



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Departament d'Arquitectura de Computadors

Tarjetas Gráficas y Aceleradores

El Pipeline Gráfico

Agustín Fernández

Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

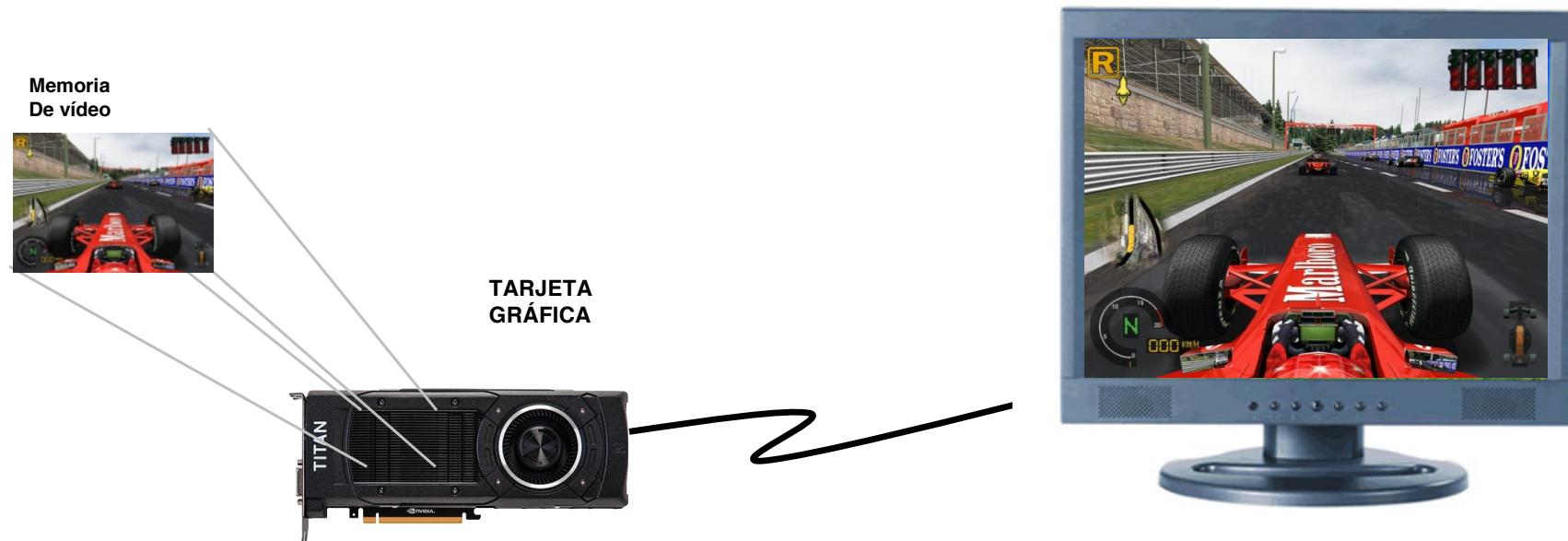
Universitat Politècnica de Catalunya



¿Qué es una Tarjeta Gráfica?

Componente que transmite al monitor la información gráfica que debe aparecer en pantalla.

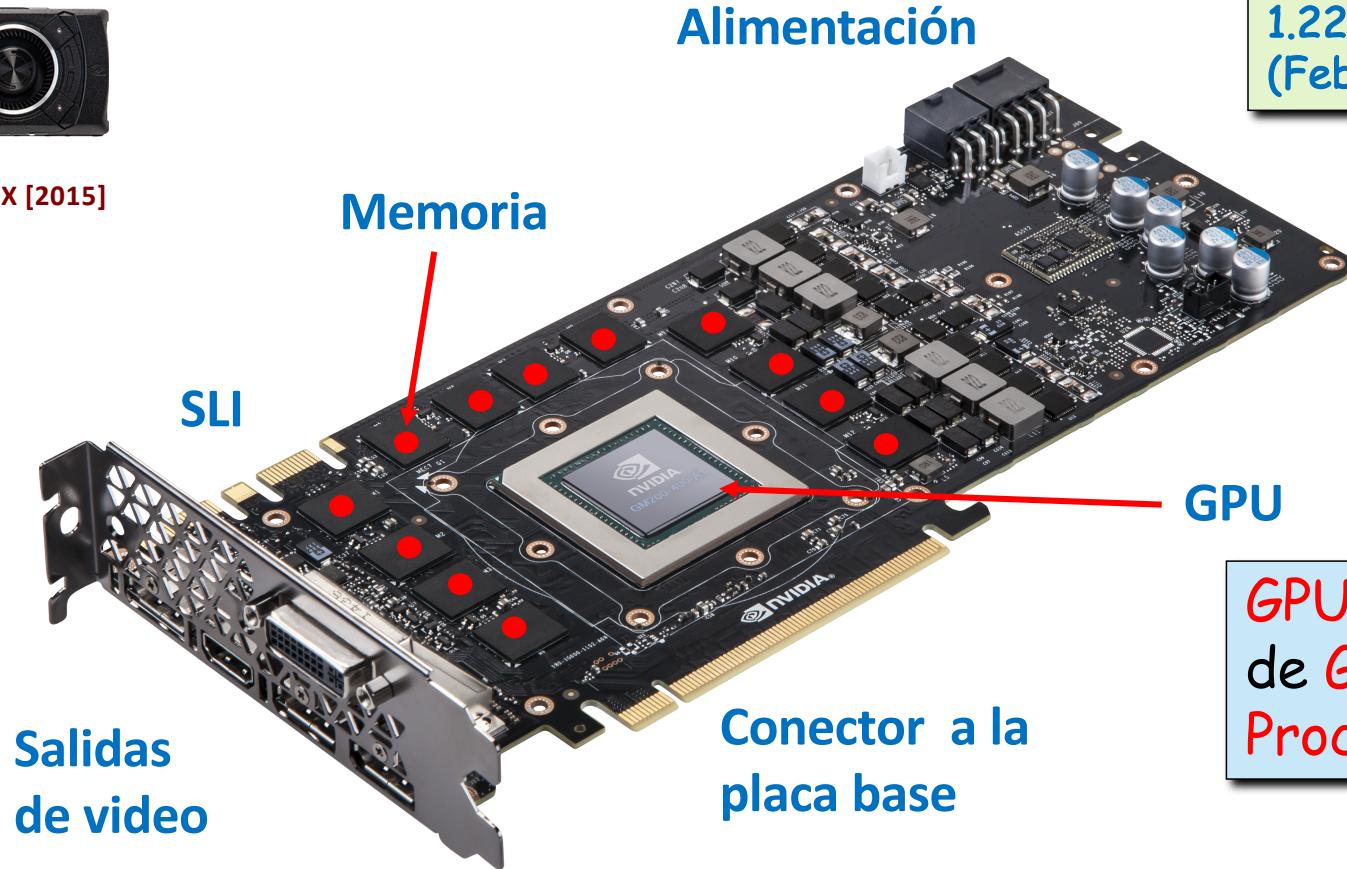
- Procesa los datos que le llegan de la CPU calculando el color de cada píxel y lo almacena en su memoria interna para poder presentarlos en la pantalla.
- Desde la memoria de vídeo, toma la salida de datos digitales resultante del proceso anterior y la transforma en una señal analógica/digital que pueda entender el monitor.



Una tarjeta gráfica



GeForce GTX Titan X [2015]



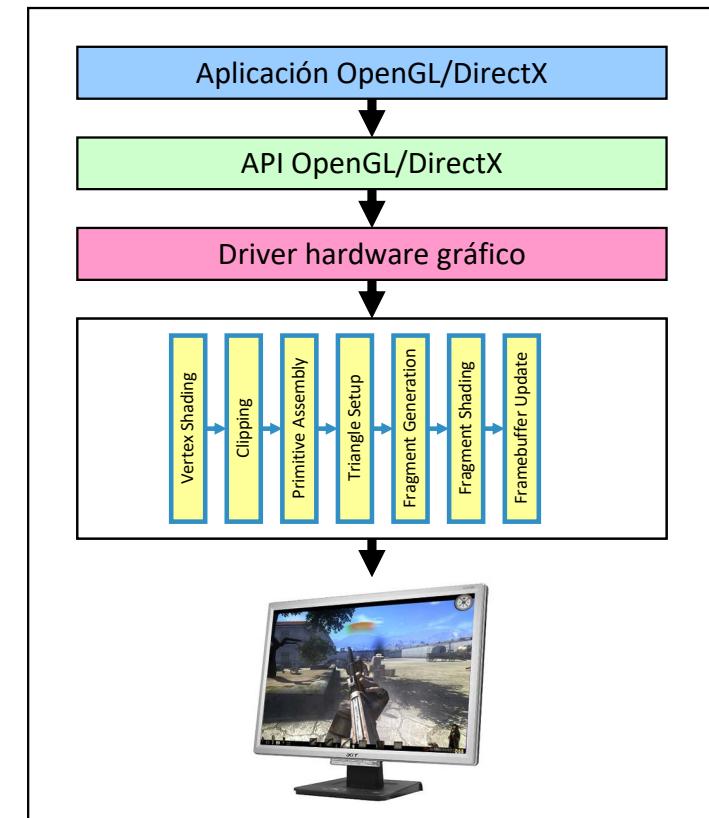
GeForce GTX Titan X, sin hardware de refrigeración.

1.221,48 € en amazon.es
(Feb/2016)

GPU son las siglas
de **Graphics**
Processing **U**nit.

¿Qué hace una GPU?

- La GPU se encarga de dibujar (renderizar) las imágenes (frames) en pantalla.
- OpenGL y DirectX son dos APIs que permiten escribir aplicaciones que produzcan gráficos 3D.
- Parten de primitivas geométricas simples: puntos, líneas, triángulos, ...
- Las APIs definen una serie de funciones y su comportamiento exacto.
- Las tarjetas gráficas son la implementación hardware de estas funciones.
- Normalmente se utilizan en el entorno de los juegos interactivos. Eso obliga a tener ciertos requerimientos de rendimiento:
 - Más de 25-30 fps (frames por segundo)
- **La existencia de OpenGL y DirectX ha permitido la evolución de las tarjetas gráficas de forma ordenada.**



Algoritmo básico de renderización

¿Qué es la renderización?

- Generar píxeles de un conjunto de imágenes o frames que componen una animación.
- Objetivo: calcular el color de cada píxel lo más rápido posible (fps)

¿Qué cálculos implica la renderización?

- Dada una BD con la escena a representar, hay que calcular el color de los objetos proyectados en la pantalla.
- Depende de la iluminación de la escena y la posición de la cámara.

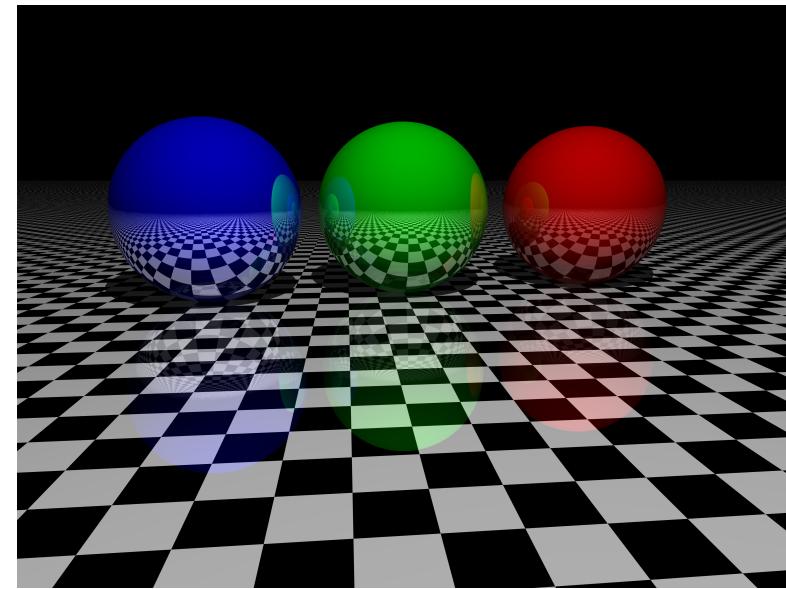
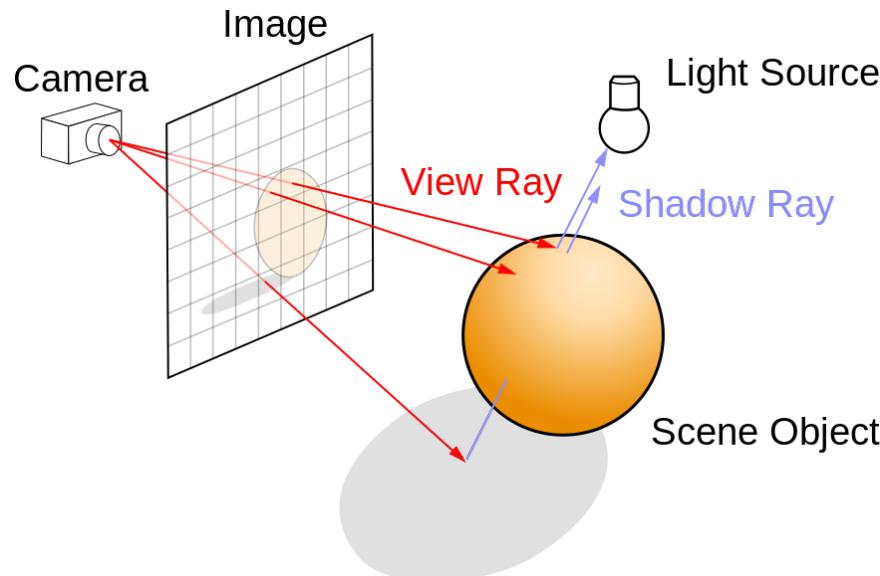


Algoritmos de renderización

Modelos complejos para generar imágenes de alta calidad

- Para cada píxel (x,y) se calcula la interacción entre los objetos y las luces:

- RayTracing, Radiosity, Photon Map
- El cálculo de cada píxel es muy costoso: Iluminación indirecta (sombras, reflexiones indirectas de los mismos objetos, ...).
- Usan algoritmos iterativos / recursivos. Poco adecuados para juegos interactivos



Algoritmos de renderización

Modelo básico de renderización:

- La interacción entre un objeto y las luces se calcula sólo en los vértices de los polígonos que forman el objeto. El color de cada píxel (x,y) se obtiene interpolando el color de los vértices.
 - *Direct Rendering* (tarjetas 3D, consolas 3D, etc.).
 - Sólo iluminación directa de las luces (El color de cada vértice es independiente)
 - Sombras y reflexiones requieren cálculo específico.



DOOM 3



Bethesda Softworks's Oblivion

Necesidades de Cálculo / Ancho de Banda de una GPU

Objetivo: HD a 50 fps

- Resolución pantalla: $1920 \times 1080 \Rightarrow 2.073.600$ píxeles $\Rightarrow 8.294.400$ bytes
- Geometría: $1,3 \cdot 10^6$ triángulos $\Rightarrow 4 \cdot 10^6$ vértices
- Sobreescritura: 8 fragmentos por píxel $\Rightarrow 16 \cdot 10^6$ fragmentos
- Necesidades de memoria (RD/WR):
 - 100 B por fragmento
 - 32 B por vértice
- Necesidades de cálculo:
 - 50 ops en coma flotante por vértice
 - 100 ops en coma flotante por fragmento
- Memoria ocupada: 700 MB + texturas

Resolución máxima de una GPU actual:

- 7680x4320 60Hz

Estas necesidades pueden aumentar:

- Aumento de la resolución
- Sistemas multipantalla
- Antialiasing agresivo
- Filtrado texturas agresivo

No está contado:

- Stencil
- Z buffer
- Función fija (p.e. Rasterización)

Objetivo 50 fps \Rightarrow necesitamos 90 GB/s y 100 GFLOPS.

Necesidades de Cálculo / Ancho de Banda de una GPU

Objetivo: 4K a 50 fps

- Resolución pantalla: $3.840 \times 2.160 \Rightarrow 8.294.400$ píxeles $\Rightarrow 33.177.600$ bytes
- Geometría: $1,3 \cdot 10^6$ triángulos $\Rightarrow 4 \cdot 10^6$ vértices
- Sobreescritura: 8 fragmentos por píxel $\Rightarrow 16 \cdot 10^6$ fragmentos
- Necesidades de memoria (RD/WR):
 - 100 B por fragmento
 - 32 B por vértice
- Necesidades de cálculo:
 - 50 ops en coma flotante por vértice
 - 100 ops en coma flotante por fragmento
- Memoria ocupada: 700 MB + texturas

Resolución máxima de una GPU actual:

- 7680x4320 60Hz

Estas necesidades pueden aumentar:

- Aumento de la resolución
- Sistemas multipantalla
- Antialiasing agresivo
- Filtrado texturas agresivo

No está contado:

- Stencil
- Z buffer
- Función fija (p.e. Rasterización)

Objetivo 50 fps \Rightarrow necesitamos 340 GB/s y 350 GFLOPS.

El Pipeline Gráfico

El diseño de las modernas GPUs gira alrededor de las siguientes ideas:

- Explotar el paralelismo
- Organización coherente
- Ocultar la Latencia con Memoria
- Mezcla inteligente de elementos de función fija con elementos programables

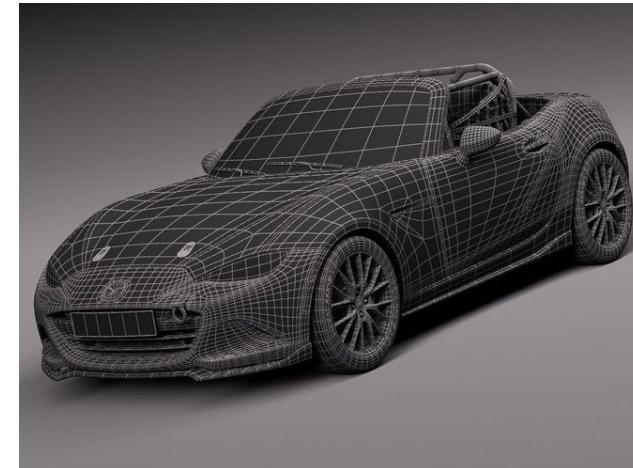
El Pipeline Gráfico

Información de partida:

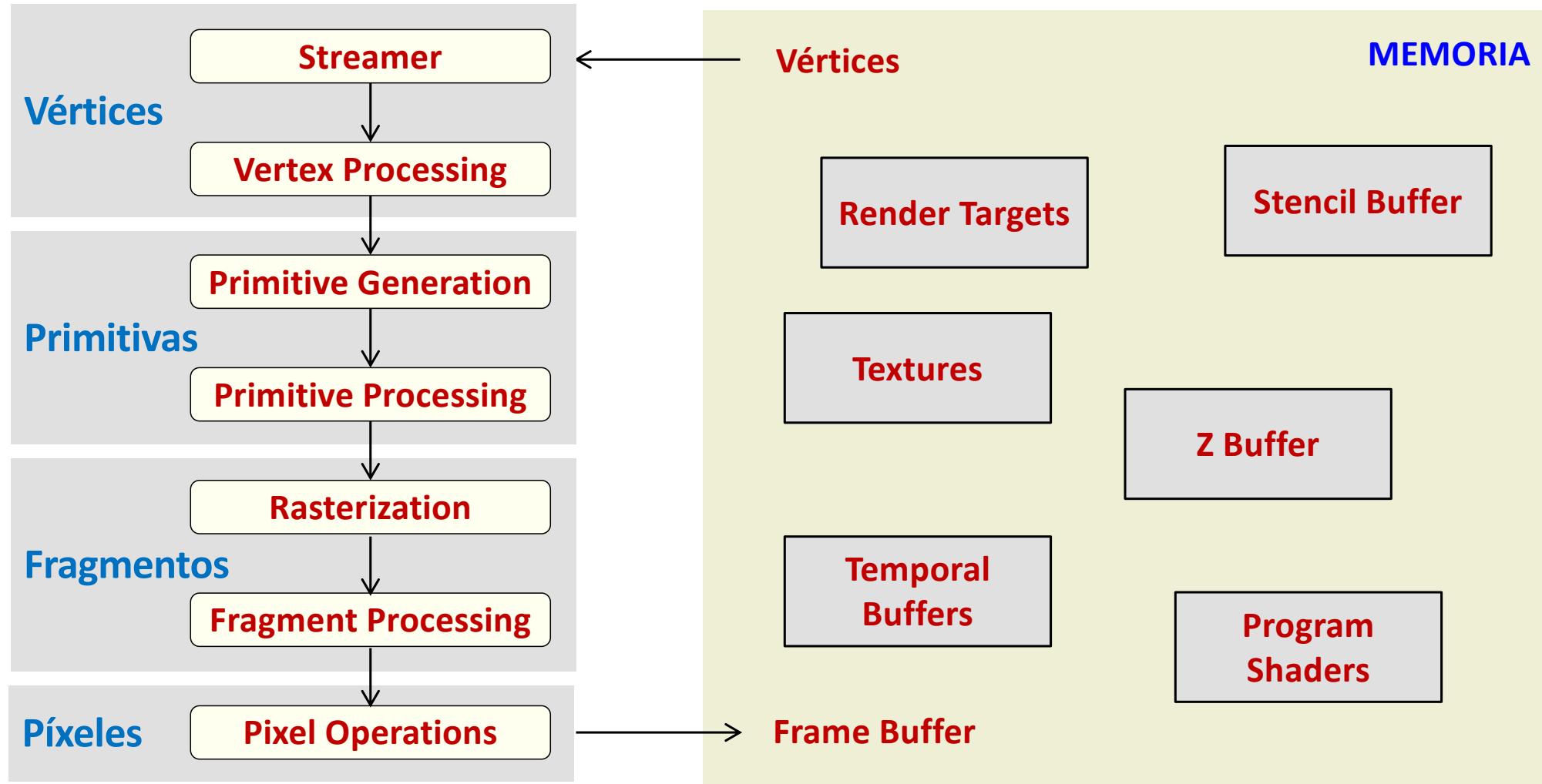
- Objetos que representan la escena:
 - Geometría definida con polígonos (en general triángulos). Para cada vértice del polígono:
 - ✓ Posición relativa
 - ✓ Color
 - ✓ Normal
 - ✓ Coordenadas de textura
 - ✓ ...
 - Texturas
 - Posición de los objetos en la escena
- Elementos de iluminación
- Punto de vista del jugador



Gran Turismo Sport for PS4



El Pipeline Gráfico



Vértices – Fragmentos – Píxeles

Píxeles

- Un punto en el frame buffer definido por su posición (x, y) y su color (R, G, B, A).
 - Coordenadas (x, y): $0 \leq x < \#columnas$, $0 \leq y < \#filas$
 - Color: $0 \leq \text{Red}, \text{Green}, \text{Blue}, \text{Alpha} \leq 255$
 - ✓ Alpha indica el nivel de transparencia: 0 (transparente), 255 (opaco)

