

Tarjetas Gráficas

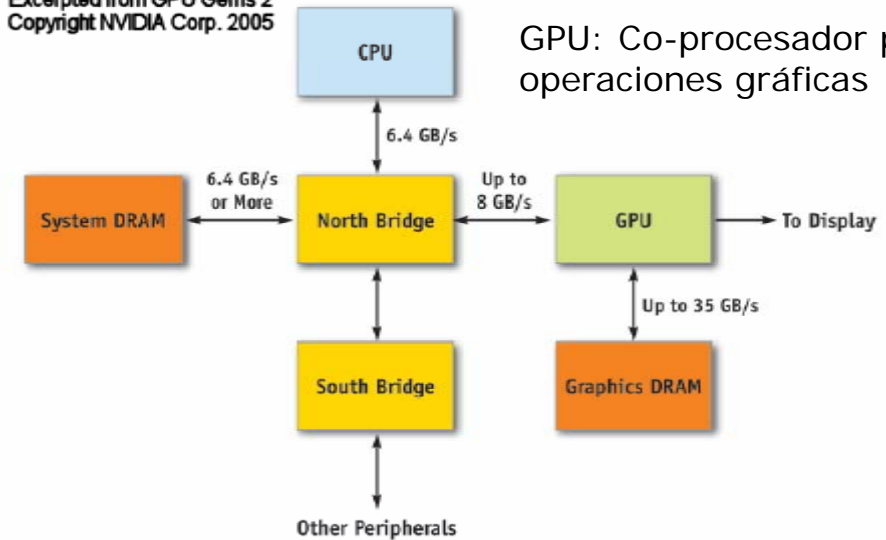
Informática Gráfica

Miguel Ángel Otaduy
26 de Noviembre de 2010



CPU – GPU

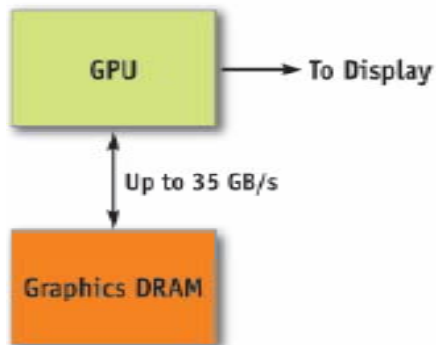
Excerpted from GPU Gems 2
Copyright NVIDIA Corp. 2005



2



CPU – GPU



Pieza clave: memoria dinámica de alta velocidad de acceso*

Almacena geometría y texturas

* NVIDIA 9600GT: 512MB a 57.6GB/s

3



Evolución de las GPUs

- Pipeline asociada a estándares OpenGL
- OpenGL 1.4: Pipeline fija para máxima eficiencia
- OpenGL 2.0: Pipeline programable para mayor versatilidad
- CUDA (NVIDIA): Arquitectura unificada, procesador paralelo general

