Programmable Graphics Hardware

Johan S. Seland

Center of Mathematics for Applications
University of Oslo

SINTEF-CMA GPU Course

12. December 2006

Offline rendering five years ago



Shrek (Dreamworks/PGI)

Interactive Rendering five years ago



Quake 3 (id Software)

Offline Rendering today



Madagascar (Dreamworks/PGI)

Interactive Rendering today



Project Gotham Racing

Enter the GPU

We all know this is because of the Graphical Processing Unit

- Fall 2001: Intel Pentium IV 2.0 GhZ
- Fall 2006: Intel Core 2 Extreme XQ6700 (4×2.67GhZ)
  - 582 million transistors
- Fall 2001: Nvidia GeForce 3 (200 MhZ, 4/2/1 Pipes/TMU/VPU)
- Fall 2006: Nvidia Geforce 8 (1.20 GhZ, 128/32/24 SPU/TMU/ROP)
  - 680 million transistors



Motivation

Logical graphics pipeline

Shading languages

Conclusion

Goals for this talk

Included:

- Have an overview of the logical graphics pipeline
- Shading languages

Not included:

- API details
- Fancy rendering techniques

Graphics pipeline

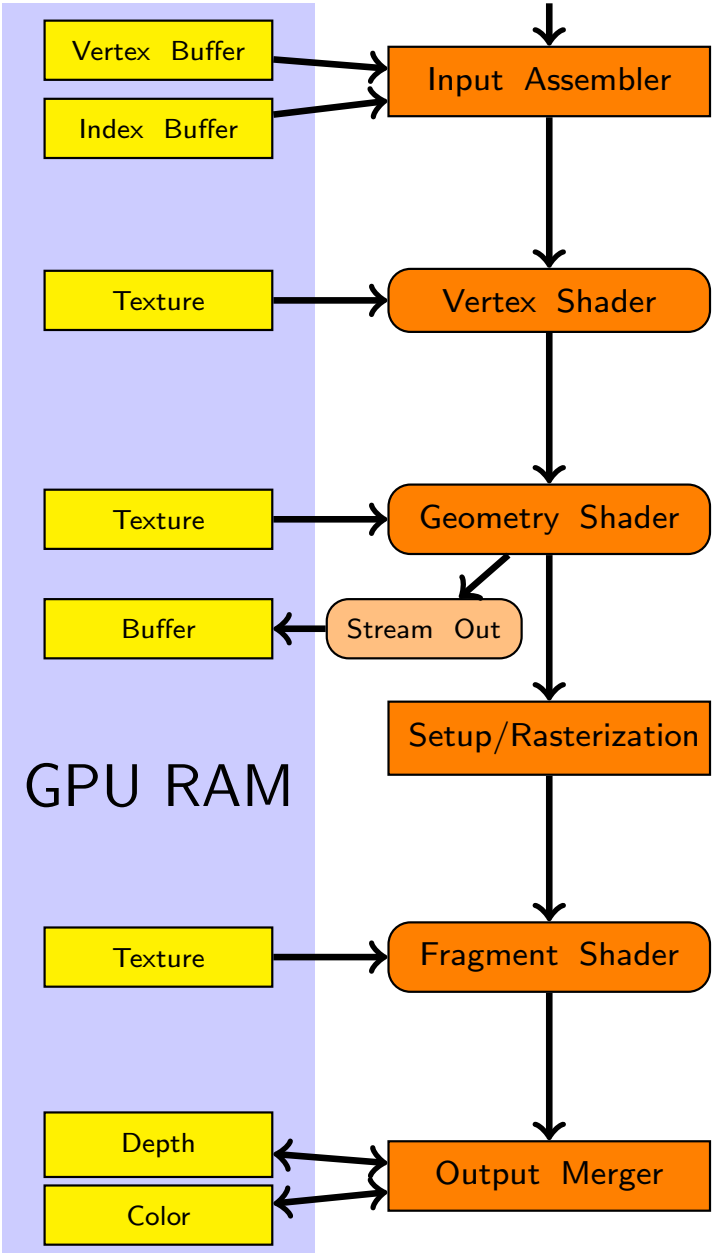
- GPUs have traditionally been programmed through graphics APIs
  - OpenGL – Big reorganizations for 2007
    - Two new versions in 2007!
  - DirectX (Direct3D) – New major release with Windows Vista
- Graphics APIs were designed to match the underlying hardware
- Graphics driver is like an OS for the GPU
- APIs and hardware constantly evolving
- Nvidia 8800 breaks with "HW *is* graphical Pipeline"
  - Still a streaming architecture
- We now have a logical graphics pipeline, for graphics
  - 2007: New programming models for computations?