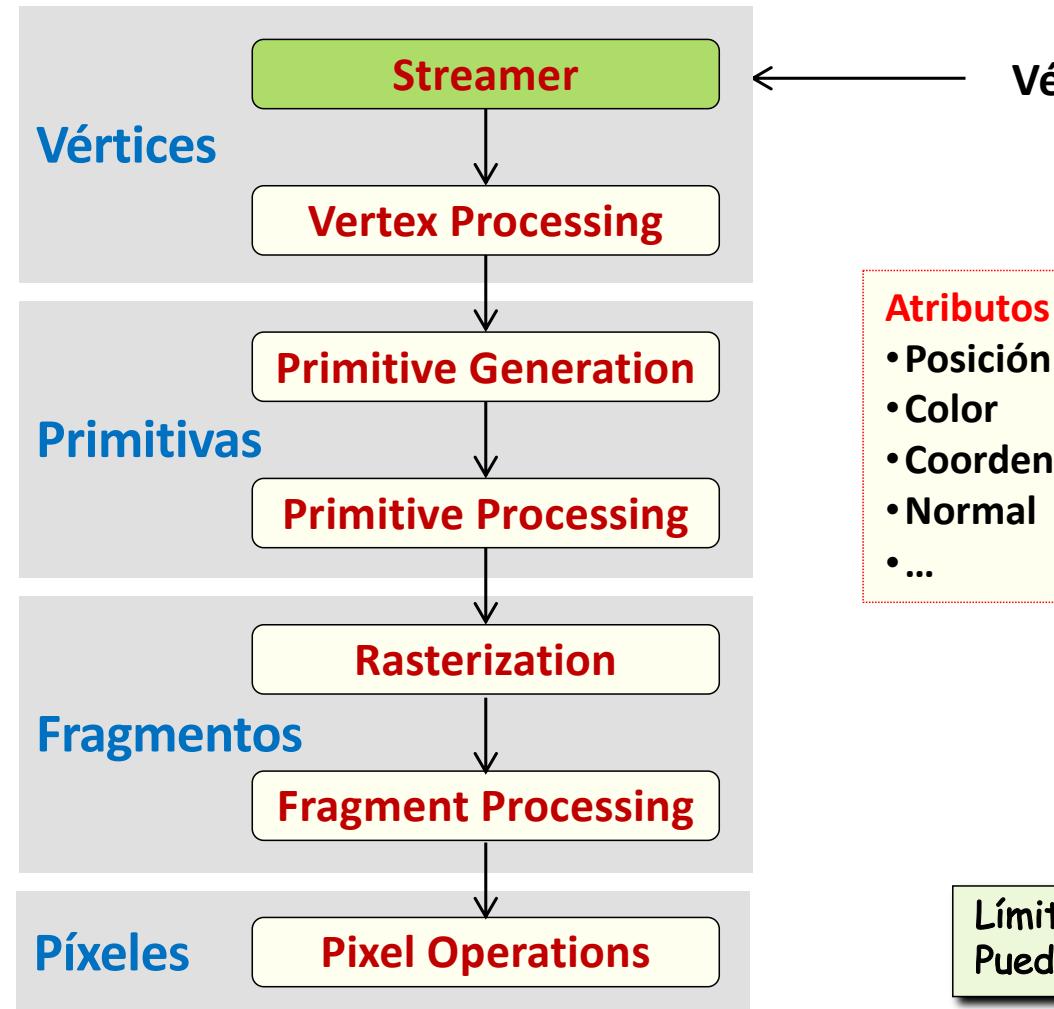
Fragmentos

- Son los puntos definidos dentro de un polígono y proyectados en el frame buffer. Contiene toda la información necesaria para generar un píxel.
 - Coordenadas (x, y, z): z es el valor de profundidad, necesario para el algoritmo de oclusión.
 - Color (R, G, B, A), Normal (x, y, z), Coordenadas de textura (s, t), ...
 - Se obtienen interpolando los atributos de los vértices

Vértices

- Son los elementos que definen los polígonos (triángulos) que forman un objeto.
 - Coordenadas (x, y, z), Color (R, G, B, A), Normal (x, y, z), Coordenadas de textura (s, t), ...

El Pipeline Gráfico: Streamer



Vértices

Vértices

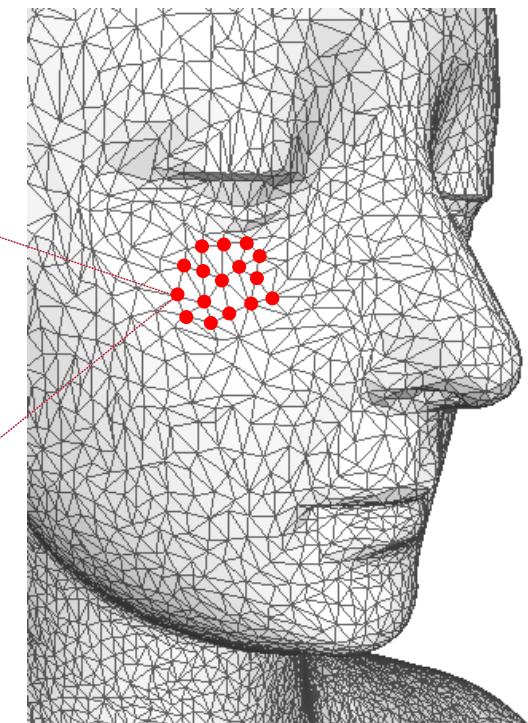
Primitivas

Fragmnetos

Píxeles

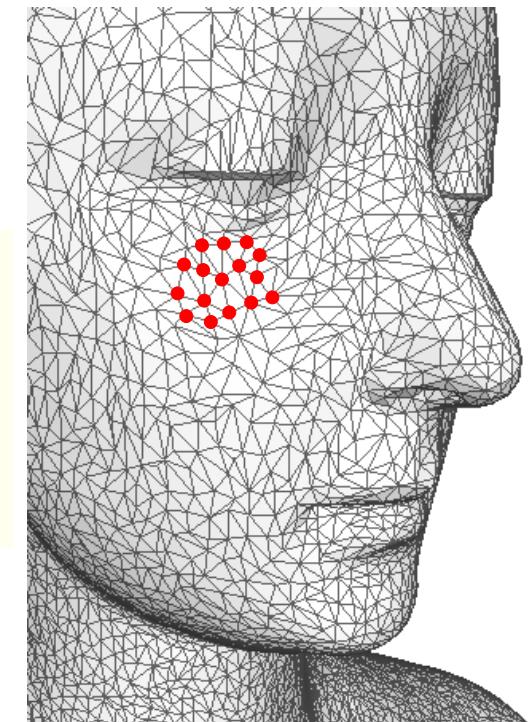
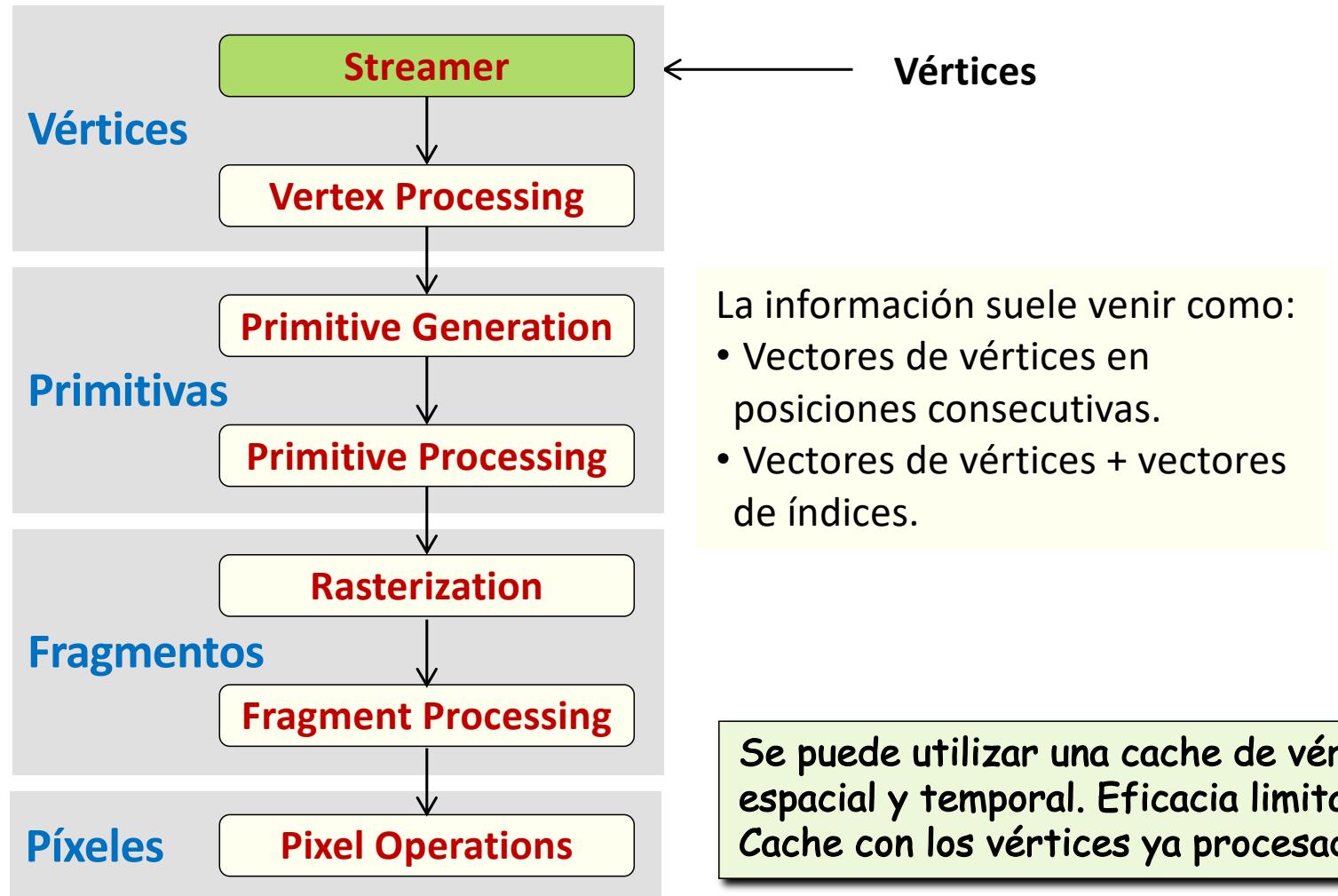
Atributos de 1 vértice:

- Posición (x, y, z)
- Color
- Coordenadas textura
- Normal
- ...

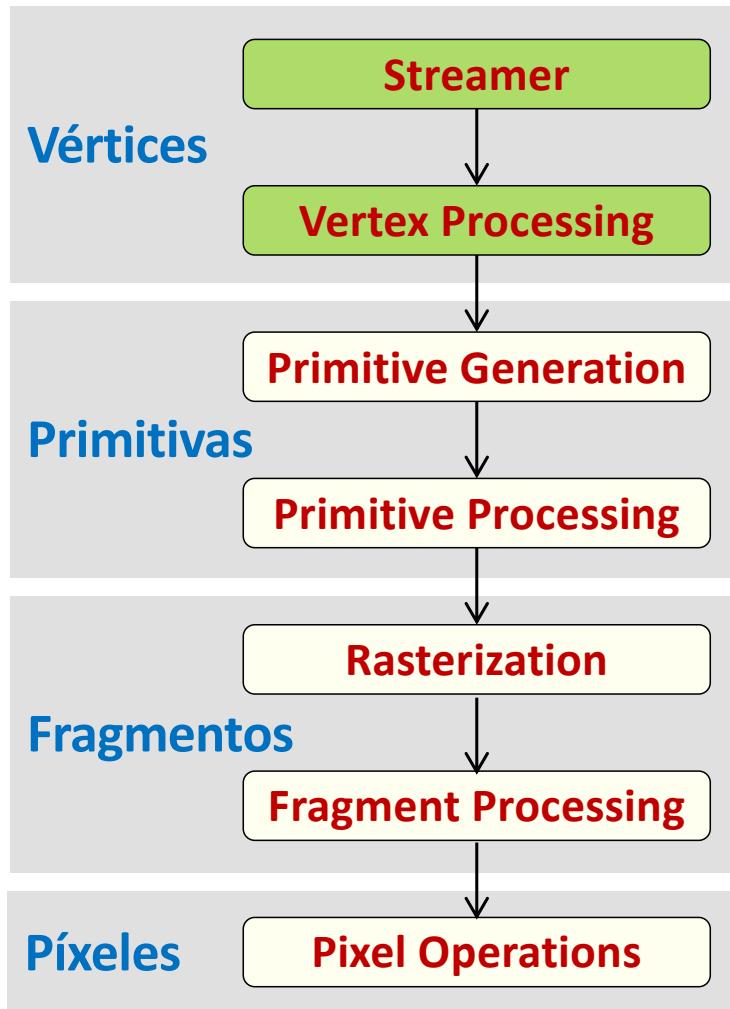


Límite en el número de atributos: **GL_MAX_VERTEX_ATTRIBS**
Puede depender del hardware disponible.

El Pipeline Gráfico: Streamer



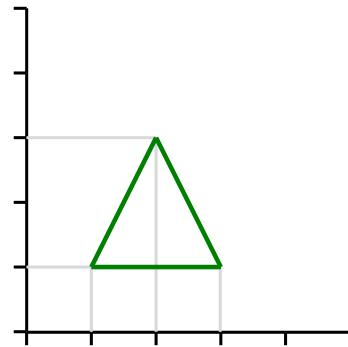
El Pipeline Gráfico: Vertex Processing



- **ELEMENTO PROGRAMABLE** en donde se proyectan los vértices sobre el plano de visión del usuario y se aplica la iluminación de la escena.
- Este elemento se solía llamar **vertex shader**.
- Se programaba directamente en ensamblador. Ahora se utilizan lenguajes específicos: HLSL (Microsoft), GLSL (OpenGL), CG (Nvidia).
- Los programas que corren en el vertex shading tienen unas pocas decenas de instrucciones.
- La mayoría de las operaciones del vertex shader se realizan sobre vectores y matrices. Cada vértice es, como mínimo, un vector de 4 componentes (x,y,z,w). [Coordenadas Homogéneas]

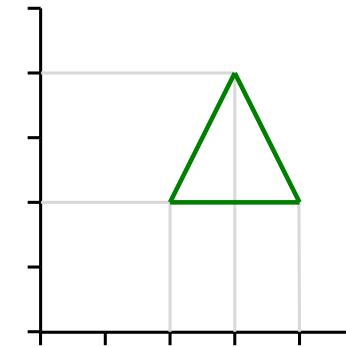
El Pipeline Gráfico: Vertex Processing

Transformaciones Básicas: TRASLACIÓN



TRASLACIÓN →

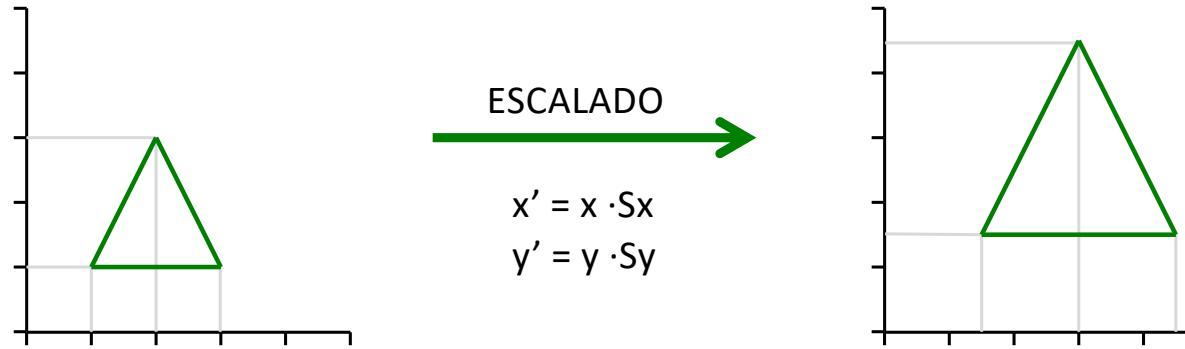
$$x' = x + Tx$$
$$y' = y + Ty$$



$$(x' \ y' \ 1) = \begin{bmatrix} 1 & 0 & Tx \\ 0 & 1 & Ty \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

El Pipeline Gráfico: Vertex Processing

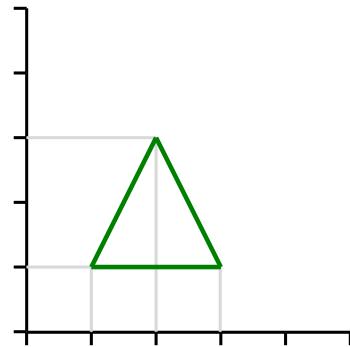
Transformaciones Básicas: ESCALADO



$$(x' \ y' \ 1) = \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

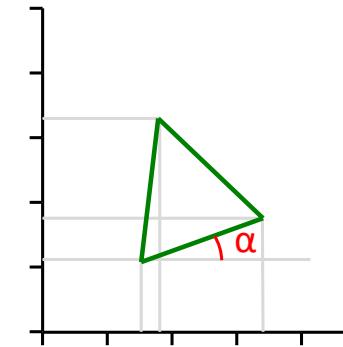
El Pipeline Gráfico: Vertex Processing

Transformaciones Básicas: ROTACIÓN



ROTACIÓN
→

$$\begin{aligned}x' &= x \cdot \cos \alpha - y \cdot \sin \alpha \\y' &= x \cdot \sin \alpha + y \cdot \cos \alpha\end{aligned}$$



$$(x' \ y' \ 1) = \begin{bmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

El Pipeline Gráfico: Vertex Processing

COMBINACIONES de Transformaciones Básicas

$$(x' \ y' \ 1) = \begin{bmatrix} 1 & 0 & Tx \\ 0 & 1 & Ty \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos \alpha & -\sin \alpha & 0 \\ \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$(x' \ y' \ 1) = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ u_{21} & u_{22} & u_{23} \\ u_{31} & u_{32} & u_{33} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Atención, el orden es importante: $A \cdot B \neq B \cdot A$

Similar en 3D

El Pipeline Gráfico: Vertex Processing

Vertex Program en Ensamblador

```
00 !ARBvp1.0
01
02 #Input
03 ATTRIB InPos = vertex.position;
04 ATTRIB InColor = vertex.color;
05
06 #Output
07 OUTPUT OutPos = result.position;
08 OUTPUT OutColor = result.color;
09
10 PARAM MVP[4] = { state.matrix.mvp };
11
12 #Transform vertex to clip space
13 DP4 OutPos.x, MVP[0], InPos;
14 DP4 OutPos.y, MVP[1], InPos;
15 DP4 OutPos.z, MVP[2], InPos;
16 DP4 OutPos.w, MVP[3], InPos;
17
18 #Output color
19 MOV OutColor, InColor;
20
21 END
```

Vertex Program en GLSL

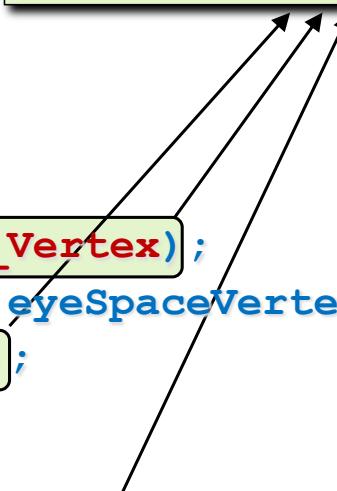
```
void main()
{
    gl_FrontColor = vec4(dot(gl_LightSource[0].position.xyz, gl_Normal));
    gl_Position = ftransform();
}
```

$$[x' \ y' \ z' \ w'] = \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ u_{21} & u_{22} & u_{23} & u_{24} \\ u_{31} & u_{32} & u_{33} & u_{34} \\ u_{41} & u_{42} & u_{43} & u_{44} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

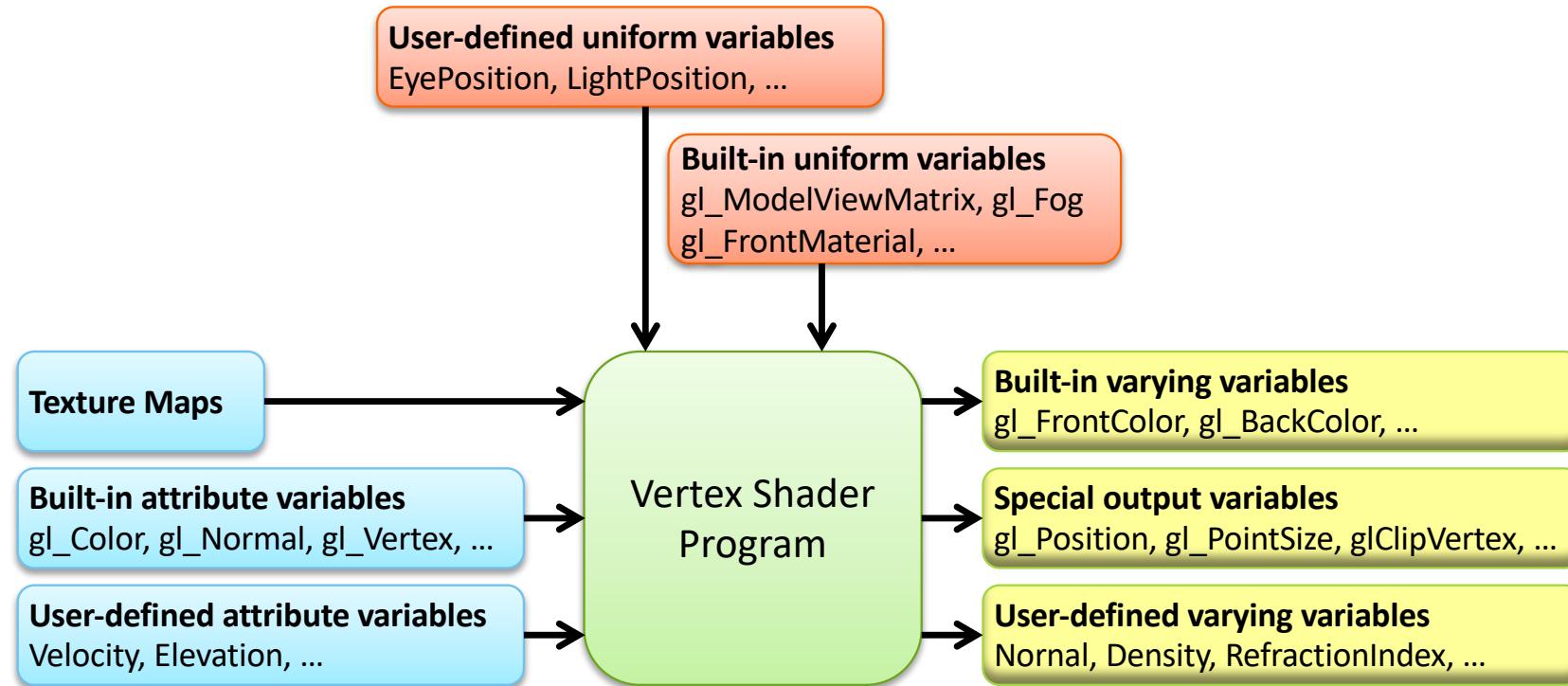
Ejemplo Vertex Program en GLSL

```
varying vec4 diffuseColor;  
varying vec3 fragNormal;  
varying vec3 lightVector;  
  
uniform vec3 eyeSpaceLightVector;  
  
void main() {  
  
    vec3 eyeSpaceVertex = vec3(gl_ModelViewMatrix * gl_Vertex);  
    lightVector = vec3(normalize(eyeSpaceLightVector - eyeSpaceVertex));  
    fragNormal = normalize(gl_NormalMatrix * gl_Normal);  
  
    diffuseColor = gl_Color;  
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;  
}
```

