Lecture 4 (10)

A bit more on CUDA

Just a few notes that didn't fit before

Shaders, GLSL and GPGPU

Why it interesting to do GPU computing with graphics APIs today, and how it works

Lecture overview

- Why care about shaders for computing?
- Shaders for graphics
- GLSL
- Computing with shaders

Lecture questions

1) What kind of shaders is most interesting for GPU computing? (What part of the pipeline?)

2) What geometry is usually used for shader-based GPU computing?

3) What is ping-ponging? Suggest an example where it is useful.

Why is classic GPGPU interesting?

- Highly suited to all problems dealing with images, computer vision, image coding etc
- Parallelization ”comes natural”, you can’t avoid it and good speedups are likely. Fewer pitfalls.
- Highly optimized (for graphics performance).
- Compatibility is vastly superior!
- Very much easier to install!

They say classic GPGPU is obsolete?

- Marketing hype!
- Mature technology - publishing opportunities limited.
- Remapping to images awkward for some problems.
- Limited access to local memory
- A lot more messy code than CUDA (but more like OpenCL)

”Any given program, when running, is obsolete”

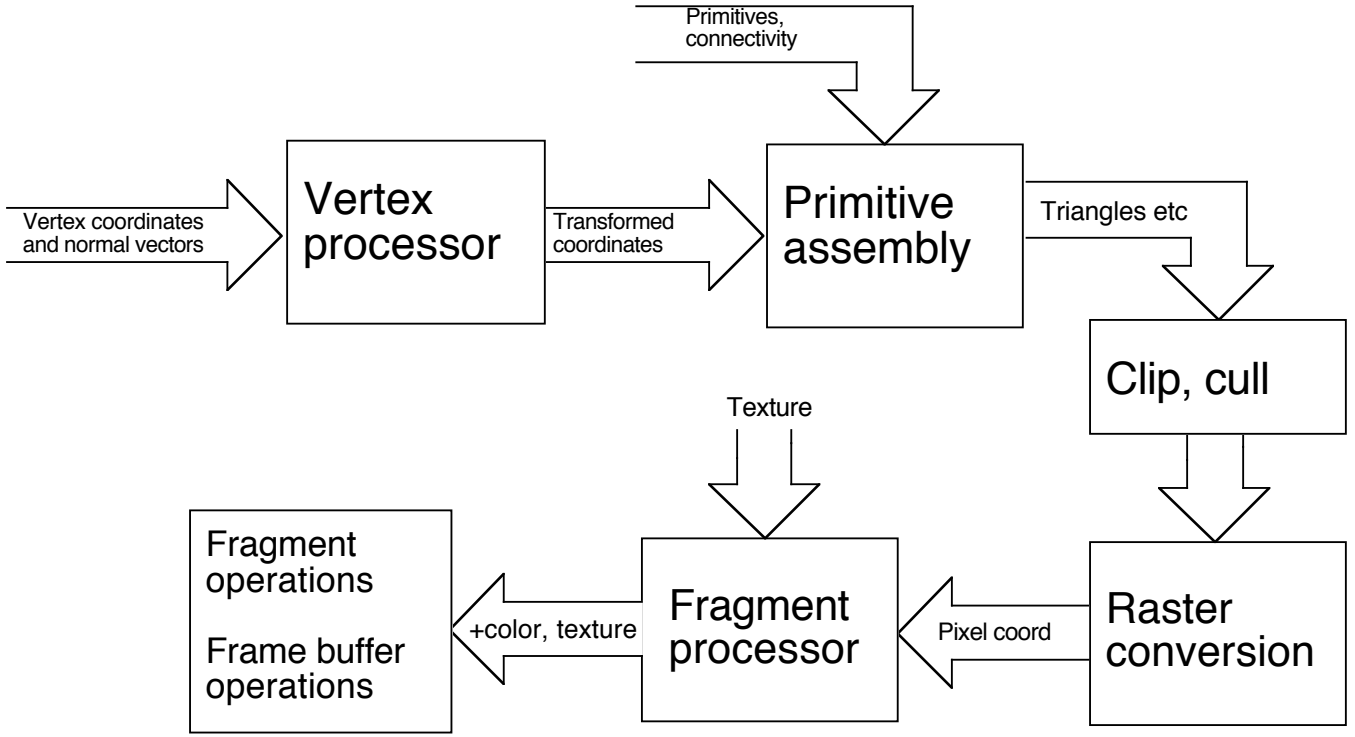
So when should we try a shader solution?

- Any time you must have an installer-free option.
- Any time you deal with images.
- When your CUDA/OpenCL solution has problems, or you need to explore all possibilities for other reasons.

