

Gràfics i Visualització de Dades

T3: Mètodes projectius: ZBuffer

Anna Puig

Índex

3.1. Introducció a ZBuffer

3.2. Pipeline de visualització

3.3. Pipeline de visualització a GL

3.4. Il·luminació usant shaders

3.5. Textures

3.6. Reflexions i Transparències

3.1. Mètodes projectius

RayTracing

Suposem que anem a visualitzar Models superficials: malles poligonals



Per a cada **pixel** (raig)

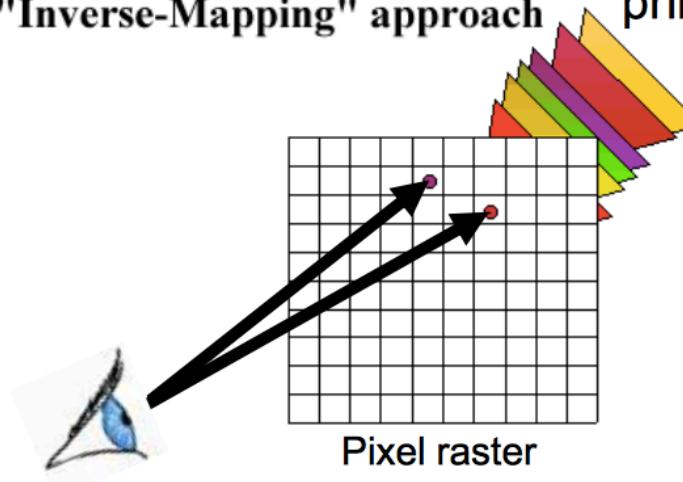
Per a cada **triangle**

Intersecta raig-triangle?

Trobar la intersecció
mes propera

"Inverse-Mapping" approach

Scene
primitives



3.1. Mètodes projectius

- RayTracing

Per a cada pixel (raig)

Per a cada triangle

Intersecta raig-triangle?

Trobar la intersecció
mes propera

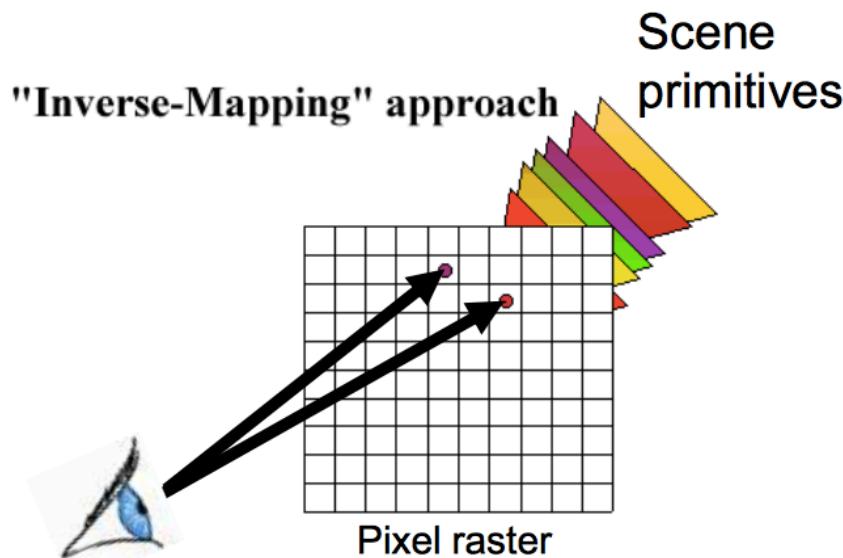
- Projectius

Per a cada triangle

Per a cada pixel

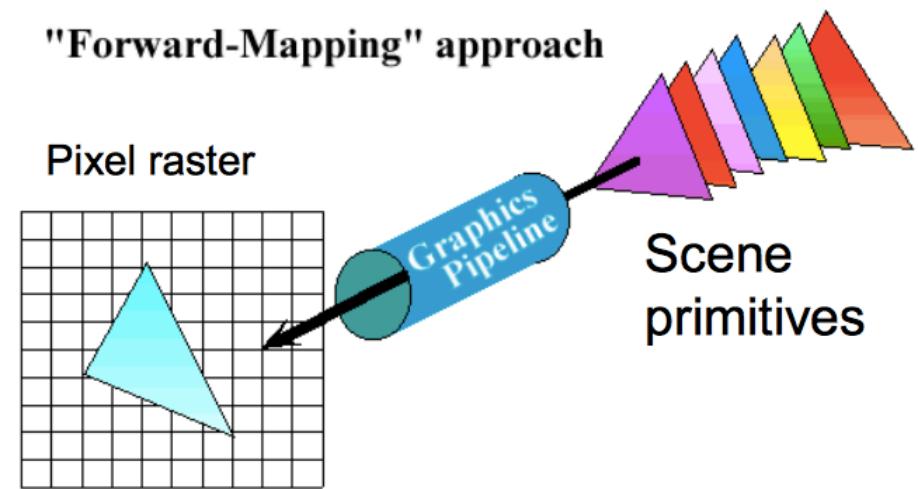
El triangle cubreix el píxel?

Trobar la projecció
més propera



"Forward-Mapping" approach

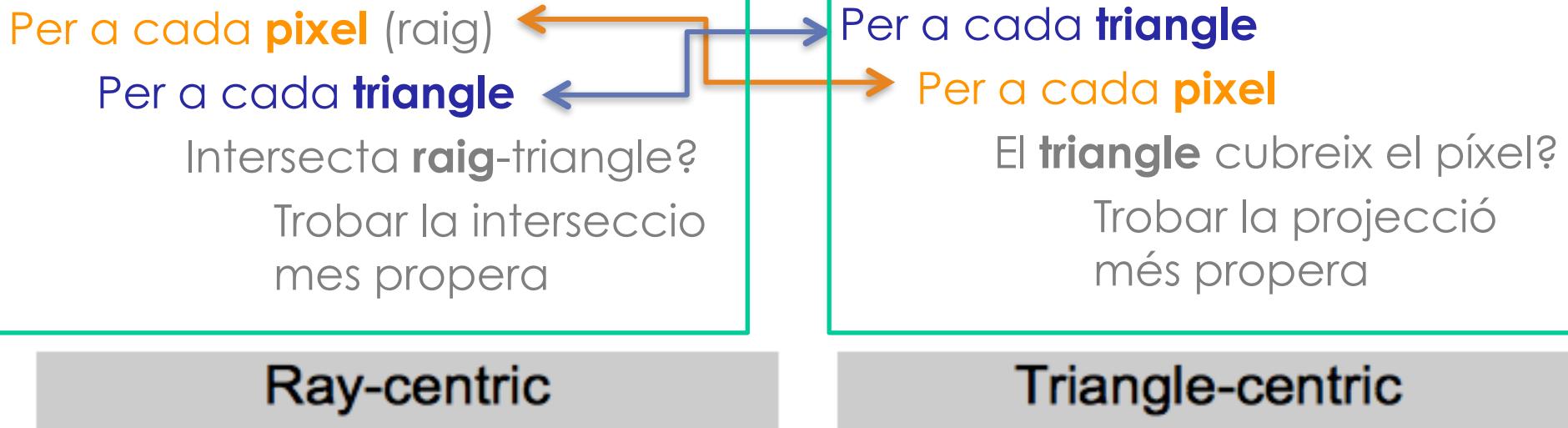
Pixel raster



https://ocw.mit.edu/courses/6-837-computer-graphics-fall-2012/53d96abf747a3c82fd3497d2fea540f5/MIT6_837F12_Lec21.pdf

3.1. Mètodes projectius

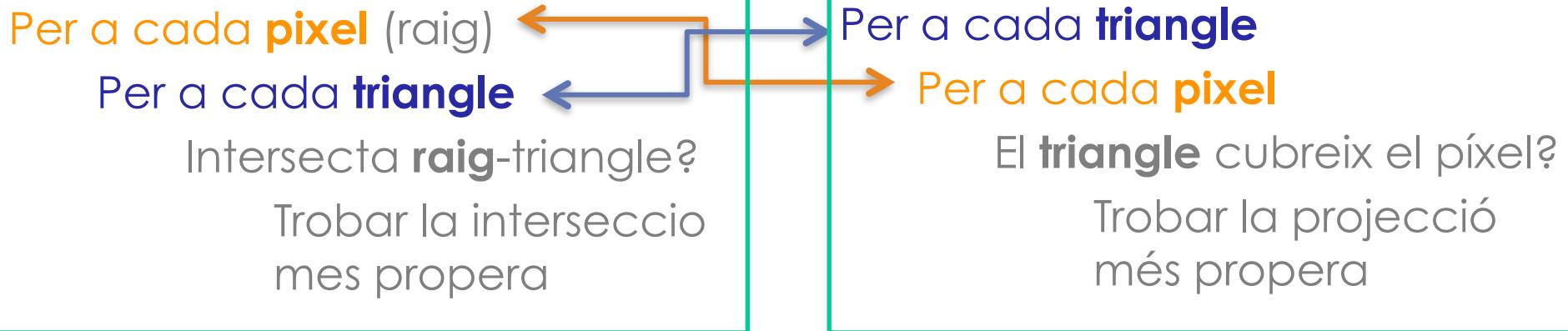
- RayTracing
- GPU (Zbuffer)



Només canvia l'ordre dels bucles

3.1. Mètodes projectius

- RayTracing
- GPU (Zbuffer)



Característiques:

- Càmera
- Rasterització
- Visibilitat

3.1. Mètodes projectius

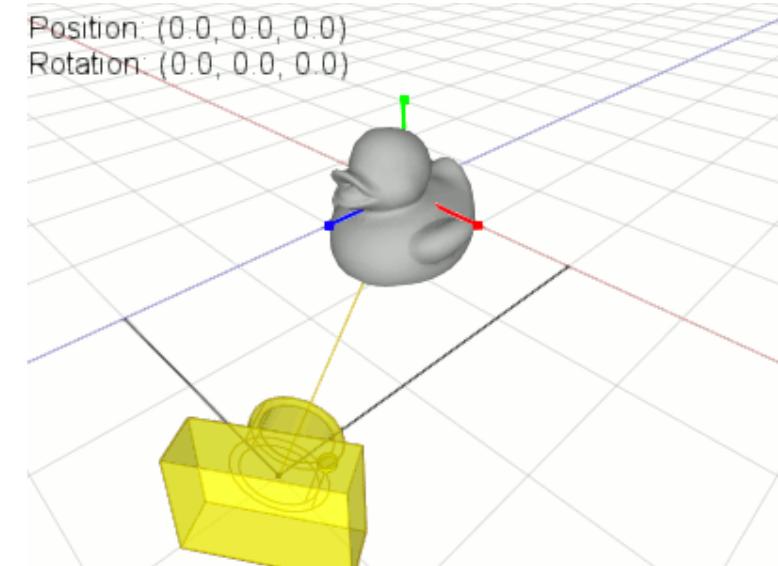
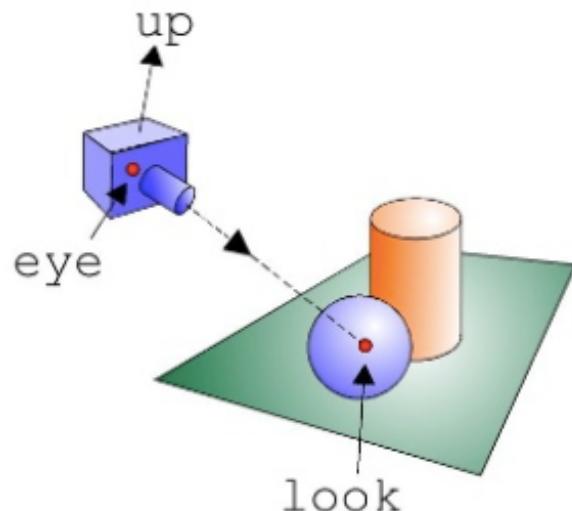
- RayTracing
- GPU (Zbuffer)

Per a cada **pixel** (raig) ←
Per a cada **triangle** ←
Intersecta **raig-triangle**?
Trobar la intersecció
mes propera

Per a cada **triangle** →
Per a cada **pixel**
El **triangle** cubreix el píxel?
Trobar la projecció
més propera

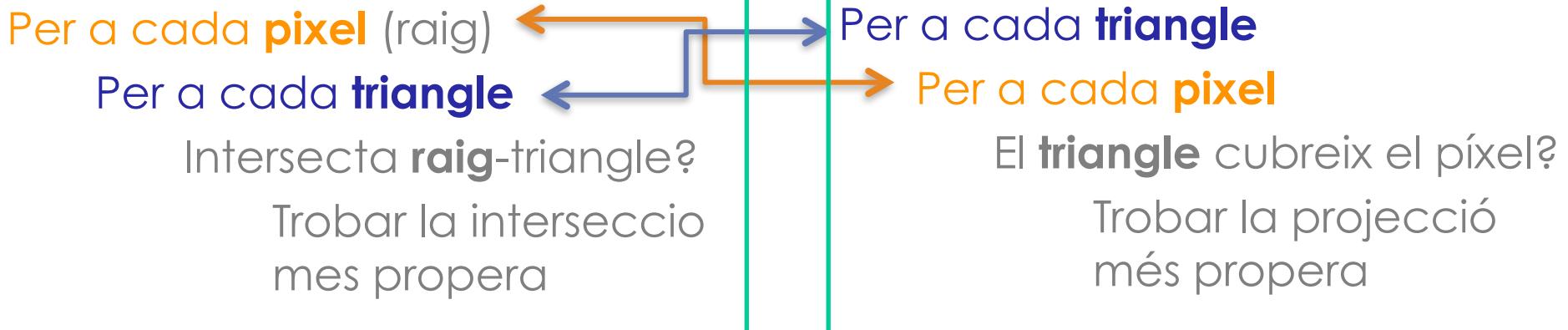
Característiques:

- Càmera



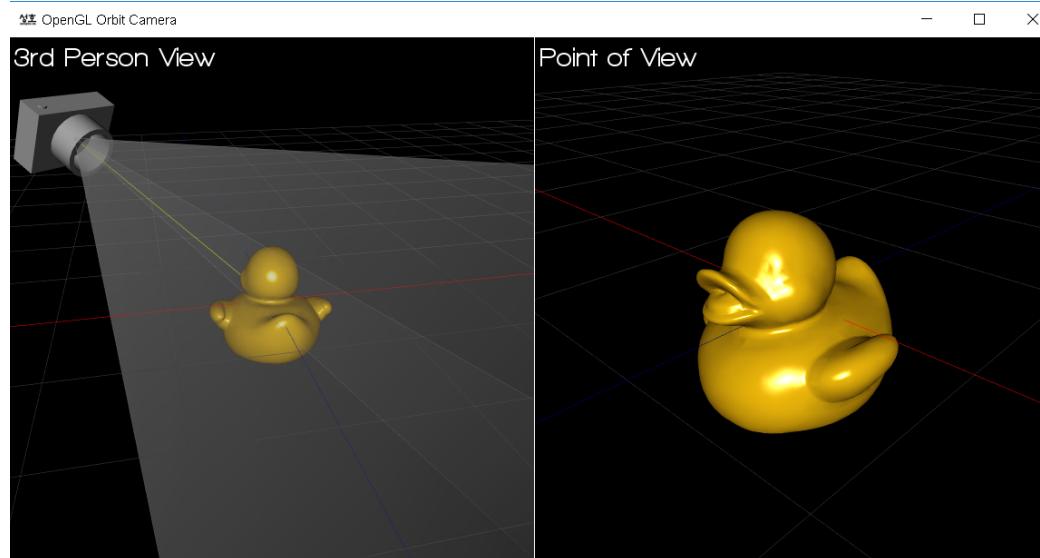
3.1. Mètodes projectius

- RayTracing
- GPU (Zbuffer)



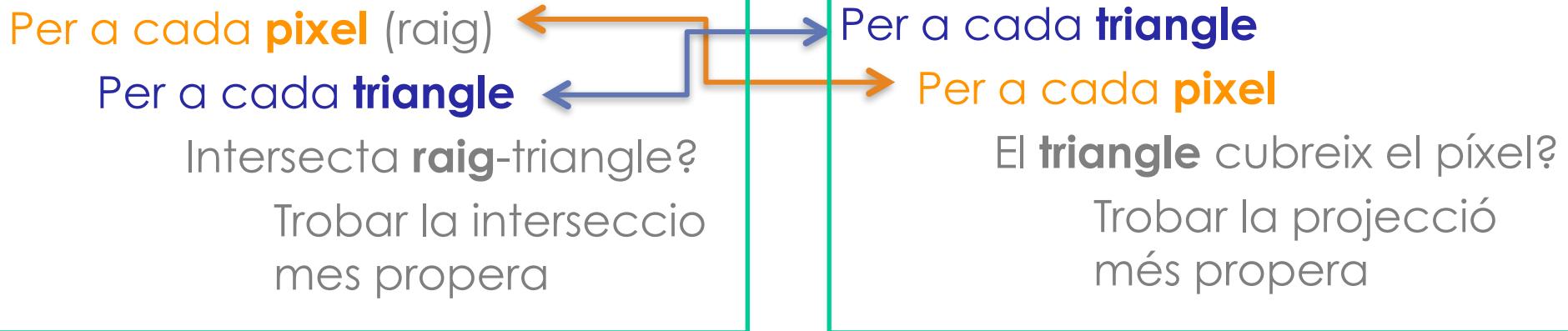
Característiques:

- Càmera



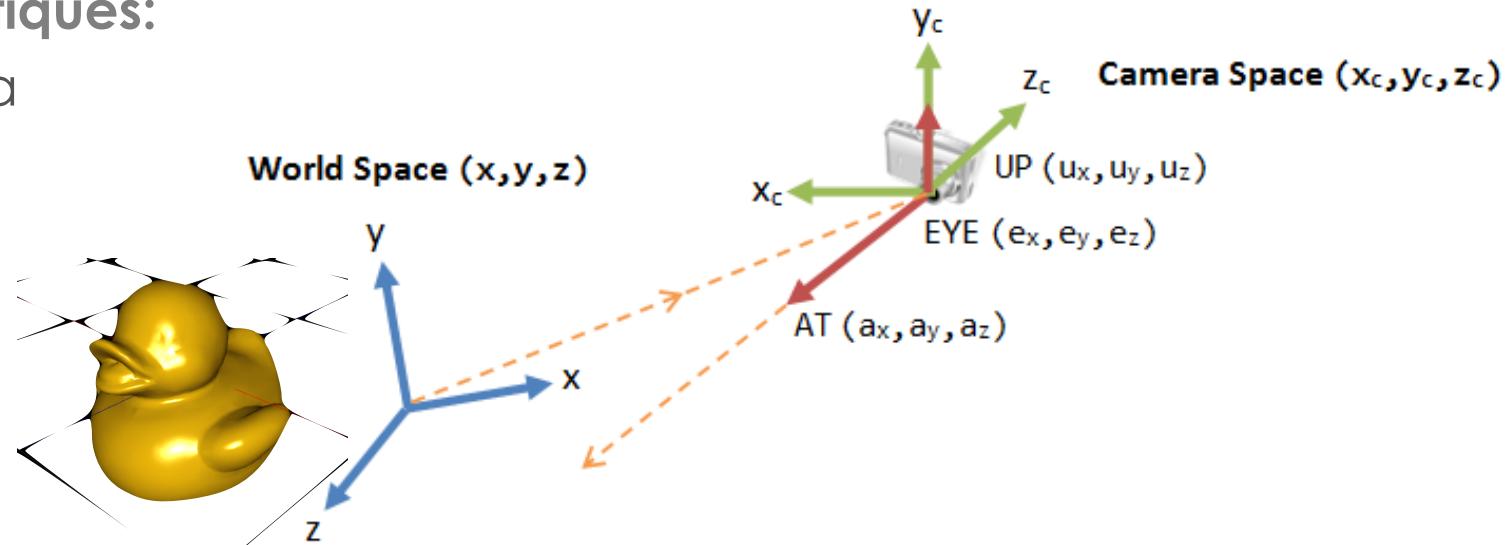
3.1. Mètodes projectius

- RayTracing
- GPU (Zbuffer)