Graphics driver

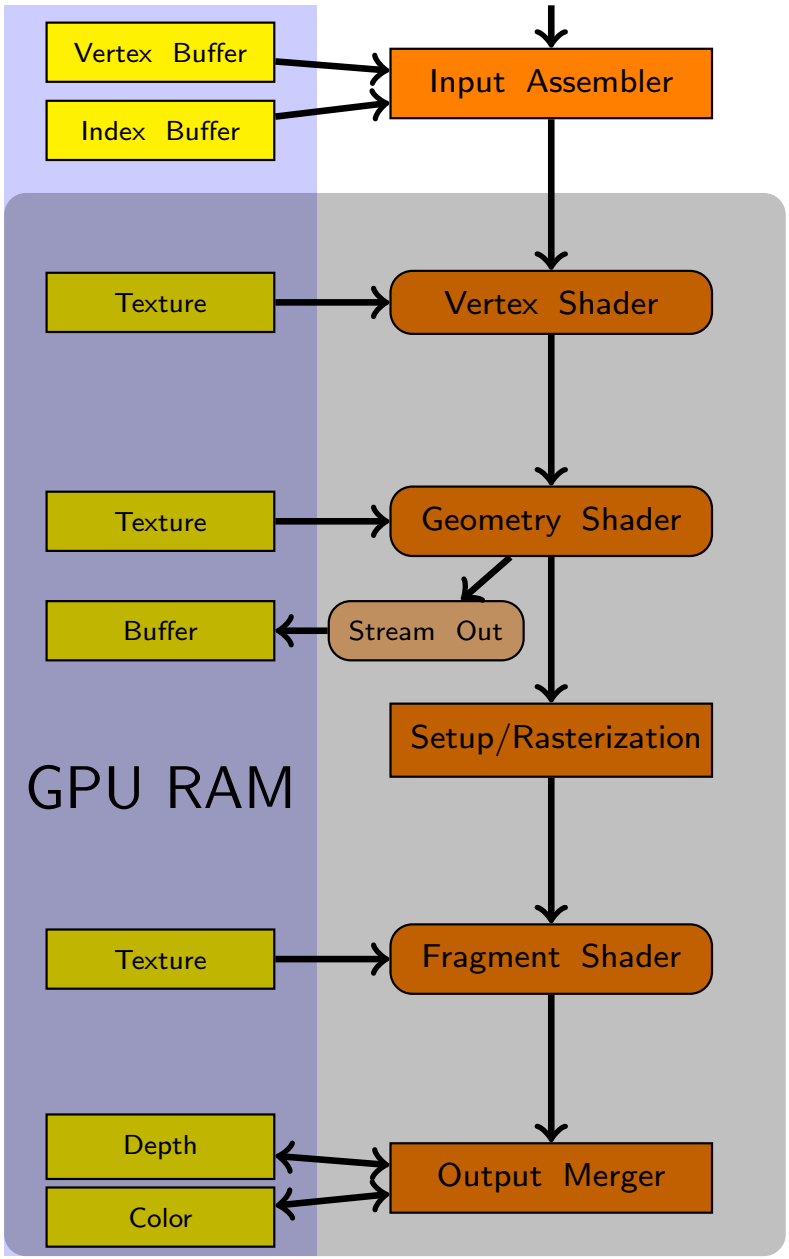
- Graphics drivers are the key to using the GPU effectively
  - Driver is like an OS
    - Thread scheduling
    - Protected memory
    - JIT compilation
  - Details they abstract are increasingly important for developers to understand
  - Hides the actual performance characteristics of the hardware
  - High-performance computing need a more direct understanding of memory performance
  - Drivers are optimized for games

Computer graphics 101

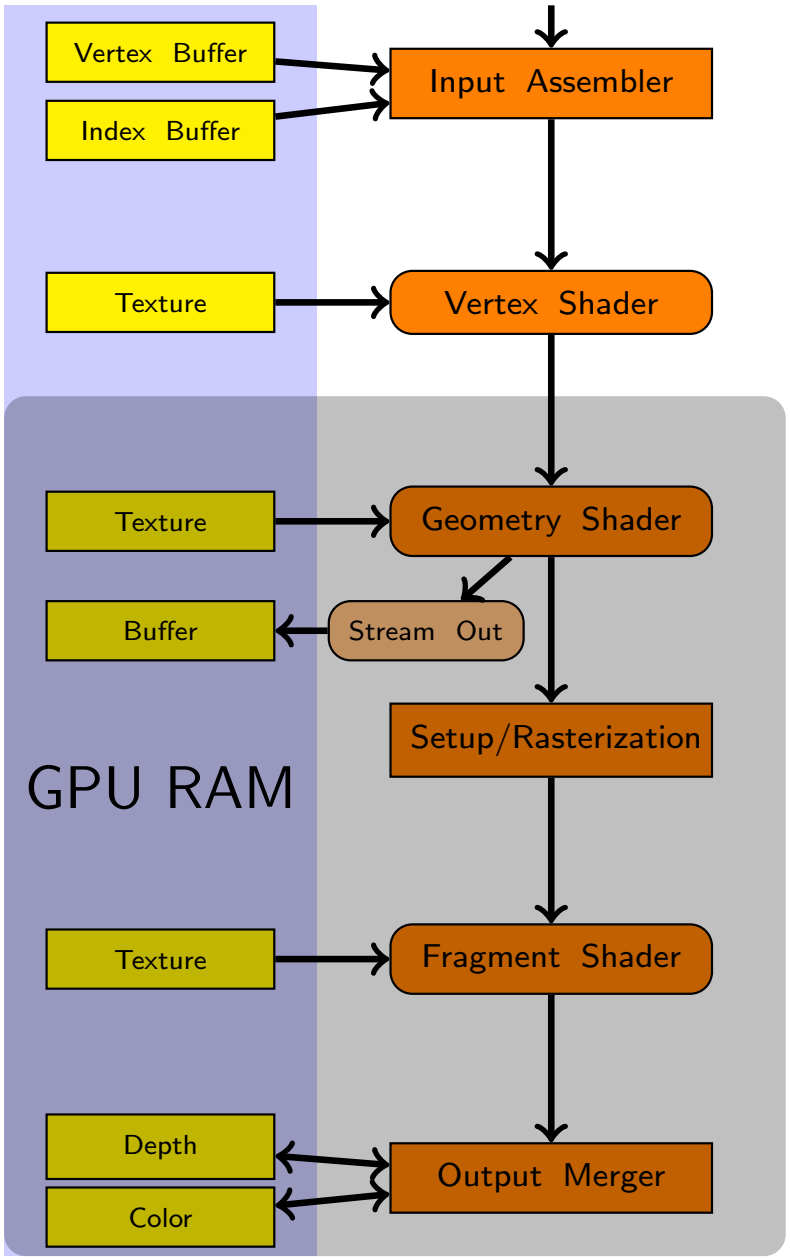
1. Models are stored/generated as a set of triangles
2. Shaders (computational kernels) are stored as C-like source code
3. CPU sends shader source code to GPU-driver, which compiles it to the underlying hardware
4. CPU sends geometry to GPU, together with camera position, light sources etc.
5. GPU projects triangles to screen
6. GPU rasterizes projected triangles
7. GPU calculates the final pixel color
8. GPU displays pixel colors to screen