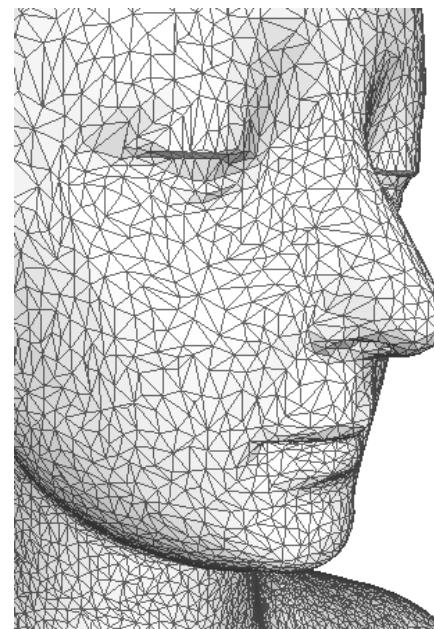
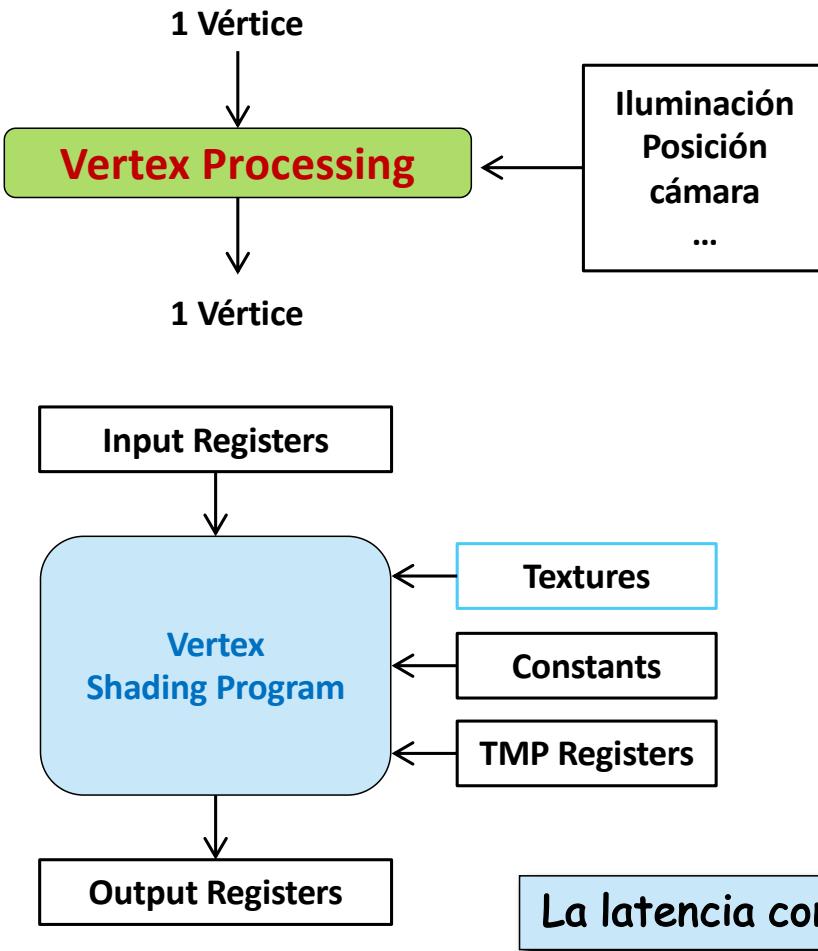
PRODUCTO de MATRICES



Vertex Shader en GLSL



El Pipeline gráfico: Vertex Processing



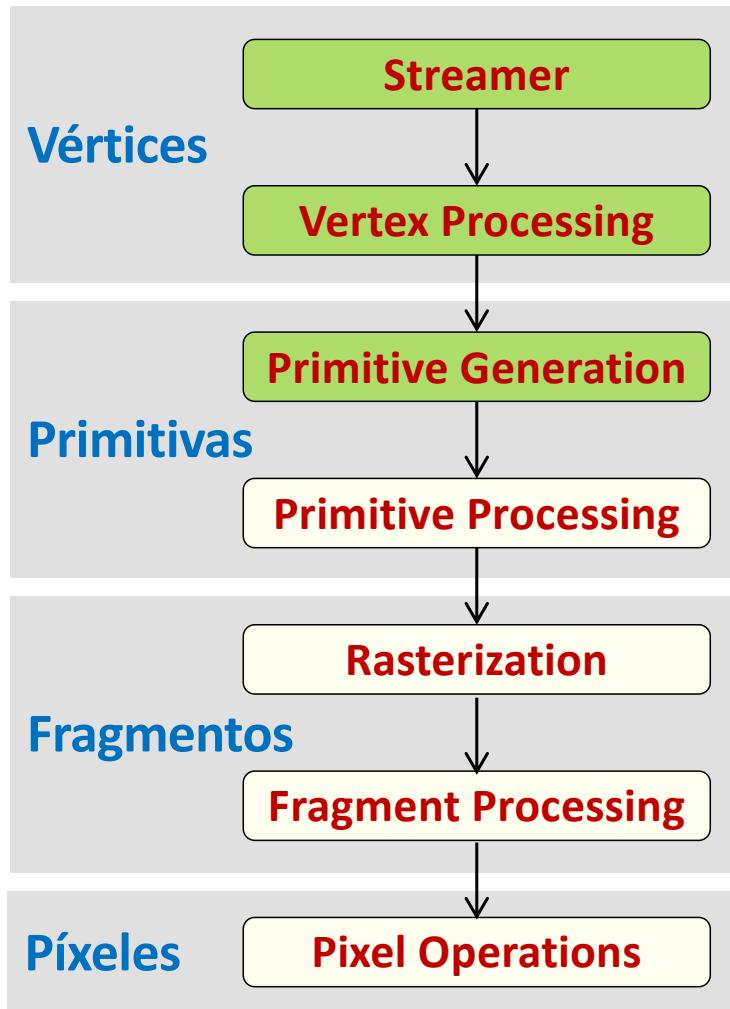
La imagen se genera batch a batch

- N vértices
- N puede ser muy grande ($N \uparrow \uparrow \uparrow$) .
- Todos los vértices son independientes entre si.
- A todos los vértices de un mismo batch se le aplica el **MISMO vertex shading program**.
- Limitado por el número de registros disponible

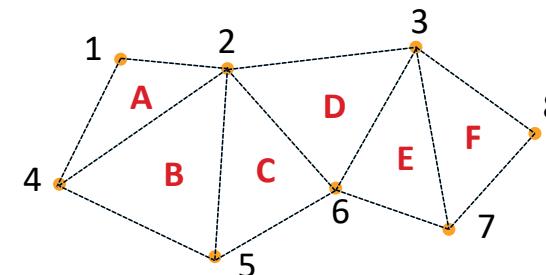
APLICACIÓN MASIVAMENTE PARALELA

La latencia con Memoria se oculta utilizando **MULTITHREADING**

El Pipeline Gráfico: Primitive Generation



Elemento no Programable: **FUNCIÓN FIJA**



3 vértices

(1, 2, 4)-(2, 4, 5)-(2, 5, 6)-(2, 3, 6)-(3, 6, 7)-(3, 7, 8)

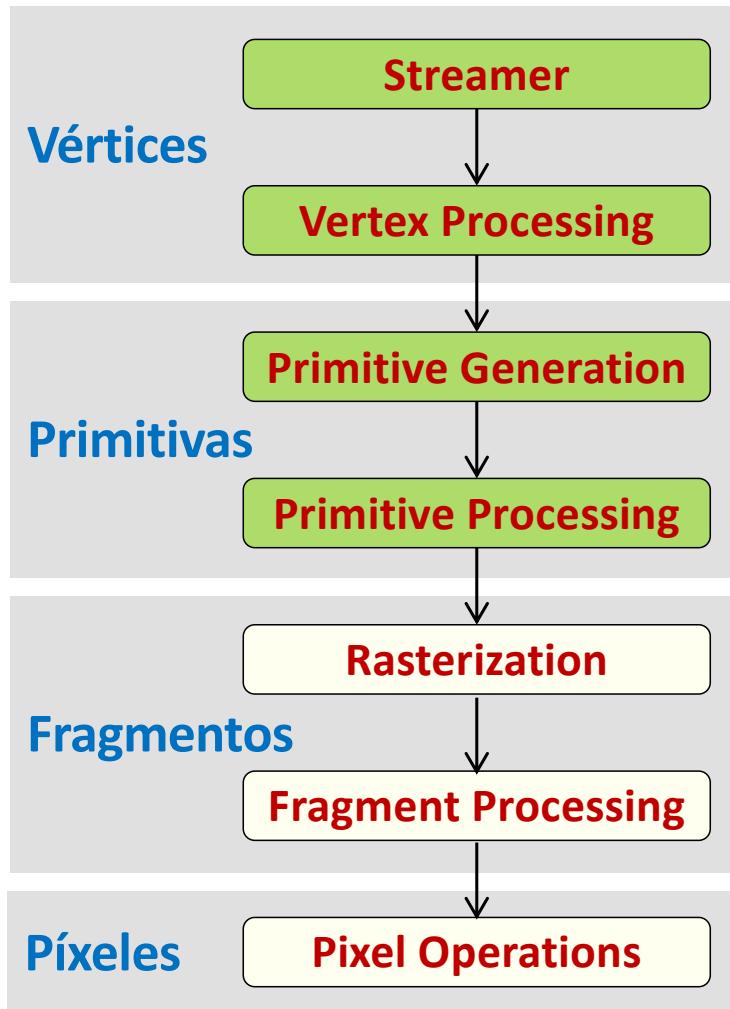
Primitive Generation

1 triángulo

(A)-(B)-(C)-(D)-(E)-(F)

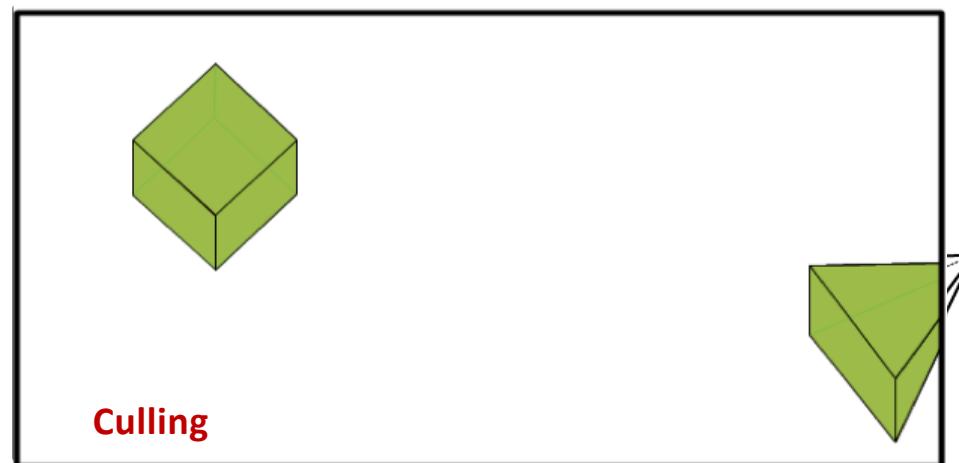
Los vértices se procesan en orden (de 3 en 3).
"Entre triángulos" no hay dependencias

El Pipeline Gráfico: Primitive Processing



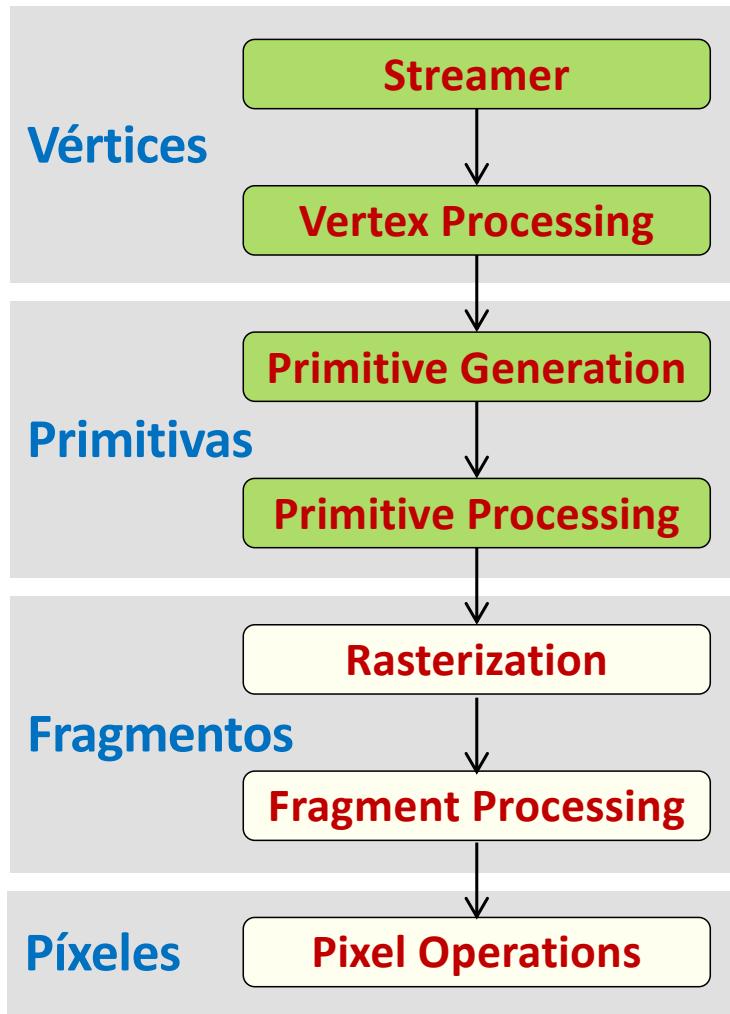
Elemento no Programable: **FUNCIÓN FIJA**

- Clipping:** **ELIMINAR** aquellas primitivas que no aparezcan en el área de visión, o que sean más pequeñas de un 1 píxel.
- Culling:** **ELIMINAR** aquellas primitivas que queden ocultas por otras primitivas **[Z-buffer]**.

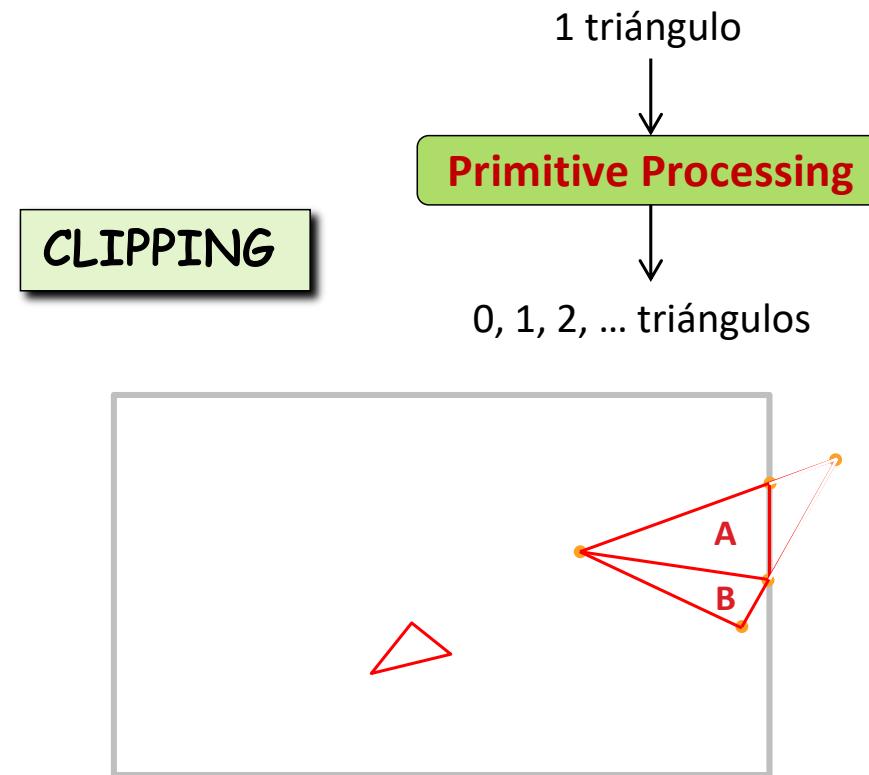


Es fundamental **ELIMINAR** lo antes posible aquellos elementos que no van a contribuir a la imagen final

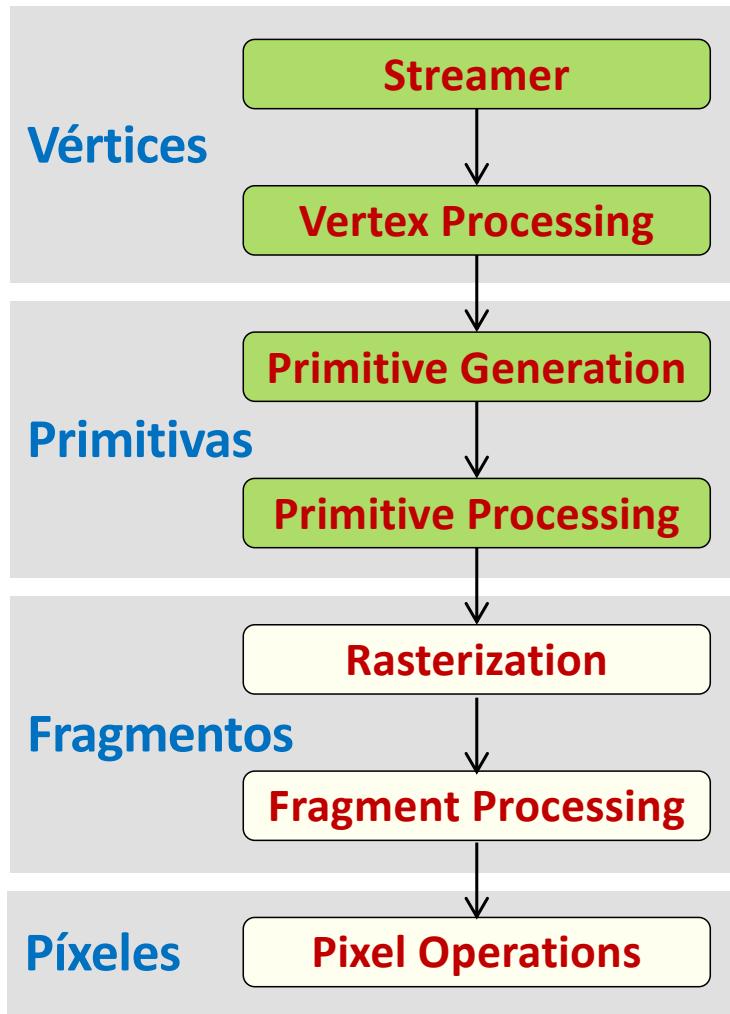
El Pipeline Gráfico: Primitive Processing



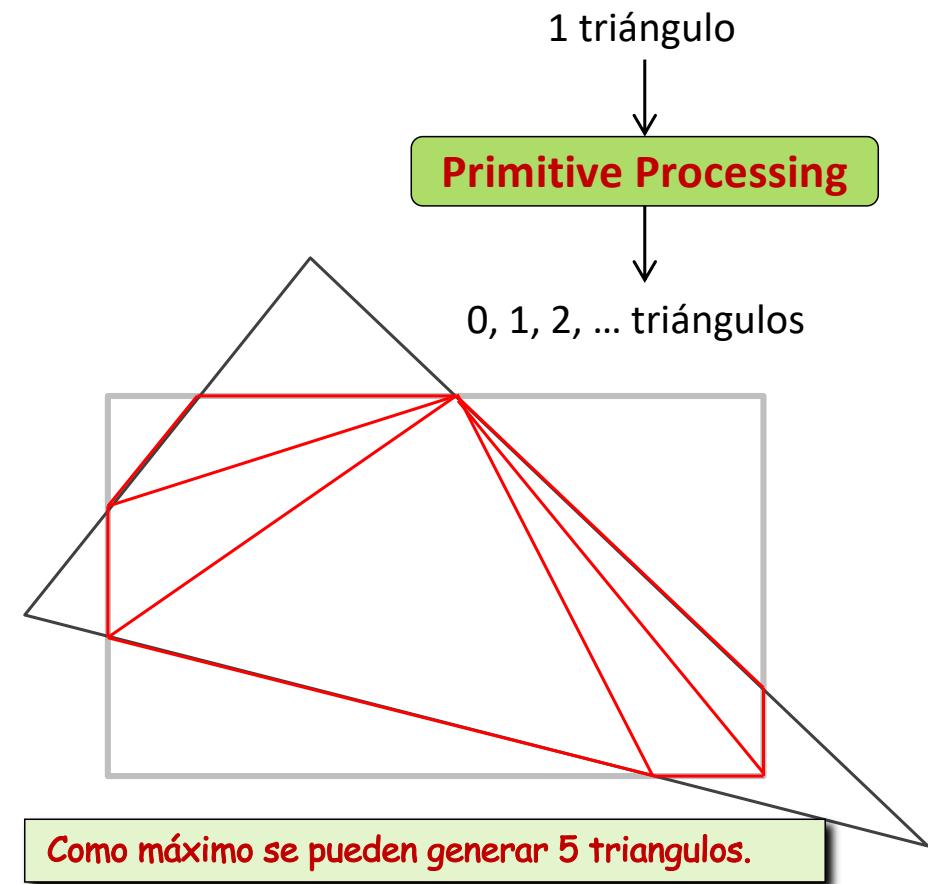
Elemento no Programable: **FUNCIÓN FIJA**



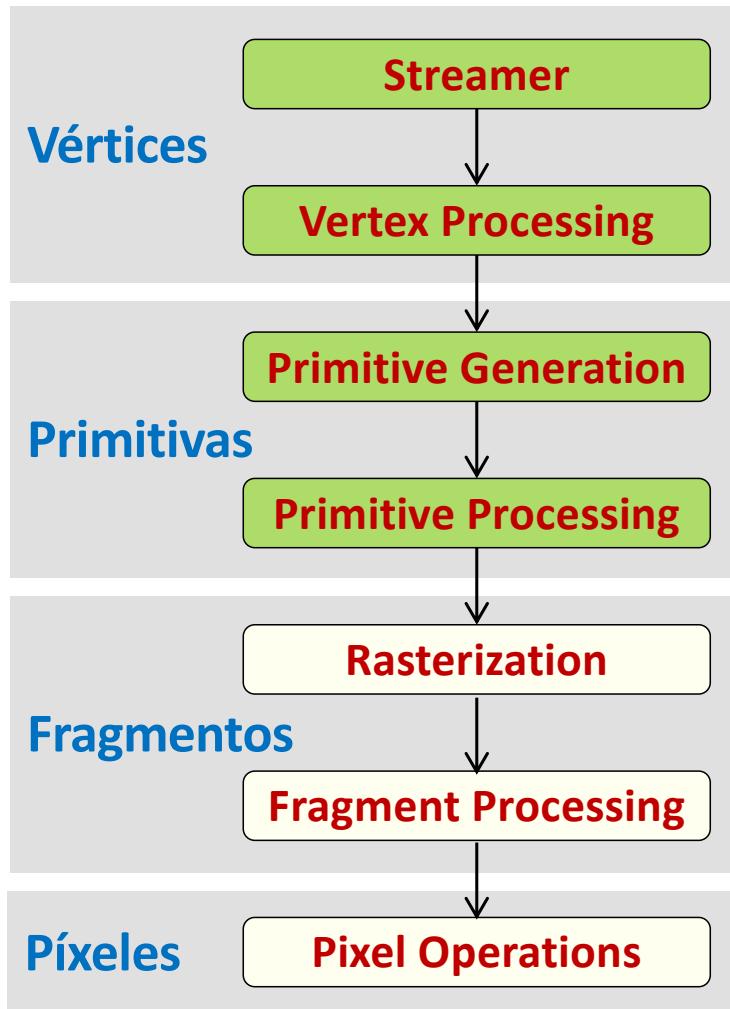
El Pipeline Gráfico: Primitive Processing