4



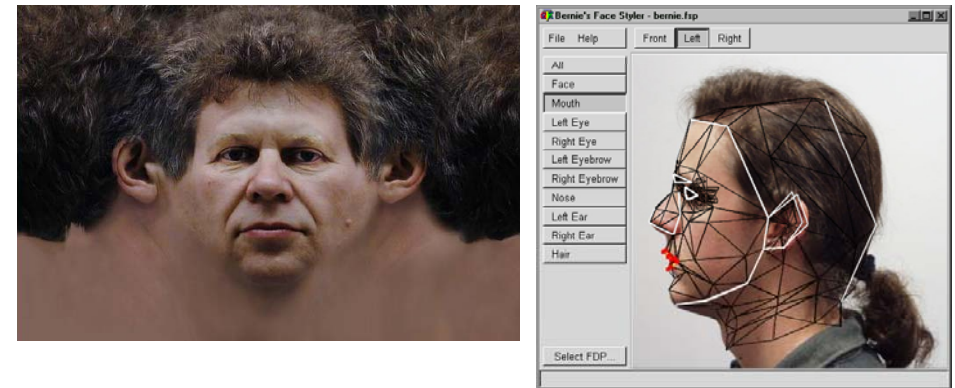
Pipeline Fija



5



Pipeline Fija

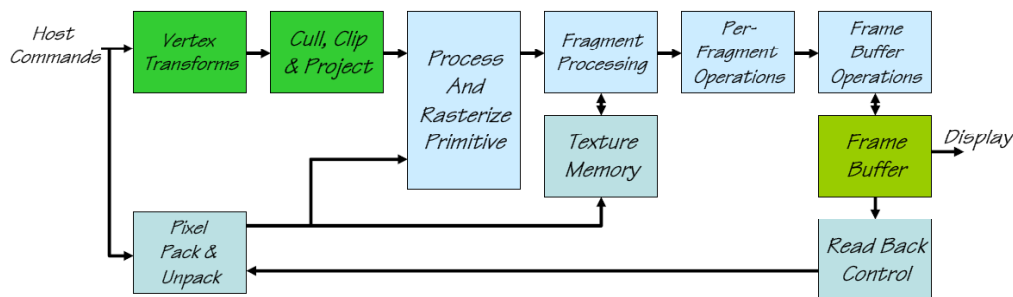


- Trabajo con texturas: impostores, environment mapping, sombras, reflexiones, transparencia...

6



Pipeline Fija



Entradas: primitivas (vértices, polígonos), texturas, comandos

Procesar sólo vértices o fragmentos (pixels). Operaciones fijas.

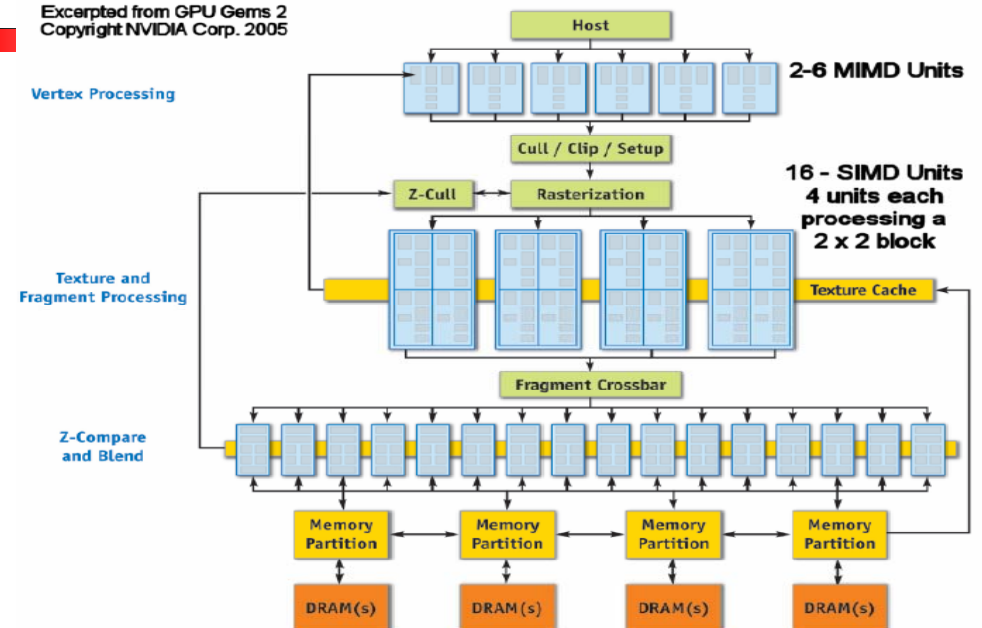
Las operaciones a realizar se determinan mediante el *estado* de OpenGL. El estado se selecciona mediante comandos.

