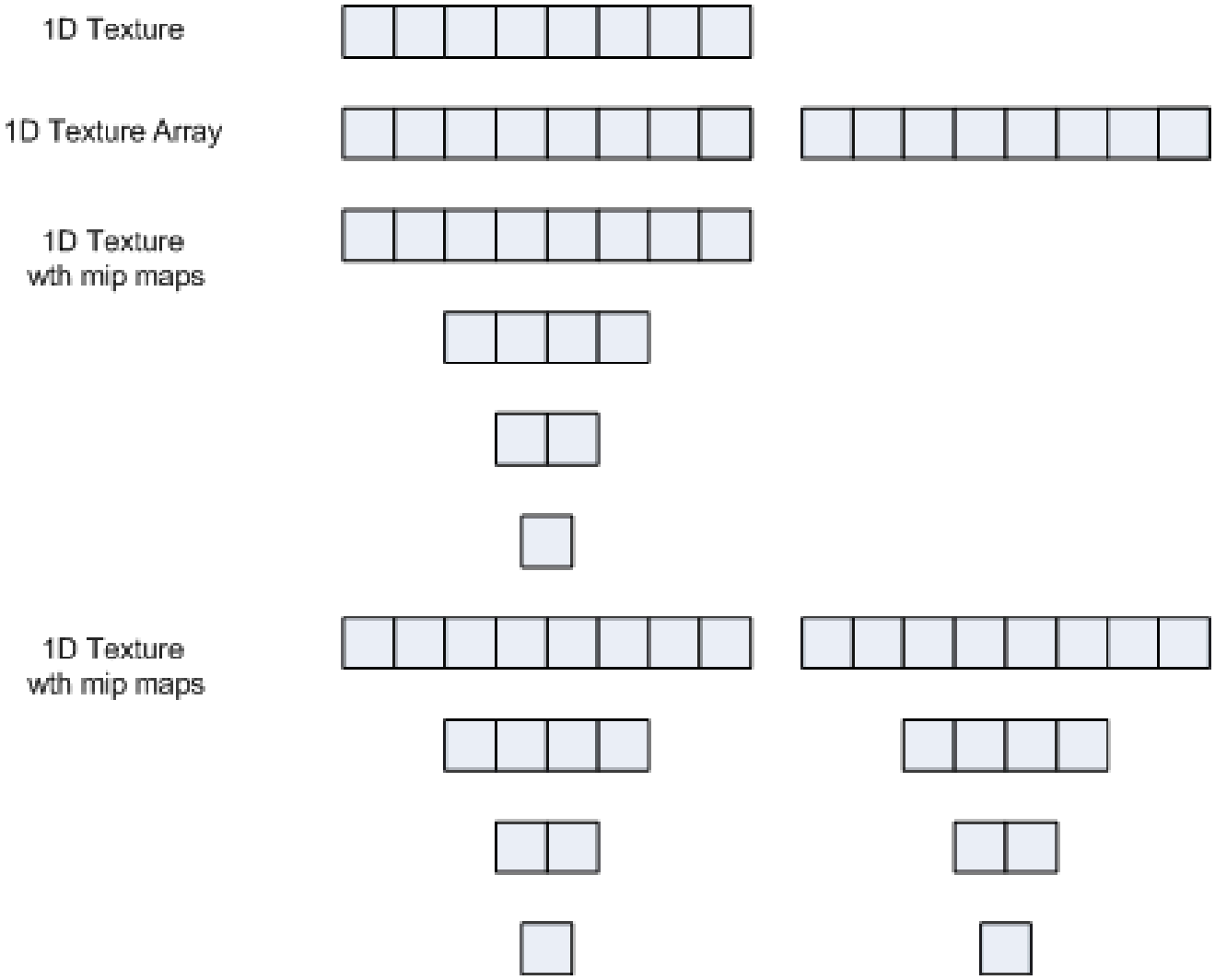
Textures in DirectX 11

- Una texture in DirectX è un array multidimensionale (uni/bi/tri) contenente dati.
- Di solito contengono colore, ma possono contenere qualsiasi valore arbitrario (descritto con DXGI_FORMAT).
  - Possono essere usate per creare effetti avanzati e memorizzare informazioni arbitrarie.