Elemento no Programable: **FUNCIÓN FIJA**



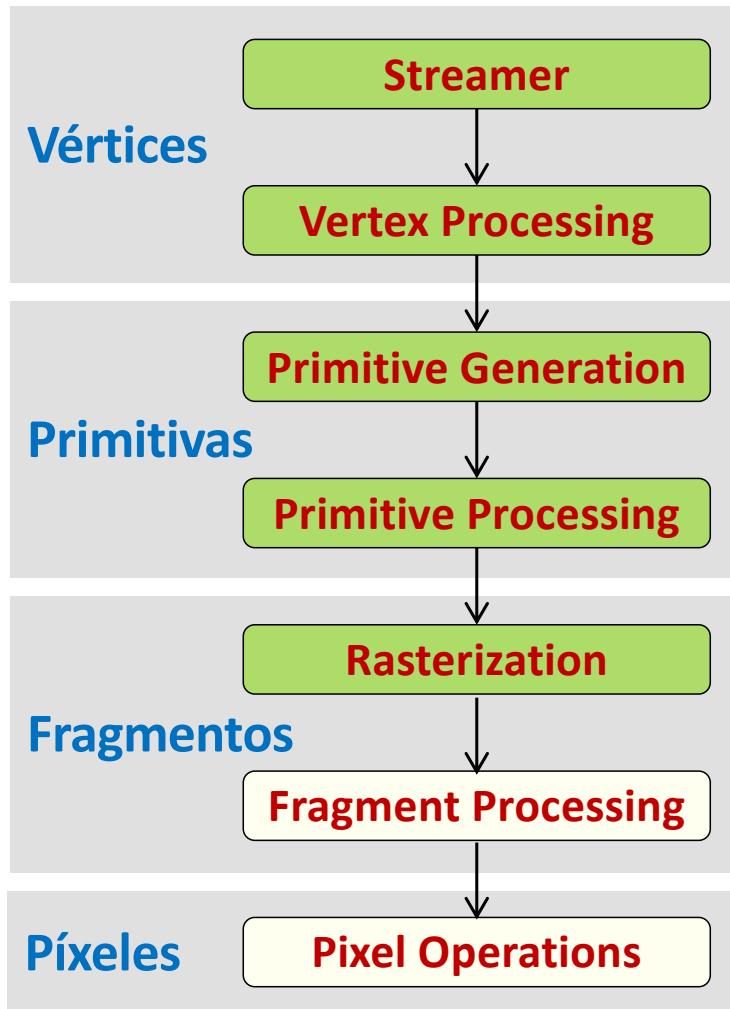
El Pipeline Gráfico: Primitive Processing



Elemento no Programable: **FUNCIÓN FIJA**

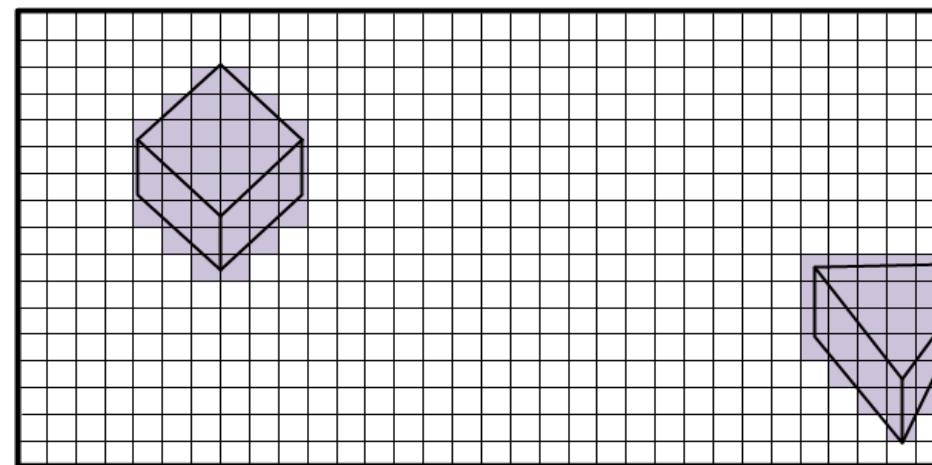
- Antes de pasar a la rasterización se realiza el **Triangle Setup**, donde se realizan todos los cálculos necesarios para preparar la siguiente fase.
- No es una operación trivial. Hay que calcular las ecuaciones de las aristas que forman el triángulo.
- El coste computacional es alto:
 - 30×3 MULT
 - 17 ADD
 - 5 RCP ($1/x$)
- Muy ineficiente con triángulos pequeños (< 4 fragmentos)

El Pipeline Gráfico: Rasterization

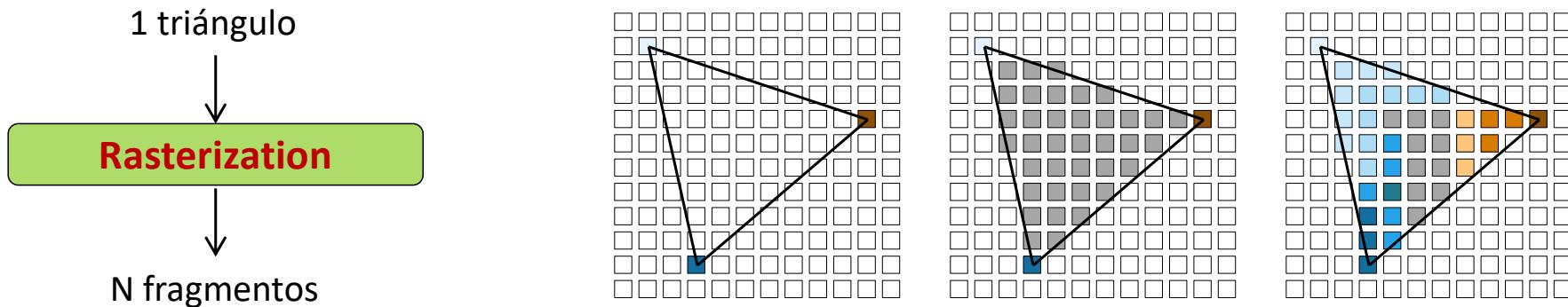


Elemento no Programable: **FUNCIÓN FIJA**

- La **Rasterización** consiste en recorrer el triángulo y generar los fragmentos que luego pueden acabar en píxeles en el frame buffer.
- Es una de las etapas que menos ha cambiado en los últimos años. Los algoritmos utilizados son inefficientes con triángulos pequeños.



El Pipeline Gráfico: Rasterization



- En la etapa de rasterización se generan los fragmentos que rellenan el triángulo.
- El número de fragmentos que puede tener un triángulo puede ser muy variable de 1 a 100's. Cada vez son más pequeños.
- La información de los fragmentos (posición, color, coordenadas de textura, ...) se obtiene interpolando la correspondiente información de los vértices.
- El número de triángulos que se pueden rasterizar por ciclo está limitado.

Esta etapa es uno de los cuellos de botella de las tarjetas gráficas actuales.

Algoritmo Básico de Renderización

- Iluminación más realista a mayor densidad de vértices.
- Necesitamos más vértices para mejorar el detalle de las superficies.
- Consecuencia:

○ \uparrow realismo $\Rightarrow \uparrow$ vértices $\Rightarrow \uparrow$ cálculo $\Rightarrow \downarrow$ fps

- Solución:
 - Definir las superficies con \downarrow vértices y
 - Especificar los detalles superficiales con texturas

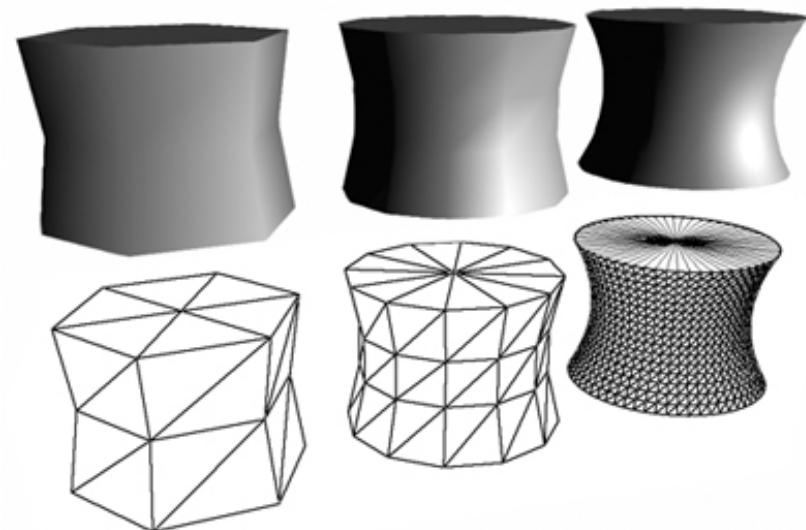


Imagen tomada de «The Cg Tutorial – The definitive Guide to Programmable Real-Time Graphics» disponible online en developer.nvidia.com

Aumentando la calidad de la imagen: Texturas

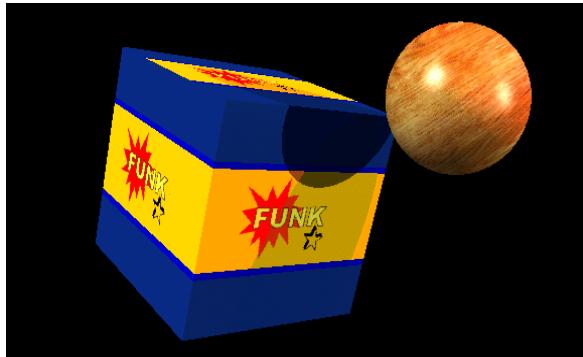


Bethesda Softworks's Oblivion

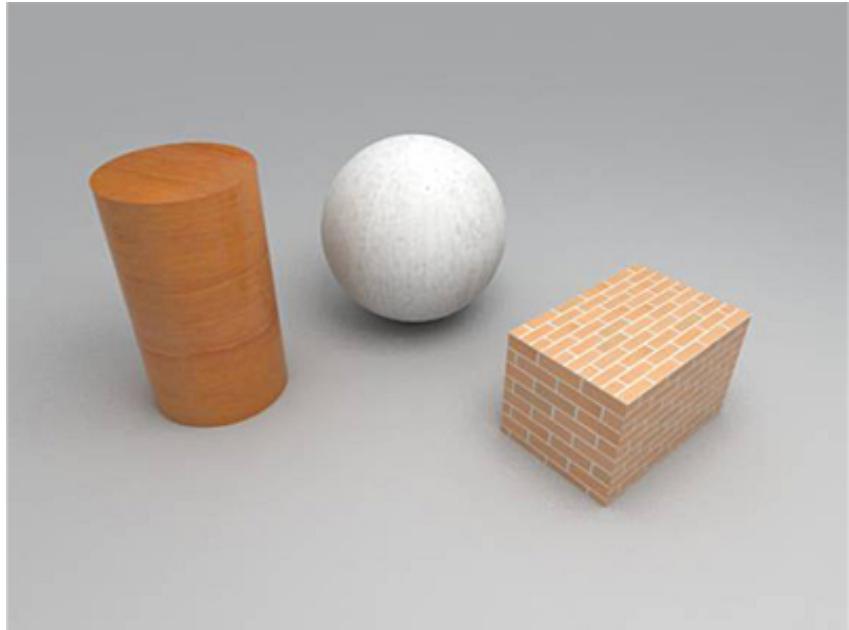
Múltiples efectos se consiguen utilizando texturas:

- Detalles superficie
- Reflejos
- Relieves
- ...

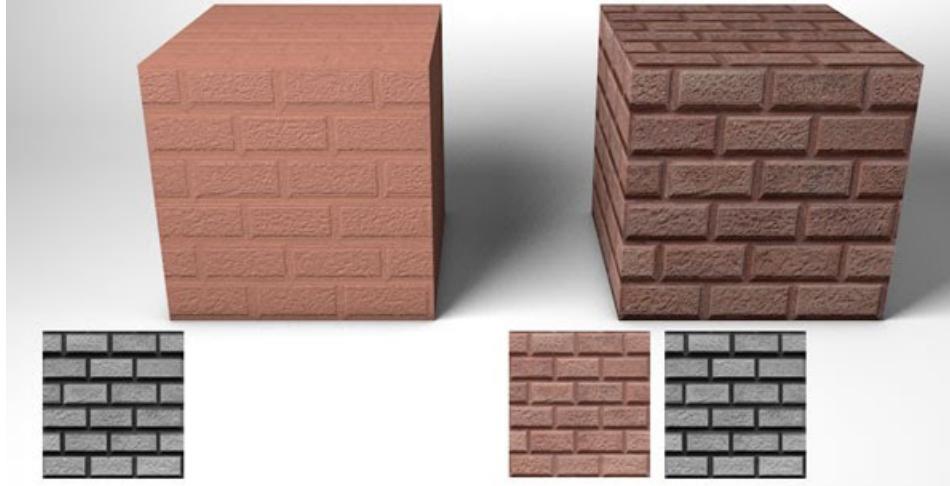
Aumentando la calidad de la imagen: Texturas



Bump Map

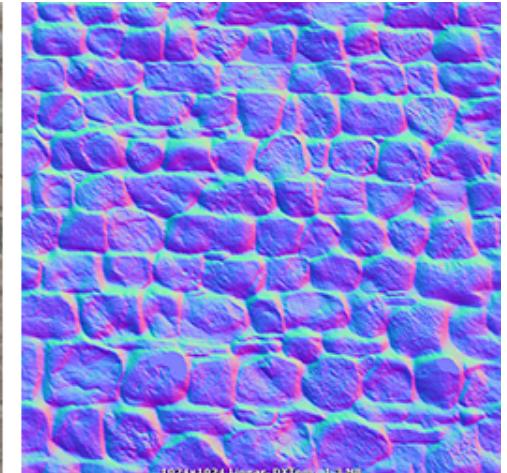
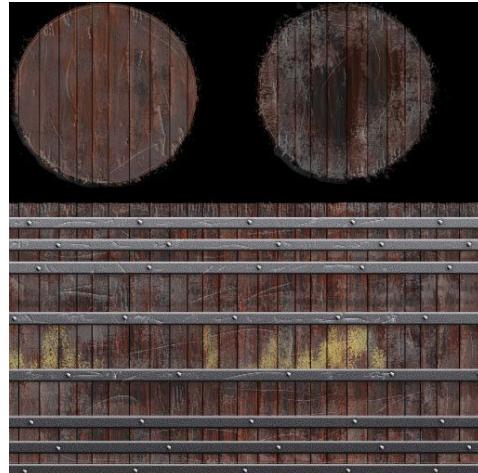
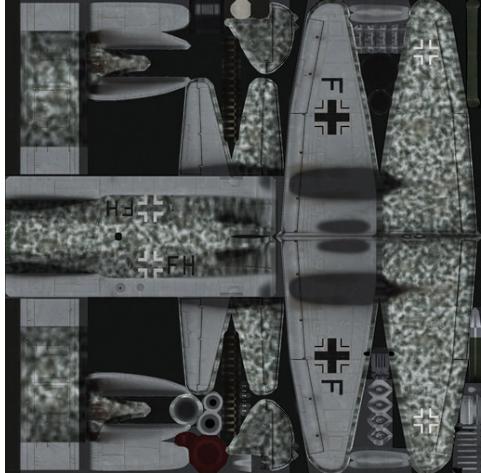


Texture Map + Bump Map



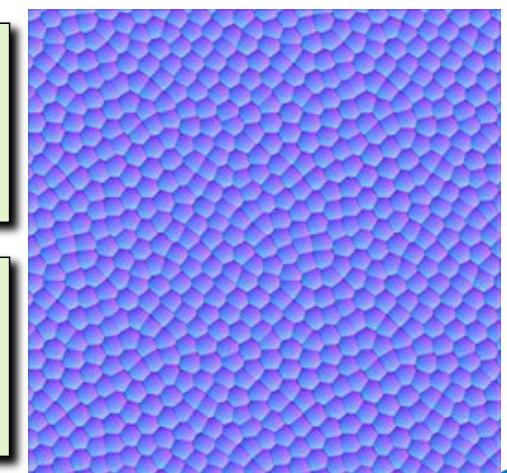
Las texturas permiten aumentar la calidad de la imagen a un coste razonable.

Aumentando la calidad de la imagen: Texturas



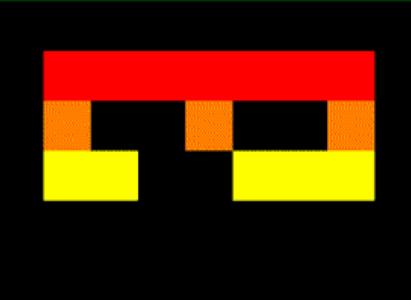
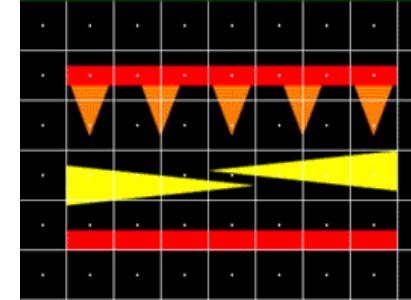
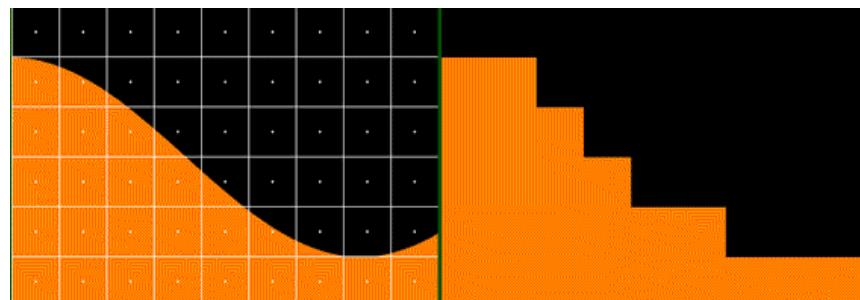
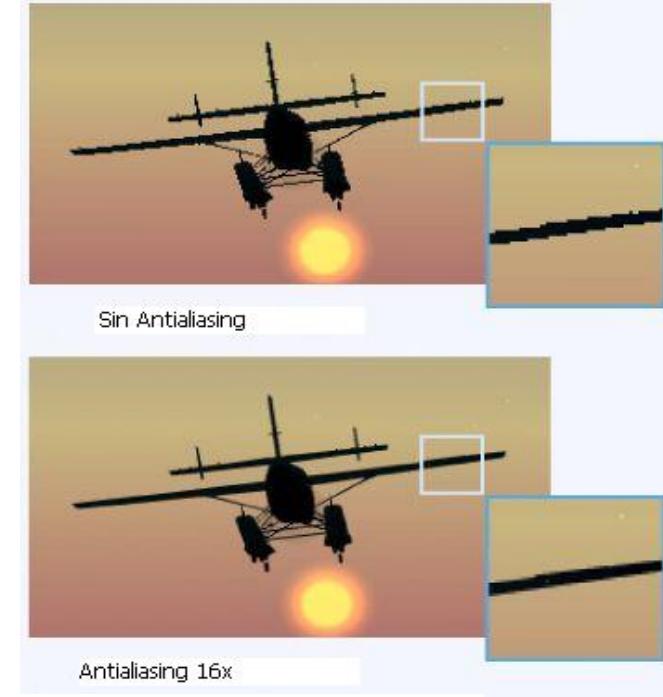
Una textura es una imagen almacenada en un fichero en un formato estándar.
También se puede calcular por programa.

No tienen porqué ser cuadradas, pero en general, las dimensiones son siempre potencia de 2 ($2^m \times 2^n$).

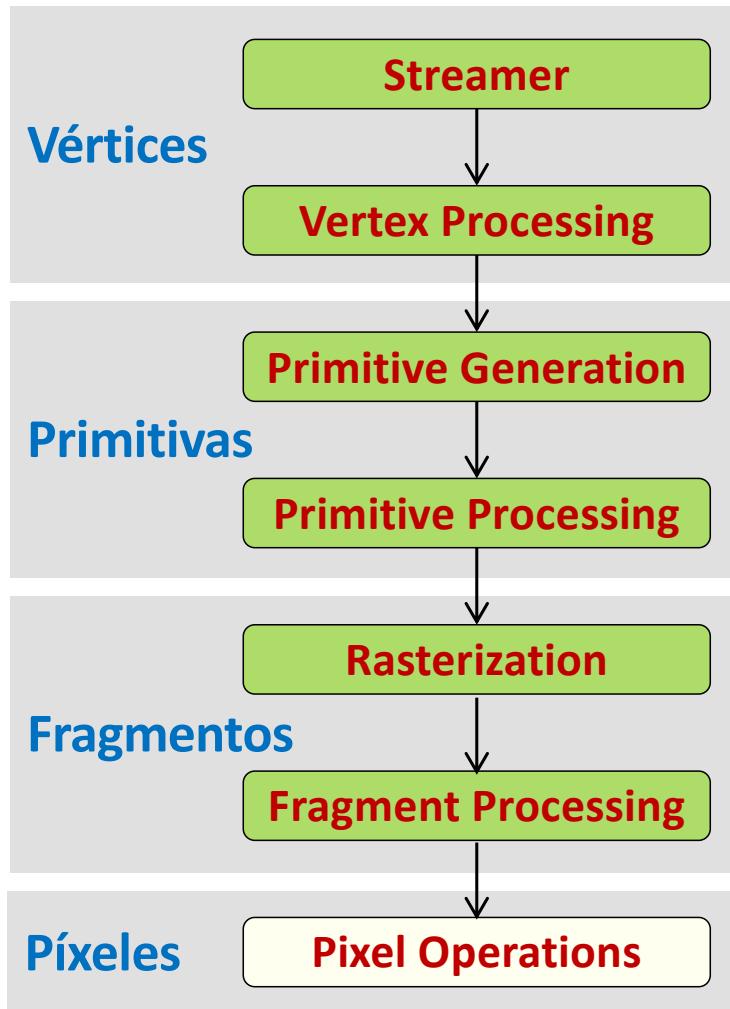


Aliasing

- Artefacto gráfico característico que hace que en pantalla ciertas curvas y líneas inclinadas presenten un efecto visual tipo "sierra" o "escalón".
- Ocurre cuando se intenta representar una imagen con curvas y líneas inclinadas, pero debido a la resolución finita del sustrato resulta que éste es incapaz de representar la curva como tal, y por tanto dichas curvas se muestran en pantalla dentadas al estar compuestas por pixeles.
- Este problema se resuelve utilizando técnicas de **multisampling**.
- Las necesidades de ancho de banda con memoria se multiplican.