7



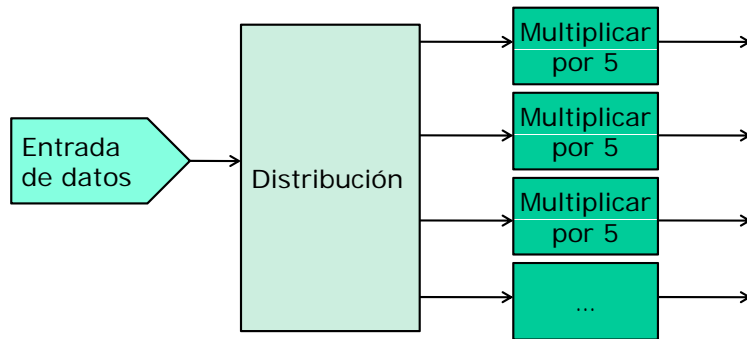
Bloques de la GPU

Excerpted from GPU Gems 2
Copyright NVIDIA Corp. 2005



Stream Processing

- SIMD: Single instruction, multiple data stream.
- MIMD: Multiple instruction...



Pipeline Programable

Precomputing Interactive Dynamic Deformable Scenes

Doug L. James
Kayvon Fatahalian

Carnegie Mellon University



10

Pipeline Programable

Interactive K-D Tree

GPU Raytracing

All images rendered at 640x480

Daniel Reiter Horn Jeremy Sugerman

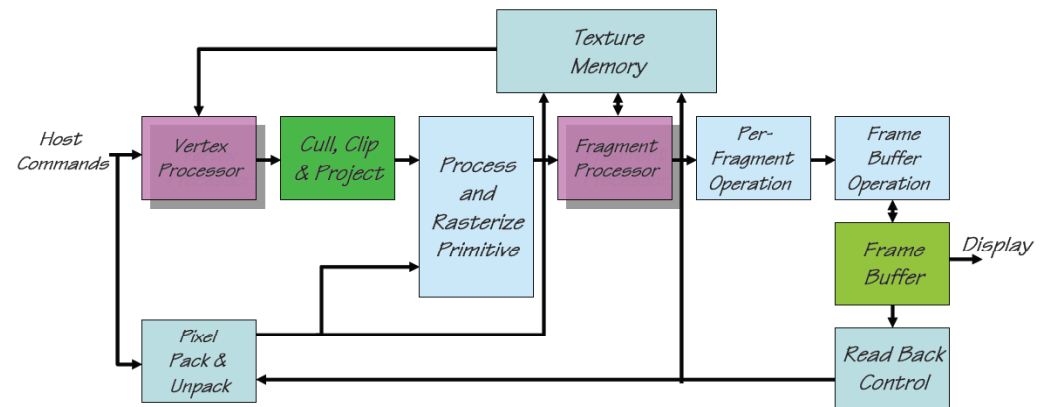
Mike Houston Pat Hanrahan

Stanford University



11

Pipeline Programable



Expone al usuario el procesamiento de vértices y fragmentos.

Memoria de textura general: accesible también en el procesado de vértices, y se puede escribir a ella desde el frame buffer!



12

Pipeline Programable

- El pipeline no cambia básicamente, pero se exponen al usuario el procesado de vértices y fragmentos
- Se pueden modificar las funciones, o se pueden diseñar funciones completamente distintas (incluso para aplicaciones no-gráficas!)

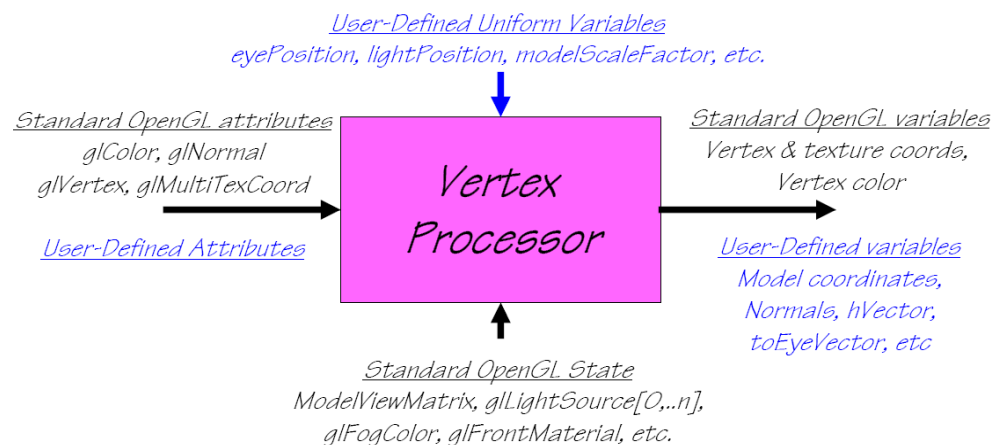


Vertex Processing

- Funciones que se pueden sustituir/modificar:
 - Transformación de coordenadas y normales
 - Normalización, escalado
 - Cálculo de iluminación
 - Aplicación de color
 - Generación y transformación de coordenadas de texturas