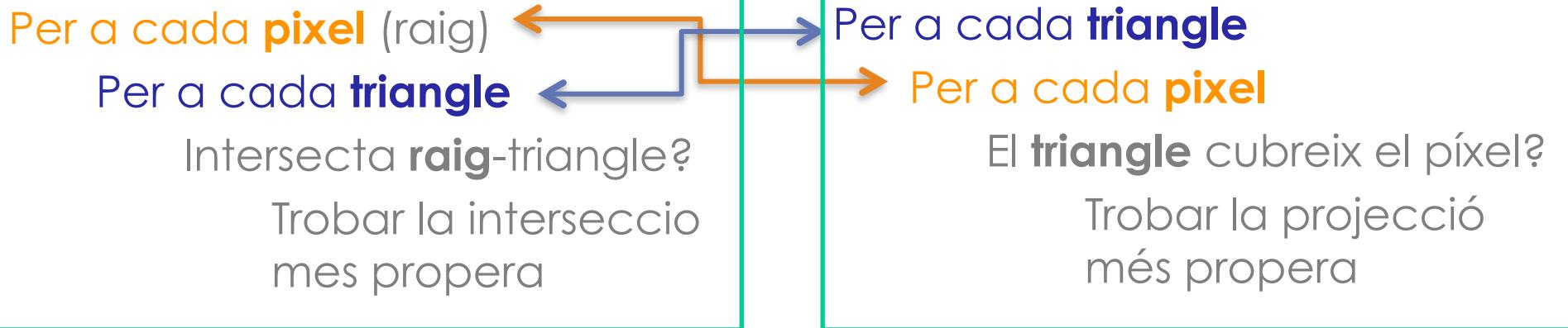
Característiques:

- Càmera



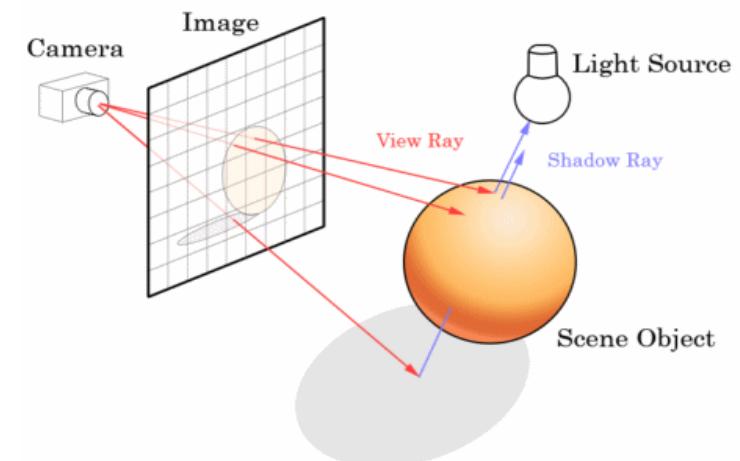
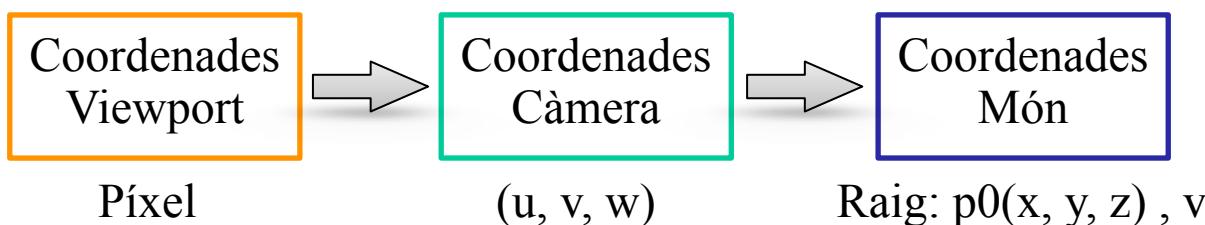
3.1. Mètodes projectius

- RayTracing
- GPU (Zbuffer)



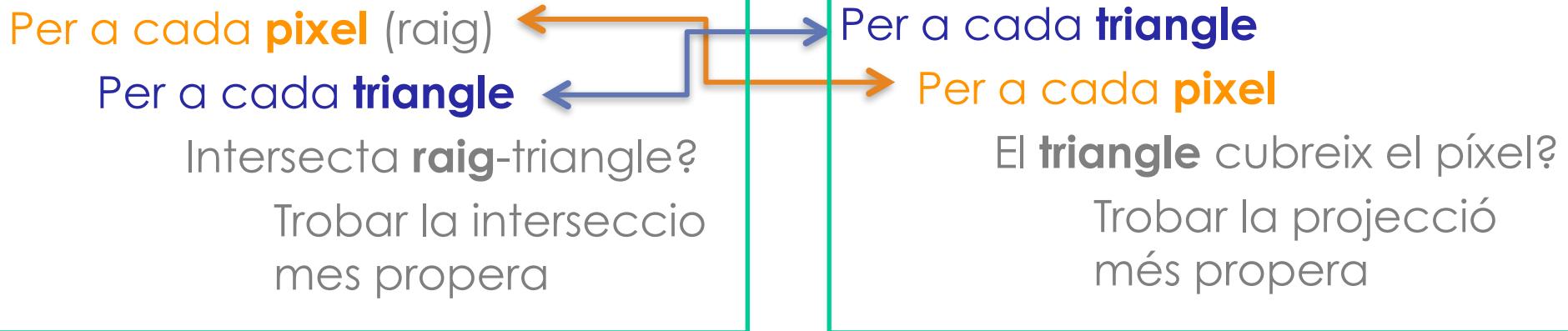
Característiques:

- Càmera:
A RayTracing:



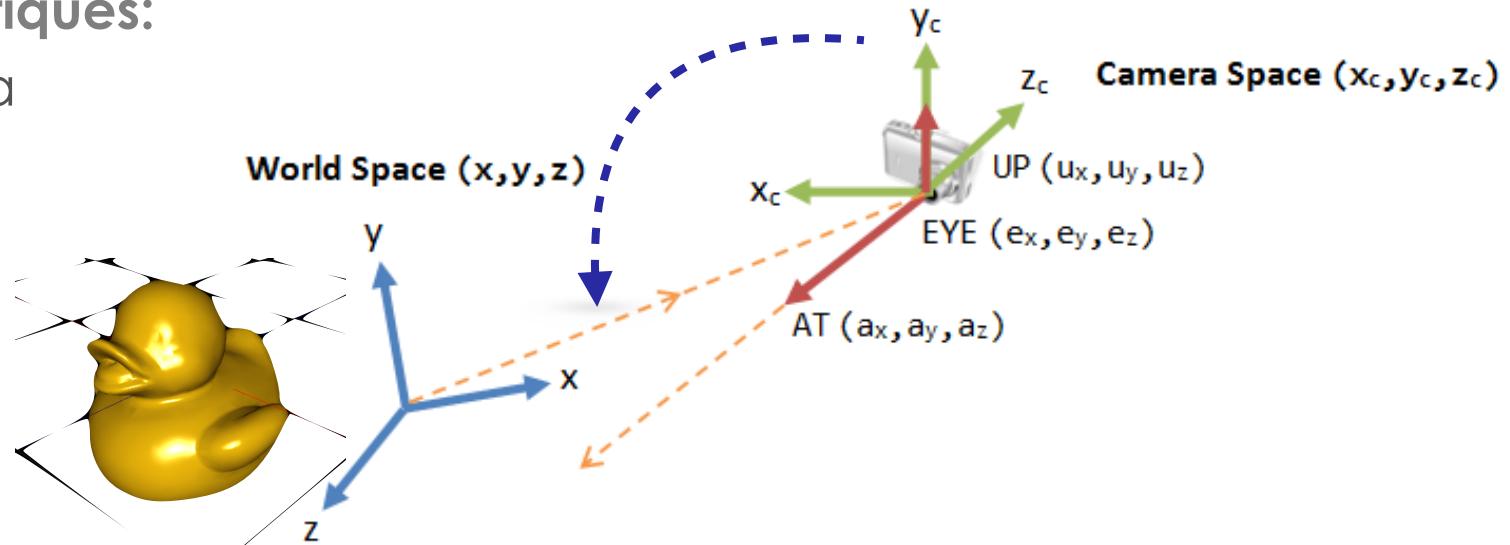
3.1. Mètodes projectius

- RayTracing
- GPU (Zbuffer)



Característiques:

- Càmera



3.1. Mètodes projectius

- RayTracing
- GPU (Zbuffer)

Per a cada **pixel** (raig)

Per a cada **triangle**

Intersecta **raig-triangle**?

Trobar la intersecció
mes propera

Per a cada **triangle**

Per a cada **pixel**

El **triangle** cubreix el píxel?

Trobar la projecció
més propera

Característiques:

- Càmera:

A Projectius:



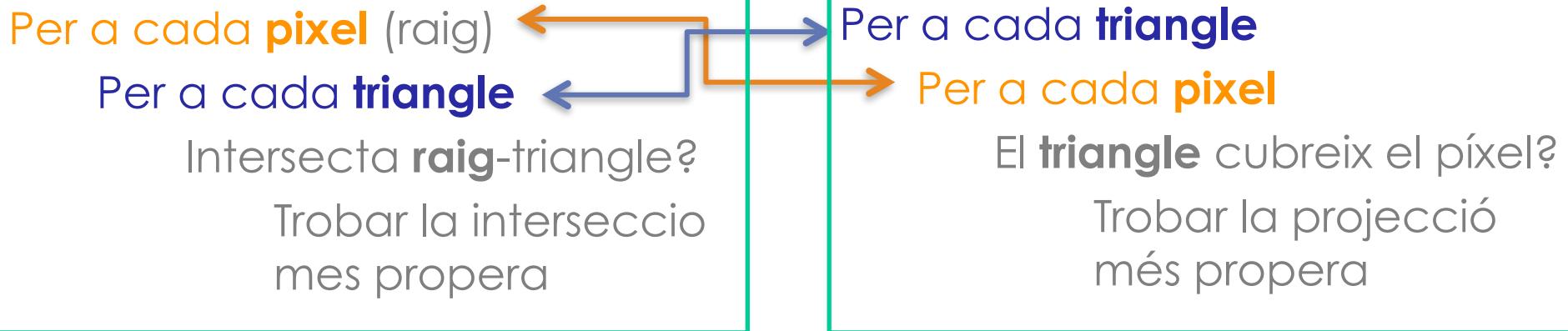
Triangle (p_1, p_2, p_3)
formats per (x, y, z)

(u, v, w)

Píxel

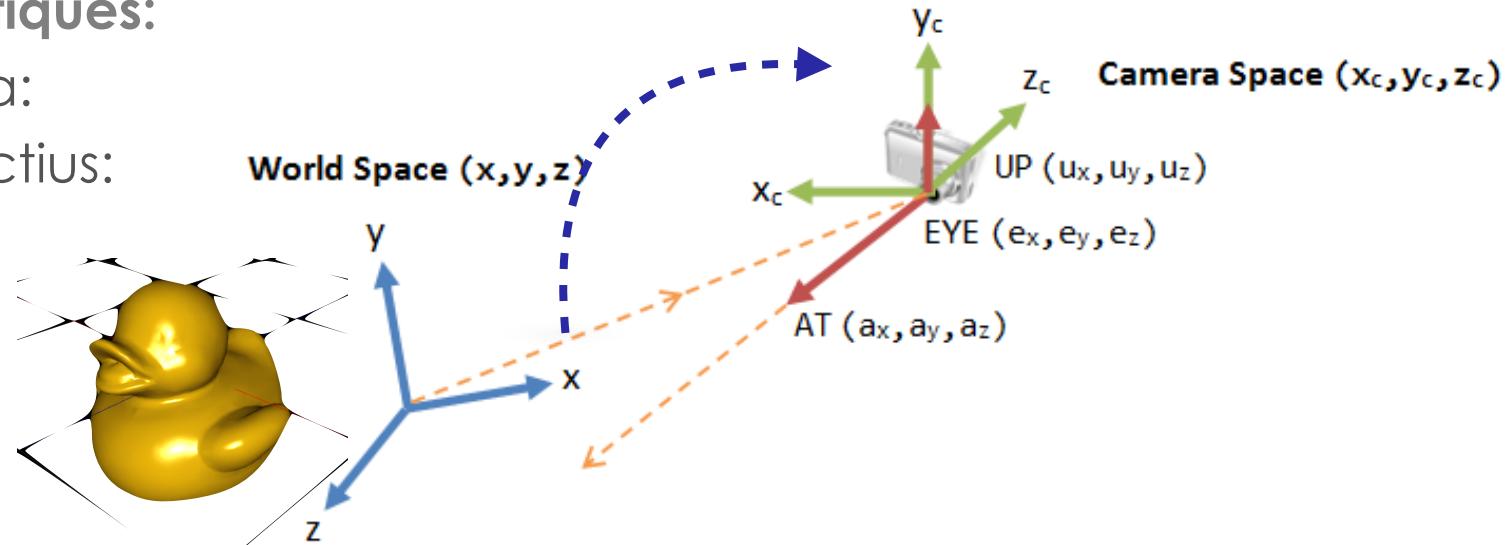
3.1. Mètodes projectius

- RayTracing
- GPU (Zbuffer)



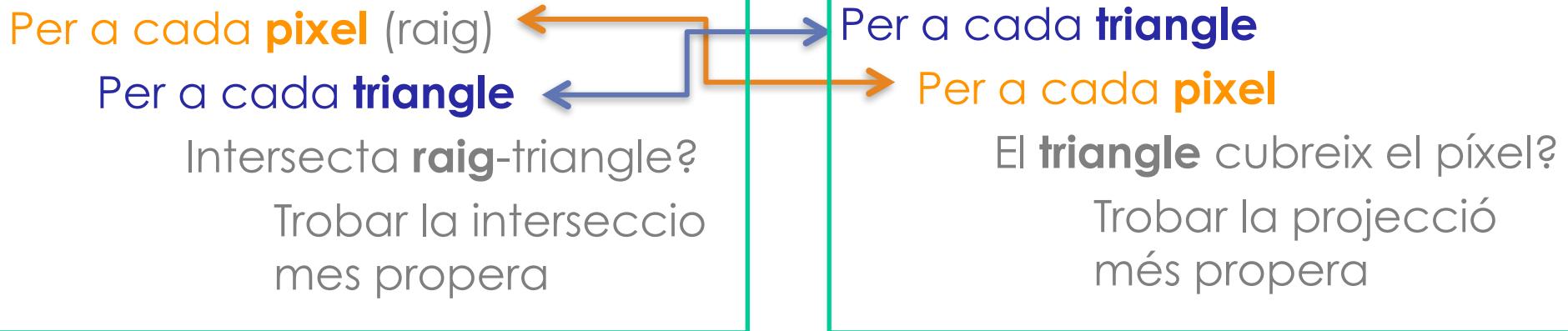
Característiques:

- Càmera:
 - A Projectius:



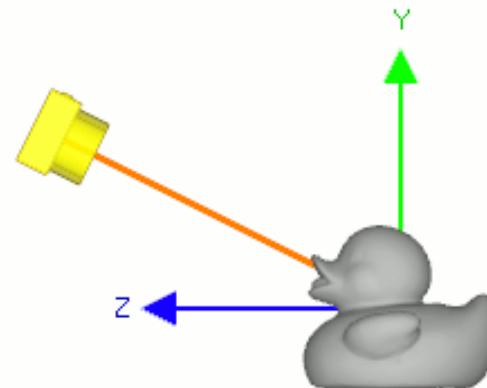
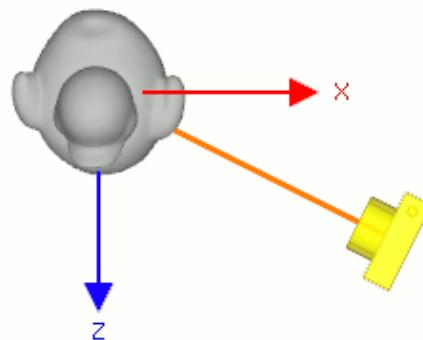
3.1. Mètodes projectius

- RayTracing
- GPU (Zbuffer)

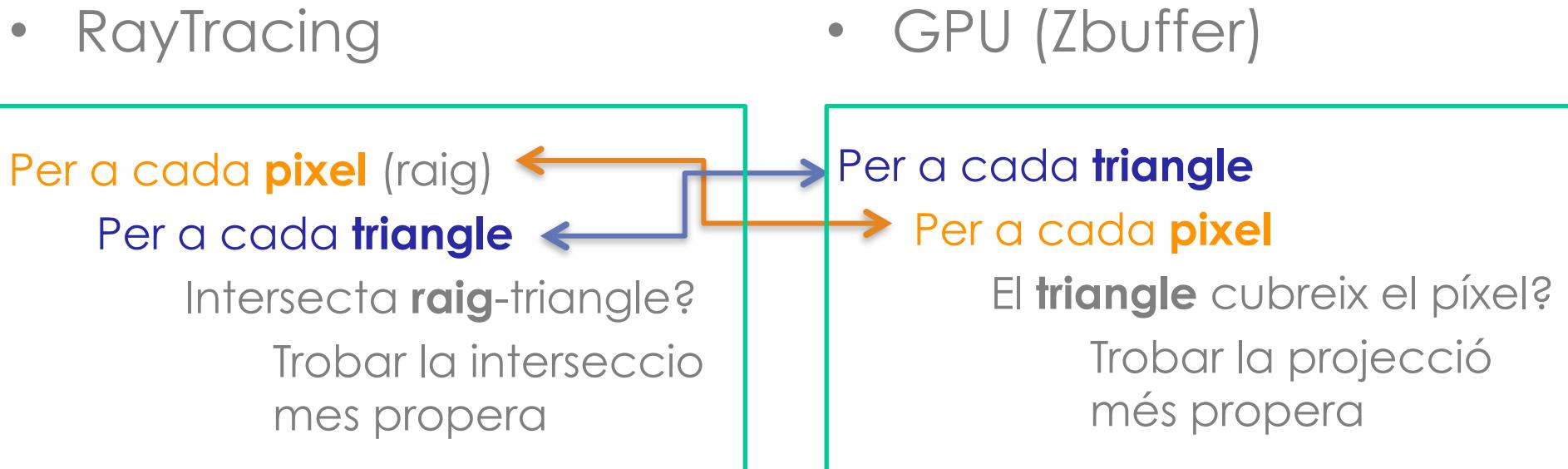


Característiques:

- Càmera A Projectius



3.1. Mètodes projectius



Característiques:

- Càmera:
A Projectius: A cada triangle se li aplica la transformació per a passar de coordenades de món (x, y, z) a coordenades de càmera (o d'observador), per acabar deixant la direcció de visió en l'eix z

matriu modelView

3.1. Mètodes projectius

- RayTracing

Per a cada **pixel** (raig)

Per a cada **triangle**

Intersecta raig-triangle?