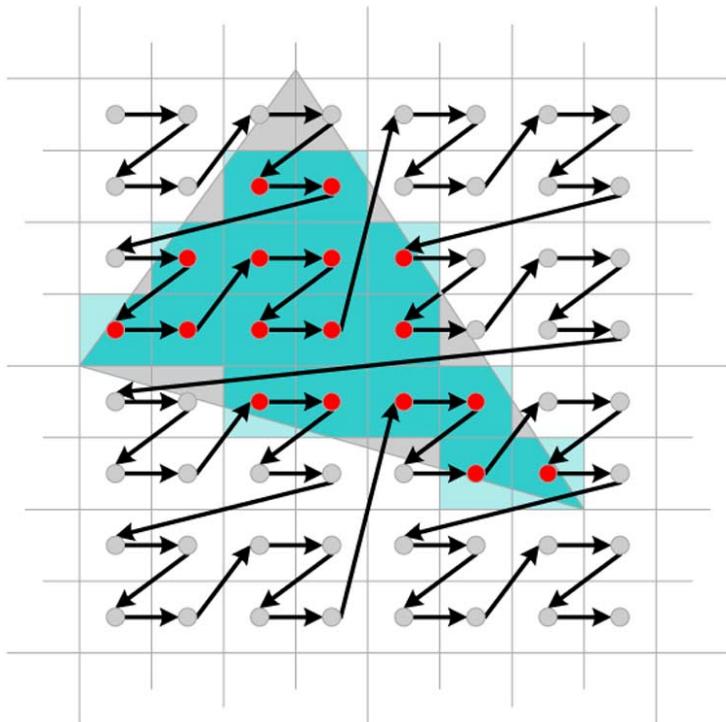
El Pipeline Gráfico: Fragment Processing



- **ELEMENTO PROGRAMABLE** que opera a nivel de fragmento. Para cada fragmento se calcula su color en función de las texturas que se apliquen, de los efectos de transparencia o niebla que tengan, ...
- Este elemento se solía llamar **fragment shader**.
- El lenguaje máquina de los shaders no incluía saltos condicionales ni posibilidad de implementar bucles.
- Todos los accesos a memoria se realizaban a través de la Unidad de Texturas (TU), que además del acceso aplica un filtro (bilinear, trilinear, anisotrópico, ...).
- Las tarjetas gráficas actuales disponen de instrucciones convencionales de acceso a memoria y saltos.

Se puede utilizar una cache de texturas. Aprovecha localidad espacial y temporal.

El Pipeline Gráfico: Fragment Processing



ORGANIZACIÓN
COHERENTE con su
USO

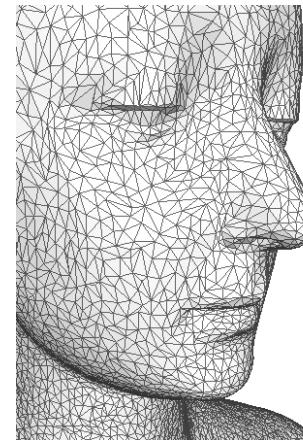
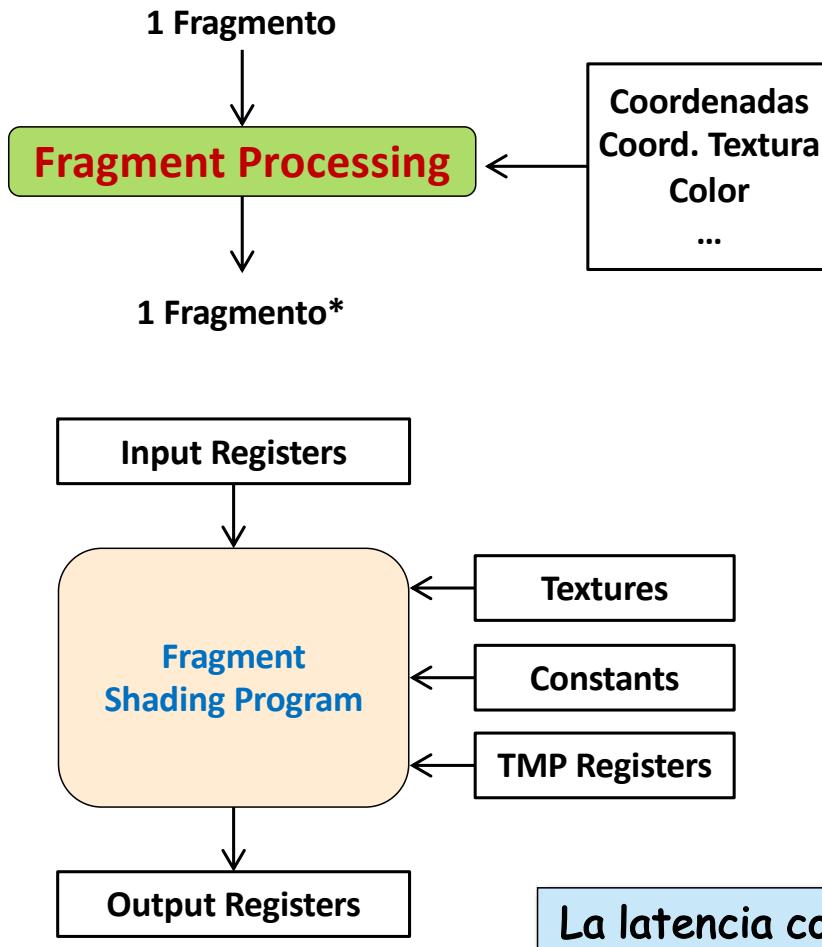
- La información de los fragmentos, texturas, ... se ha de almacenar en memoria siguiendo un patrón especial.
Objetivo: Aprovechar la localidad espacial
- Elementos que se procesarán conjuntamente (p.e. los fragmentos de un mismo triángulo) han de estar en posiciones cercanas de memoria.
- Hay que utilizar un patrón de almacenamiento en memoria diferente al que estamos acostumbrados.
- Si la información de fragmentos y texturas se almacenara igual que se almacena una matriz de datos en una CPU convencional, no se podría aprovechar la localidad espacial de los datos.

Ejemplos Fragment Shader

```
varying vec4 diffuseColor;
varying vec3 lightVector;
varying vec3 fragNormal;
void main() {
    float perFragmentLighting = max(dot(lightVector,fragNormal),0.0);
    gl_FragColor = diffuseColor * lightingFactor;
}

uniform sampler2D myTexture;
#define epsilon 0.0001
void main (void) {
    vec4 value = texture2D(myTexture, vec2(gl_TexCoord[0]));
    if (value[0] > 1.0-epsilon) && (value[2] > 1.0-epsilon)
        discard;
    gl_FragColor = value;
}
```

El Pipeline Gráfico: Fragment Processing



La imagen se genera batch a batch

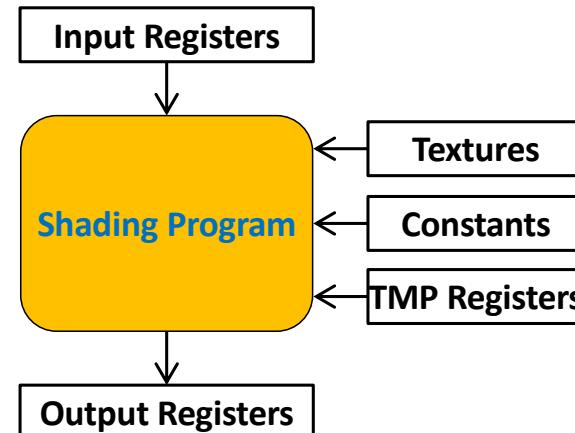
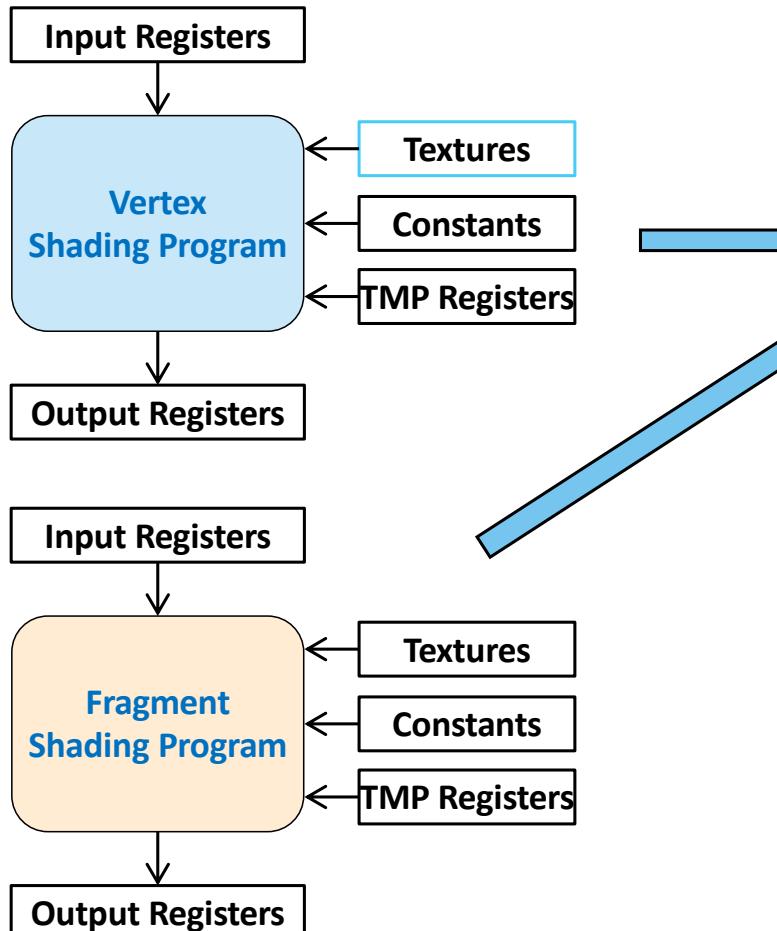
- M fragmentos
- M puede ser enorme ($M \uparrow \uparrow \uparrow$) .
- No hay dependencias entre fragmentos.
- Se pueden procesar en paralelo.
- A todos los fragmentos de un mismo batch se le aplica el **MISMO fragment shading program**.

(*) El número de elementos a procesar en el pipeline no hace más que crecer. Existen numerosos tests dedicados a descartar aquellos elementos que no se van a ver en la imagen final y por tanto no es necesario procesarlos.

APLICACIÓN MASIVAMENTE PARALELA

La latencia con Memoria se oculta utilizando **MULTITHREADING**

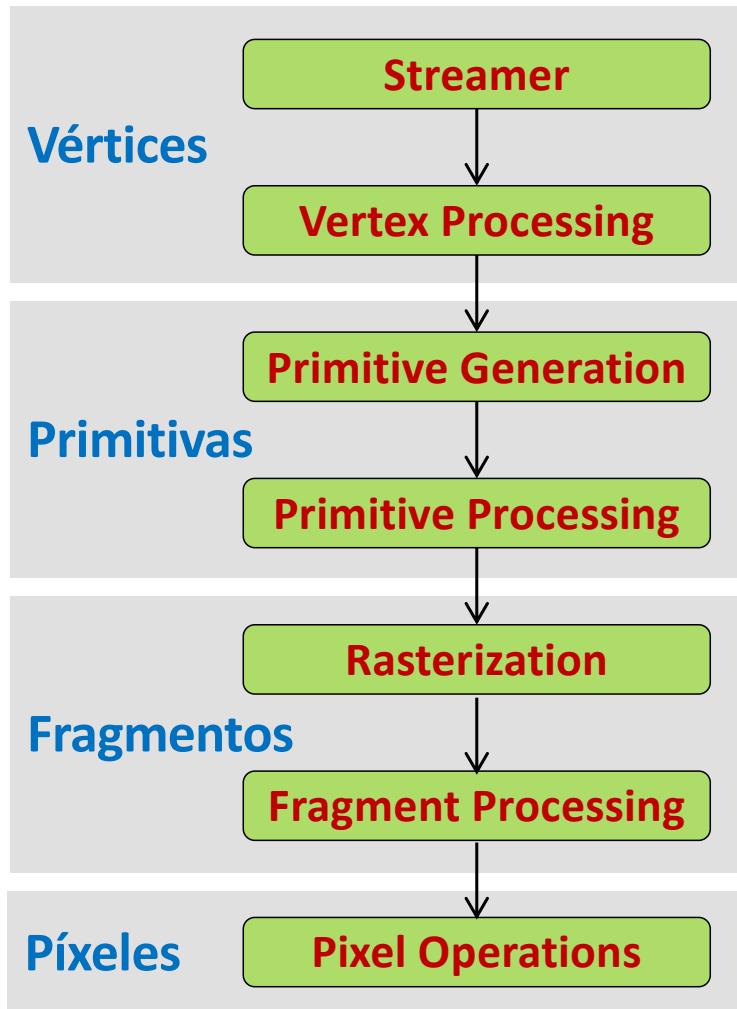
El Pipeline Gráfico: Unified Shaders



Esta Unificación ha provocado un cambio radical en la estructura del pipeline gráfico.

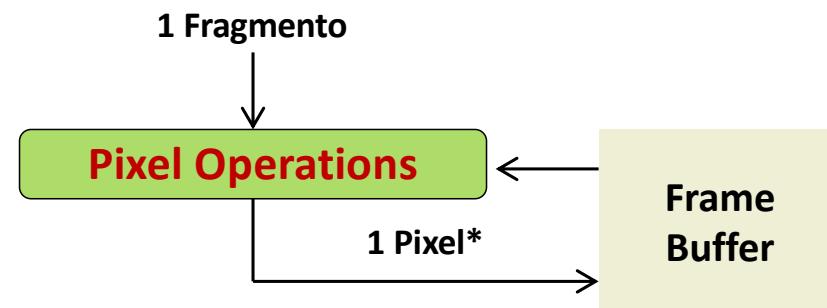
- Originalmente, el vertex shader y el fragment shader eran diferentes. Diferentes usos y diferentes lenguajes máquina.
- Hoy, no existe esta diferencia, **ESTÁN UNIFICADOS**.
- El mismo shader puede procesar: **vértices, fragmentos, primitivas, threads de CUDA/OpenCL, ...**

El Pipeline Gráfico: Pixel Operations

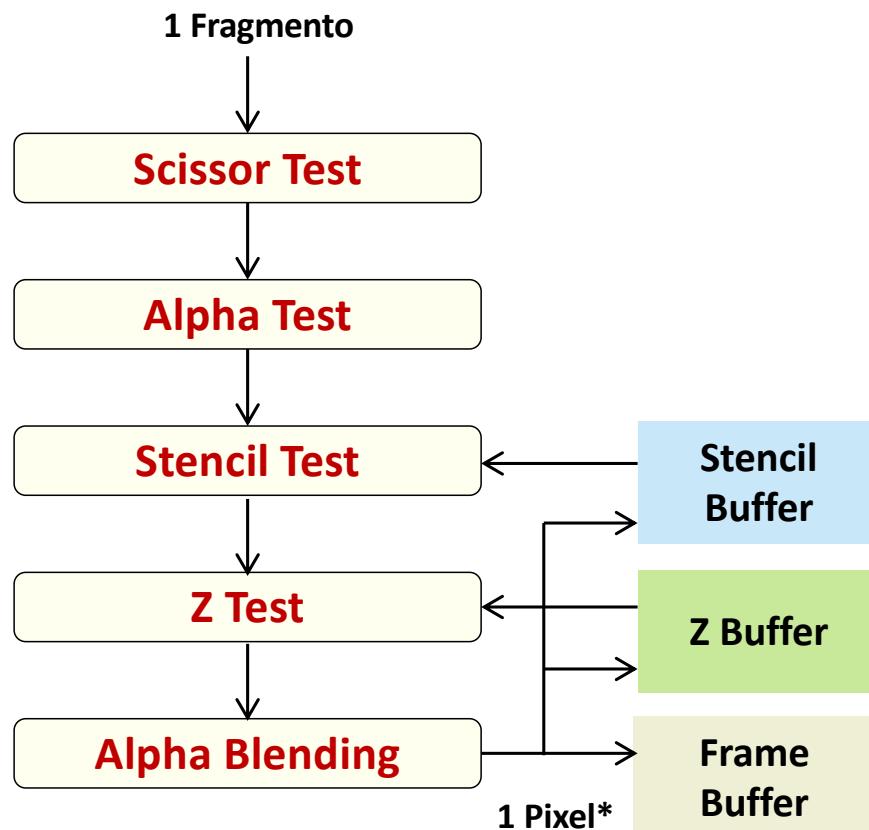


Elemento no Programable: **FUNCIÓN FIJA**

- En la etapa final se realizan las operaciones sobre el **frame buffer**.
- El frame buffer es la zona de memoria destinada a almacenar la imagen.
- Dos tipos de operaciones:
 - Test de “Aceptación / Rechazo” del fragmento/píxel
 - Operación Combinación



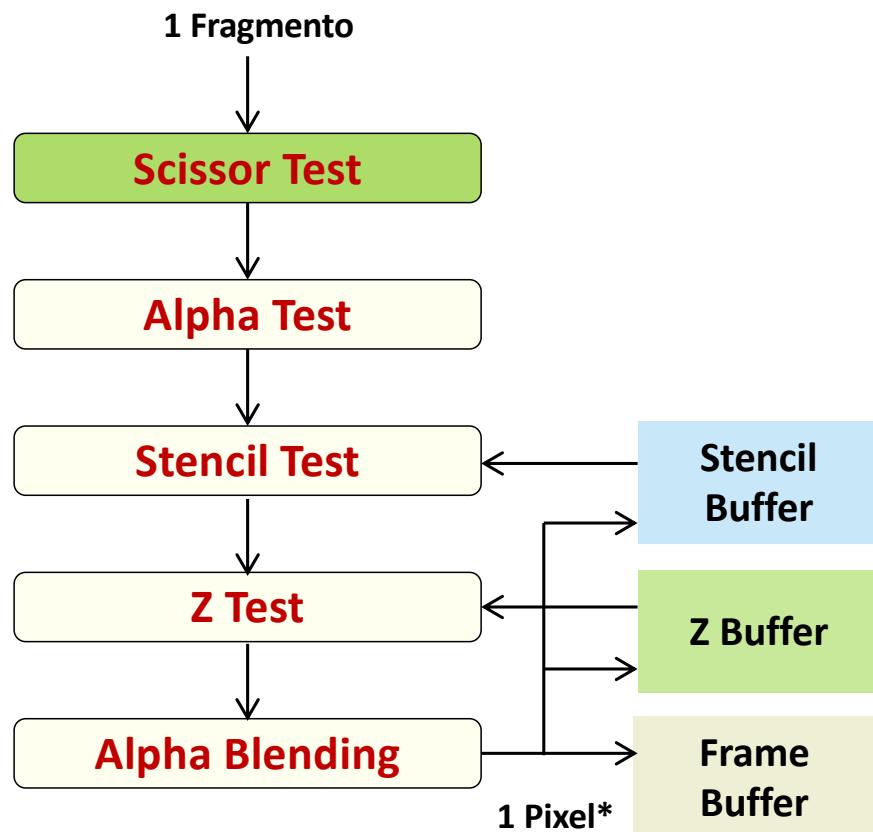
El Pipeline Gráfico: Pixel Operations



Operación	Tipo	Uso
Scissor Test	Aceptación Rechazo	Limitar área renderización
Alpha Test	Aceptación Rechazo	Limitar área de renderización Transparencias (sin orden)
Stencil Test	Aceptación Rechazo	Limitar área renderización Implementación sombras
Z Test	Aceptación Rechazo	Oclusión Objetos
Alpha Blending	Combinación	Transparencias (orden)

Esta fase se ejecuta en las ROPs
(Render OutPut Units)

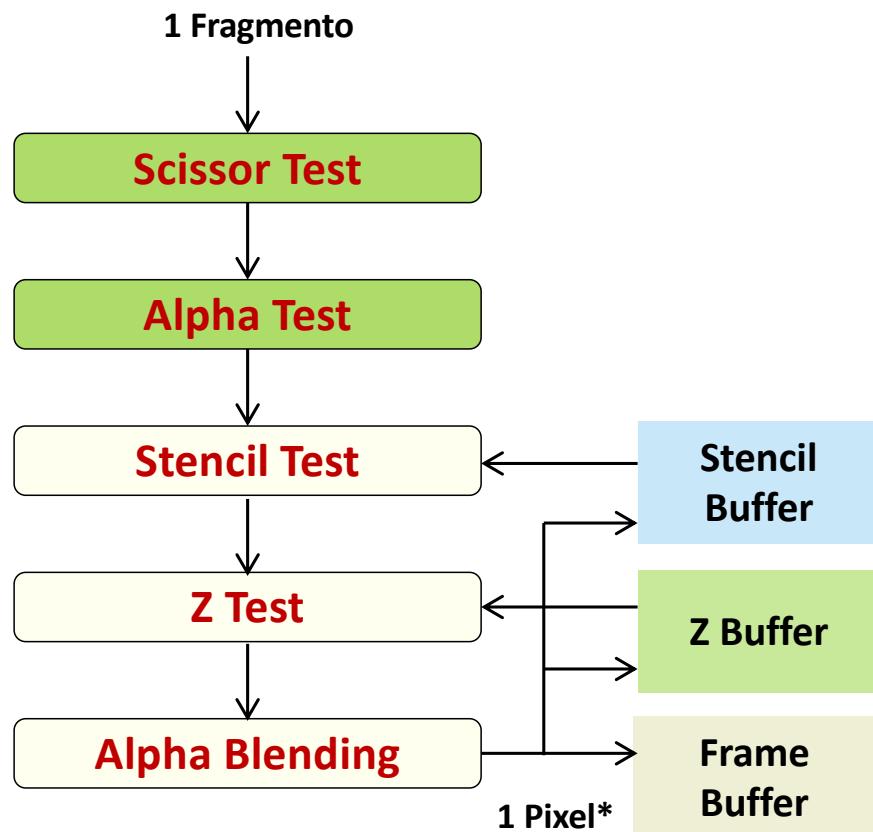
El Pipeline Gráfico: Pixel Operations



Scissor Test

- El Scissor Test simplemente descarta los píxeles que están fuera del área rectangular de renderizado.
- La única información que necesita son las dimensiones del área de renderizado.
- No sólo se renderiza el frame buffer.