Vertex Processing

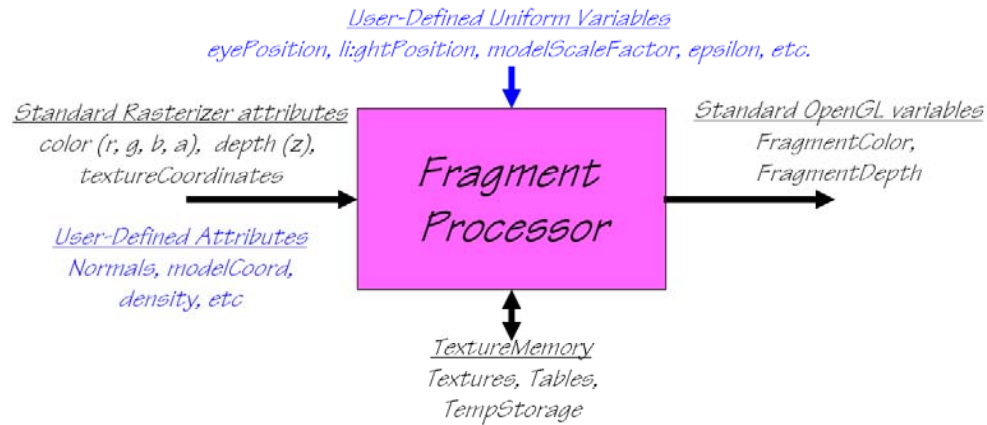


Fragment Processing

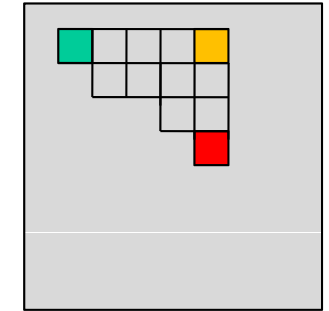
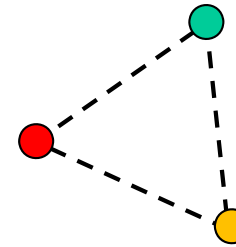
- Funciones que se pueden sustituir/modificar:
 - Acceso y aplicación de texturas
 - Suma y mezcla de colores (blending)
- Funciones que NO se sustituyen:
 - Z-test (profundidad)
 - Posición del fragmento (scan-conversion)
 - ...



Fragment Processing



Limitaciones



No se puede crear geometría (nuevos vértices)

No se puede cambiar la posición de los fragmentos



Multi-Pass Rendering

- Para crear efectos complejos, realizar varios pases del pipeline, y en cada pase una operación distinta
- El resultado de un pase se puede escribir directamente a la memoria de textura (render to texture)
- En un nuevo pase, se leen los datos escritos a las texturas
- Programas: Frame Buffer Object (FBO)



Ejemplo 1: Motion Blur (Efectos de Movimiento)



4 pases



16 pases



Ejemplo 2: Depth of Field (Enfoque de la Cámara)



Sin enfoque

Con enfoque



Ejemplo 3: Soft Lighting (Luces con Área)



Luz puntual

Luz con área



Lenguajes de Shading

- Cg (NVIDIA): se programa y compila por separado, y se carga desde la aplicación
- GLSL (OpenGL): se programa directamente en la aplicación, y se compila *on-the-fly*; se puede modificar!
- HLSL (DirectX): similar pero de Microsoft