Textures in Directx10/11

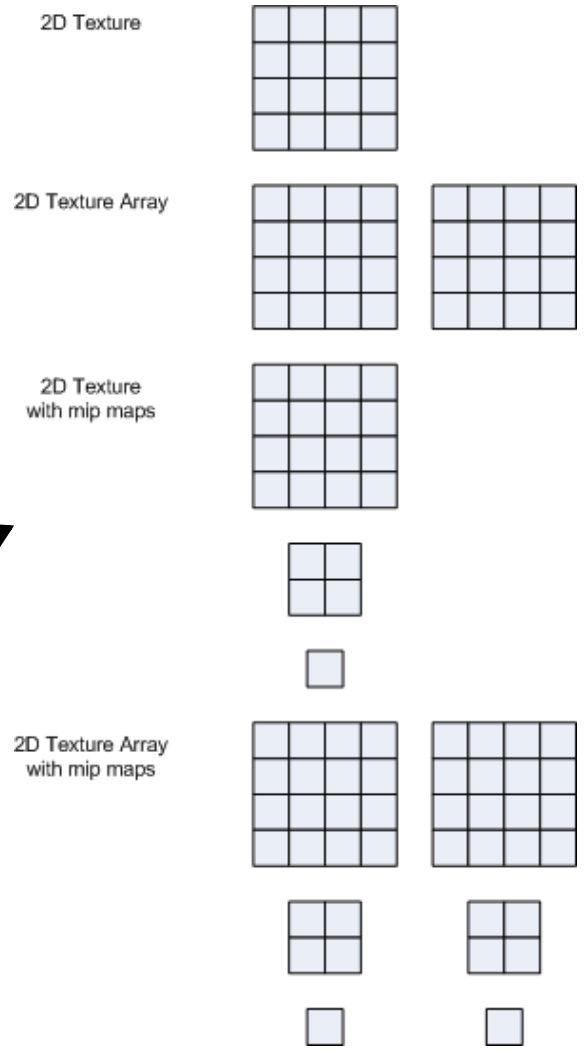
- Ad ogni texture in DirectX10/11 è associato un oggetto vista (view).
- Una *Vista* è un oggetto che descrive come una texture deve essere memorizzata dal device.
  - Definisce i parametri di mipmap per la lettura in memoria.
  - Specifica dove una texture è letta/collegata. Dato che le nostre texture sono utilizzate negli shader parleremo di *Shader Resource View*.

Texture unimensionali

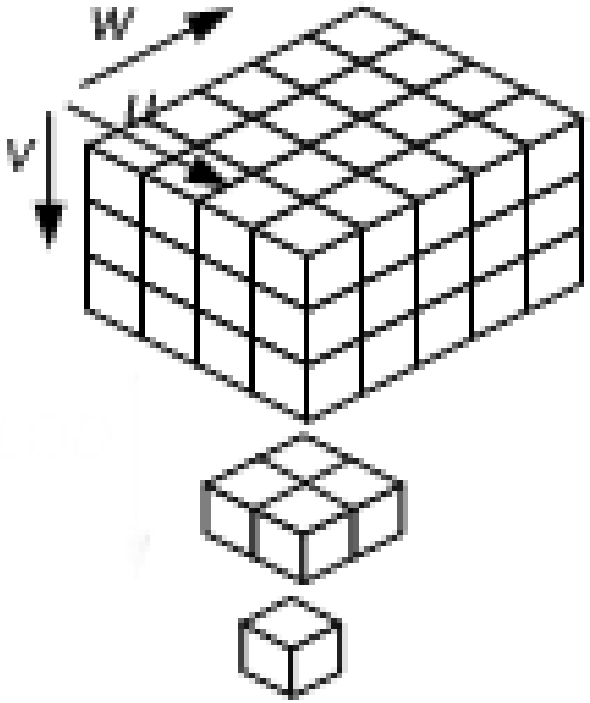


Texture bidimensionali

- Il tipo di texture di colore più comune sono quelle bidimensionali a 3-4 canali + mipmapping.



Texture tridimensionali



Creazione Textures

- Per utilizzare una texture dovremmo creare la texture e associare una shader resource view.
- Per creare delle textures, caricandole da file, utilizzeremo una libreria di utility (DirectXTK) che creerà la texture e la shader resource view per noi.
- È possibile creare delle texture vuote da utilizzare come output (render target).
- È possibile «riempire» una texture utilizzando un'area di memoria (dati generati algoritmicamente o letti da file).

CreateDDSTextureFromFile

- La funzione per creare da file una texture con relativa vista in un solo colpo è:

```
HRESULT DirectX::CreateDDSTextureFromFile(ID3D11Device* d3dDevice,
                                          const wchar_t* fileName,
                                          ID3D11Resource** texture,
                                          ID3D11ShaderResourceView** textureView,
                                          size_t maxsize )
```

- In textureView sarà memorizzato il puntatore ad una vista **ID3D11ShaderResourceView** .
  - Contiene anche le informazioni sulla texture relativa.
  - Dovremo collegare tale oggetto nella pipeline (agli shader), e dovremo ricordarci di rilasciarlo quando non più necessario in modo che elimini la texture associata

Binding texture shader

- Una texture creata può essere collegata allo shader attraverso la funzione **PSSetShaderResources** (per pixel shader).

```
void PSSetShaderResources( UINT StartSlot,  UINT NumViews,
            ID3D11ShaderResourceView *const *ppShaderResourceViews
);
```

- Si utilizza la shader resource view relativa alla texture creata.