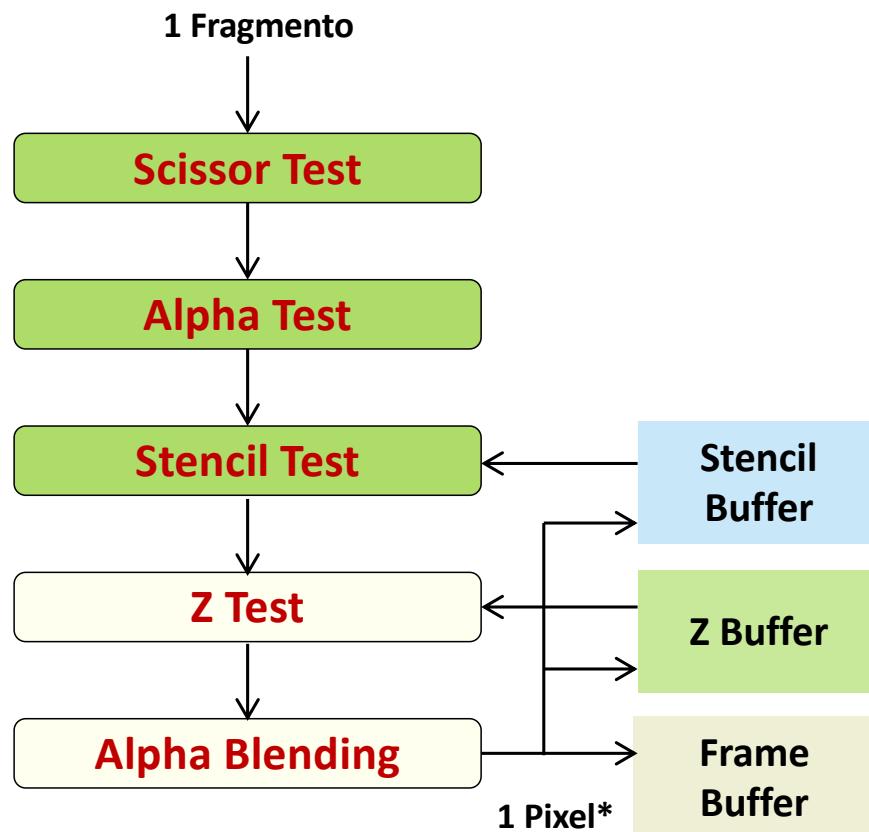
El Pipeline Gráfico: Pixel Operations



Alpha Test

- Junto con la información del color de cada fragmento (R G B) se almacena un 4º dato: canal Alfa (A).
- Corresponde con la transparencia del fragmento:
 - 0, totalmente transparente,
 - 1, totalmente opaco.
- Los fragmentos con un valor Alpha muy pequeño se descartan.

El Pipeline Gráfico: Pixel Operations



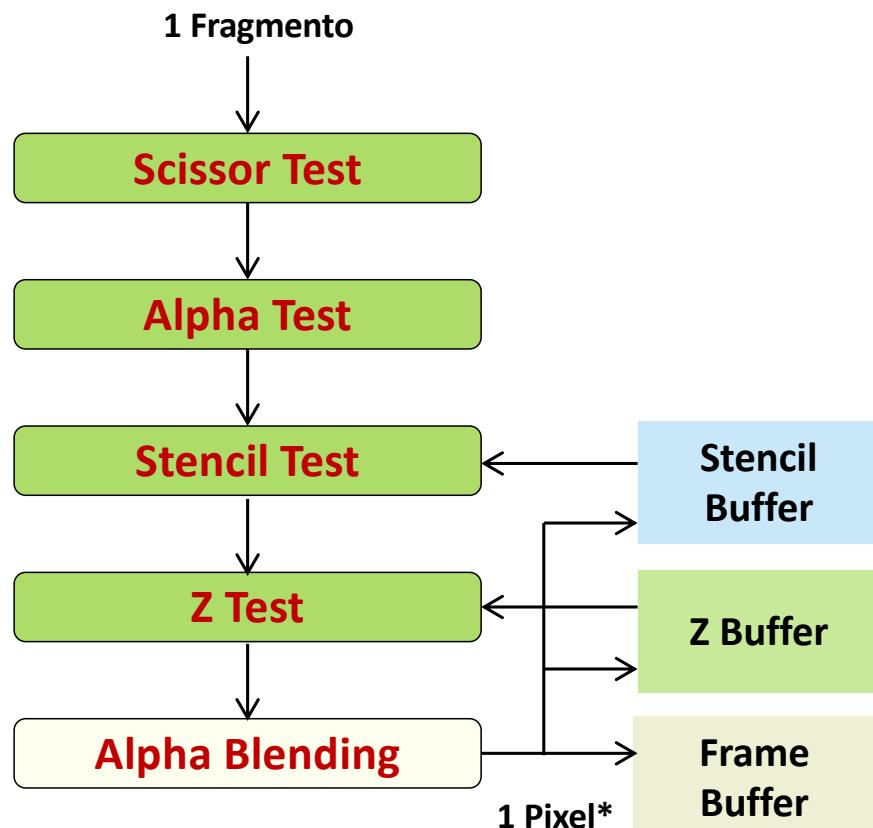
Stencil Test

- El uso más habitual del stencil es la generación de sombras.
- Idea: calcular el volumen de la región sombreada respecto a los puntos de iluminación.
- Proyectar esas sombras sobre el stencil buffer. Marcando a 1 los elementos sombreados y a 0 los iluminados.
- Renderizar la imagen en 2 pasadas:
 - Iluminación ON, sólo dibuja fragmentos con stencil = 0
 - Iluminación OFF, sólo dibuja fragmentos con stencil = 1



El coste en ancho de banda **NO** es despreciable

El Pipeline Gráfico: Pixel Operations



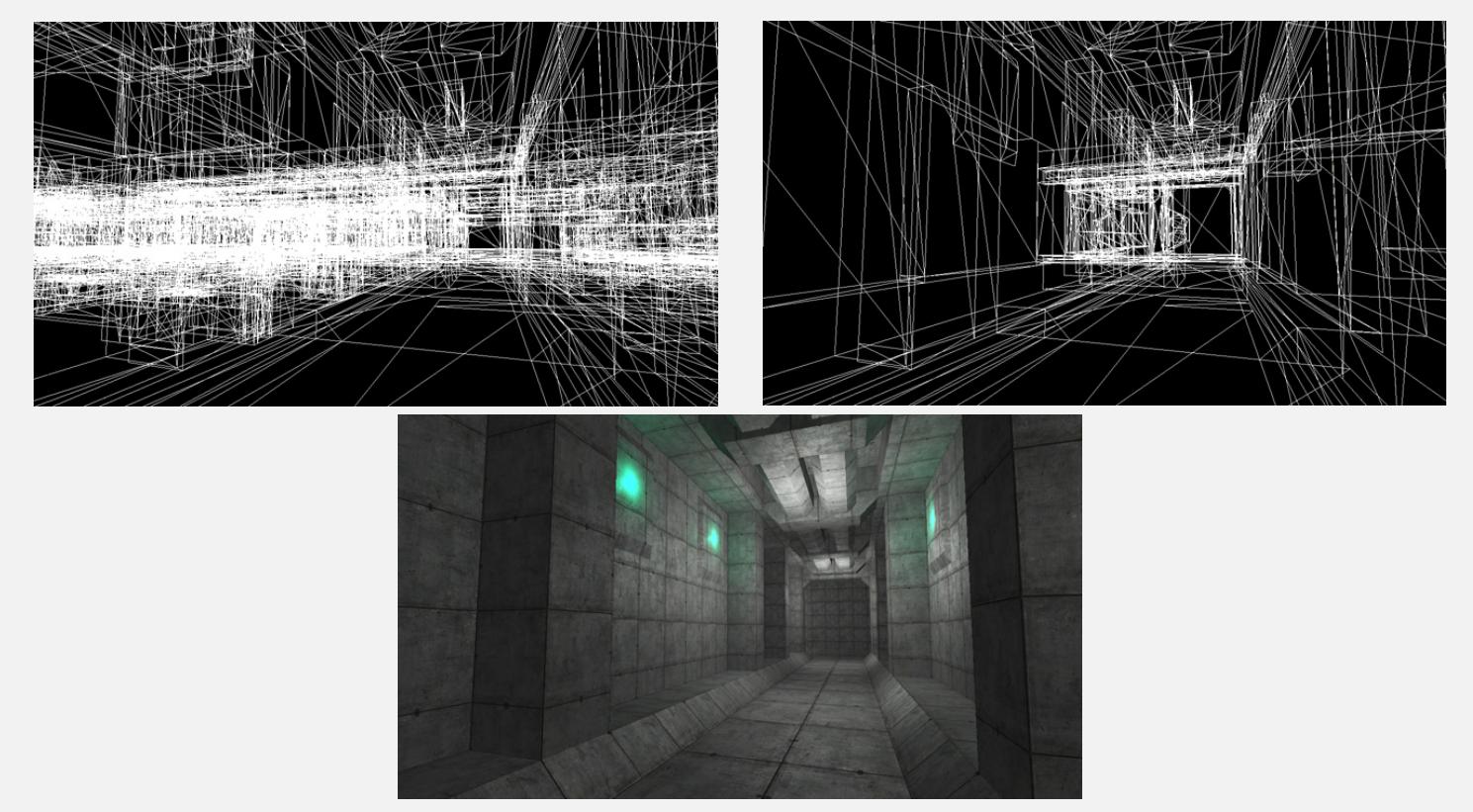
Z Test

- Cada fragmento además de su posición (x,y), conserva información de la coordenada z.
- Es un valor entre 0 y 1, que informa de la distancia del fragmento respecto del observador. [0: más cercano]
- El Z buffer, del tamaño de la imagen, inicialmente todo a 1, almacena el valor de z de cada píxel dibujado.
- Si valor $z(x,y) > \text{BufferZ}(x,y)$ se descarta el fragmento

Test fundamental en el proceso de Renderizado.
Garantiza, sin cálculos adicionales, que los elementos ocultos no sean mostrados.
El procesado de fragmentos es independiente y pueden ser **PROCESADOS en PARALELO**.

Z Test

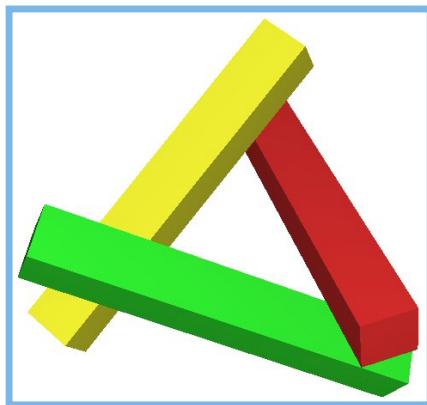
- En una escena 3D, determinar que objetos son visibles y que objetos no lo son, es una tarea muy importante.



Z Test

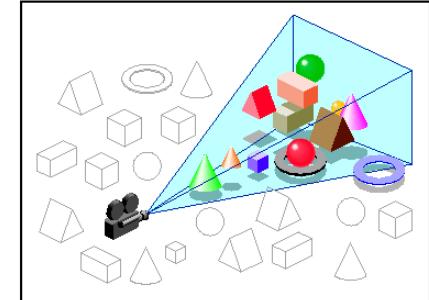
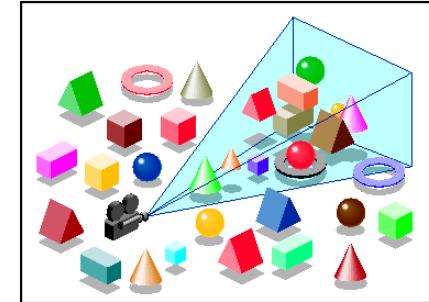
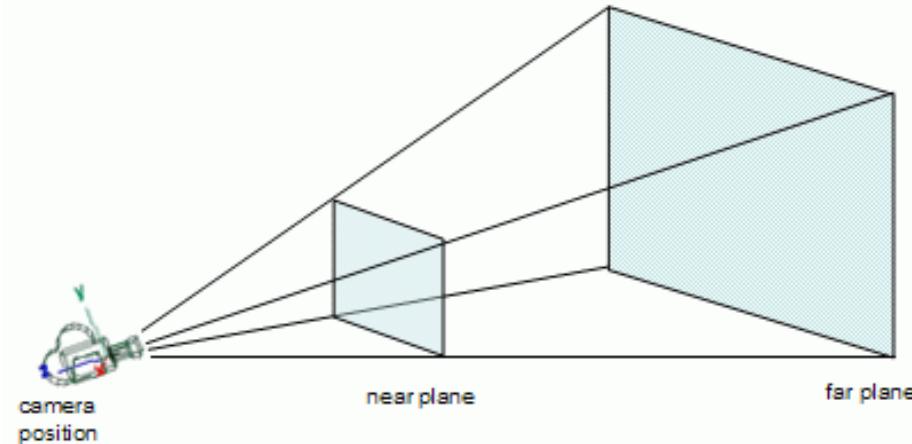
OCCCLUSION CULLING, Algoritmo para eliminar las zonas ocultas de una imagen.

- Existen múltiples algoritmos, en general muy costosos.



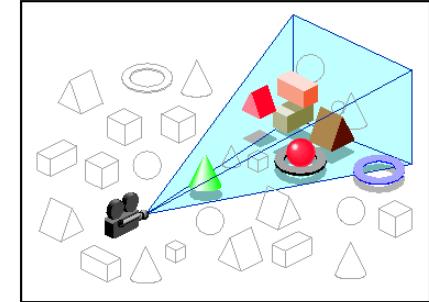
Z Test

- El área de visión (Frustum View) es el primer paso para decidir que objetos son visibles y cuales son descartados.



- La cantidad de trabajo a realizar es muy grande, descartar lo antes posible los elementos no visibles es muy importante.

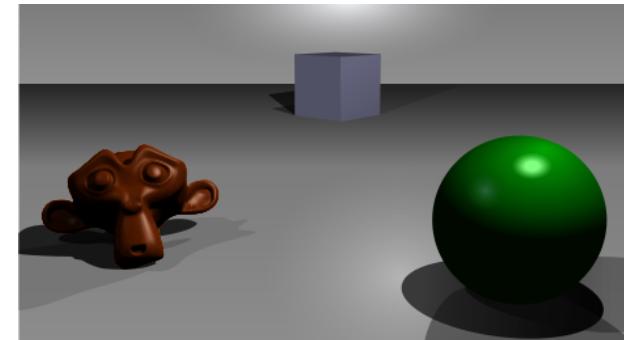
IDEA BÁSICA: NO DIBUJAR lo que NO se vaya a VER.



Z Buffer

- El Z Buffer es un array 2D del tamaño de la imagen a representar.
- Cada posición representa la profundidad (coordenada Z) de un pixel.

```
void InitScreen() {  
    for (i=0; i<N; i++)  
        for (j=0; j<M; j++) {  
            Screen[i][j] = ...;  
            Zbuffer[i][j] = ∞;  
        }  
  
    void DrawPixel(x,y,z,COL) {  
        if (z <= Zbuffer[x][y]) {  
            Screen[x][y] = COL;  
            Zbuffer[x][y] = z;  
        }  
        else DescartarPixel;  
    }  
}
```



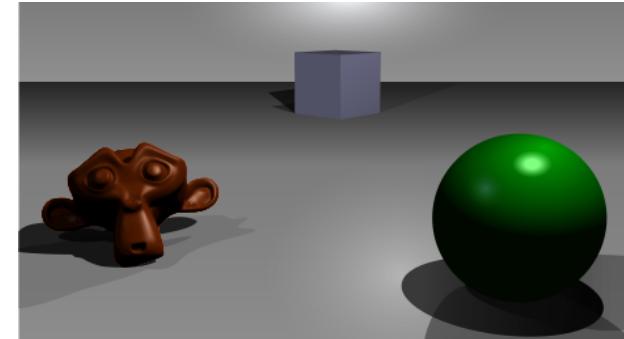
A simple three-dimensional scene



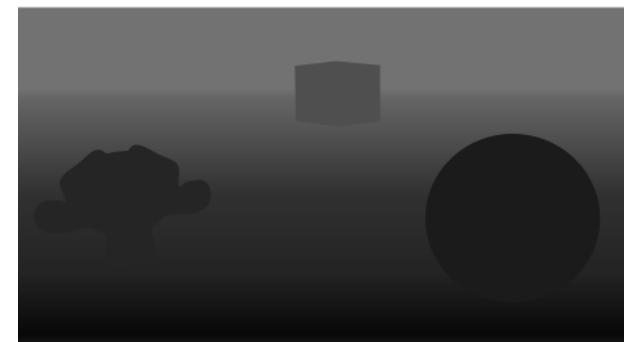
Z-buffer representation

Z Buffer

- En el Z-Buffer se almacena la coordenada de profundidad normalizada a 1
 - 0: NEAR
 - 1: FAR (en realidad 0xFFFFFFFF con 32 bits)
- Utiliza aritmética entera sin signo, aunque represente un número real.
- Problemas de precisión afectan la calidad del algoritmo. Depende de la relación entre los planos far/near del frustum, que se usan al normalizar z.
- Algoritmo sencillo de implementar en hardware.
- Resuelve muchos problemas de visibilidad.
- Problemas con las transparencias y las sombras.



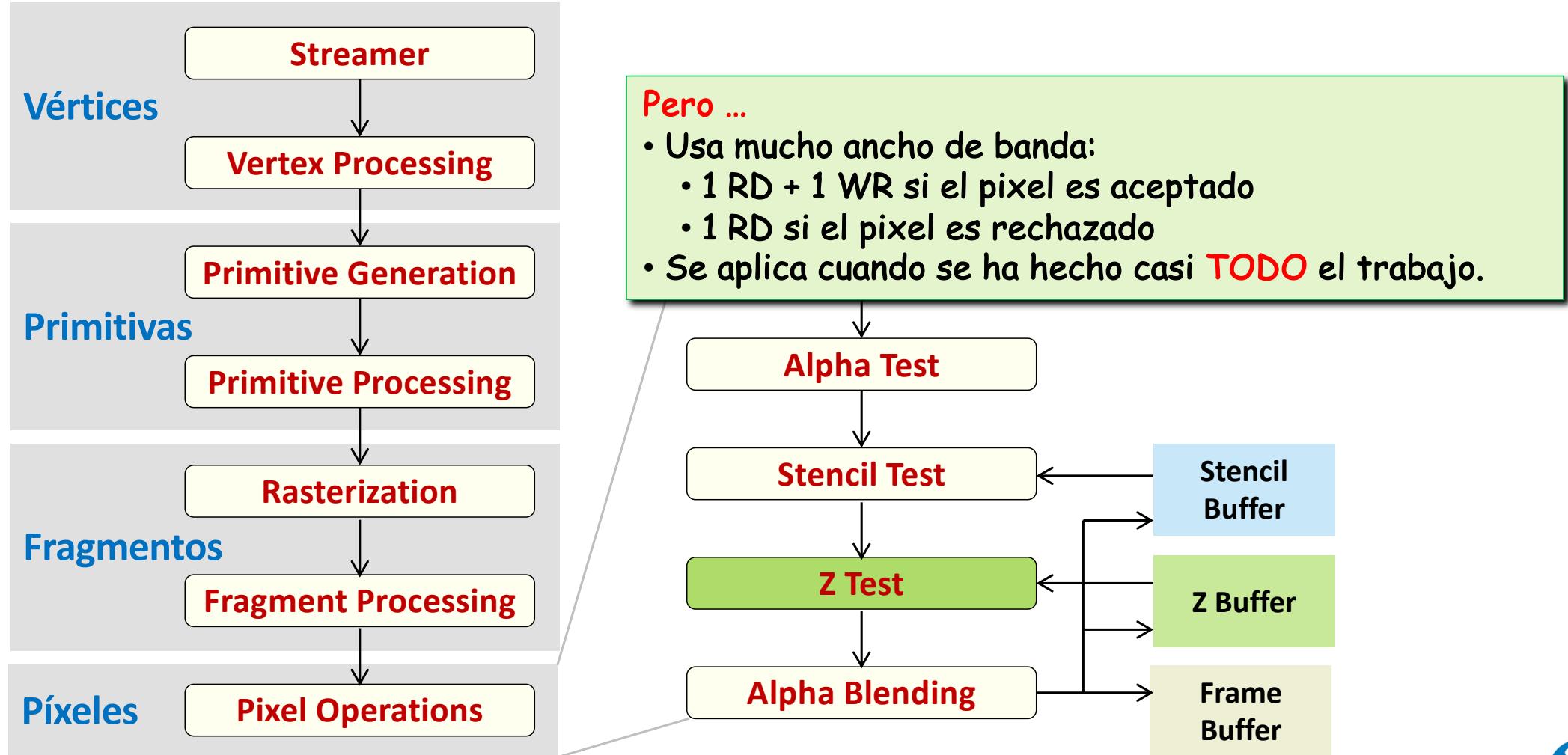
A simple three-dimensional scene



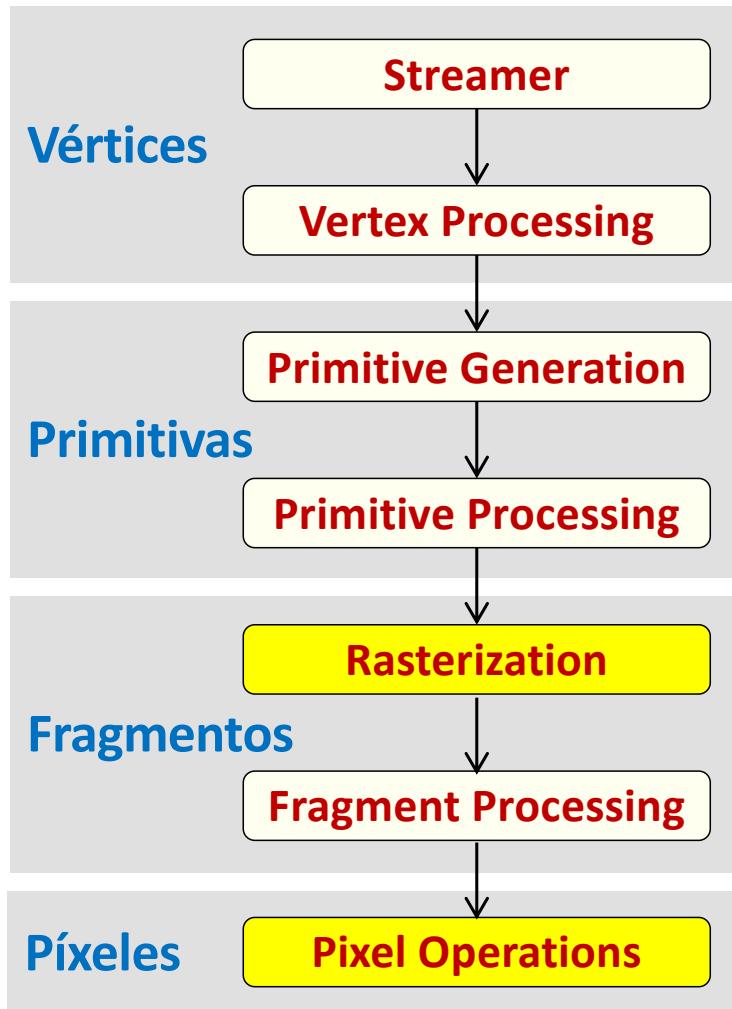
Z-buffer representation

¡ El coste (**ancho de banda**) es muy importante !

Z Buffer



Mejoras Z Buffer



Early Z

- Hacer el test antes del **Fragment Processing**.
- Puede incluirse dentro del proceso de rasterización.
- **Z test** se mantiene.
- El rendimiento depende del orden en que se dibujan las cosas [front to back / back to front].

Compresión Z + cache Z

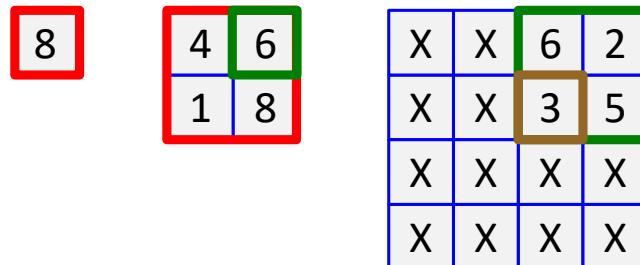
- Reduce necesidad de ancho de banda

Fast Z Buffer Clear

- Reduce necesidad de ancho de banda
- Reduce tiempo de inicialización

Hierarchical Z-Buffer

Idea Básica: mantener varios buffers de Z con resoluciones variables.