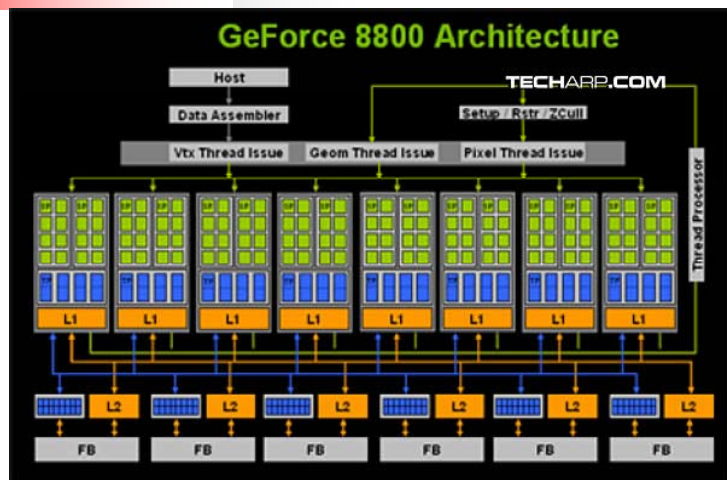


Ejemplo Cg

```
// input vertex
struct VertIn {
    float4 pos : POSITION;
    float4 color : COLOR0;
};
// output vertex
struct VertOut {
    float4 pos : POSITION;
    float4 color : COLOR0;
};
// vertex shader main entry
VertOut main(VertIn IN, uniform float4x4 modelViewProj) {
    VertOut OUT;
    OUT.pos = mul(modelViewProj, IN.pos); // calculate output coords
    OUT.color = IN.color; // copy input color to output
    OUT.color.z = 1.0f; // blue component of color = 1.0f
    return OUT;
}
```



Pipeline Unificada

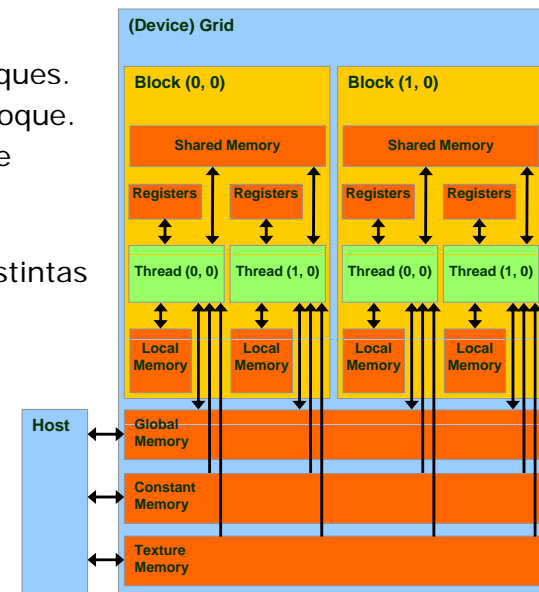


Massive parallel processing: multitud de procesadores ejecutando lo mismo. Desde la CPU, lanzar hebras e indicar cuántas se ejecutan, dónde, y qué memoria utilizan.



Bloques de Hebras

- Hebras distribuidas por bloques.
- Comparten memoria por bloque.
- Identificador de bloque y de hebra.
- 3 tipos de memoria, con distintas propiedades y aplicabilidad



Comunicación CPU-GPU

Creación/destrucción de memoria y transferencia de datos (parecido a C):

- `cudaMalloc((void**) &Md.elements, size);`
- `cudaFree(Md.elements);`
- `cudaMemcpy(Md.elements, M.elements, size, cudaMemcpyHostToDevice);`
- `cudaMemcpy(M.elements, Md.elements, size, cudaMemcpyDeviceToHost);`



Código CUDA

- En general, se programa como en C
- Funciones (indicando quién las llama y dónde se ejecutan):

	Ejecutada en	Llamada por
<code>__device__ float DeviceFunc()</code>	GPU	GPU
<code>__global__ void KernelFunc()</code>	GPU	CPU
<code>__host__ float HostFunc()</code>	CPU	CPU



Ejemplo CUDA



29

Próximo Nivel: Fermi

- Procesamiento MIMD: Multiple Instruction, Multiple Data.
- Una misma GPU puede tener bloques de procesadores dedicados a tareas diferentes en paralelo.
- Fermi es de nVidia. Otros esfuerzos 'libres': OpenCL.

30