Trobar la intersecció
mes propera

- Projectius

Per a cada **triangle**

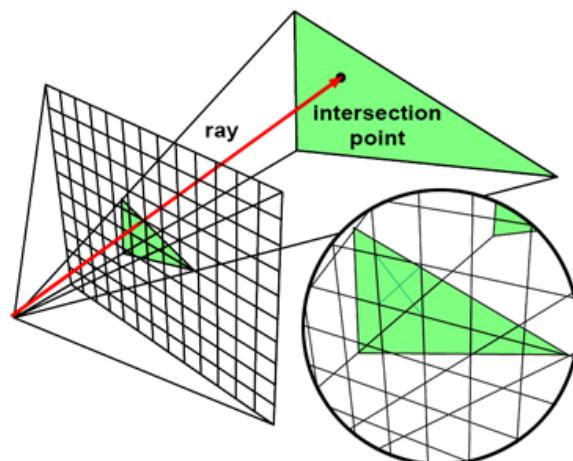
Per a cada **pixel**

El triangle cubreix el píxel?

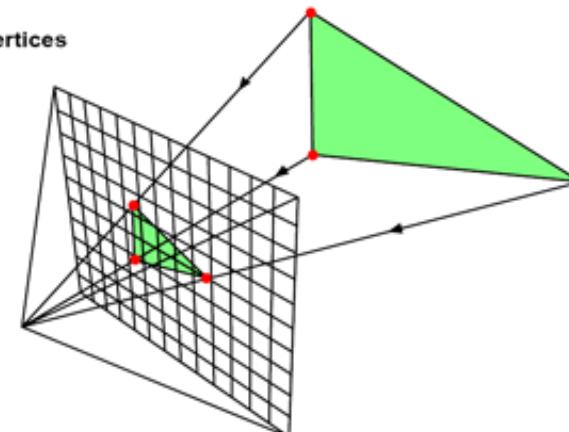
Trobar la projecció
més propera

Característiques:

- Rasterització: calcular els píxels que cubreix un triangle



1) Project vertices



© www.scratchapixel.com

3.1. Mètodes projectius

- RayTracing

Per a cada **pixel** (raig)

Per a cada **triangle**

Intersecta raig-triangle?

Trobar la intersecció
mes propera

- Projectius

Per a cada **triangle**

Per a cada **pixel**

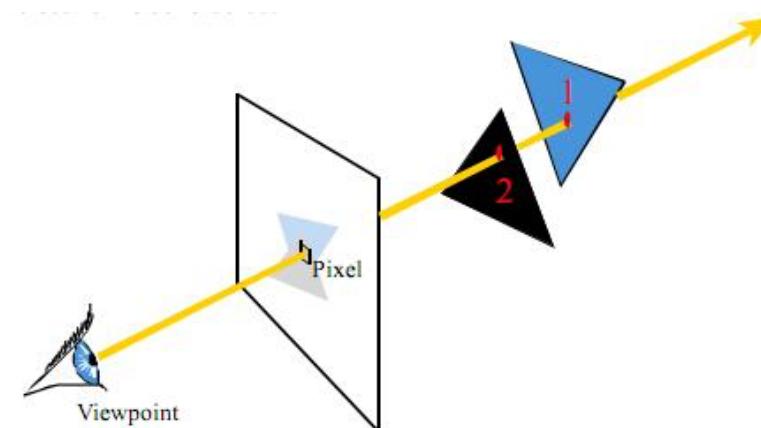
El triangle cubreix el píxel?

Trobar la projecció
més propera

Característiques:

- Visibilitat:

Raytracing: intersecció
raig primari i tots els
objectes de l'escena,
agafant la t minima



3.1. Mètodes projectius

- RayTracing

Per a cada **pixel** (raig)

Per a cada **triangle**

Intersecta raig-triangle?

Trobar la intersecció
mes propera

- Projectius

Per a cada **triangle**

Per a cada **pixel**

El triangle cubreix el píxel?

Trobar la projecció
més propera

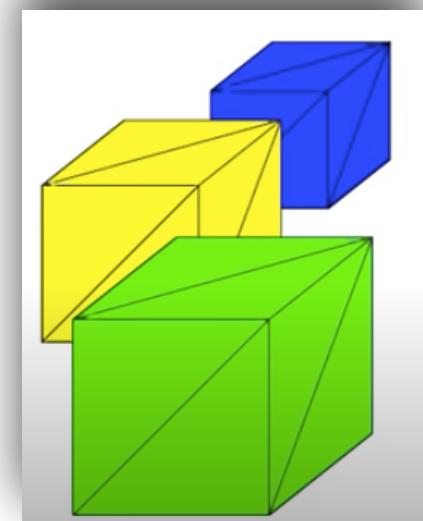
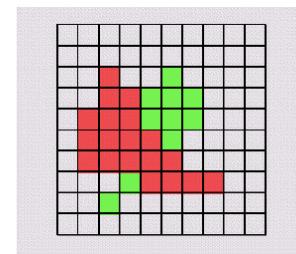
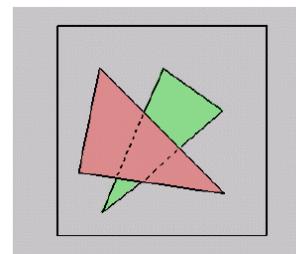
Característiques:

- Visibilitat:

Projectius: l'objectiu és acabar pintant el triangle més proper a l'observador.

Dos estratègies:

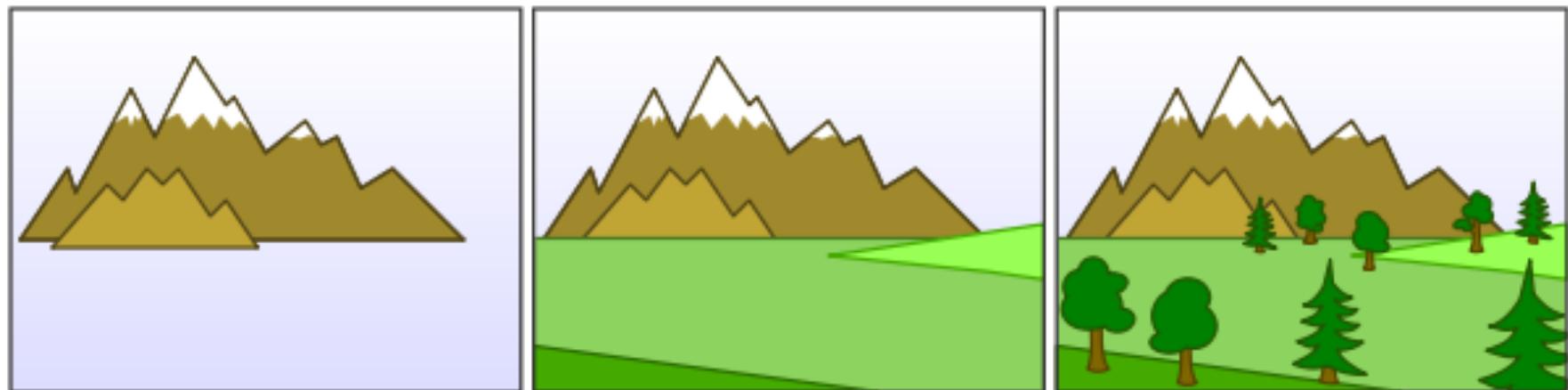
- algorisme del **Pintor**
- algorisme de **ZBuffer**



3.1. Mètodes projectius: Pintor

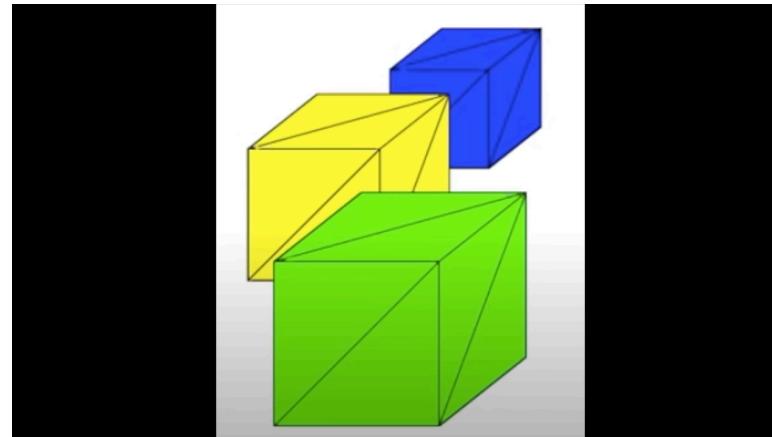
Algorisme del Pintor (Painter's Algorithm)

- Inspirat de com pinten els pintors
- S'ordenen els objectes o triangles de z's més llunyanes a z's més properes
- Es projecten els triangles en aquest ordre

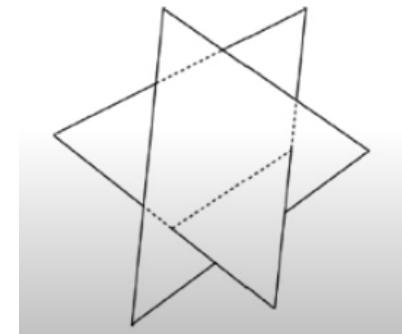


3.1. Mètodes projectius: Pintor

Algorisme del Pintor (Painter's Algorithm)



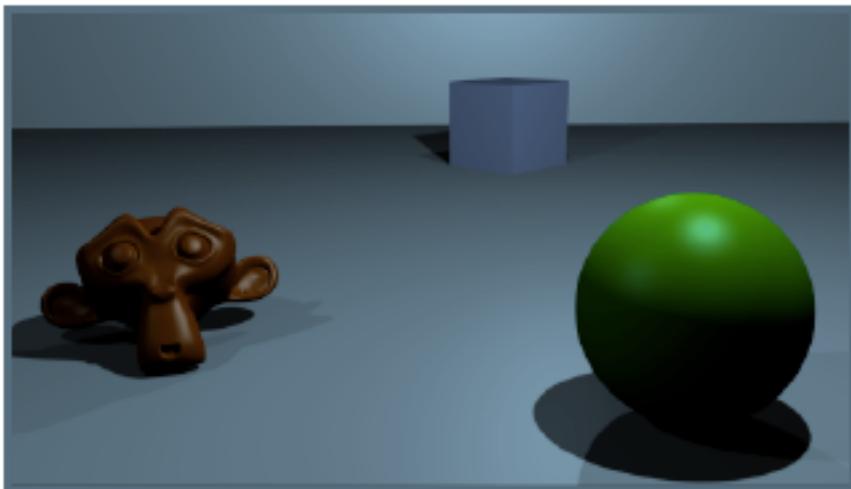
Problemes: Depèn de com estiguin situats els triangles, aquest algorisme no funciona bé



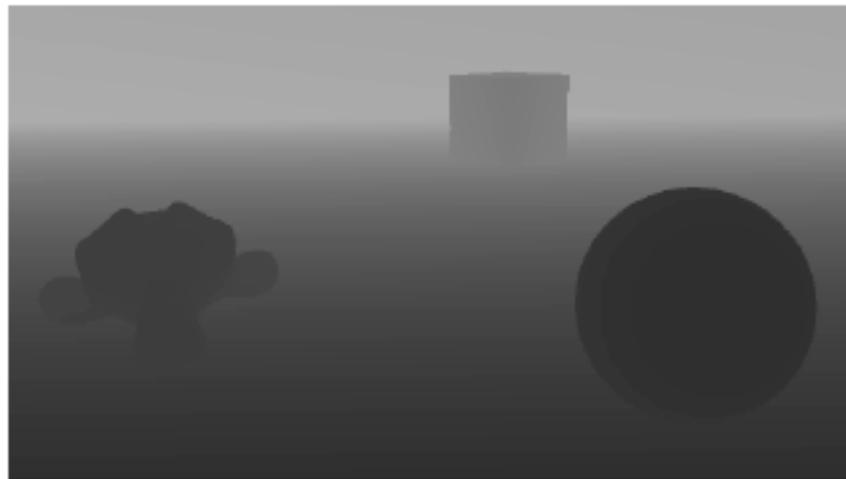
3.1. Mètodes projectius: ZBuffer

Z-Buffer

- Aplicat en coordenades de càmera
- Les z's augmenten a mesura que s'allunyen de l'observador



Escena



Depth-Buffer o z-buffer
(fosc: proper, clar: lluny)

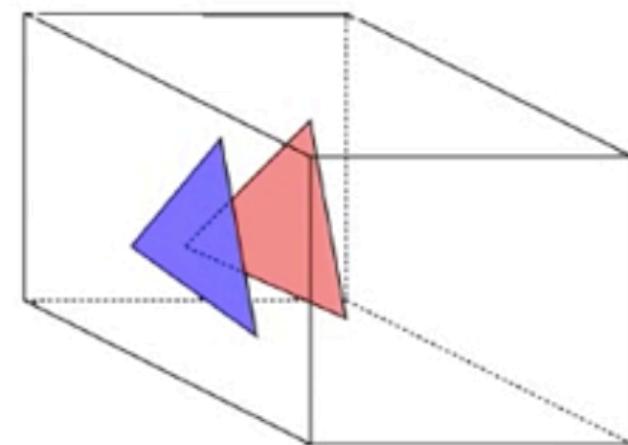
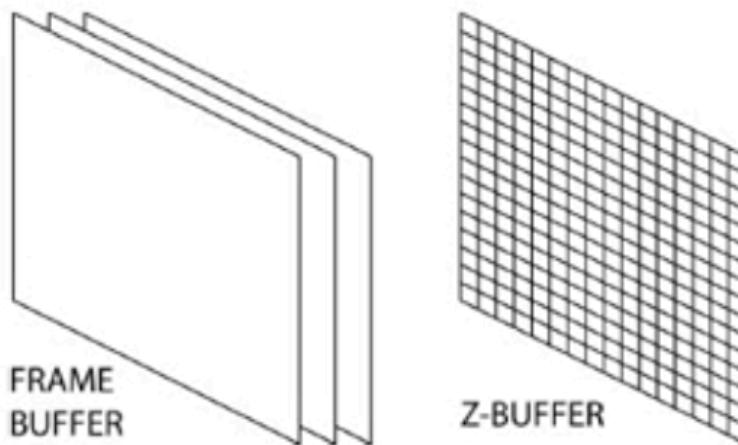
3.1. Mètodes projectius: ZBuffer

Z-Buffer

- Aplicat en coordenades de càmera
- Les z's augmenten a mesura que s'allunyen de l'observador

Estructures:

- Frame Buffer (on es guarden els colors finals)
- Z-Buffer (on es guarden les profunditats)



3.1. Mètodes projectius: ZBuffer

Z-Buffer

Inicialitzar el frame buffer (**colors**) al color de background

Inicialitzar el depth buffer (**depth**) a profunditats més allunyades

Per a cada triangle t

Per a cada píxel (i, j) de la rasterització del triangle t

$z = \text{calculProfunditat}(i, j)$

$c = \text{calculColorTriangle}(i, j)$

si $z < \text{depth}(i, j)$ llavors

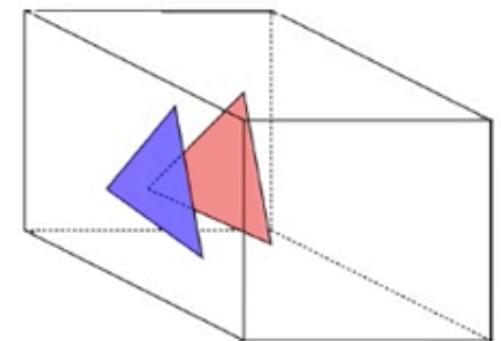
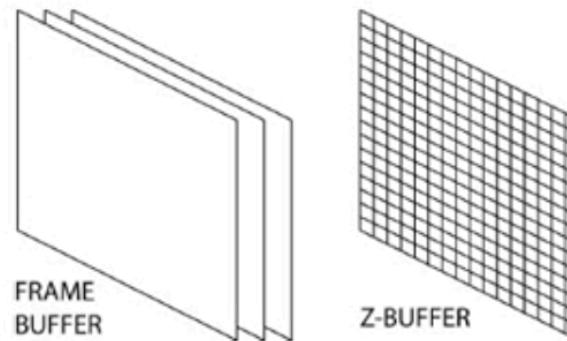
$\text{depth}(i, j) = z$

$\text{color}(i, j) = c$

fsi

fper

fper



3.1. Mètodes projectius: ZBuffer

Z-Buffer

Inicialitzar el frame buffer (**colors**) al color de background

Inicialitzar el depth buffer (**depth**) a profunditats més allunyades

Per a cada triangle t

Per a cada píxel (i, j) de la rasterització del triangle t

$z = \text{calculProfunditat}(i, j)$

$c = \text{calculColorTriangle}(i, j)$

si $z < \text{depth}(i, j)$ **llavors**

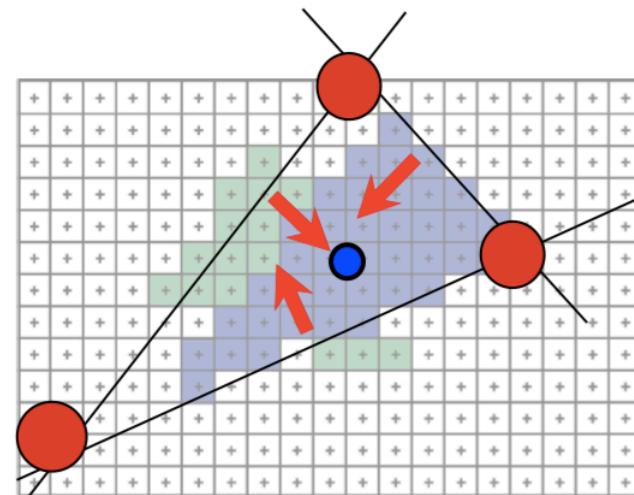
$\text{depth}(i, j) = z$

$\text{color}(i, j) = c$

fsi

fper

fper



depth dels píxels dels vèrtexs del triangle **coneuguda**
depth dels píxels interns, **interpolada**

3.1. Mètodes projectius: ZBuffer

Z-Buffer

Els valors de Z sempre són positius i van des del pla de clipping anterior (z_{near}) al posterior (Z_{far}).