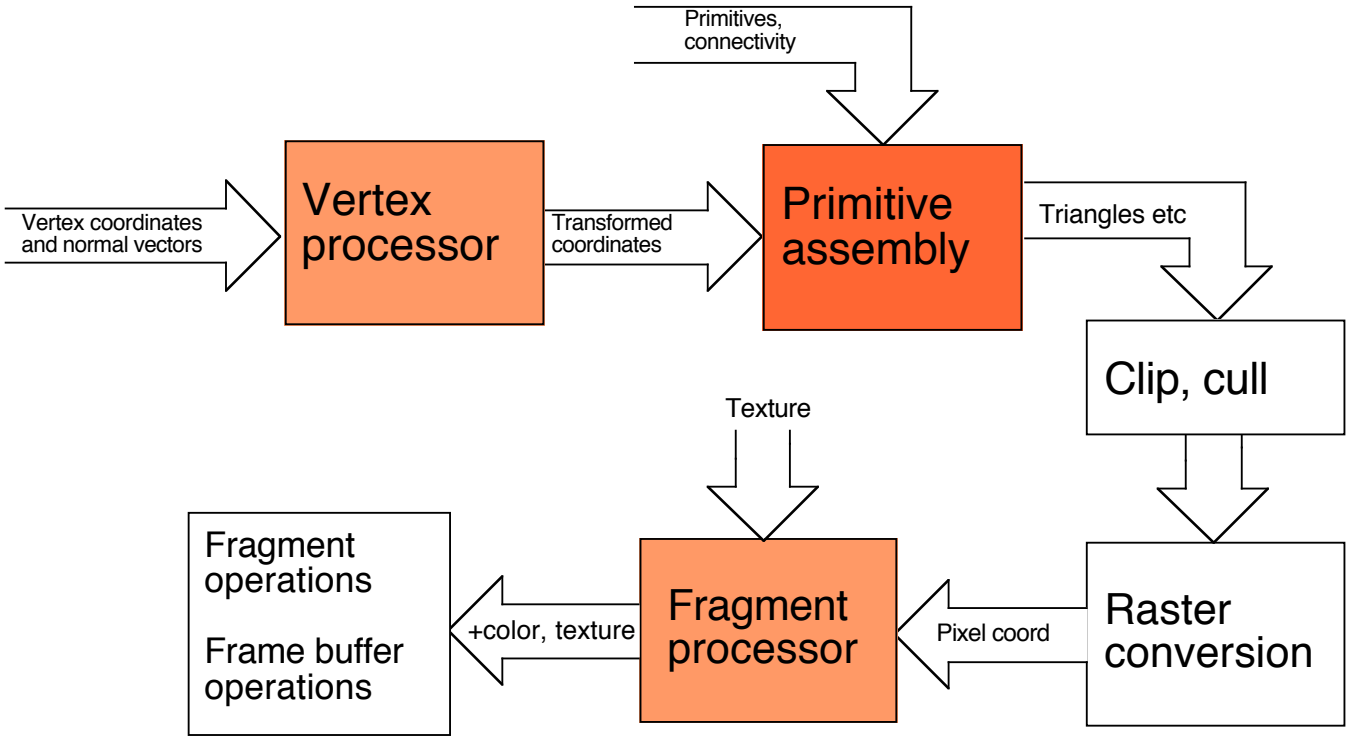
Shaders and GLSL

Let’s have a look at how it works!

The OpenGL pipeline



Out of these, three are programmable!



Shader programs

Program snippets that are executed per vertex or per fragment, on the GPU!

Two programs cooperate, one vertex program and one fragment program.

“Shader” implies that the goal is lighting, but that is only one of the goals!.

Vertex transform
Vertexcolor, vertex-level lighting

Can be done in a vertex shader

Texturing
Color and light per pixel

Can be done in a fragment shader

Vertex shader

Replaces the fixed functionality of the vertex processor.

It can:
- transform vertices, normals and texture coordinates
- generate texture coordinates
- calculate lighting per vertex
- set values for interpolation for use in a fragment shader

It knows nothing about:
- Perspective, viewport
- Frustum
- Primitives (!)
- Culling

Fragment shader
(a.k.a pixel shader)

Replaces the fixed functionality of the fragment processor.

It can:
- **set the fragment color**
- **get color values from textures**
- **calculate fog and other color calculations**
- **use any kind of interpolated data from the vertices**

It can not
- **change the fragment coordinates**
- **write into textures**
- **affect stencil, scissor, alpha, depth...**

Shader languages

Four different:

Assembly language: Old solution, no longer updated. Dead! But used in old GPGPU research.

Cg: “C for graphics”, NVidia
HLSL: “High-level shading language”, Microsoft
GLSL: “OpenGL shading language”

Choice depends on flatform and needs (and taste).
GLSL superior for compatibility!

Typical shader examples in graphics

Vertex manipulation (deformations)

Lighting calculations

Multitexturing

Bump mapping