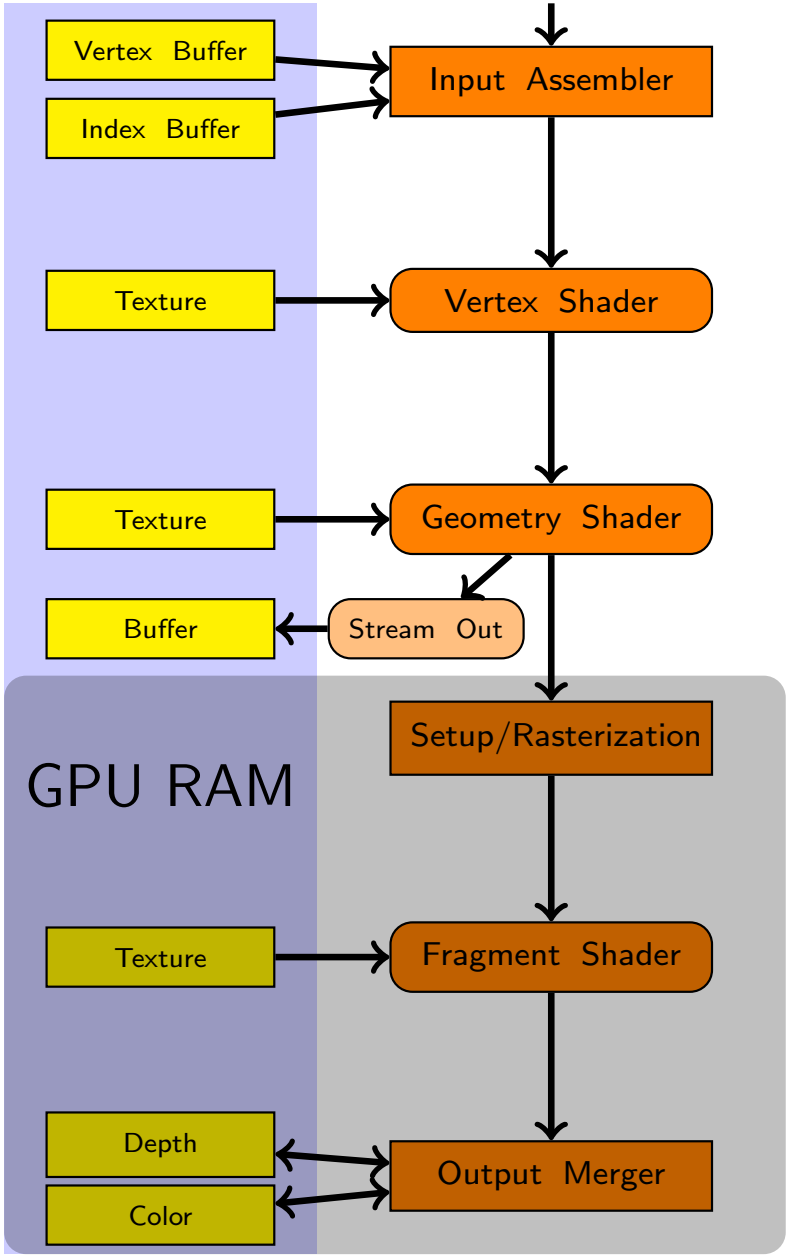
- Logical pipeline
  - Programmers View
  - Not the way the hardware works anymore