El z-buffer enmagatzema enters positius
 $b = \{0, 1, \dots, B-1\}$

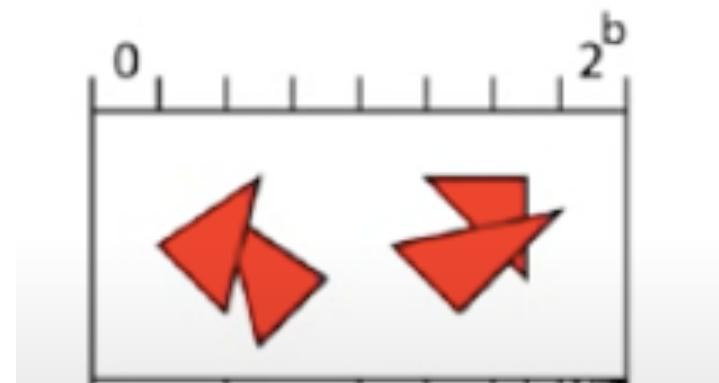
Els valors de z's en coordenades de càmera es mapegen a aquests valors, en intervals regulars de mida

$$\Delta z = \frac{(Z_{far} - Z_{near})}{B}$$

I un z d'un punt es calcula com:

$$\frac{z - z_{near}}{Z_{far} - z_{near}} * (B - 1)$$

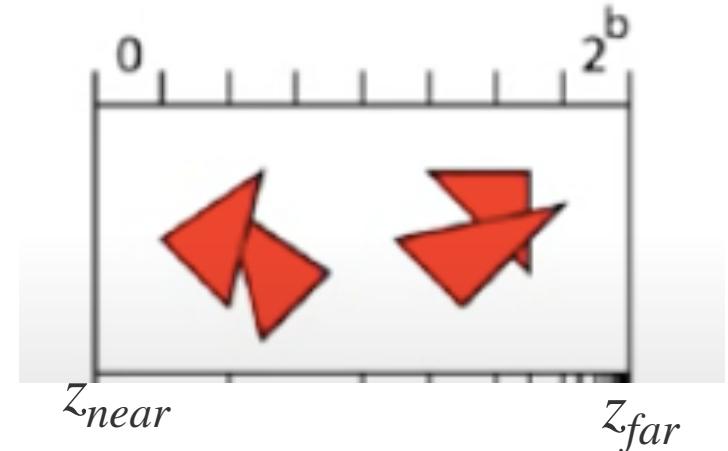
Si el zbuffer és un canal de b bits, es tenen 2^b buckets



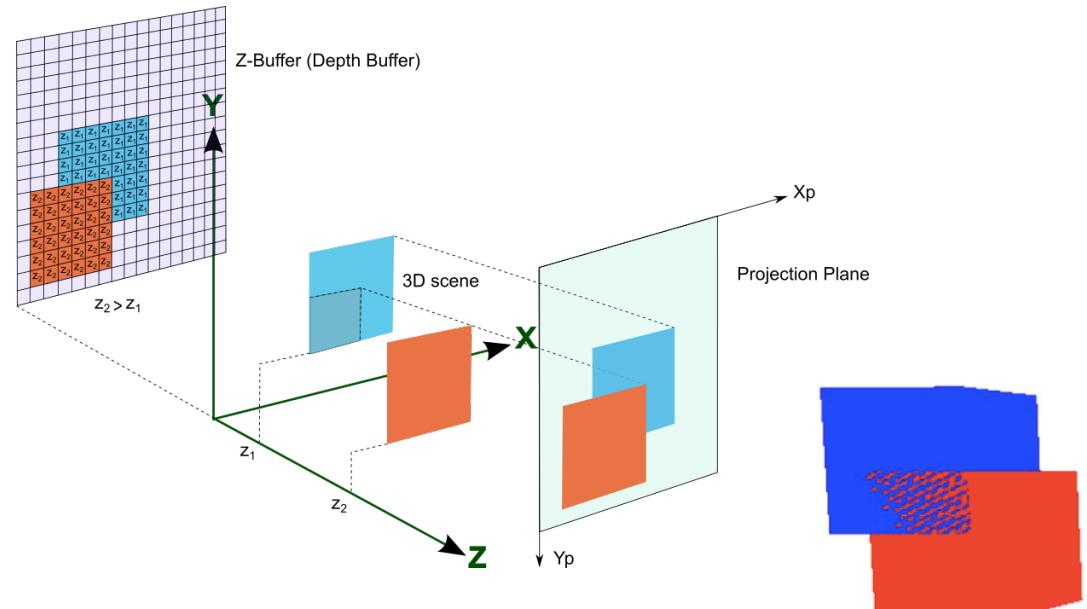
3.1. Mètodes projectius: ZBuffer

Z-Buffer:

Z-fighting problem: Problema que es dóna quan dos o més triangles van a parar al mateix bucket en z, i no es pot decidir qui d'ells són més propers



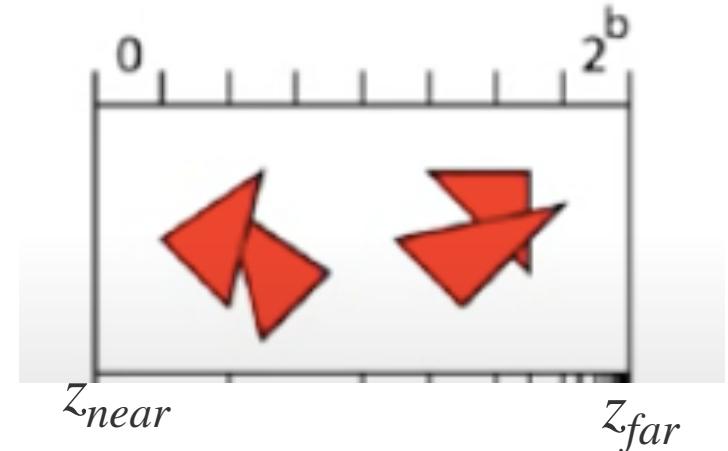
Com es podria solucionar?



3.1. Mètodes projectius: ZBuffer

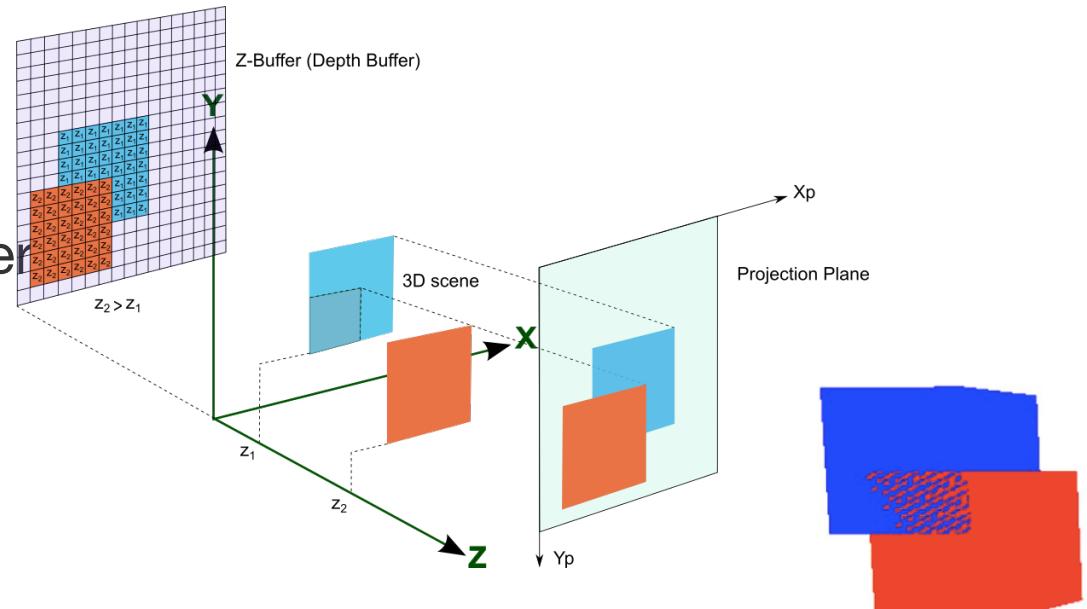
Z-Buffer:

Z-fighting problem: Problema que es dóna quan dos o més triangles van a parar al mateix bucket en z, i no es pot decidir qui d'ells són més propers

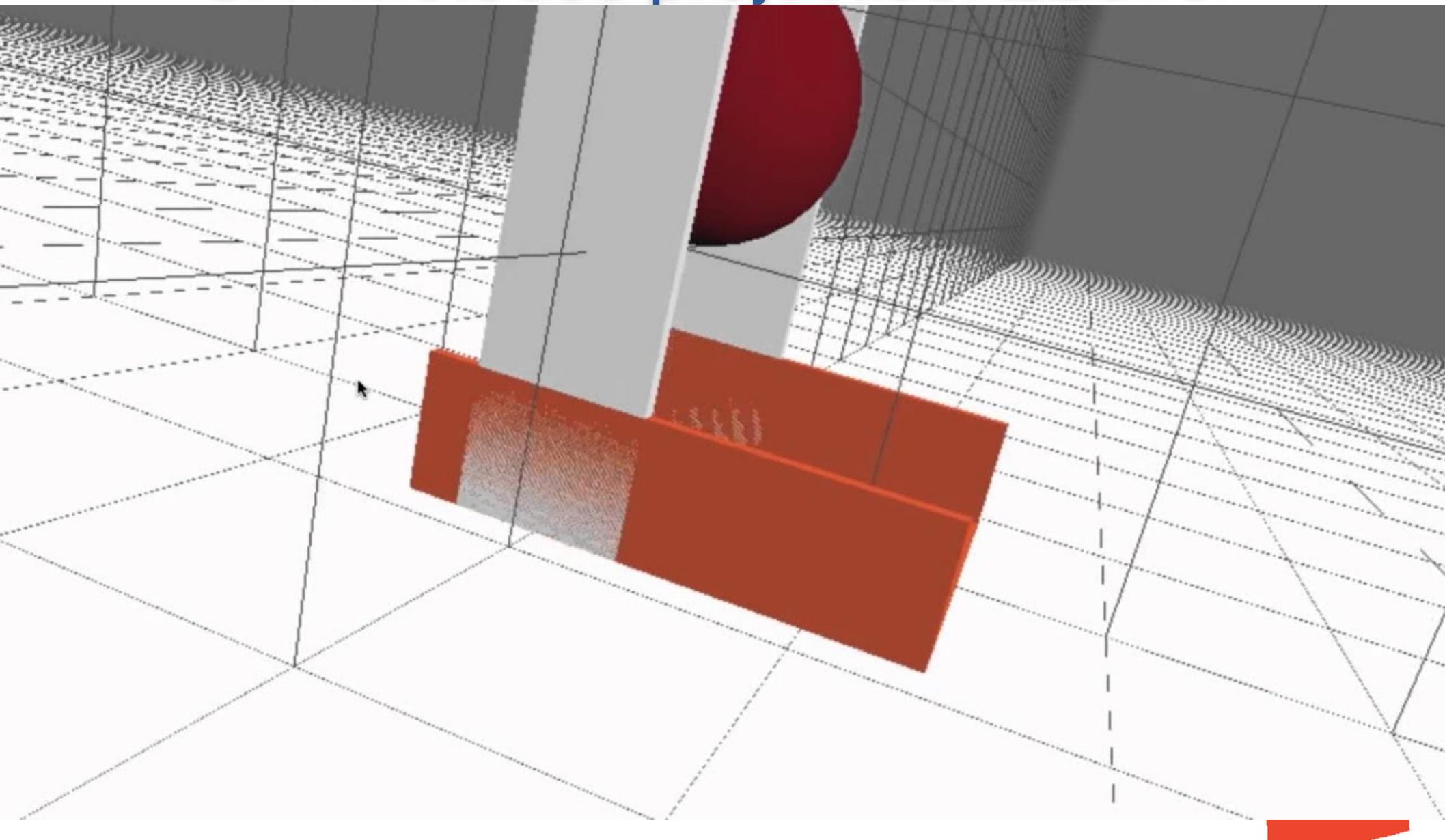


Com es podria solucionar?

- Posant més bits de canal al ZBuffer
- Allunyant el z_{near}
- Apropant el z_{far}



3.1. Mètodes projectius: ZBuffer



3.1. Mètodes projectius: ZBuffer



Fallout - New Vegas

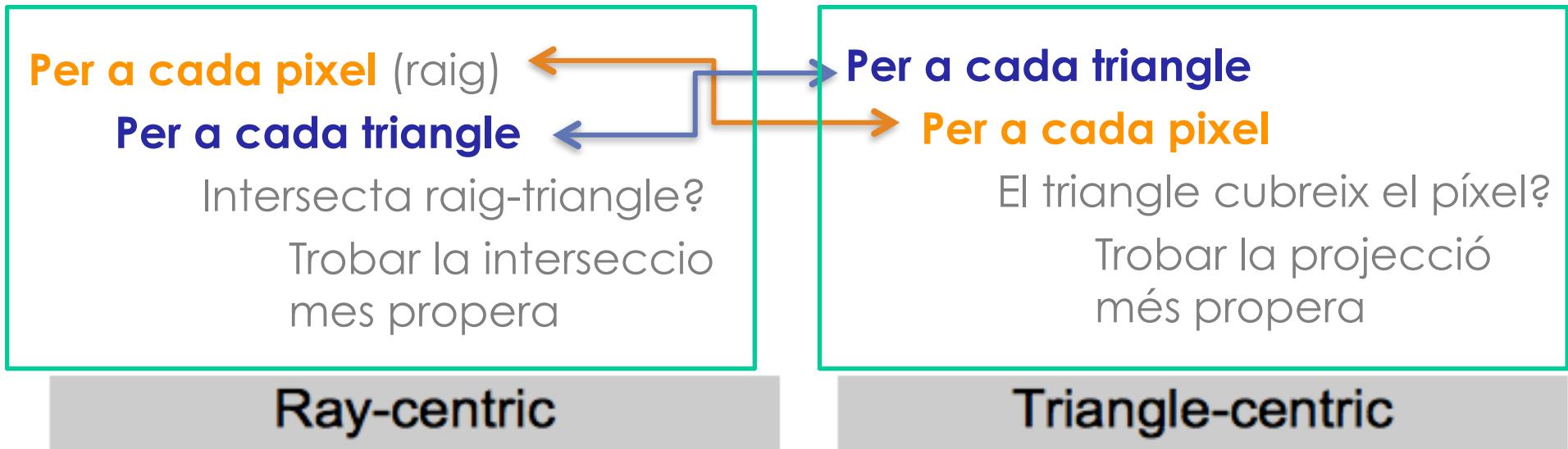
- Posant més bits de canal al ZBuf
- Allunyant el z_{near}
- Apropant el z_{far}

Skyrim - Elder Scrolls



3.1. Mètodes projectius: ZBuffer

- RayTracing
- GPU (Zbuffer)



Quina **memòria** es necessita a cada cas per a tenir la informació d'un píxel?

- **RayTracing:** Necessita tota l'escena en memòria en un cert instant
- **Zbuffer:** Necessita només un triangle a la vegada i una imatge amb les profunditats.

Una escena es sovint més complicada que una imatge.

Una imatge de 1920x1080 pixels de colors codificats amb 16 bits per canal de color (RGBA) i un depth buffer 32-bits suposaria uns 24MB.

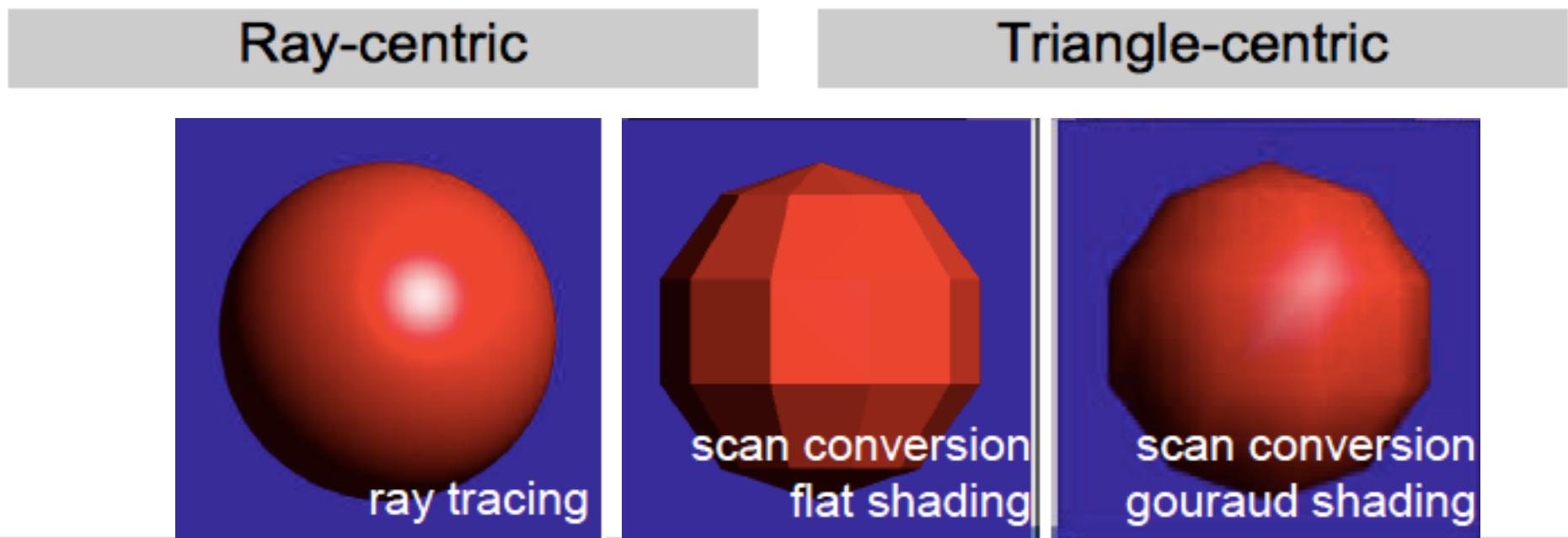
Si hi hagués més d'una mostra per píxel (4) serien aprox 100 MB

3.1. Mètodes projectius: ZBuffer

- RayTracing
- GPU (Zbuffer)

Per a cada pixel (raig)
Per a cada triangle
Intersecta raig-triangle?
Trobar la intersecció
mes propera

Per a cada triangle
Per a cada pixel
El triangle cubreix el píxel?
Trobar la projecció
més propera



3.1. Mètodes projectius: ZBuffer

RayTracing

- Avantatges:

- General: es pot visualitzar tot objecte que es pot calcular la intersecció amb un raig
- Recursiu de forma que es poden calcular **ombres, reflexions, transparències**

- Desavantatges:

- Difícil d'implementar amb hardware. L'escena ha de cabre en memòria, dependències entre dades, etc.
- Lent per fer visualitzacions interactives sino es disposa d'una **gràfica molt especialitzada** (Nvidia Titan V amb arquitectura Volta GPU)

<https://devblogs.nvidia.com/introduction-nvidia-rtx-directx-raytracing/>

ZBuffer

- Avantatges:

- Permet **paral·lelitzar** i fer streaming dels triangles en **targetes gràfiques asequibles**
- Els objectes **no** tenen perquè estar **ordenats** en profunditat
- No hi ha dependències entre dades.