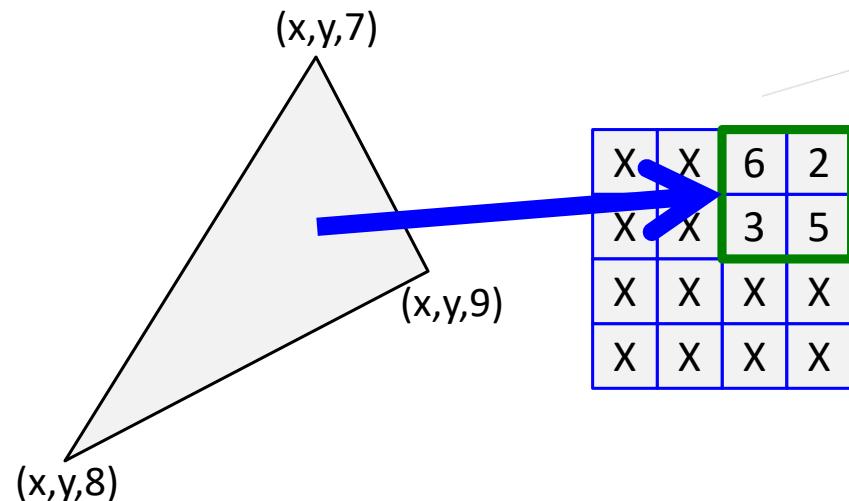
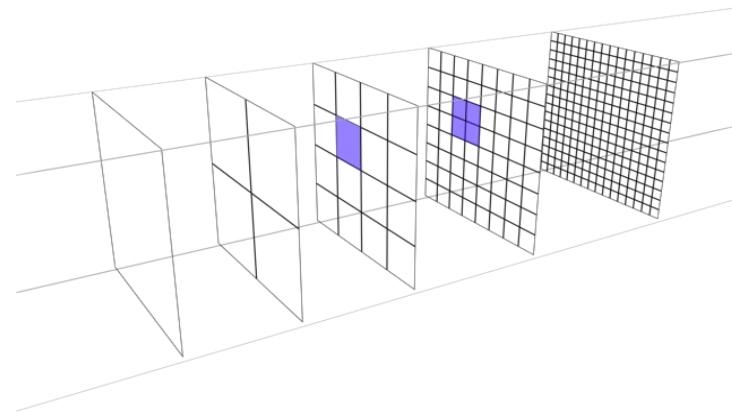


X	X	X	X	4	2	1	2
X	X	X	X	6	3	2	1
X	X	X	X	2	1	3	4
X	X	X	X	3	1	0	5
X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X
X	X	X	X	X	X	X	X

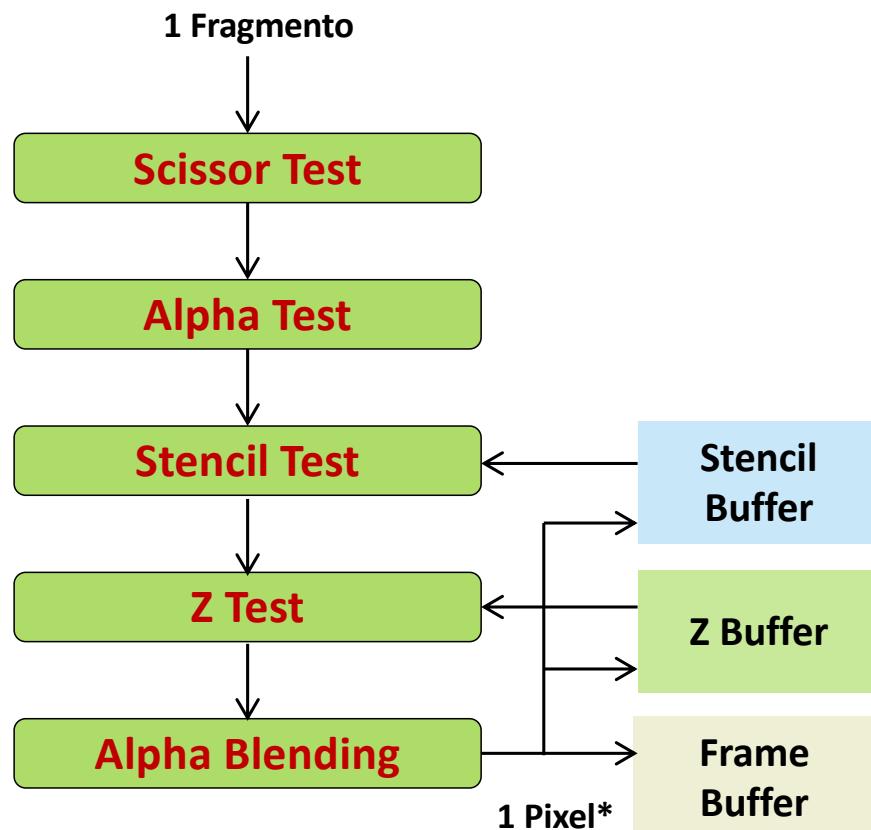
- Algoritmo muy elegante, pero muy costoso en tiempo.
- Si el pixel es aceptado. Hay que acceder a todos los niveles de la jerarquía y además actualizar algunos de ellos.

Hierarchical Z-Buffer

- El algoritmo se podría utilizar con pocos niveles.
 - p.e. Un solo nivel de 32x32
- Este algoritmo, con algunas variantes, se podría utilizar para descartar triángulos completos en la etapa de **Primitive Generation**



El Pipeline Gráfico: Pixel Operations



Alpha Blending

- ❑ Soporte a transparencias utilizando el canal Alpha del color de cada fragmento.
- ❑ Hay que interpolar el color del píxel del frame buffer con el color del píxel recién calculado.



Evolución del Pipeline Gráfico: AMD

AMD	Año	VS	FS	TU	ROP	Mpix/s	Mtex/s
Rage 128 Pro	1999	0	2	2	2	250	250
Radeon 7200	2000	0	2	6	2	333	966
Radeon 7500	2001	0	2	6	3	580	1740
Radeon 9000 Pro	2002	1	4	4	4	1.100	1.100
Radeon 9800 XT	2003	4	8	8	8	3.296	3.296
Radeon X850 XT	2004	6	16	16	16	8.320	8.320
Radeon X1800 XT	2005	8	16	16	16	10.000	10.000
Radeon X1950 XT	2006	8	48	16	16	10.000	10.000
Radeon HD 2900 XT	2007	320		16	16	11.900	11.900
Radeon HD 4870	2008	800		40	16	12.000	30.000
Radeon HD 4890	2009	800		40	16	13.600	34.000
Radeon HD 5870	2009	1.600		80	32	27.200	68.000
Radeon HD 6970	2010	1.536		96	32	28.200	84.500
Radeon HD 6930	2011	1.280		80	32	24.000	60.000
Radeon HD 7970	2012	2.048		128	32	29.600	118.400
Radeon HD 8970	2013	2.048		128	32	33.600	134.400
Radeon R9 290X	2013	2.816		176	64	64.000	176.000
Radeon R9 280	2014	1.792		112	32	26.500	92.600
Radeon R9 Fury X	2015	4.096		256	64	67.200	268.800
Radeon RX 480	2016	2.304		144	32	35.800	161.300
Radeon RX Vega 64	2017	4.096		256	64	98.900	395.800
Radeon RX 590	2018	2.304		144	32	47.000	211.500
Radeon VII	2018	3.840		240	64	115.000	432.000
Radeon RX 5700 XT	2019	2.560		160	64	102.700	256.800
Radeon RX 6900 XT	2020	5.120		320	128	288.000	720.000

Geometría en la CPU o función fija

Primeras GPUs programables

Shaders diferenciados

Shaders unificados

VS: Vertex Shaders

FS: Fragment Shaders

TU: Texture Units

ROP: Render OutPut Units
(Pixel Operations)

Evolución del Pipeline Gráfico: NVIDIA

NVIDIA	Año	VS	FS	TU	ROP	Mpix/s	Mtex/s
Riva TNT2	1999	0	2	2	2	250	250
GeForce 256 DDR	2000	0	4	4	4	480	480
Geforce2 Pro	2000	0	4	8	4	800	1.600
GeForce3	2001	1	4	8	4	800	1.600
GeForce4 Ti 4600	2002	2	4	8	4	1.200	2.400
GeForce FX 5950 Ultra	2003	3	4	8	4	1.900	3.800
GeForce 6800 GT	2004	6	16	16	16	5.600	5.600
GeForce 6800 Ultra	2005	6	16	16	16	6.400	6.400
GeForce 7900 GTX	2006	8	24	24	16	10.400	15.600
GeForce 8800 Ultra	2007	128		32	24	14.700	39.200
GeForce 9800 GTX	2008	128		64	16	10.800	43.200
GeForce GTS 150	2009	128		64	16	20.736	51.840
GeForce GTX 285	2009	240		80	32	11.808	47.232
GeForce GT 340	2010	96		32	8	4.400	17.600
GeForce GTX 480	2010	480		60	48	33.600	42.000
GeForce GTX 560 Ti	2011	384		60	48	29.280	32.210
GeForce GTX 680	2012	1.536		128	32	32.200	128.800
GeForce GTX Titan	2013	2.688		224	48	40.200	187.500
GeForce GTX Titan Black	2014	2.880		240	48	42.700	213.400
GeForce GTX Titan X	2015	3.072		192	96	96.000	192.000
Nvidia Titan X Pascal	2016	3.584		224	96	136.000	317.400
Nvidia Titan V	2017	5.120		320	96	153.600	384.000
Nvidia Titan RTX	2018	4.608		288	96	129.600	388.800
GeForce RTX 2080 Super	2019	3.072		192	64	105.600	316.800
GeForce RTX 3090	2020	10.496		328	112	134.400	459.200

Geometría en la CPU o función fija

Primeras GPUs programables

Shaders diferenciados

Shaders unificados

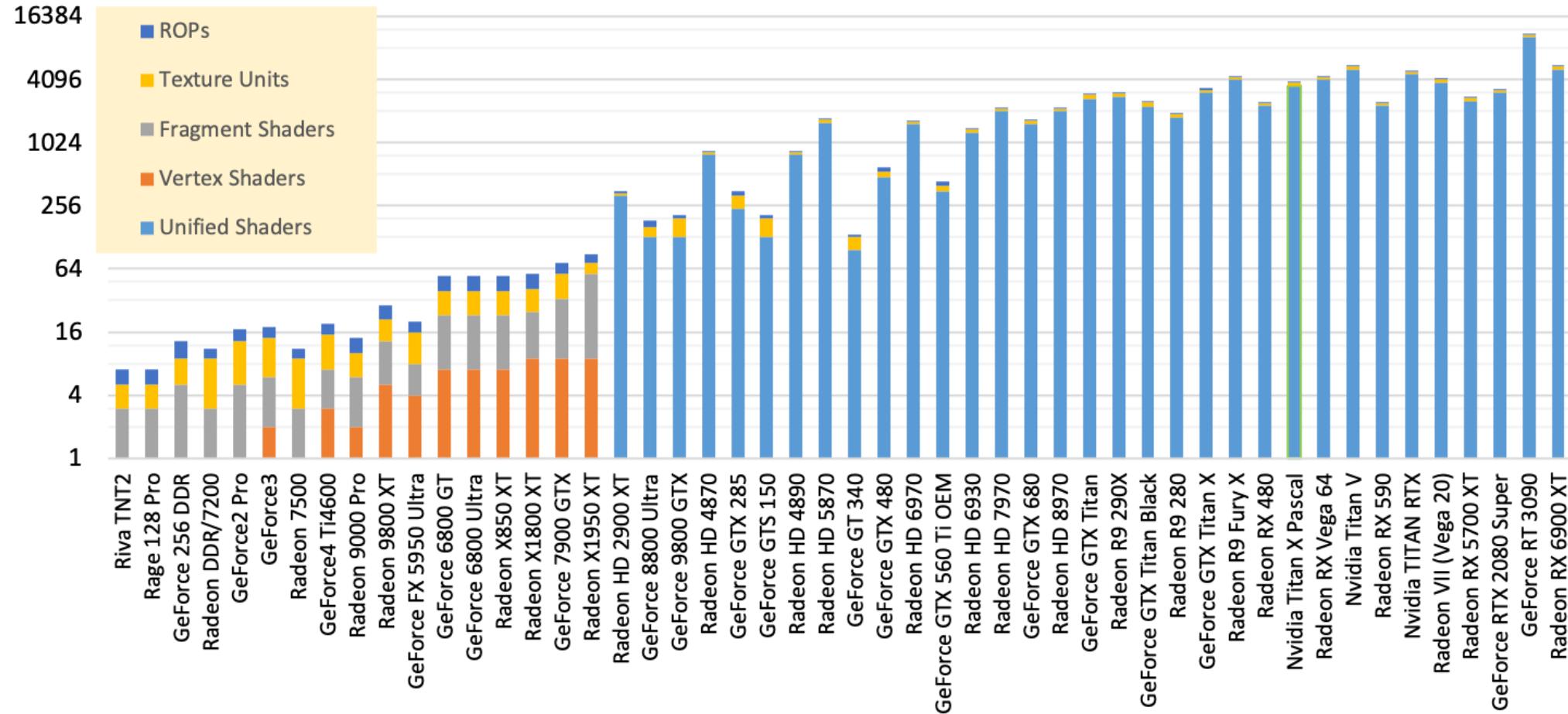
VS: Vertex Shaders

FS: Fragment Shaders

TU: Texture Units

ROP: Render OutPut Units
(Pixel Operations)

Elementos de cálculo



Selección de GPUs ordenada por orden cronológico. A partir de 2007 los shaders están unificados.

Los nuevos tiempos: NVIDIA Titan RTX

GPU
Familia: Turing
Nombre: TU102
Tecnología: 12 nm
Transistores: $18 \cdot 600 \cdot 10^6$
Die Size: 754 mm ²
GPU Clock: 1350 MHz
Boost Clock: 1770 MHz

Tarjeta
Lanzamiento: Dec 2018
Bus: PCIe x16 3.0
Output: 1 HDMI, 3 DisplayPort, 1 USB-C
Power Input: 2x 8-pin



TECHPOWERUP

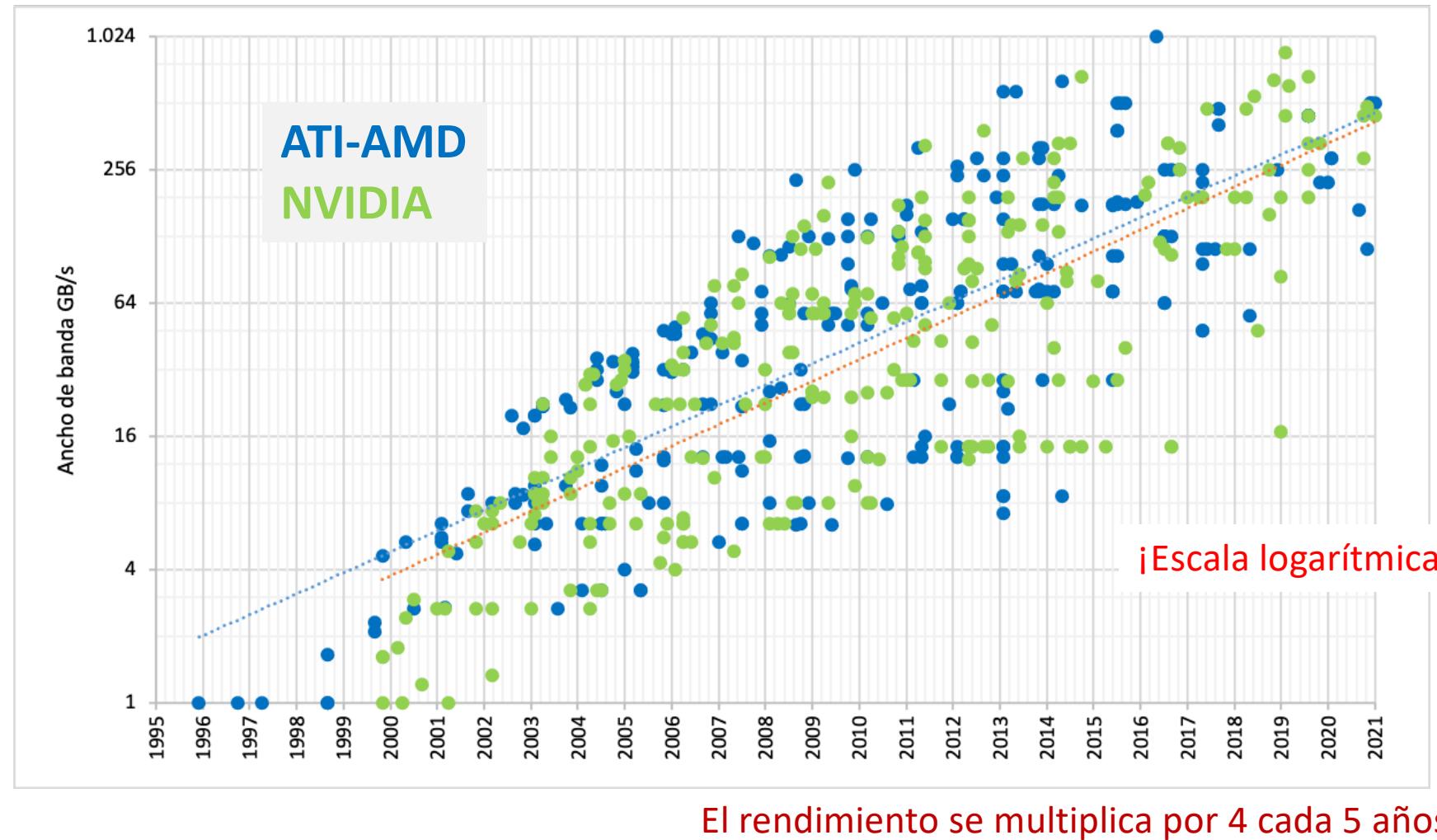
API
DirectX: 12.0
OpenGL: 4.6
OpenCL: 2.0
Shader Model: 6.3

Rendimiento
Pixel rate: 169.9 Gpix/s
Texture rate: 509,8 Gtex/s
GFLOPs FP32: 16.312
GFLOPs FP64: 509,8
GFLOPs FP16: 32.625
Consumo: 275 W
G3D mark: 16.367

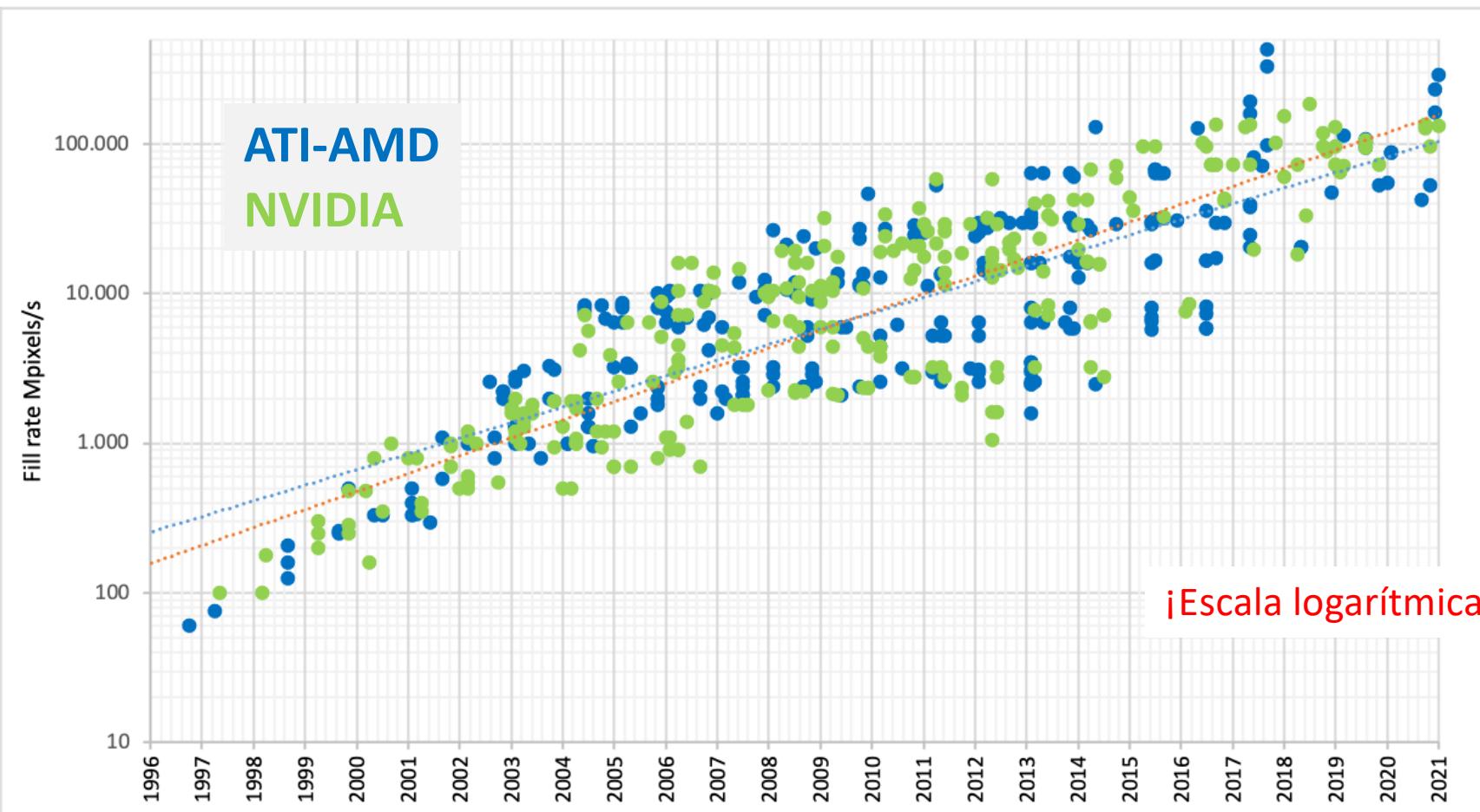
Memoria
Tamaño: 24 GB
Tipo: GDDR6, 1.75GHz
Bandwidth: 672 GB/s
Bus: 384 bits

Configuración
SMM: 72
Shaders: 4608
TMUs: 288
ROPs: 26
Tensor Cores: 576
RT Cores: 72

Ancho de Banda (GB/s)