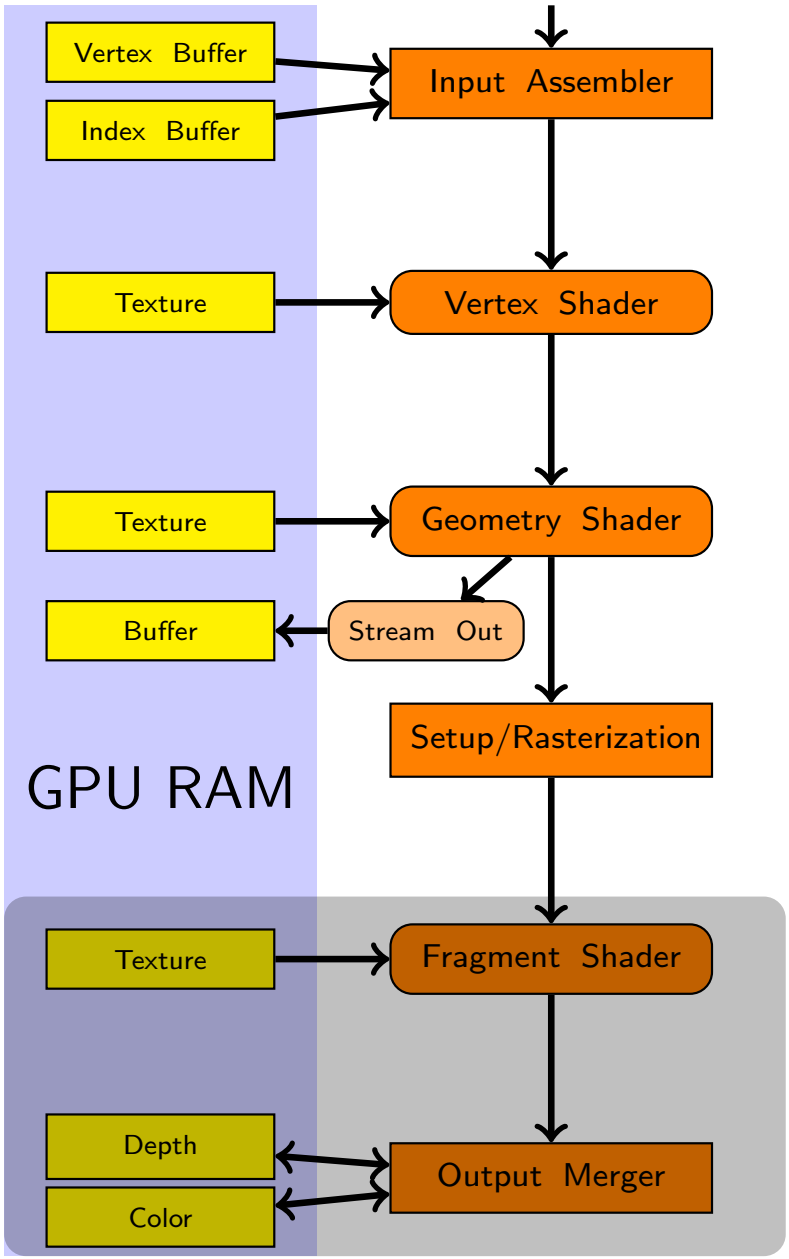
- Input Assembler
  - Fixed-function
  - Canonicalize vertex data
  - Generate IDs
    - Primitive
    - Vertex
    - Instance



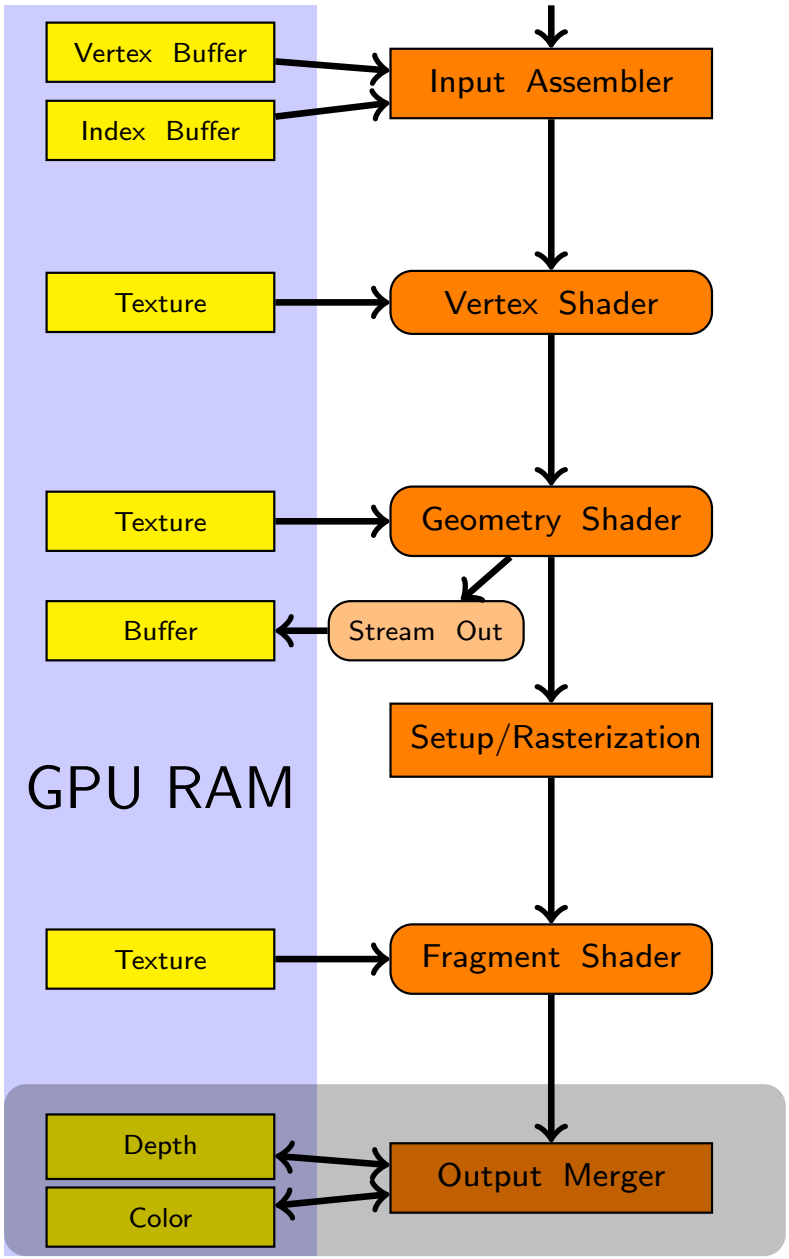
- Vertex Shader
  - Programmable
  - Per vertex processing (same shader for all)
  - No knowledge of the type of primitive
  - No knowledge of neighboring vertices
  - Can change a vertex (x, y, z, w) position
  - 1 primitive in, 1 primitive out
  - Read from memory
- Usage
  - Transform vertex into eye coordinates
  - Calculate values to be linearly interpolated across primitive (like texture coordinates)
  - Move vertex position