- Desavantatges:

- **Restringit** a objectes formats per primitives que es pugui rasteritzar
- Artefactes de *shading*
- **Ombres/reflexions/transparències** no es tracten de forma directa
- Problema de redibuixat un mateix píxel

3.1. Mètodes projectius: ZBuffer



Índex

3.1. Introducció a ZBuffer

3.2. Pipeline de visualització

3.3. Pipeline de visualització a GL

3.4. Il·luminació usant shaders

3.5. Textures

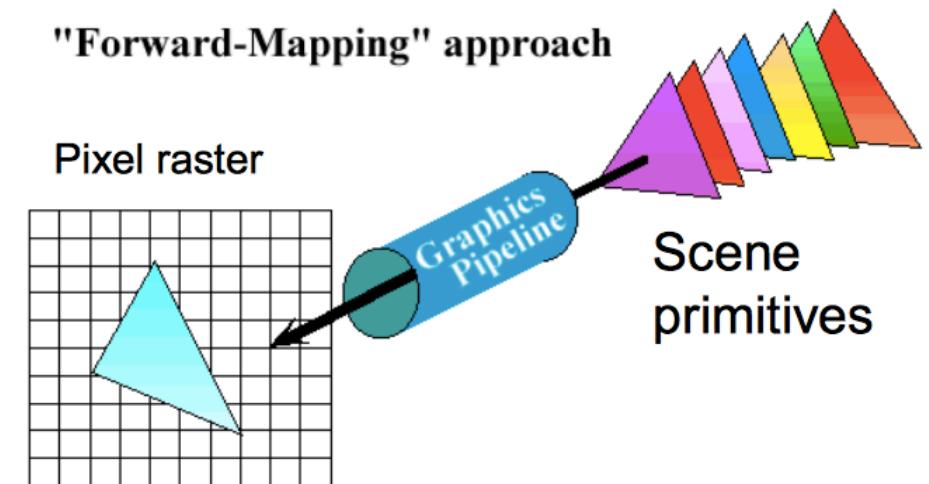
3.6. Reflexions i Transparències

3.2. Pipeline de visualització

- GPU (Zbuffer)

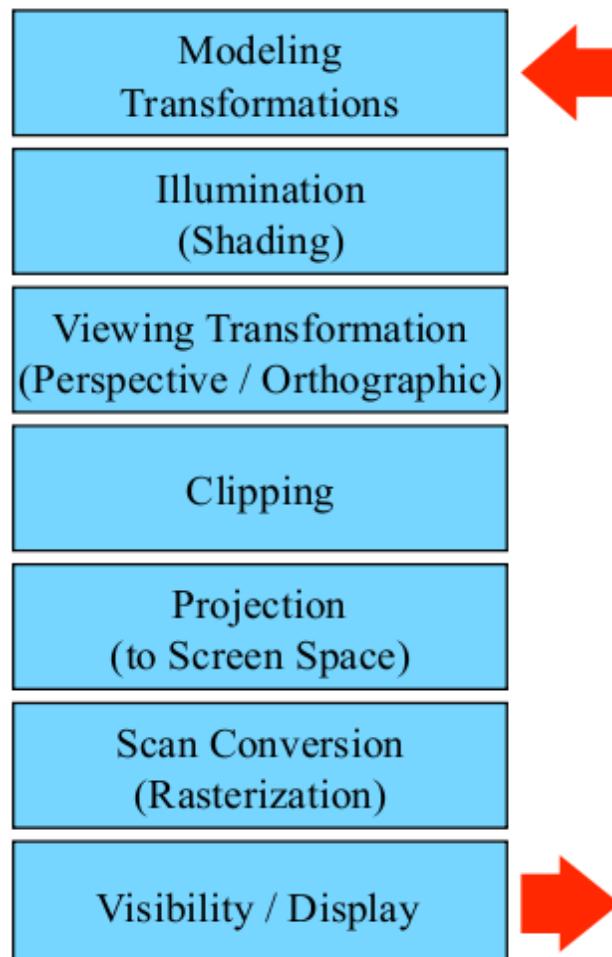
El conjunt de passos o etapes que cal realitzar per a generar una imatge 2D com sortida a partir d'una escena virtual usant un mètode **projectiu** (o Zbuffer) amb suport de la targeta gràfica (GPU) s'anomena **pipeline de visualització** ("graphics pipeline").

Per a cada triangle
Per a cada pixel
El triangle cubreix el píxel?
Trobar la projecció
més propera

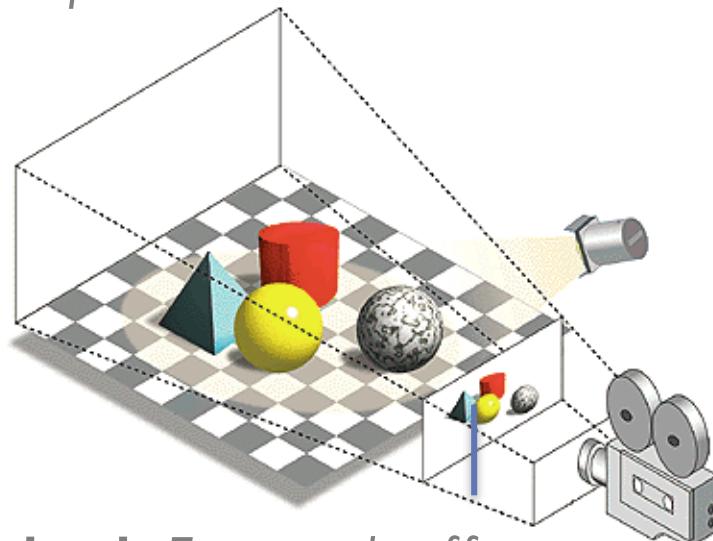


3.2. Pipeline de visualització

Pipeline de visualització: conjunt d'etapes* que a partir de l'escena 3D, llums, càmera i viewport, permet obtenir la imatge 2D amb els colors finals.

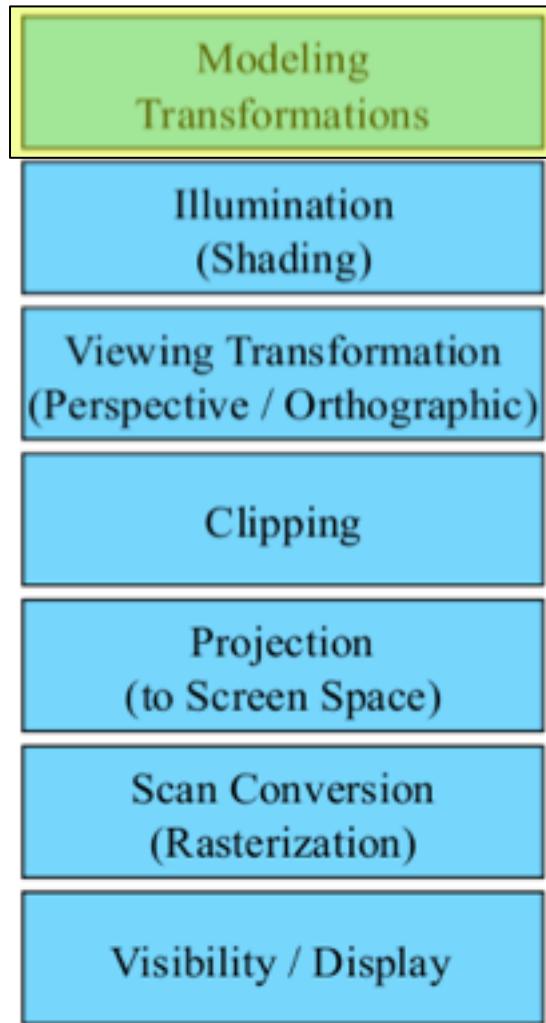


- **Input:** Objectes, Llum, Càmera i viewport

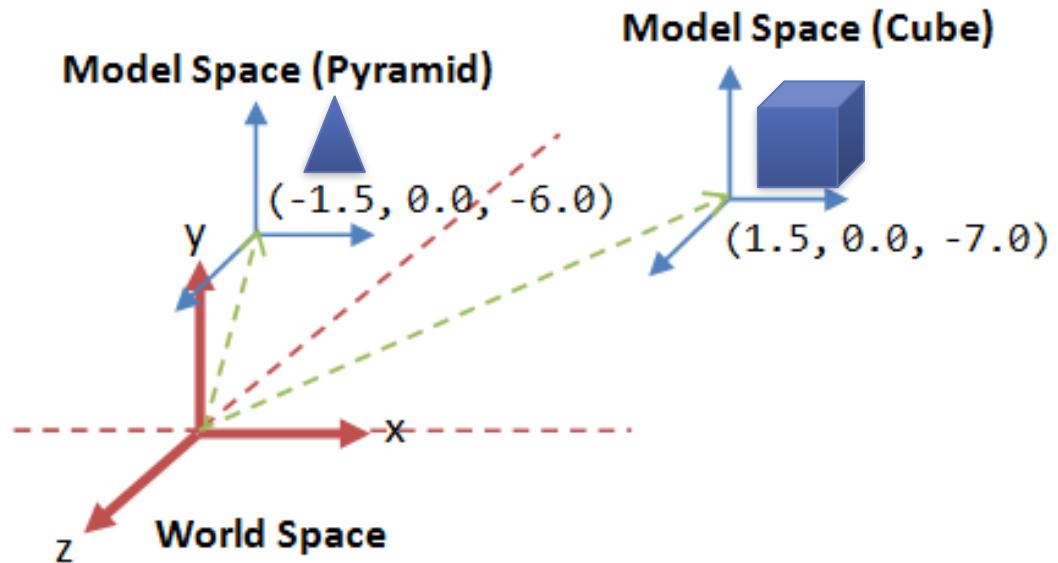


- **Output:** Frame buffer
(Colors/Intensitats (ex. 24-bit RGBA a cada píxel))

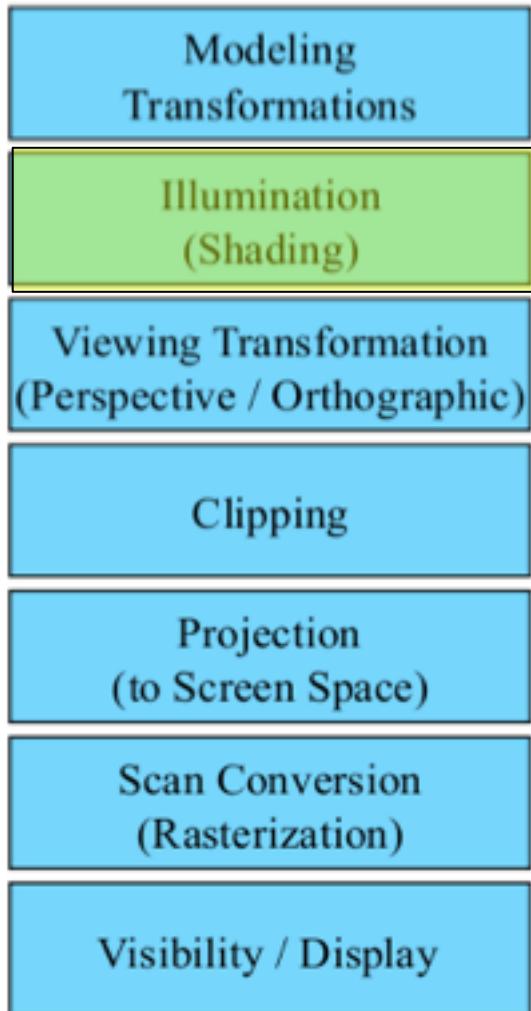
3.2. Pipeline de visualització



- **Transformacions de Modelatge:** A partir dels models 3D i les llums en els seus **sistemes de coordenades locals** (o de model) es calculen les seves posicions en el **sistema de coordenades global** (o de món)

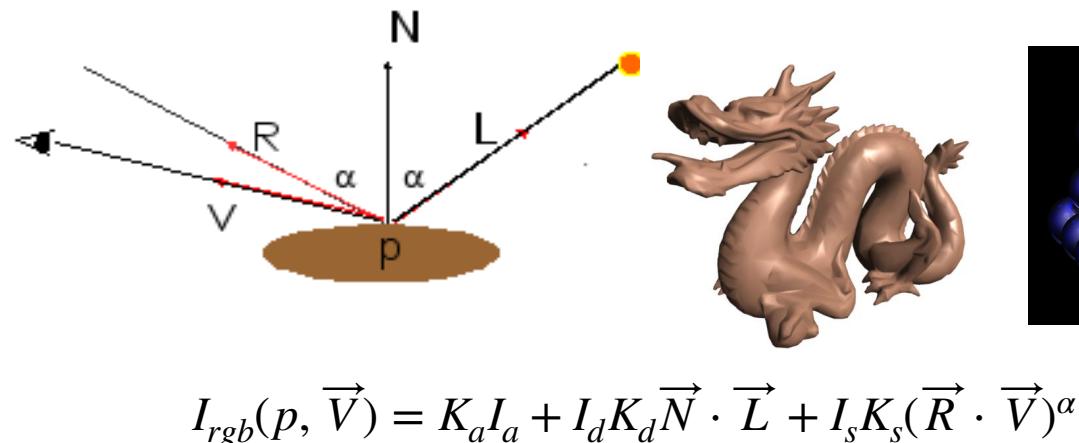


3.2. Pipeline de visualització



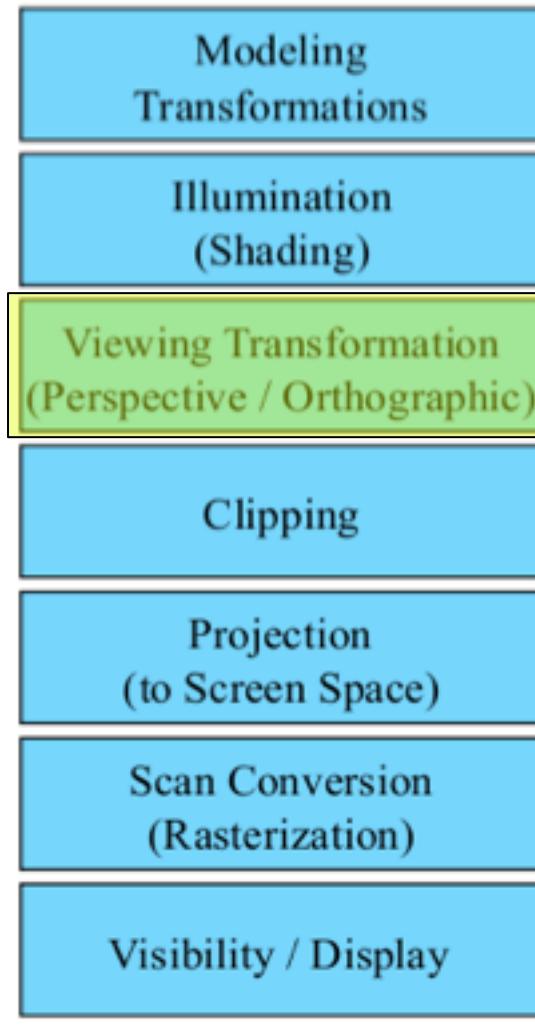
• Il·luminació:

- A partir dels vèrtexs dels objectes ja transformats, es calcula il·luminació (**shading**) de cadascun segons les propietats dels materials, la geometria, les textures i les llums i l'**orientació de la superfície (vector normal)**
- S'usen models d'il·luminació locals (Diffuse, Ambient, Phong, etc.)

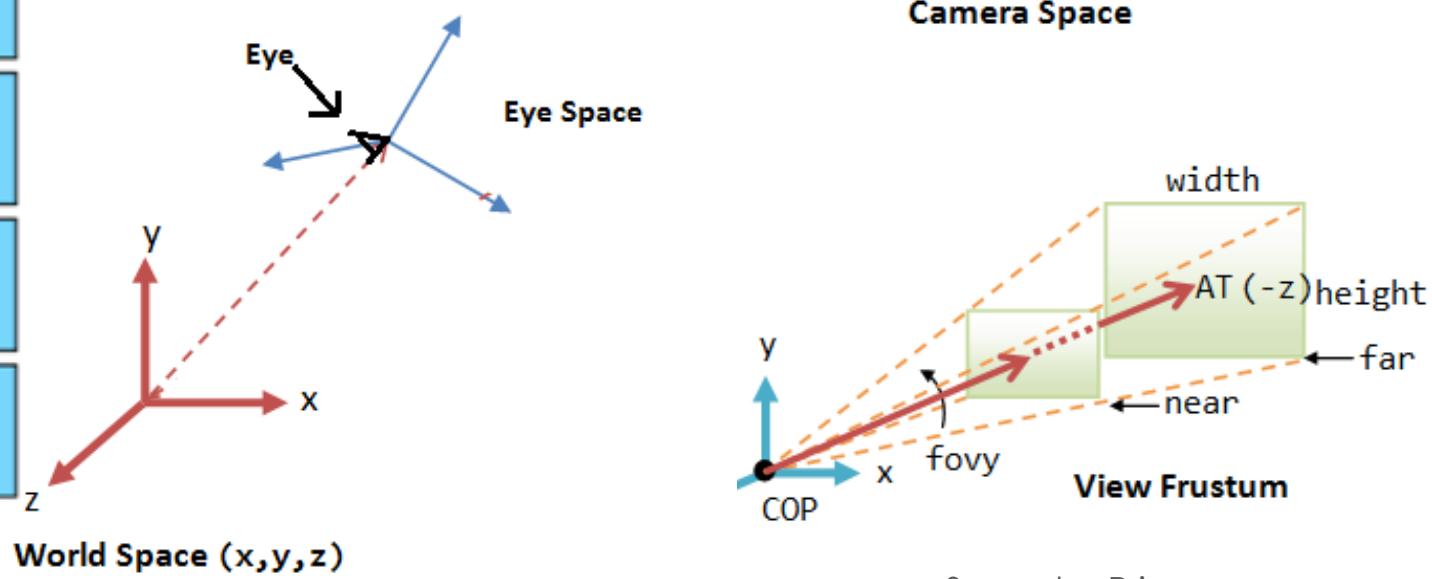


3.2. Pipeline de visualització

- **Transformacions de càmera:**

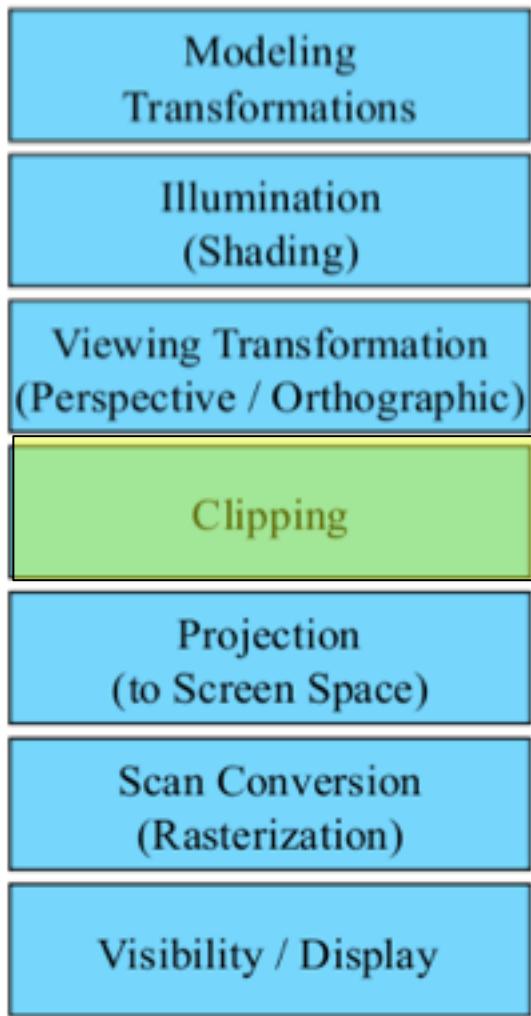


- Es fan les transformacions per a “girar” l'escena segons la posició de la càmera (matriu **model-view**)
- Es projecten els vèrtexs 3D segons el tipus de projecció en el pla de projecció (matriu **projection**)
- S'obtenen punts 2D en l'espai continu