Binding texture shader

- Abbiamo imparato come collegare una resource view ad una risorsa definita nello shader.
  - Ma dobbiamo definire le risorse texture nei nostri shader.

Texture negli shaders

- Anche negli Shader possiamo definire texture uni/bi/tridimensionali:
  - **Texture1D** tx1;
  - **Texture2D** tx2;
  - **Texture3D** tx3;
- Ovviamente dovremo effettuare il binding tra queste variabili e una Resource Shader View relativa ad una texture di uguali dimensioni.

Sampler states

- Oltre all'oggetto texture dovremo definire uno stato che determini i parametri di sampling, ovvero:
  - come vengono ottenuti i valori della texture?
  - Che tipo di interpolazione viene usata?
- Dovremmo creare un Sampler state di tipo **ID3D11SamplerState**.

Sampler states

- Per creare un sampler state utilizzeremo la funzione:

```
HRESULT CreateSamplerState(
        const D3D11_SAMPLER_DESC *pSamplerDesc,
        ID3D11SamplerState **ppSamplerState
);
```

- D3D11_SAMPLER_DESC è una struttura che contiene la descrizione dello stato che vogliamo creare.

Sampler states

- AddressU
- AddressV
- AddressW
- BorderColor
- Filter
- MaxAnisotropy
- MaxLOD
- MinLOD
- MipLODBias

Sampler states

- **AddressU/V/W**
  - Identifica la tecnica per risolvere le coordinate che cadono fuori dal range [0,1]
  - Possono essere (già viste in precedenza) D3D11_TEXTURE_ADDRESS_
    - WRAP
    - MIRROR
    - CLAMP
    - BORDER
    - MIRROR_ONCE
  - BorderColor definisce il colore di bordo nel caso sia selezionato D3D11_TEXTURE_ADDRESS_BORDER

Sampler states

- **MipLODBias**
  - Offset per calcolare la texture nella gerarchia di mipmap. Ad esempio, se DirectX seleziona la texture al livello 2 e impostiamo **MipLODBias** a 3, sarà seleziona la texture al livello 2+3 = 5. Default a 0.
- **MaxAnisotropy**
  - Valore massimo nel rapporto di anisotropia
- **MinLOD, MaxLOD**
  - Valore minimo/massimo di livello che è possibile selezionare nella gerarchia di mipmap.

Sampler states

- **Filter**
  - Filtro da utilizzare per il sampling (D3D11_FILTER_):

    MIN_MAG_MIP_POINT
    MIN_MAG_POINT_MIP_LINEAR
    MIN_POINT_MAG_LINEAR_MIP_POINT
    MIN_POINT_MAG_MIP_LINEAR
    MIN_LINEAR_MAG_MIP_POINT
    MIN_LINEAR_MAG_POINT_MIP_LINEAR,
    MIN_MAG_LINEAR_MIP_POINT
    MIN_MAG_MIP_LINEAR
    ANISOTROPIC

  - MIN = Minification, MAG = Magnification,

    MIP = MipMap (se generata)
  - Point = Nearest Neighbor, Linear = Bilinear

Sampler states

- Analogamente alle texture dobbiamo collegare il sampler state allo shader con la funzione (pixel shader):

```
void PSSetSamplers( UINT StartSlot, UINT NumSamplers,
        ID3D11SamplerState *const *ppSamplers
);
```

Sampler state - shader

- La sintassi per definire degli stati di Sampling in DX11 è:
- *SamplerState Name : register (sn)*;
  - Name è il nome del sampler state.

Sampling

- Per effettuare il sampling, ovvero ottenere il valore da una texture, basta richiamare il metodo Sample dall'oggetto texture.

```
DXGI_FORMAT Object.Sample(  sampler_state  S, float Location  [, int Offset] );
```

- Location conterrà le coordinate texture. Float nel caso di oggetto texture1d, float2 nel caso di texture2d e float3 nel caso di texture3d.
- Ritorna il valore nella forma memorizzata nella texture (di solito float3 o float4 - colore R32G32B32 o R32G32B32 A32)

Esempio 05

- Proviamo ora ad applicare quanto visto.
- Costruiamo un esempio, sulla base del modello di illuminazione delle esempio precedente (phong esempio 04), che carichi una texture per un modello e moduli l'illuminazione con il colore della texture.
- Come modello, specifichiamo manualmente le coordinate del cubo, in modo da vedere come configurare/bindare le coordinate texture nel vertex buffer e caricare un'immagine texture.
  - Alternativamente una mesh spesso contiene già texture+coordinate.