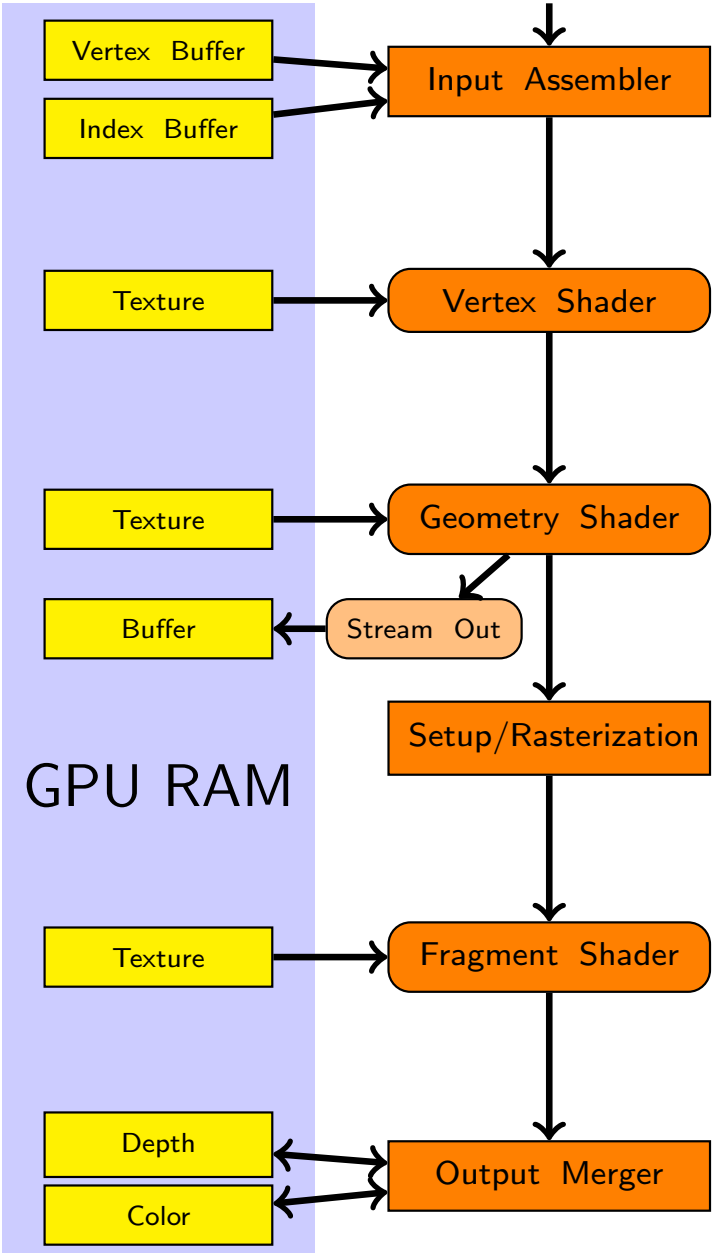
- Geometry Shader
  - New for 2007
  - Programmable
  - Per primitive processing
  - 1 primitive in, n primitives out
  - Read from memory
  - Pipeline may terminate here
- Usage:
  - Adjacency computations
  - Kill primitive?
  - Quadratic interpolation?



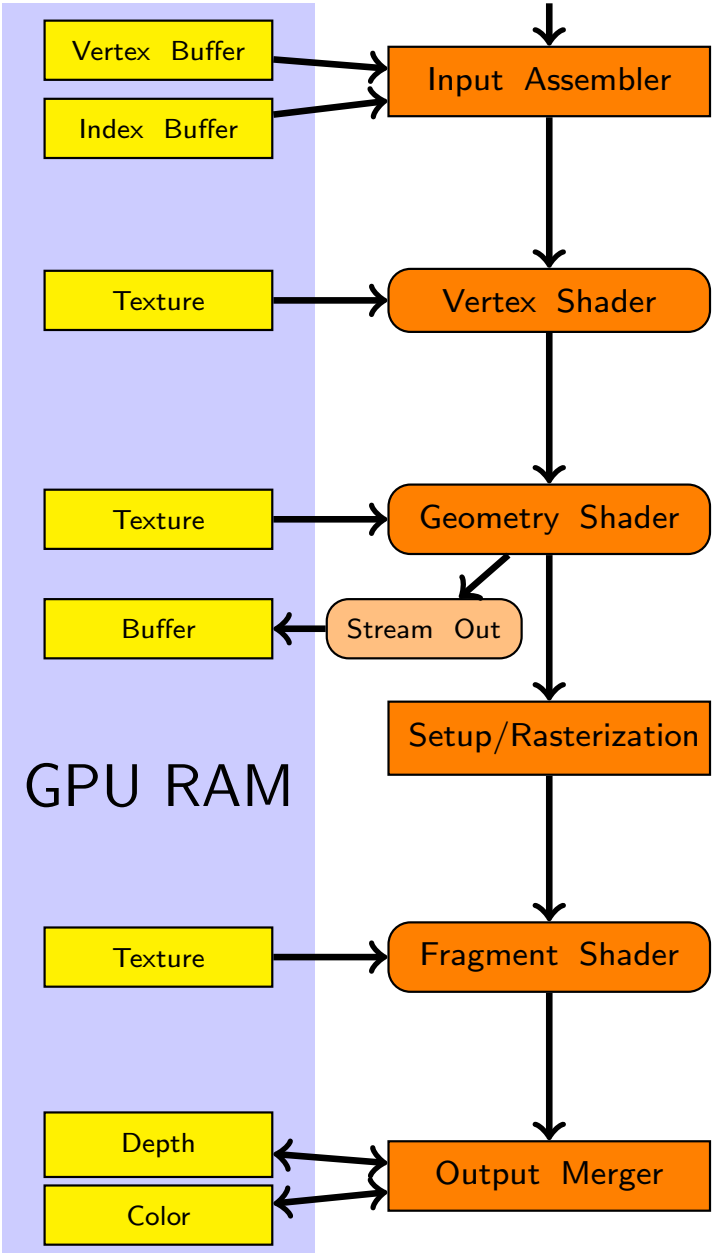
- Setup/Rasterization
  - Fixed-function
  - Clipping, perspective transform
  - Convert primitives into fragments