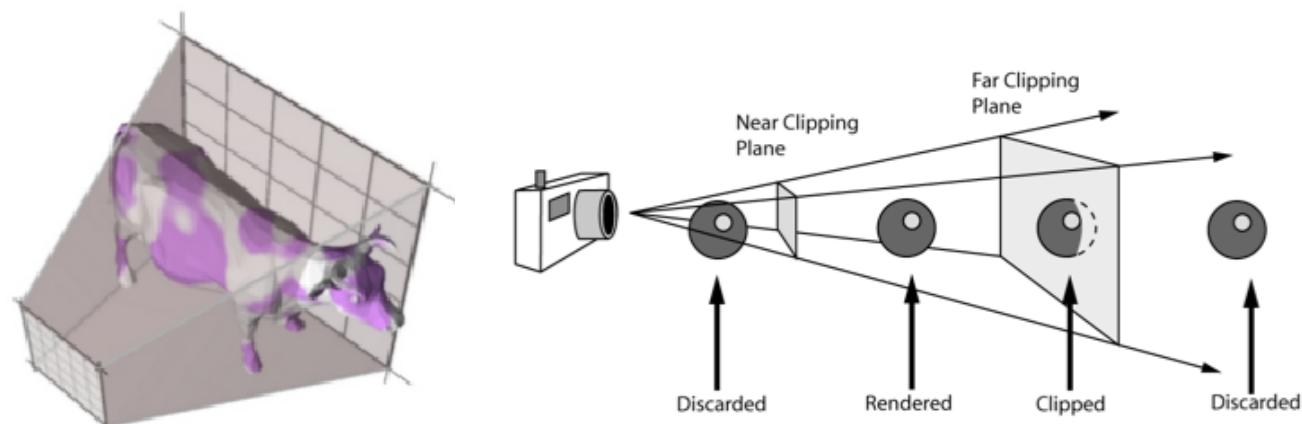
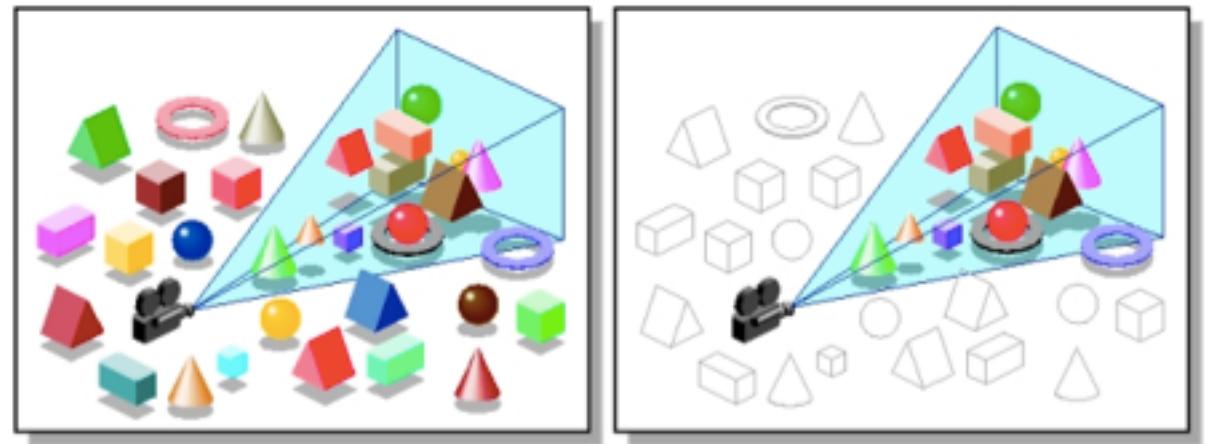


3.2. Pipeline de visualització

- **Clipping o retallat:**



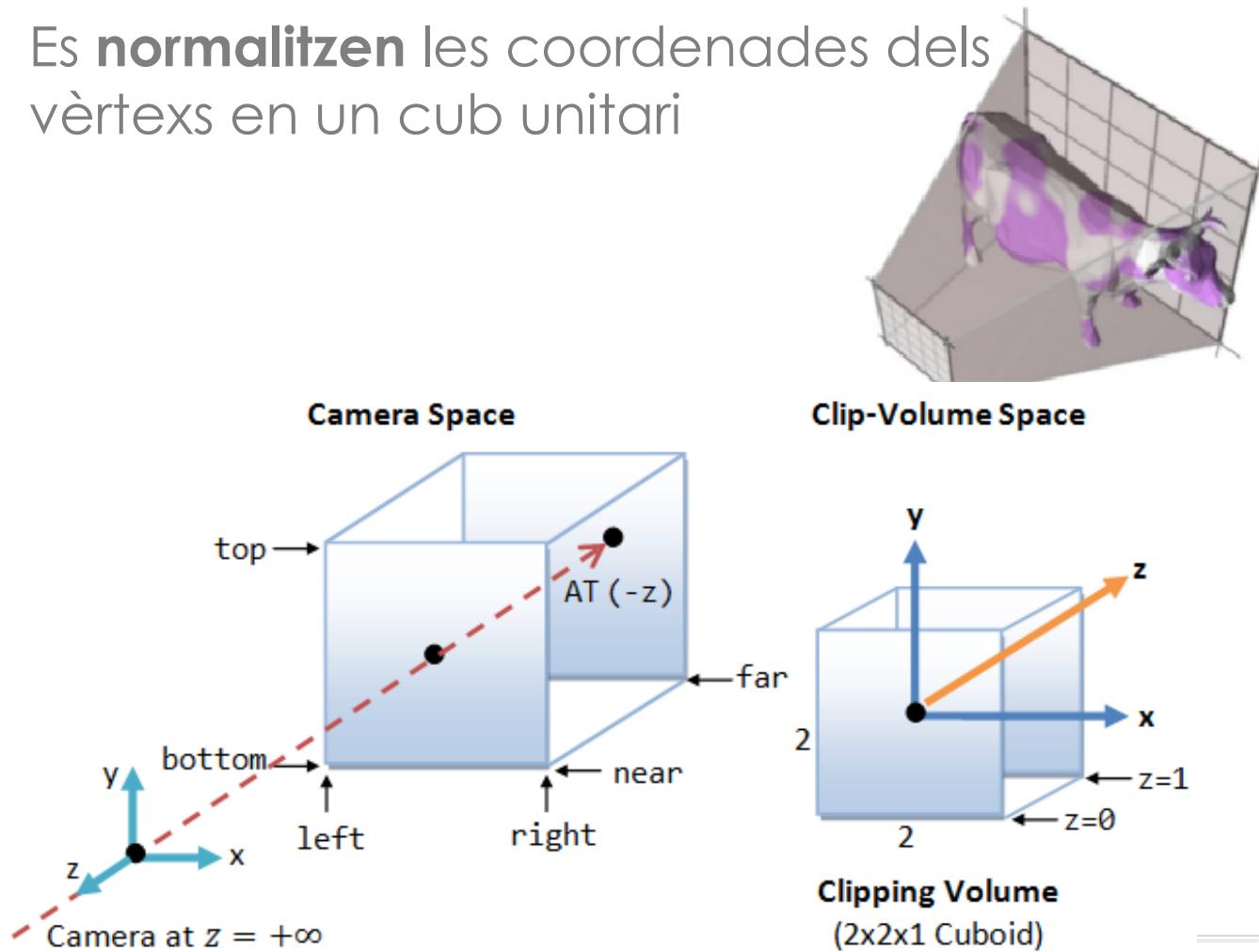
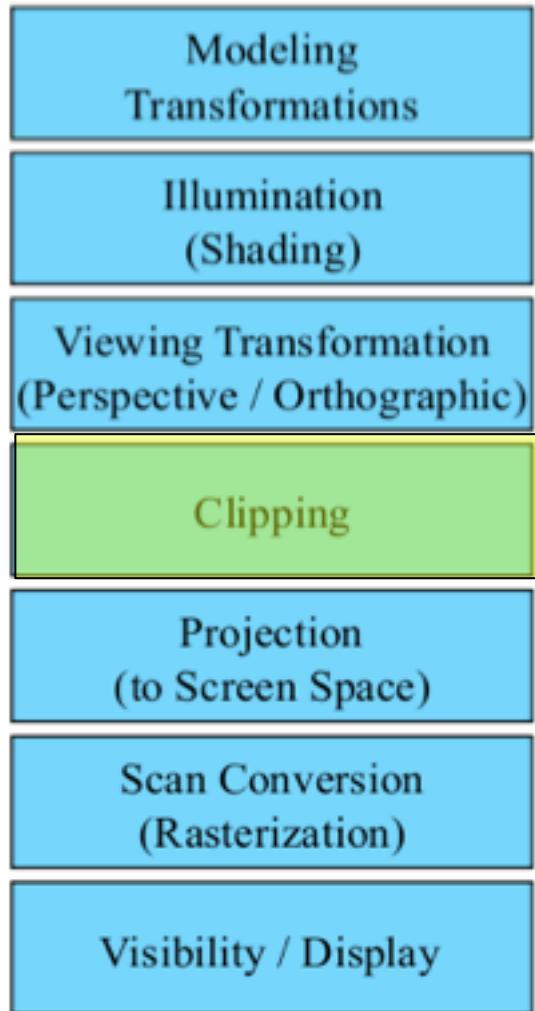
- Tots els objectes de fora del *frustum* s'eliminen per eliminar-los



3.2. Pipeline de visualització

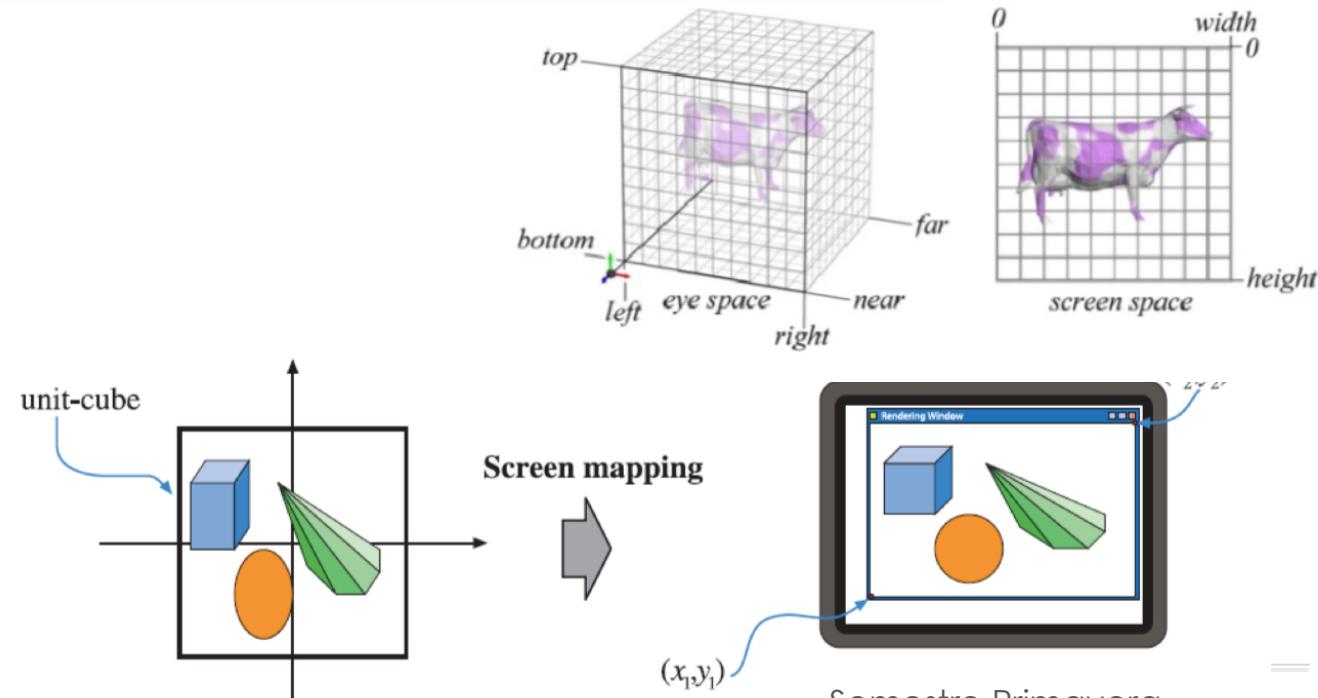
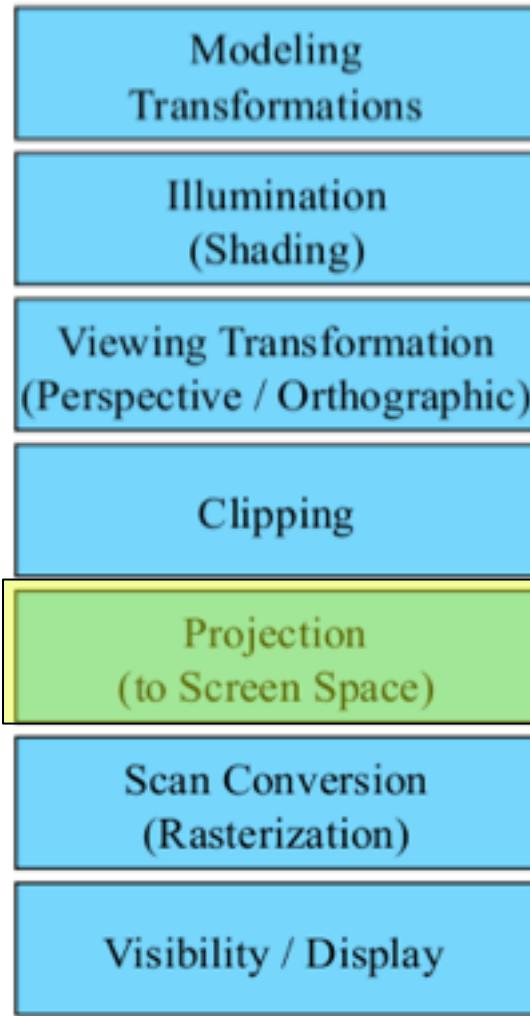
- **Clipping o retallat:**

- Tots els objectes de fora del frustum s'eliminen per eliminar-los
- Es **normalitzen** les coordenades dels vèrtexs en un cub unitari



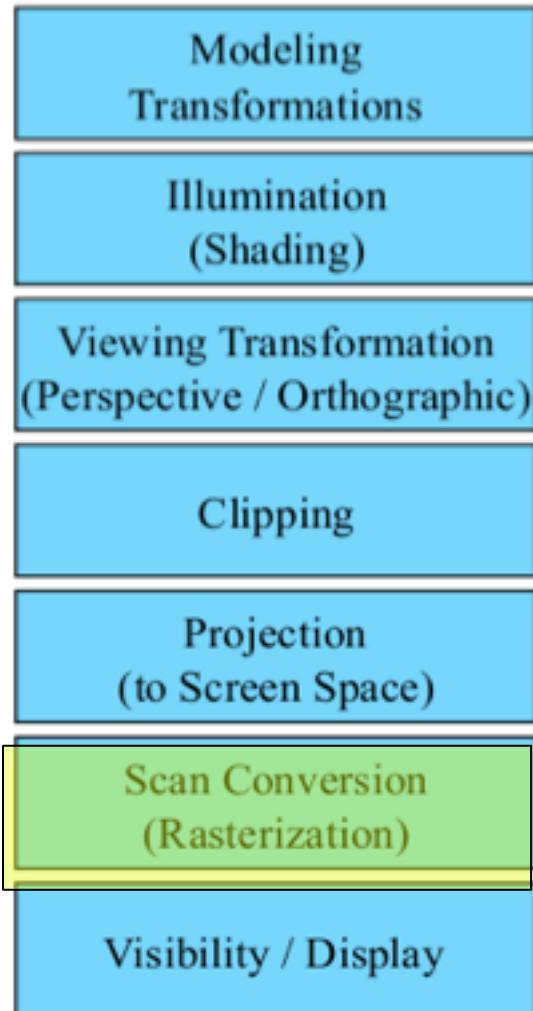
3.2. Pipeline de visualització

- **Projecció de window a viewport:**
 - Es projecta cada vèrtex de l'espai 2D continu (**window**) al frame buffer (o **viewport**)

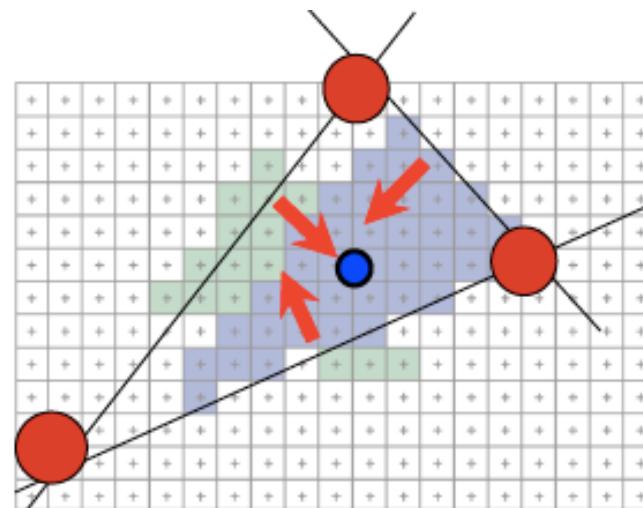


3.2. Pipeline de visualització

• Assemblatge de primitives i Rasterització

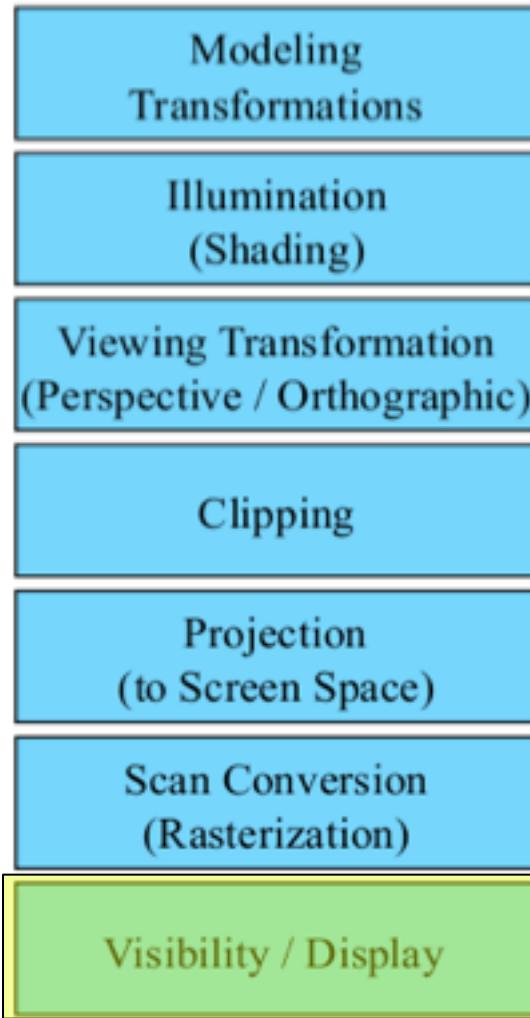


- Donats els píxels on s'han projectat els vèrtexs es fa la connexió entre ells i es calcula la discretització del triangle en píxels
- A cada píxel es fa el test contra les tres arestes del triangle, si tots donen cert, el píxel dibuixa i s'interpolia la seva z a partir de les z's dels píxels del vèrtexs
- S'anomena **fragment** a cada píxel cobert per la discretització