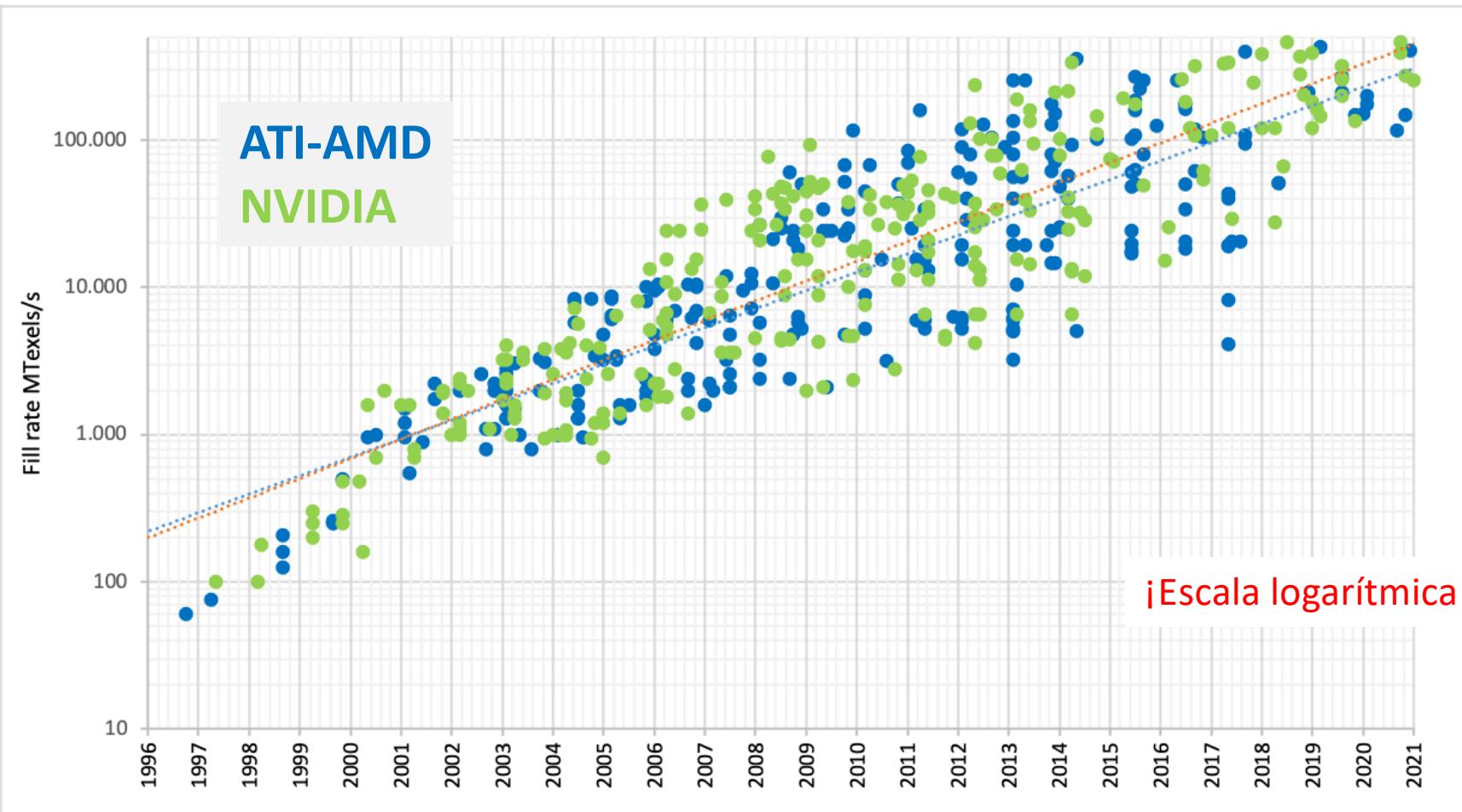


Fill Rate (MPíxeles/s)



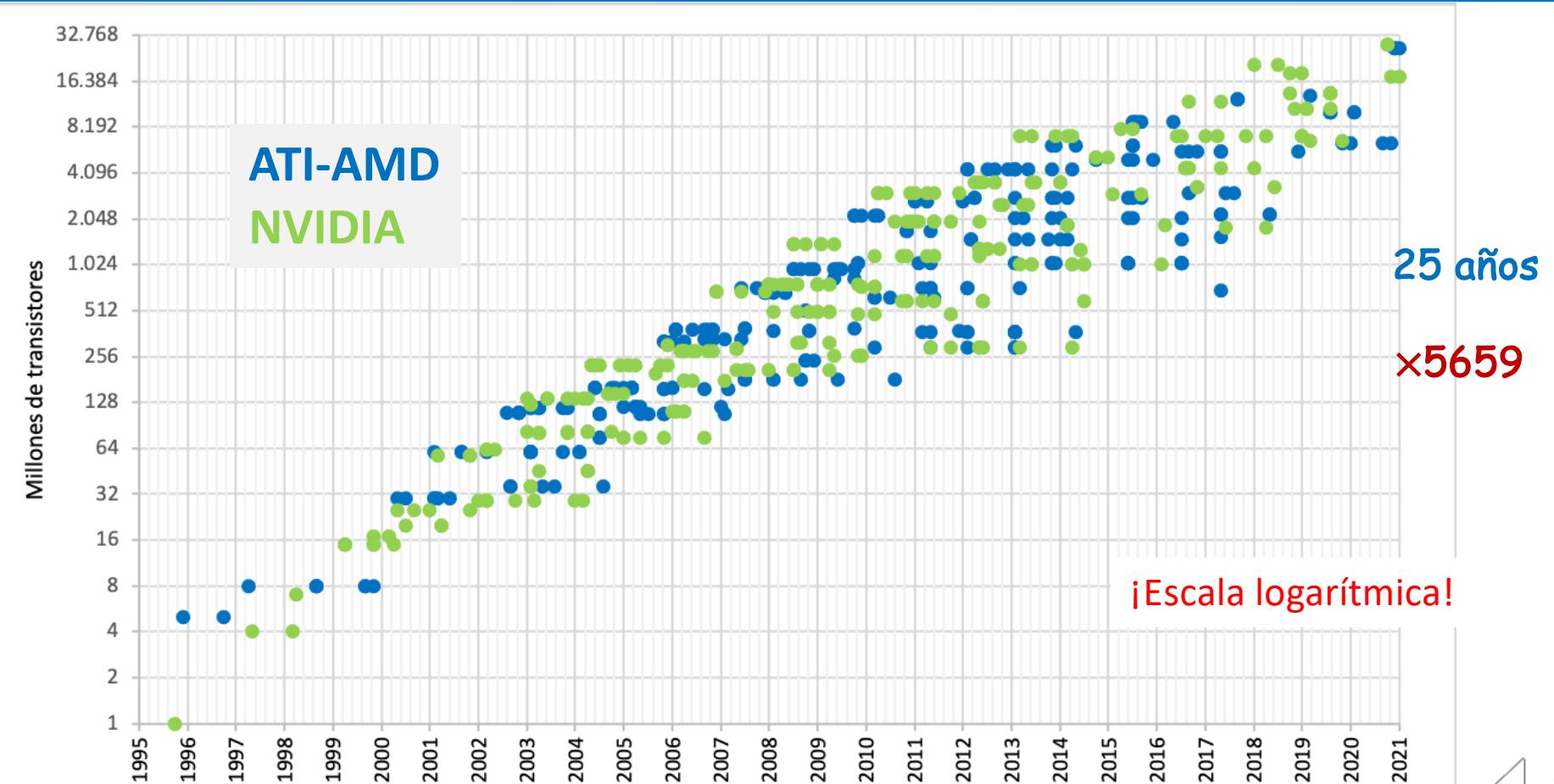
Fill Rate (Mpíxeles/s): número de píxeles que una GPU puede renderizar y escribir en el frame buffer por segundo.

Fill Rate (MTexels/s)

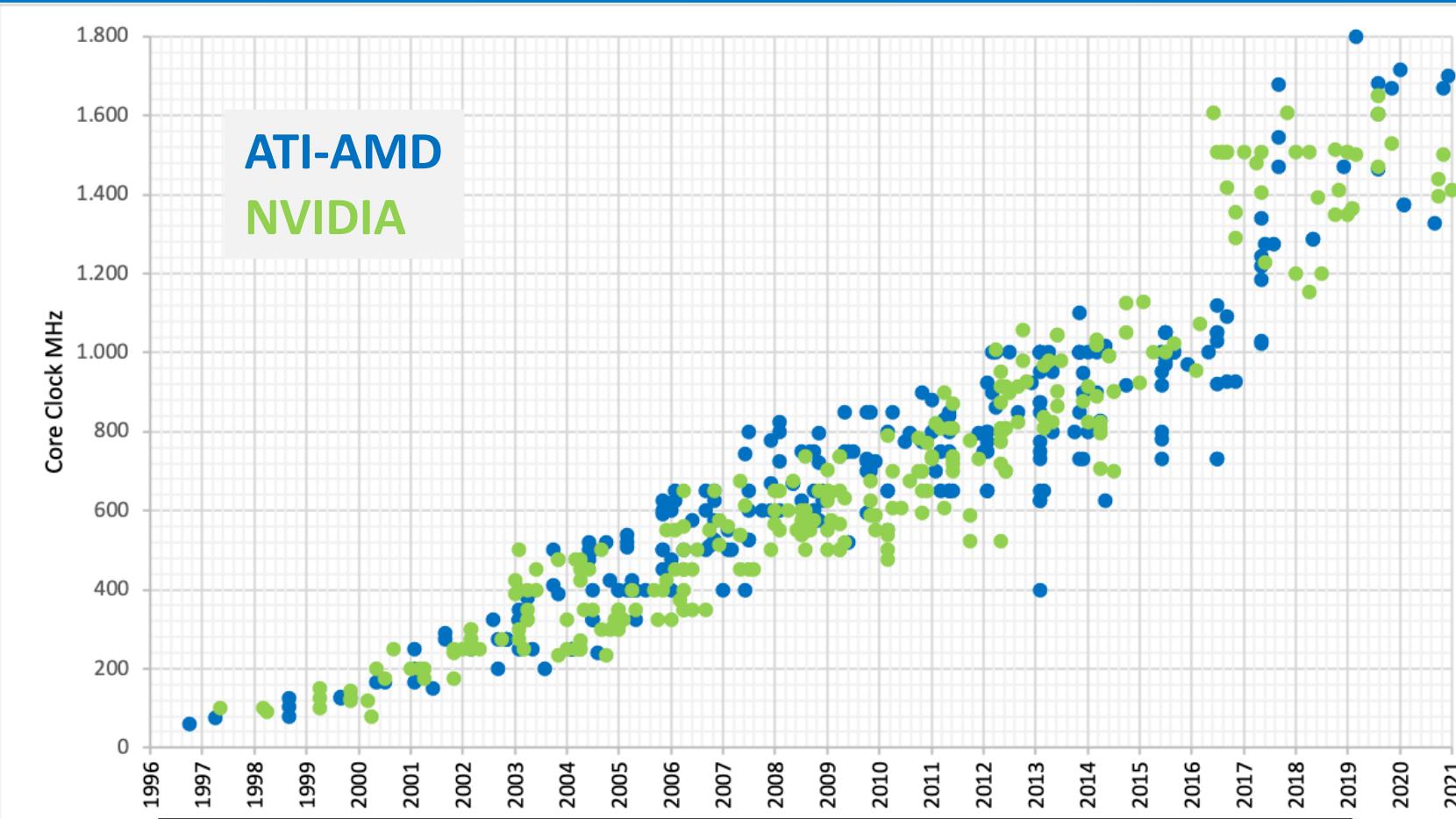


Fill Rate (MTexels/s): número de texels (unidad mínima de textura) que una GPU puede procesar por segundo.

Transistores (10^6)

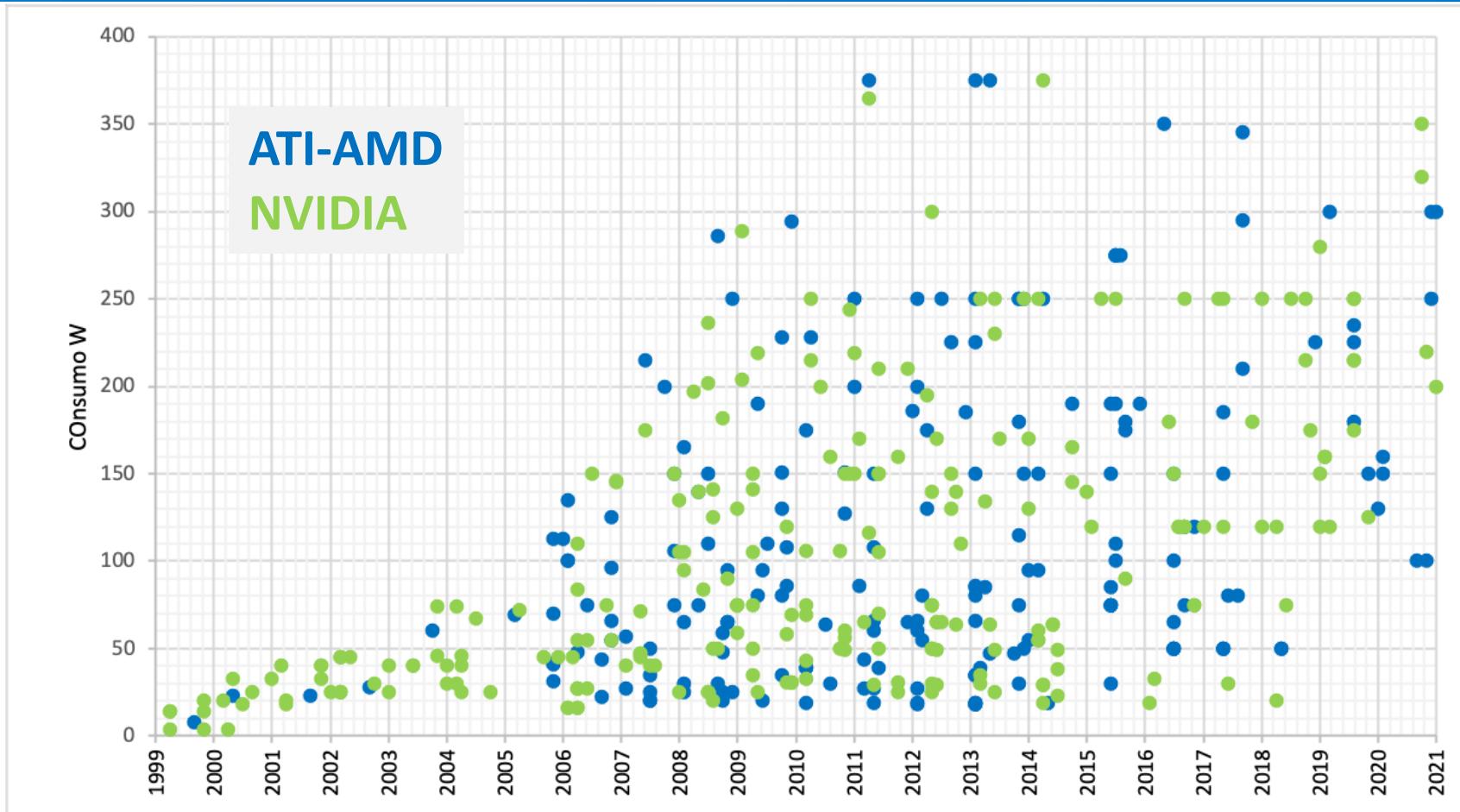


Core Clock (MHz)



Core Clock: Frecuencia mínima de funcionamiento de los shaders. En las GPUs modernas, se puede incrementar alrededor de un 10-15% (Boost Core Clock).

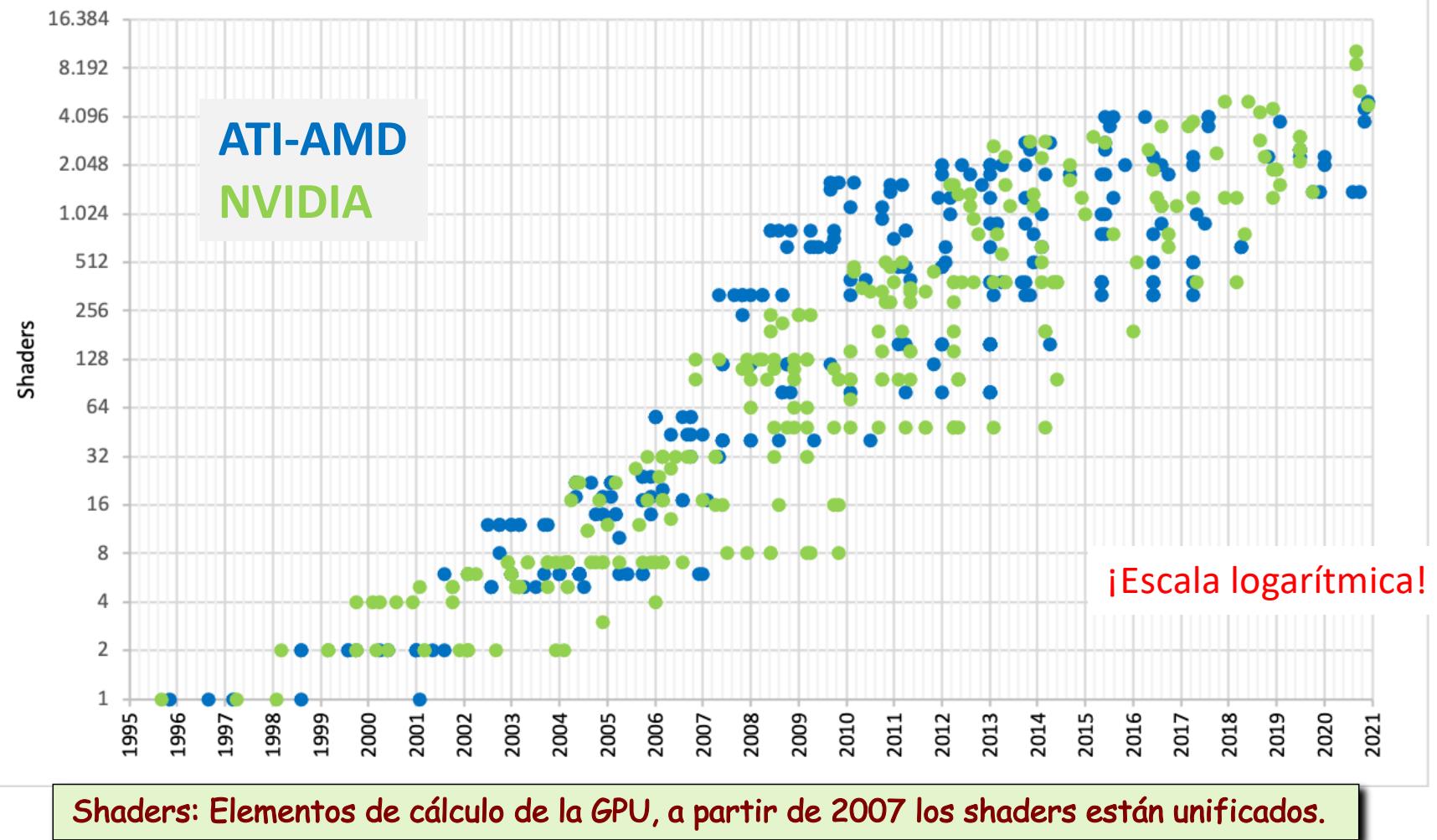
Consumo (W)



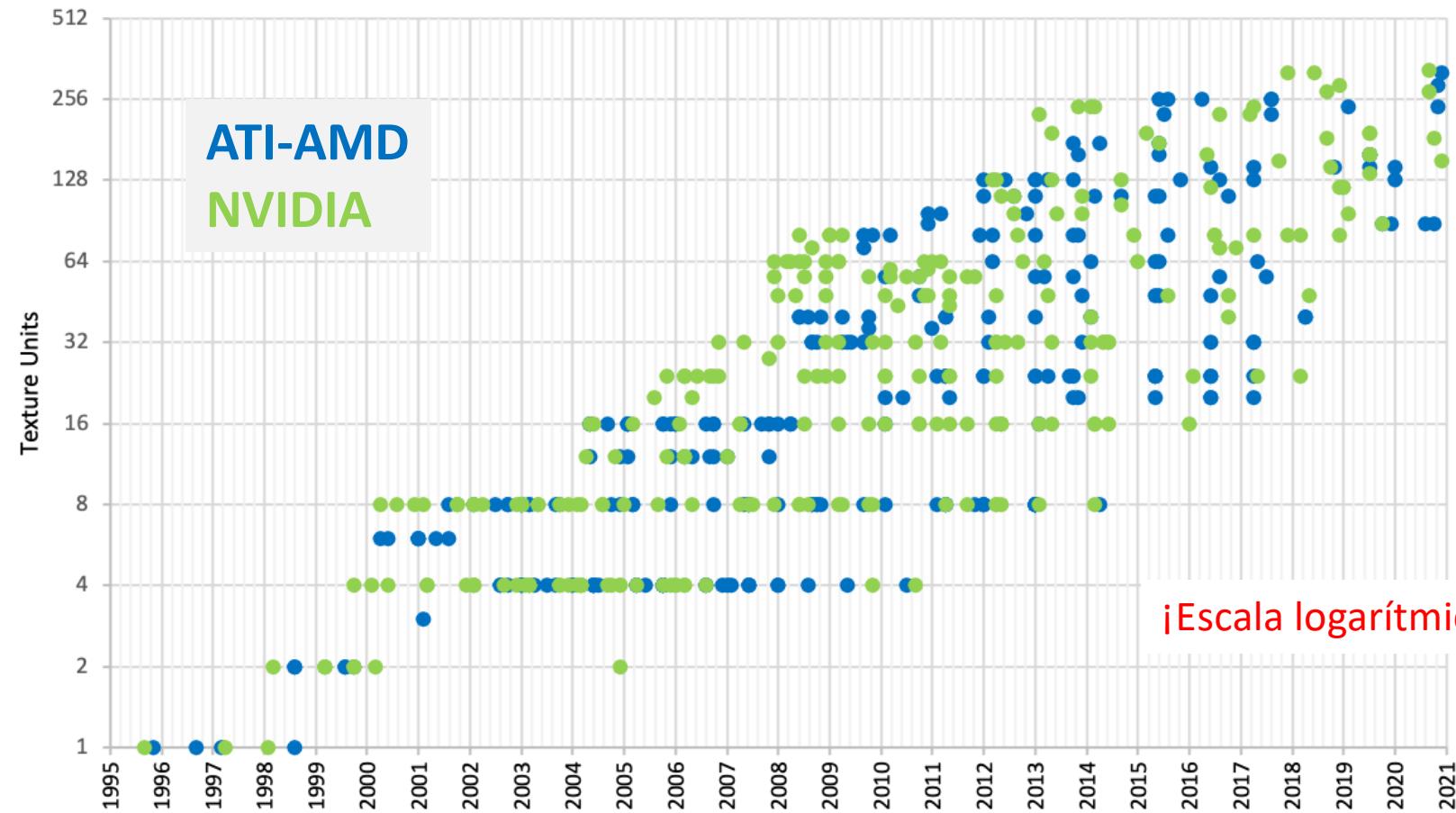
En cualquier época hay GPUs de rendimientos (y consumos) muy diversos. Datos Incompletos.



Shaders



Texture Units



¡Escala logarítmica!

Texture Units: elementos encargados de leer y filtrar texturas. Se usan exclusivamente para aplicaciones gráficas.

Conclusión

El diseño de las modernas GPUs gira en las siguientes ideas:

- Explotar el paralelismo
- Organización coherente
- Ocultar la Latencia con Memoria
- Mezcla inteligente de elementos de función fija con elementos programables

Lectura Complementaria

- David Blythe.
Rise of the Graphics Processor
Proceedings of the IEEE, 96(5):761-778, May 2008.
- Cheng-Hsien Chen and Chen-Yi Lee
“Two-level hierarchical Z-buffer with compression technique for 3D graphics hardware”
The Visual Computer, December 2003



UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Departament d'Arquitectura de Computadors

Tarjetas Gráficas y Aceleradores

El Pipeline Gráfico

Agustín Fernández

Departament d'Arquitectura de Computadors

Facultat d'Informàtica de Barcelona

Universitat Politècnica de Catalunya