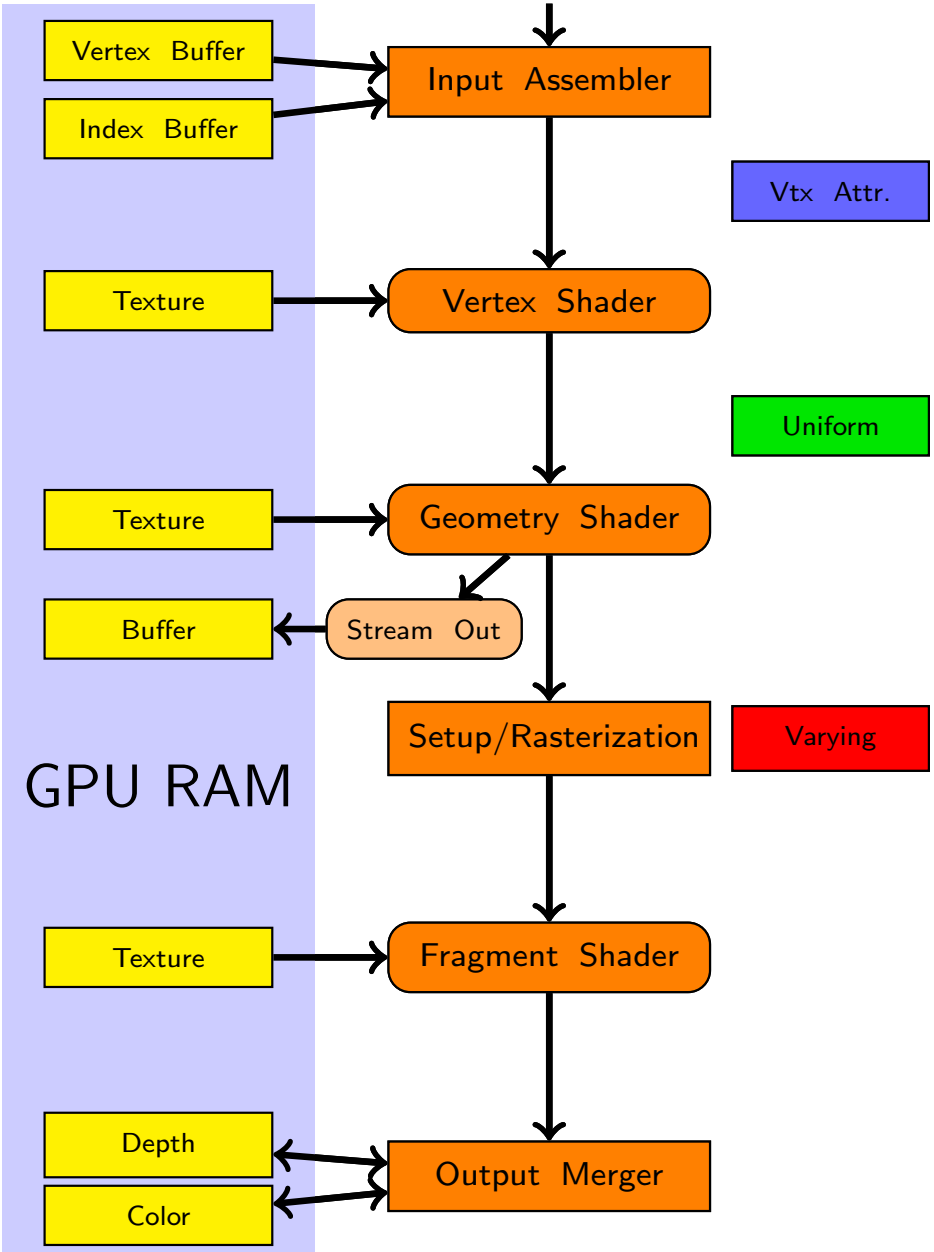
- Fragment Shader
  - Programmable
  - Execute same shader for all fragments
  - Has no knowledge of neighboring fragments
  - Can not change the (x, y) coordinate of a fragment
  - Can change a fragments (r, g, b, α) value
  - 1 fragment in, 0 or 1 fragment out
  - Up to 33 scalar values of output per pixel
    - $8\times$rgbα + depth
  - Read from memory



- Output Merger
  - Fixed-function
  - Depth/Stencil test
  - Color buffer blending
  - Read/write/modify memory