3.2. Pipeline de visualització

- **Test de visibilitat i càlcul del color final:**

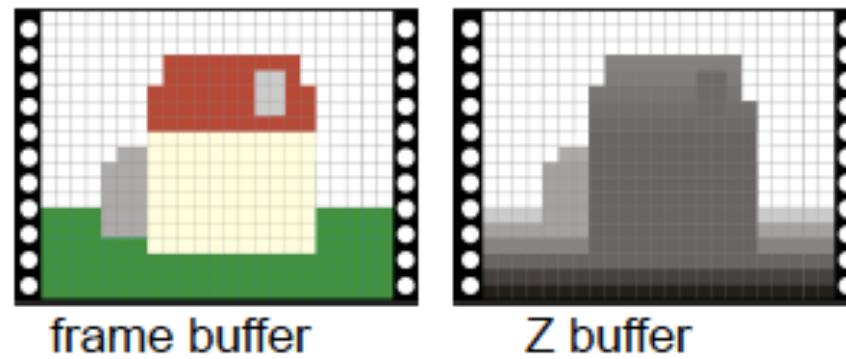


- Es té un Z-Buffer on es vol guardar la mínima distància a la càmera en un Z-Buffer
- Es té el frameBuffer on es guardaran els colors de cada píxel

si $Z_{\text{new}} < \text{Zbuffer}[x,y]$

$\text{Zbuffer}[x,y] = Z_{\text{new}}$

$\text{frameBuffer}[x, y] = \text{newColor}$



Índex

3.1. Introducció a ZBuffer

3.2. Pipeline de visualització

3.3. Pipeline de visualització a GL

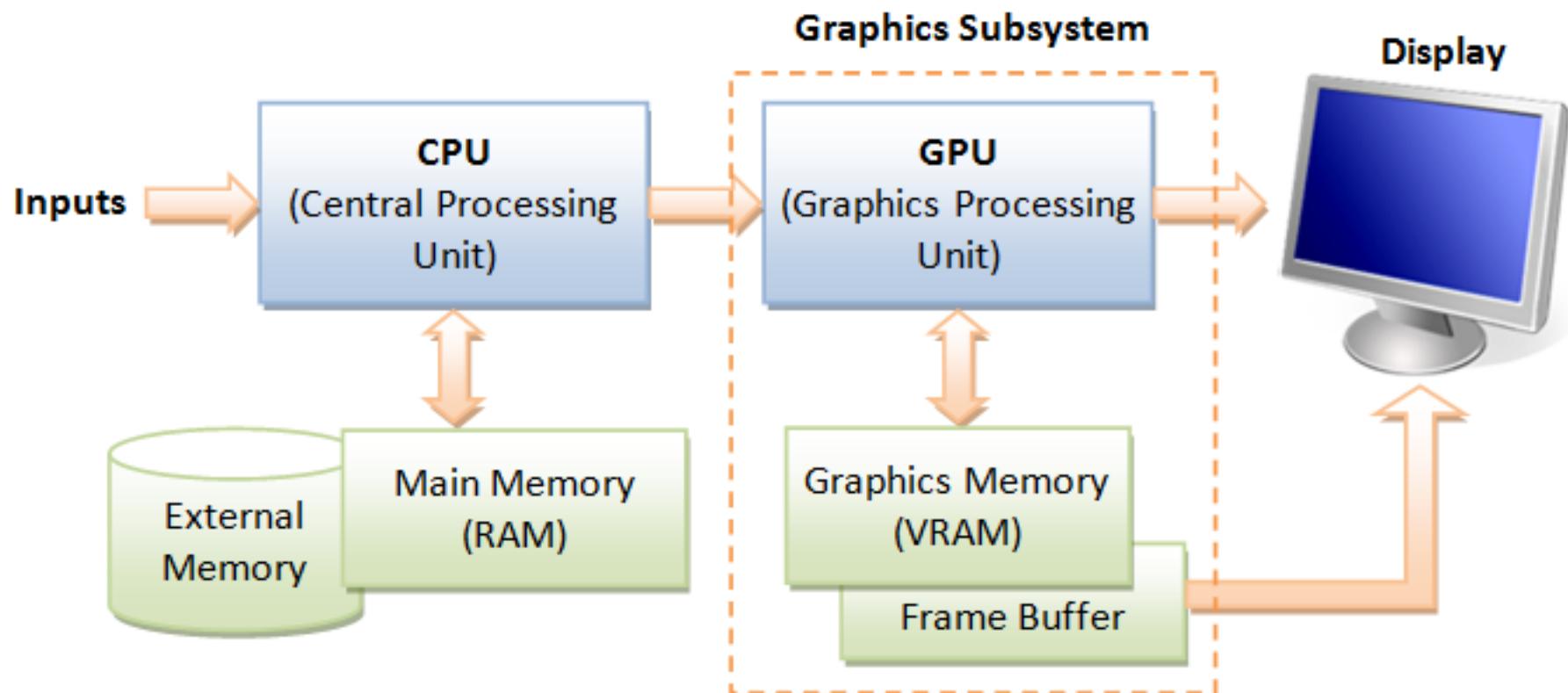
3.4. Il·luminació usant shaders

3.5. Textures

3.6. Reflexions i Transparències

3.3. Pipeline integrat a OpenGL

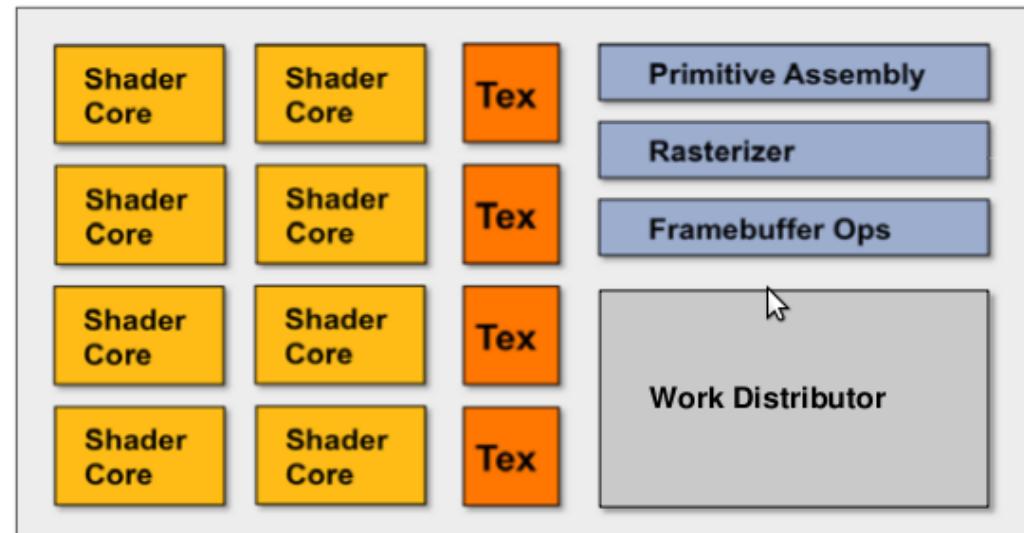
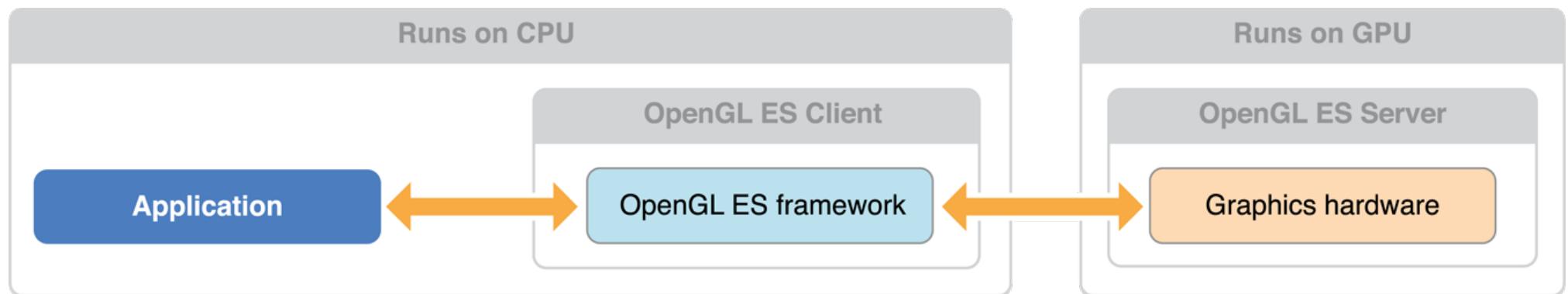
- Arquitectura OpenGL (2.0 programmable)



3.3. Pipeline integrat a OpenGL

- Arquitectura OpenGL

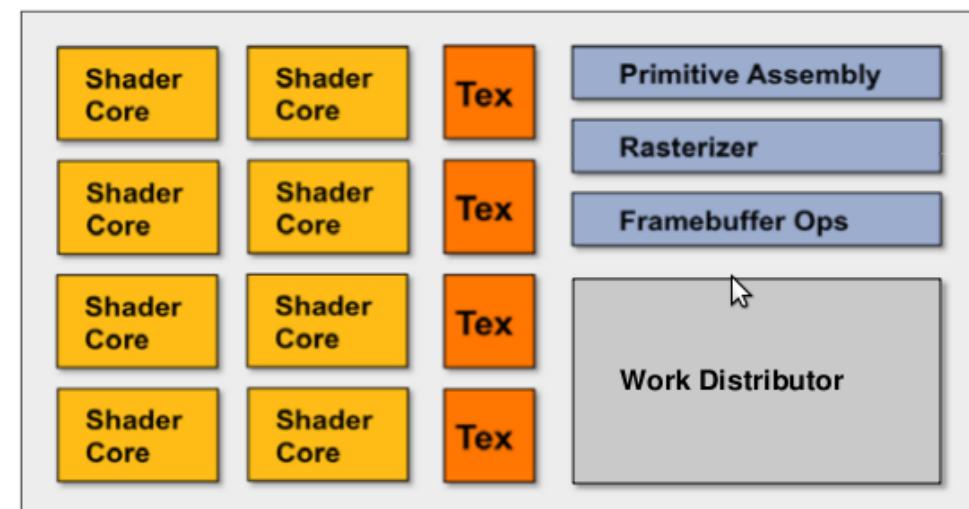
Arquitectura client-servidor



Arquitectura GPU

Com està dissenyada la GPU?

- La GPU està formada per un conjunt de processadors, que permeten paral·lelitzar a diferents nivells (són els **shader cores**)
- Es tracten en paral·lel els **vèrtexs** que s'envien des de la CPU
- Es paral·lelitzan els **fragments** que es generen internament



Per una explicació més detallada consulteu la secció 1.7 del llibre [Angel2011]

Pipeline fixe: OpenGL

Com es pinta un cub en GL?

```
typedef float Point[3];
```

```
Point verts[8] = {
```

```
    {-1.,-1.,-1.},
```

```
    { 1.,-1.,-1.},
```

```
    { 1., 1.,-1.},
```

```
    {-1., 1.,-1.},
```

```
    {-1.,-1., 1.},
```

```
    { 1.,-1., 1.},
```

```
    { 1., 1., 1.},
```

```
    {-1., 1., 1.},
```

```
};
```

```
face(int a, int b, int c, int d) {
```

```
    glBegin(GL_POLYGON);
```

```
    glVertex3fv(verts[a]);
```

```
    glVertex3fv(verts[b]);
```

```
    glVertex3fv(verts[c]);
```

```
    glVertex3fv(verts[d]);
```

```
    glEnd();
```

```
}
```

```
// Note consistent ccw orientation!
```

```
cube() {
```

```
    face(0,3,2,1);
```

```
    face(2,3,7,6);
```

```
    face(0,4,7,3);
```

```
    face(1,2,6,5);
```

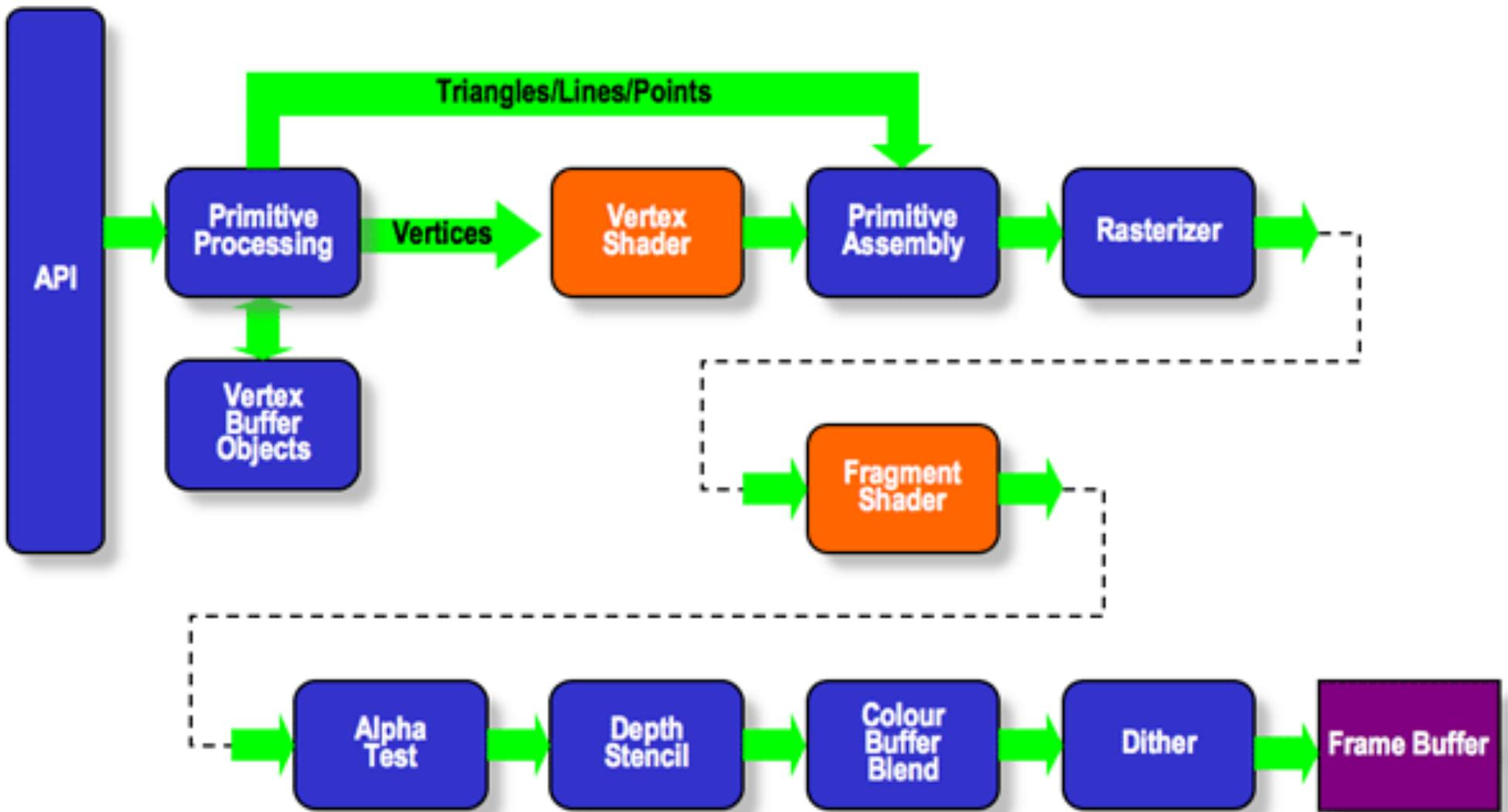
```
    face(4,5,6,7);
```

```
    face(0,1,5,4);
```

```
}
```

3.3. Pipeline integrat a OpenGL

OpenGL ES 2.0 Programmable Pipeline



Pipeline fixe: OpenGL

Exemple amb càmera i pintat:

```
glClear( GL_COLOR_BUFFER_BIT );      /* color de background */
glEnable(GL_DEPTH_TEST);            /* activació del test d'eliminació de parts amagades */
glShadeModel(GL_FLAT);             /* model d'il.luminacio */
```

```
glMatrixMode( GL_PROJECTION );      /* Matriu de projecció */
glLoadIdentity();                   /* Inicialització de la matriu de projecció */
                                    /* identity */
glFrustum( -1, 1, -1, 1, 1, 1000 ); /* Projecció perspectiva */
```