Input layout

- L'input layout dovrà descrivere l'input per vertici come contenente Posizioni, Normali e Coordinate texture.

```
// Definiamo l'input layout
D3D11_INPUT_ELEMENT_DESC layout[]  = {
   { "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D11_INPUT_PER_VERTEX_DATA, 0 },
   { "NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12, D3D11_INPUT_PER_VERTEX_DATA, 0 },
   { "TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 24, D3D11_INPUT_PER_VERTEX_DATA, 0 },
};
```

Creazione vertex buffer

- Definiamo i vertici (posizioni/normali/coord.texture):

```
mcg::PositionNormalTex2 vertices[] =
{
    { XMFLOAT3( -1.0f, 1.0f, -1.0f ), XMFLOAT3( 0.0f, 1.0f, 0.0f ), XMFLOAT2( 1.0f, 0.0f ) },
    { XMFLOAT3( 1.0f, 1.0f, -1.0f ), XMFLOAT3( 0.0f, 1.0f, 0.0f ), XMFLOAT2( 0.0f, 0.0f ) },
    { XMFLOAT3( 1.0f, 1.0f, 1.0f ), XMFLOAT3( 0.0f, 1.0f, 0.0f ), XMFLOAT2( 0.0f, 1.0f ) },
    { XMFLOAT3( -1.0f, 1.0f, 1.0f ), XMFLOAT3( 0.0f, 1.0f, 0.0f ), XMFLOAT2( 1.0f, 1.0f ) },

....
};
```

Caricamento della texture

- Carichiamo la texture da file con la funzione di utilità:

```
CreateDDSTextureFromFile(mPd3dDevice, L"./05-TextureMapping/colormap.dds",
                         nullptr, &mColorTexResourceView);
```

- Questa funzione ci permette di leggere texture caricate da file con estensione dds.

Creazione del sampler state

- Creiamo un sampler desc:

```
D3D11_SAMPLER_DESC sampDesc;
  ZeroMemory( &sampDesc, sizeof(sampDesc) );
  sampDesc.Filter = D3D11_FILTER_MIN_MAG_MIP_LINEAR;
  sampDesc.AddressU = D3D11_TEXTURE_ADDRESS_WRAP;
  sampDesc.AddressV = D3D11_TEXTURE_ADDRESS_WRAP;
  sampDesc.AddressW = D3D11_TEXTURE_ADDRESS_WRAP;
  sampDesc.ComparisonFunc = D3D11_COMPARISON_NEVER;
  sampDesc.MinLOD = 0;
  sampDesc.MaxLOD = D3D11_FLOAT32_MAX;
```

- Creiamo il sampler state:

```
pd3dDevice->CreateSamplerState( &sampDesc, &mColorTexSamplerState );
```

Shaders

- Vediamo ora come modificare gli shaders scritti precedentemente per l'illuminazione per includere la texture del materiale.
- Dobbiamo modulare (moltiplicare) il valore di illuminazione per quello della texture sia per lo shading di Gourad che per lo shading di Phong.

Definizione variabili

- Definiamo la variabile texture e la variabile di definizione del sampling:

```
Texture2D colorTex; // Texture diffusiva
SamplerState texSampler : register( so ); // Sampler state
```

Strutture Input/Output

- L'input e l'output del vertex shader dovranno contenere anche l'informazioni sulle coordinate texture.

```
// Strutture input dei Vertex shaders
struct VertexShaderInput
{
        float3 pos: POSITION;
        float3 norm: NORMAL;
        float2 texCoord: TEXCOORD;
};
```

```
// Strutture output  dei Vertex shaders
struct PixelShaderInput
{
        float4 pos: SV_POSITION;
        float2 texCoord: TEXCOORD0;
        float3 wNormal: NORMAL;
        float3 viewDirection: VIEWDIRECTION;
        float3 wPos: WORLDPOSITION;
};
```

Pixel shader Phong

- Dovremo aggiungere il valore estratto dalla texture nel pixel shader sia per il Gourad che per il Phong shading.

```
float4 ColorMapPS(PixelShaderInput input) : SV_TARGET
{
   float3 materialColor = colorTex.Sample(texSampler, input.texCoord).rgb;
   return CalcLightinig(input.wPos, normalize(input.wNormal),
                                normalize(input.viewDirection), materialColor);
}
```

Esempio 05 - texturing

- Esercizio 05: Provate a cambiare il sampler states.