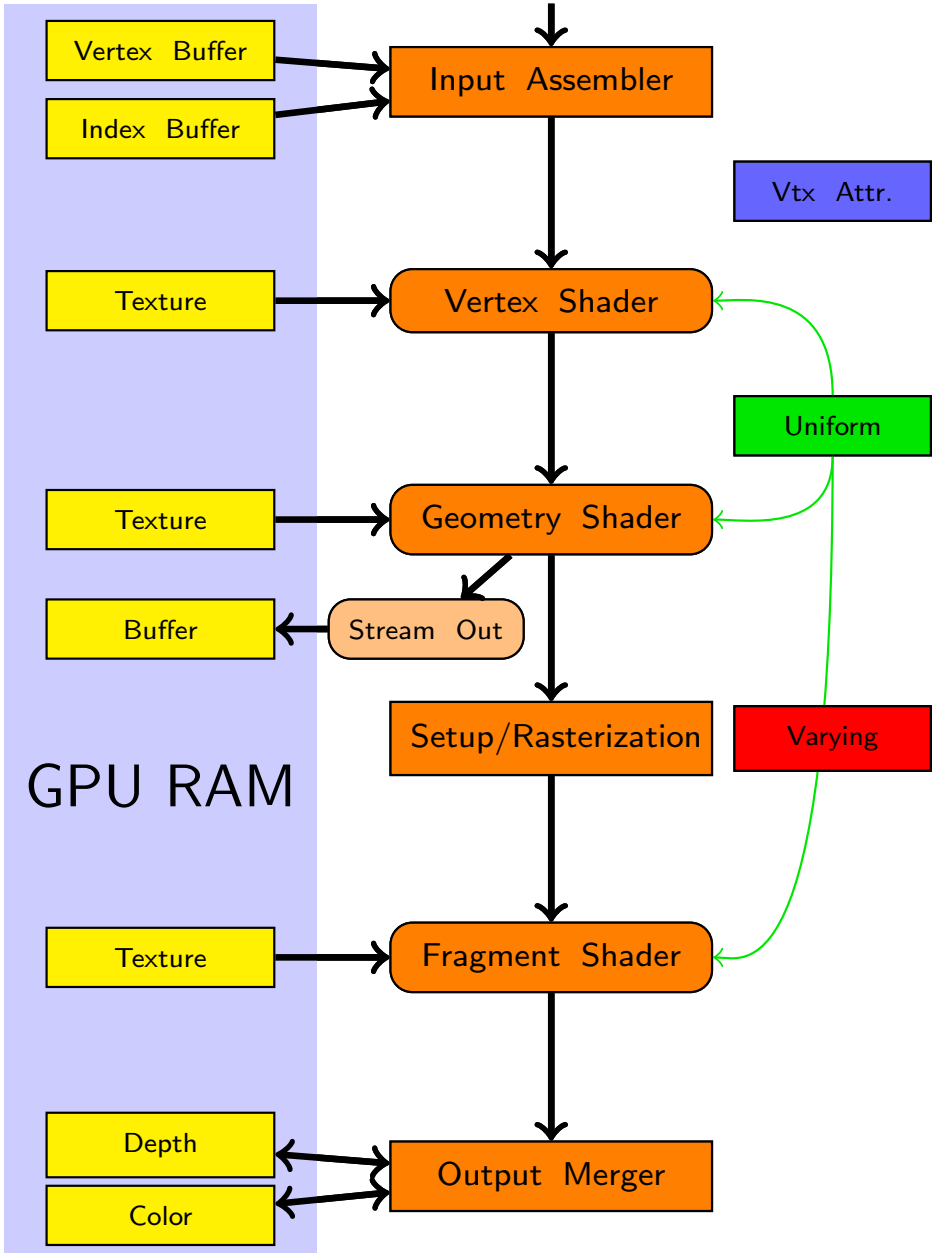


- Common programmable "core"
- Flexible memory objects
- Array forms of memory objects
  - Indices generated in shaders