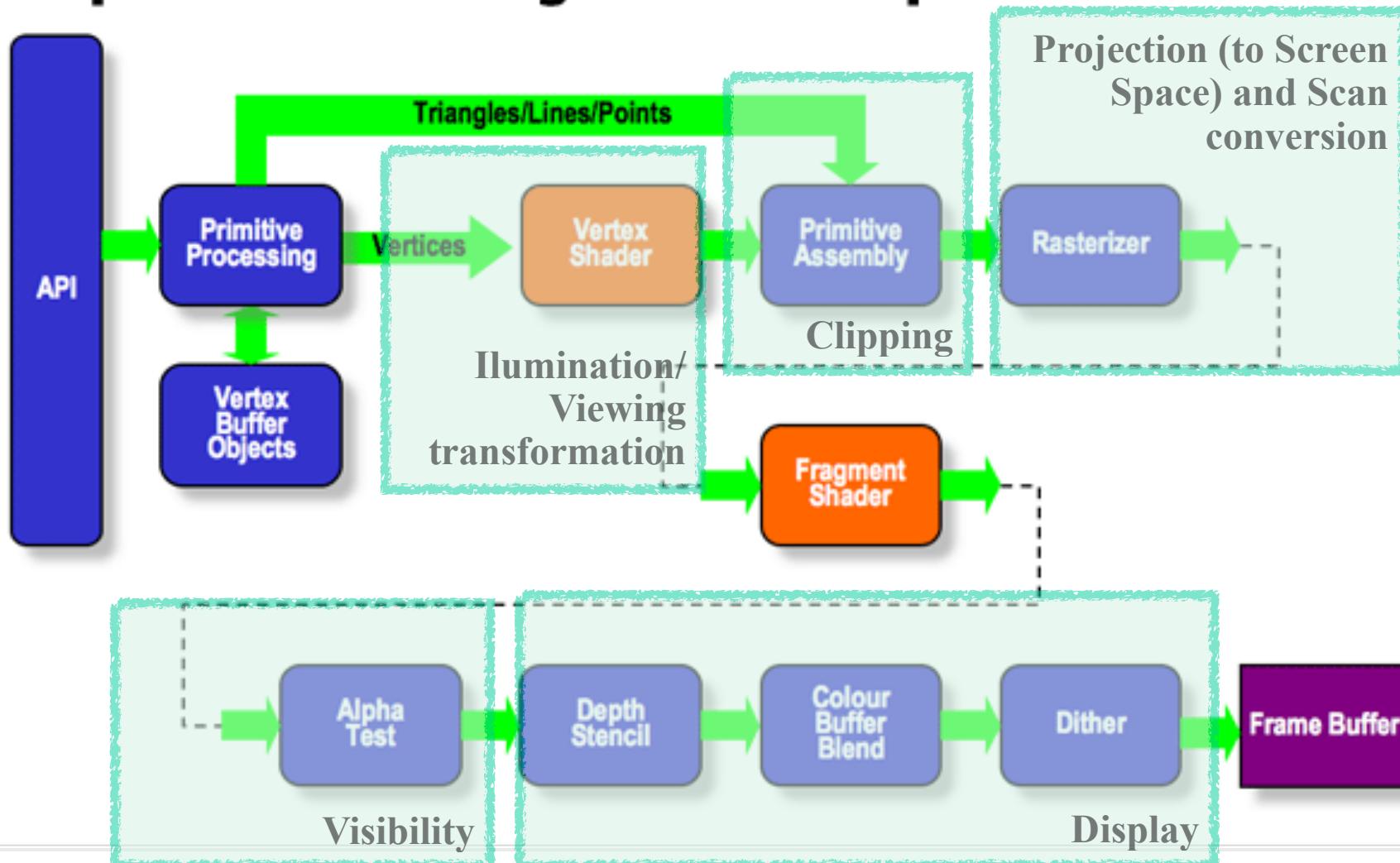
```
glMatrixMode( GL_MODELVIEW );       /* Model view */
glLoadIdentity();                   /* Inicialització de la matriu de visió */
glTranslatef( 0, 0, -3 );          /* Traslació dels vertexs, encara que també es
                                    poden rotar segons la posició de la càmera */
```

```
glBegin( GL_POLYGON );            /* Pintat de vertexs */
glColor3f( 0, 1, 0 );              /* Set the current color to green */
glVertex3f( -1, -1, 0 );           /* Issue a vertex */
glVertex3f( -1, 1, 0 );             /* Issue a vertex */
glVertex3f( 1, 1, 0 );              /* Issue a vertex */
glVertex3f( 1, -1, 0 );             /* Issue a vertex */
glEnd();                           /* Enviament de tot a la targeta gràfica */
```

3.3. Pipeline integrat a OpenGL

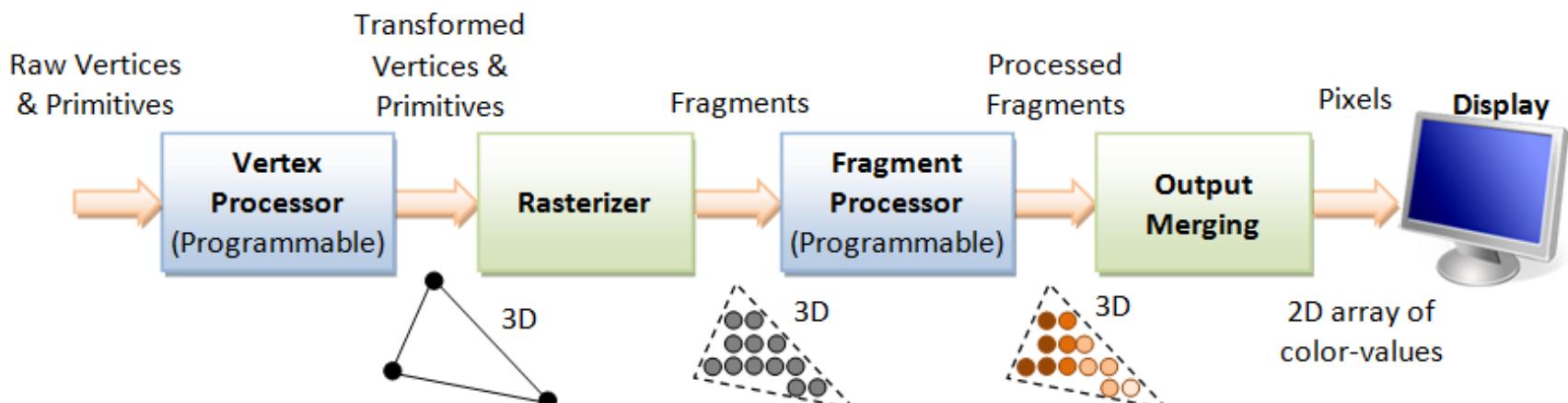
- Com es mapeja el pipeline amb OpenGL?

OpenGL ES 2.0 Programmable Pipeline

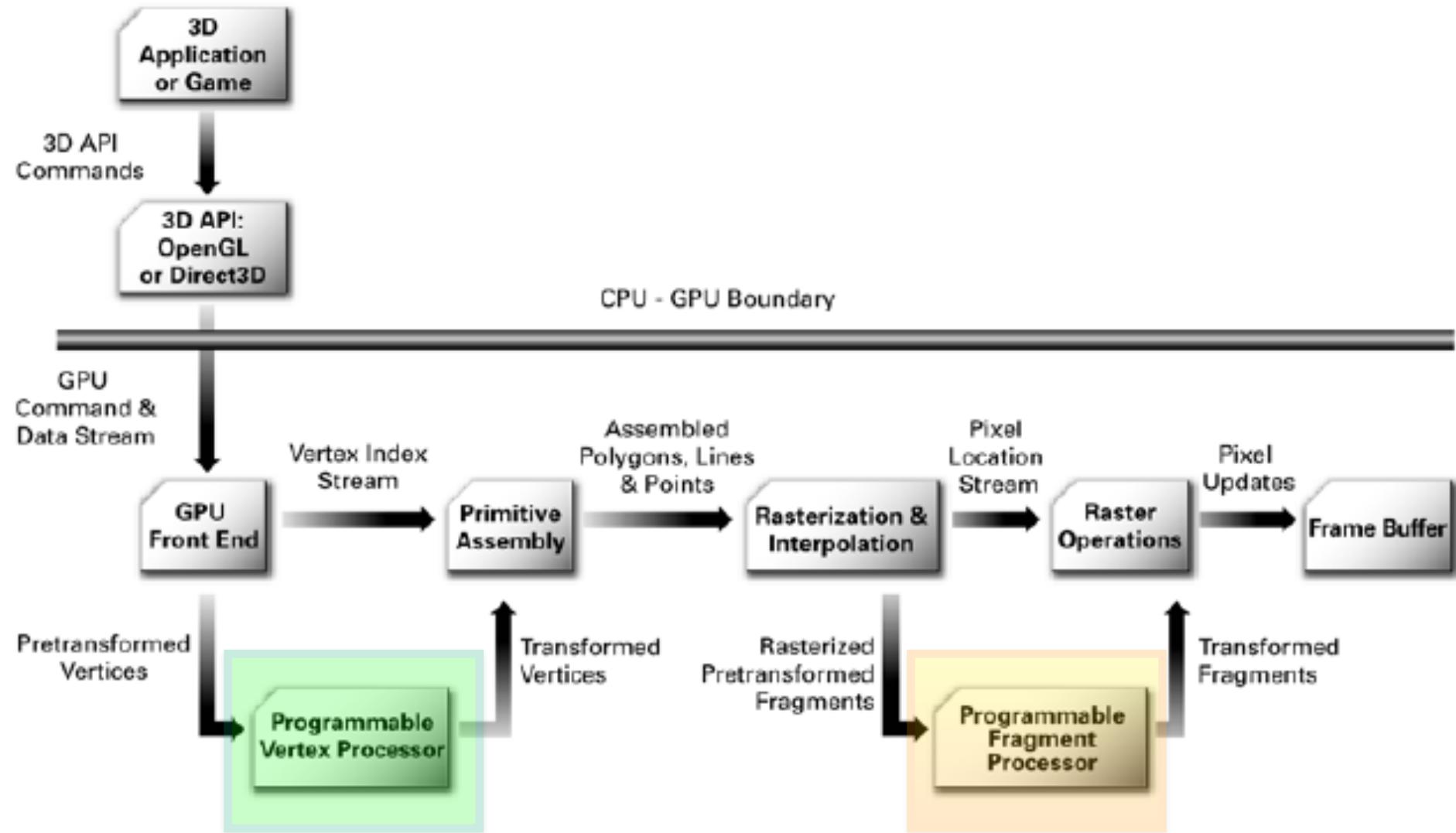


Pipeline programable a OpenGL

- Les etapes programables són:
 - Vertex processor
 - Fragment processor
- Es programen amb un llenguatge anomenat **GLSL**, molt semblant al C
- Els programes s'executen en els *Shader cores* de la GPU
- Aquests programes s'anomenen *vertex shader* i *fragment shader*, respectivament

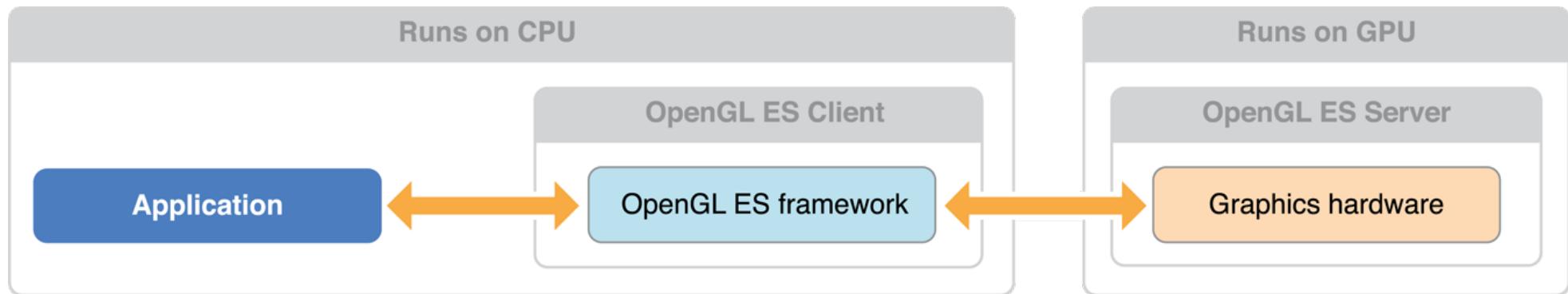


3.3. Pipeline integrat a OpenGL



3.3. Pipeline integrat a OpenGL

- Arquitectura OpenGL programant els shaders
Arquitectura client-servidor



Client:

1. S'inicialitzen els programes o **shaders** a ser executats a la GPU (*vertex shader* i *fragment shader*)
2. S'envien les dades a la GPU (vèrtexs, normals, llums, materials): uniform/ in
3. S'executa el pintat (glDraw)

Servidor:

- 1. Carrega els *shaders* a cada core de la GPU
- 2. Es guarden a memòria les dades
- 3. S'executa el *pipeline* de GL

3.3. Pipeline integrat a OpenGL

Vertex shader

```
layout (location = 0) in vec4 vPosition;
layout (location = 1) in vec4 vColor;

uniform mat4 model_view;
uniform mat4 projection;

out vec4 color;

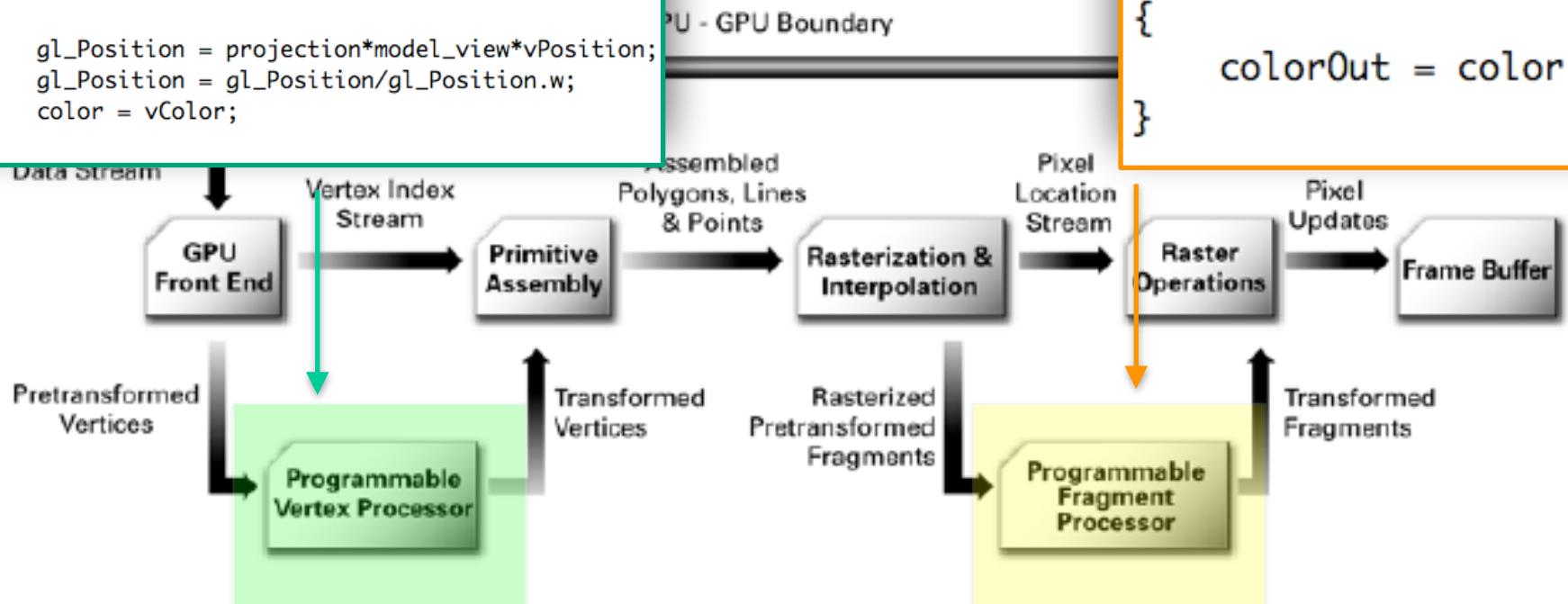
void main()
{
    gl_Position = projection*model_view*vPosition;
    gl_Position = gl_Position/gl_Position.w;
    color = vColor;
}
```

Fragment shader

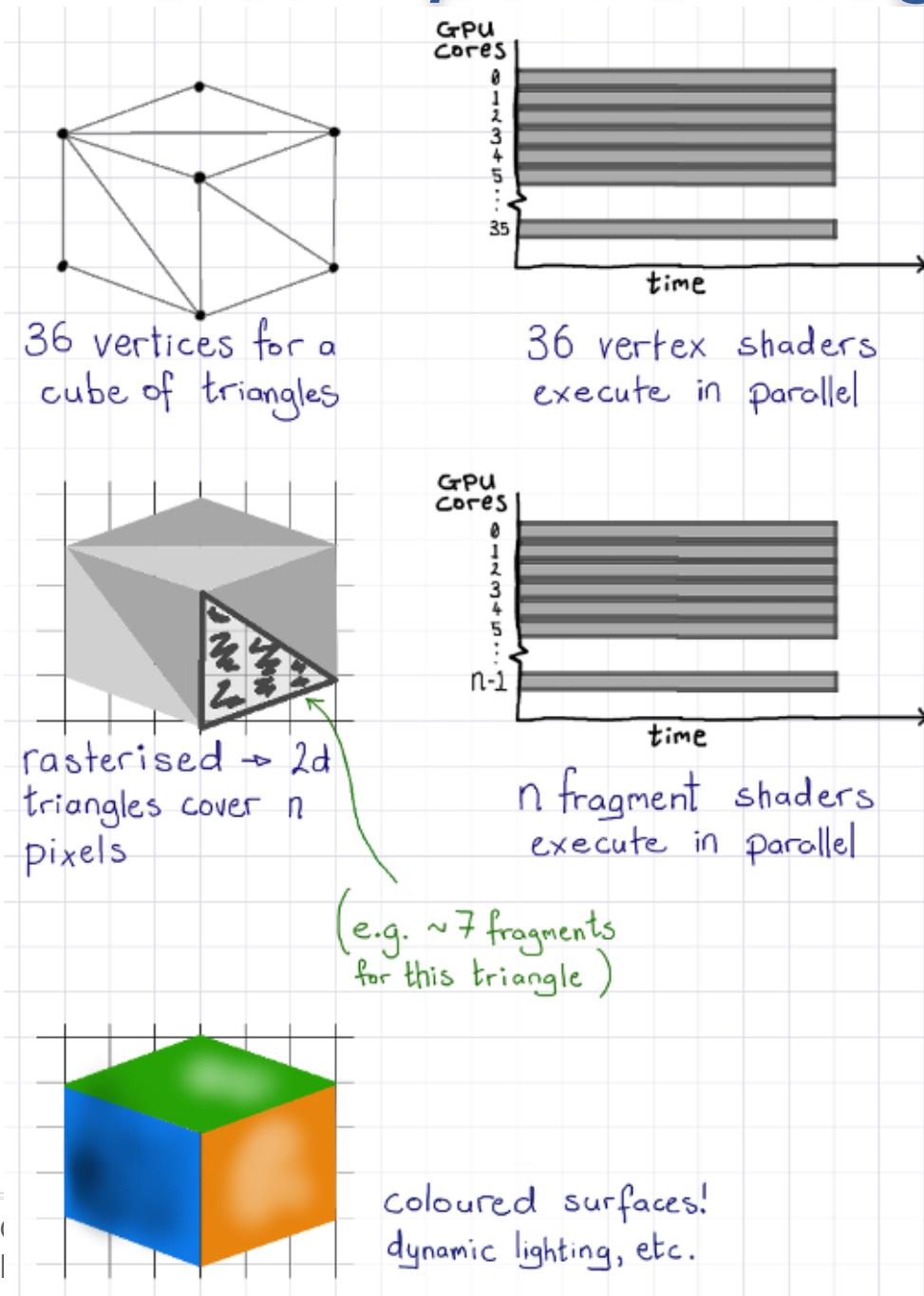
```
#version 330

in vec4 color;
out vec4 colorOut;
```

```
void main()
{
    colorOut = color;
}
```



3.3. Pipeline integrat a OpenGL



- Diferència entre **fragment** (àrea equivalent a un píxel d'una superfície) i **píxel** (àrea del frame buffer)
- GPU's: <https://www.notebookcheck.net/Comparison-of-Laptop-Graphics-Cards.130.0.html>