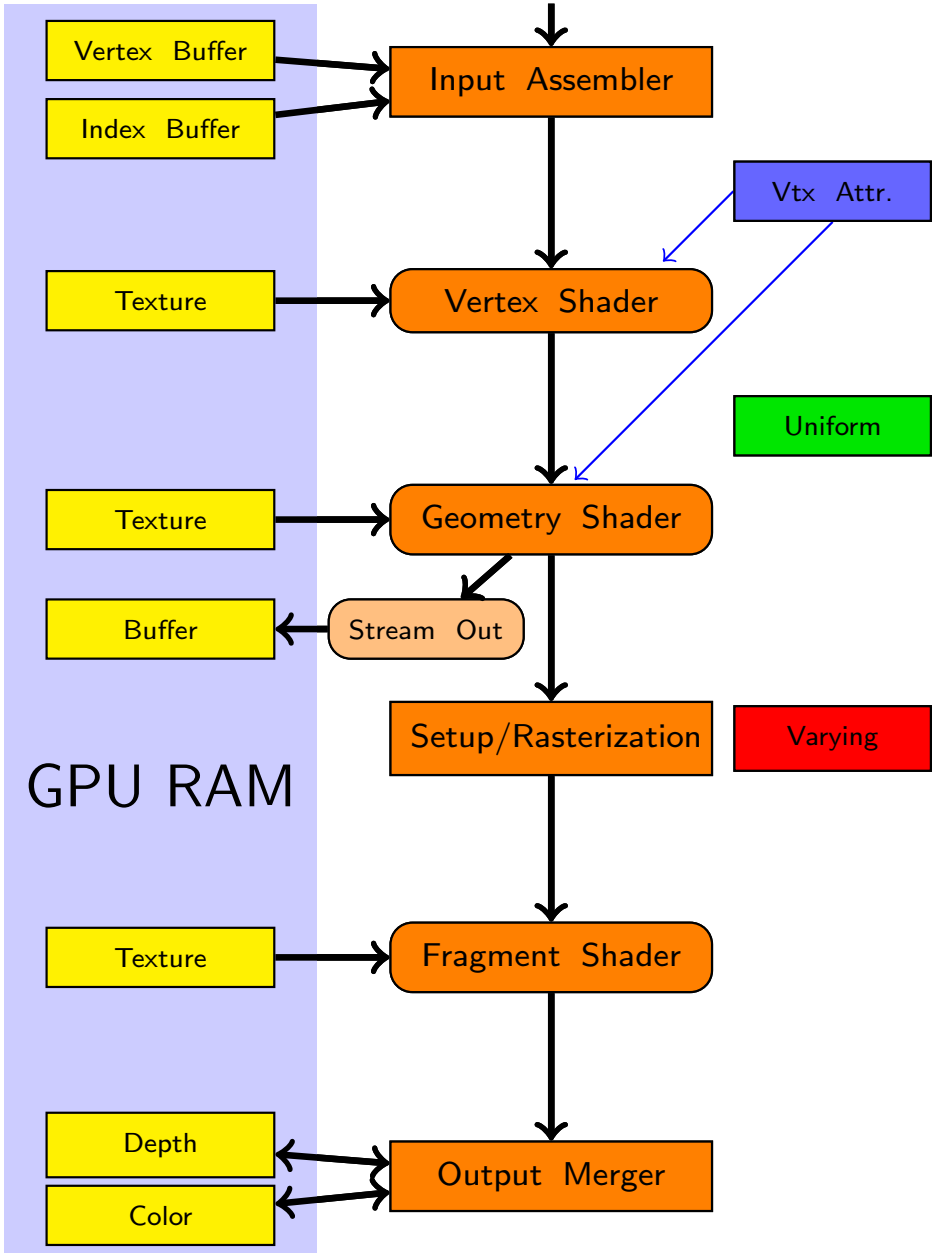
Communication in the pipeline

- Uniform variables are set by the CPU and are accessible in all shaders
- Vertex attributes are encoded in the geometry stream and accessible in {vertex,geometry} shaders
- Varying variables are set by {vertex,geometry}-shaders, interpolated across primitive and read in fragment shader.



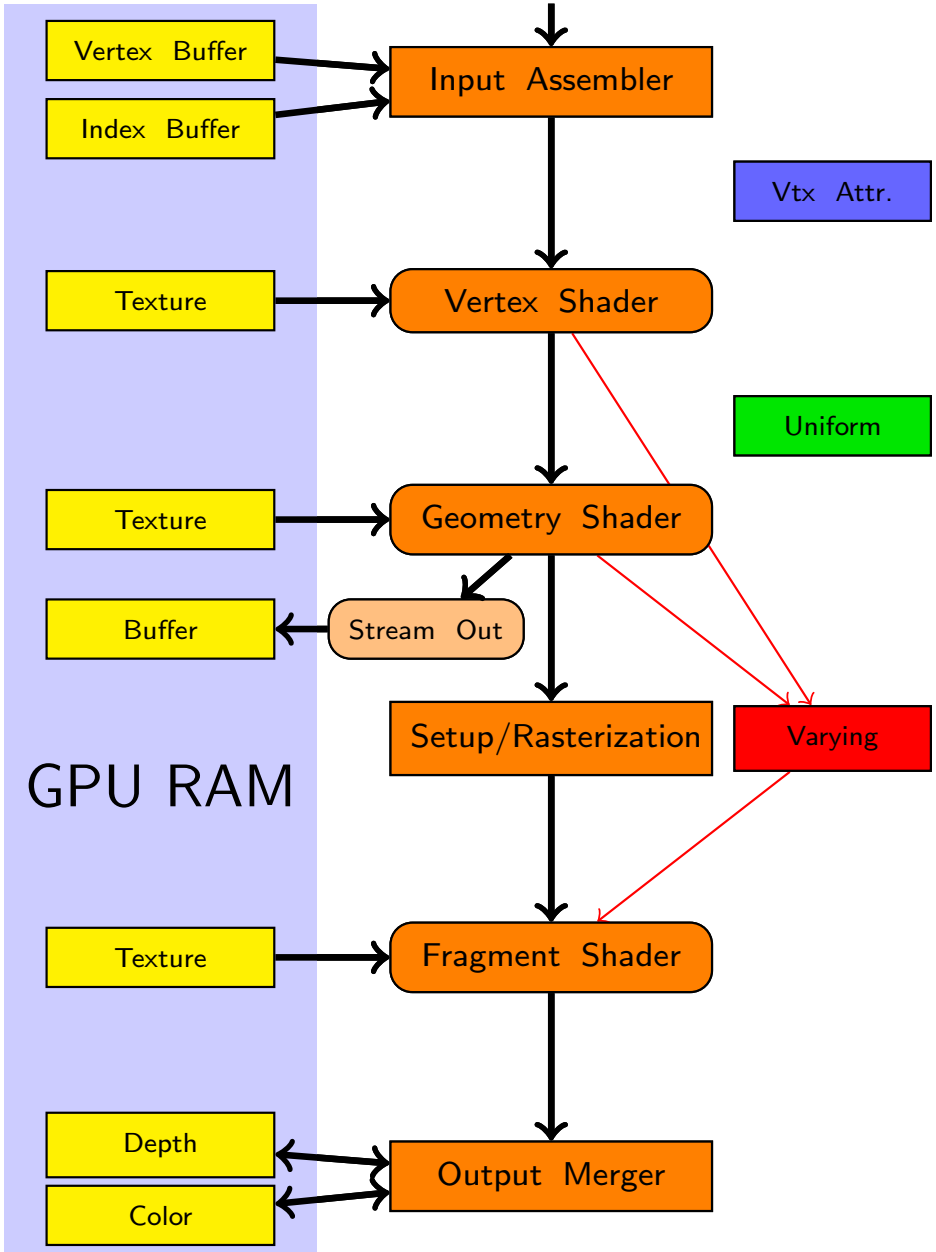
Communication in the pipeline

- Uniform variables are set by the CPU and are accessible in all shaders
- Vertex attributes are encoded in the geometry stream and accessible in {vertex,geometry} shaders
- Varying variables are set by {vertex,geometry}-shaders, interpolated across primitive and read in fragment shader.



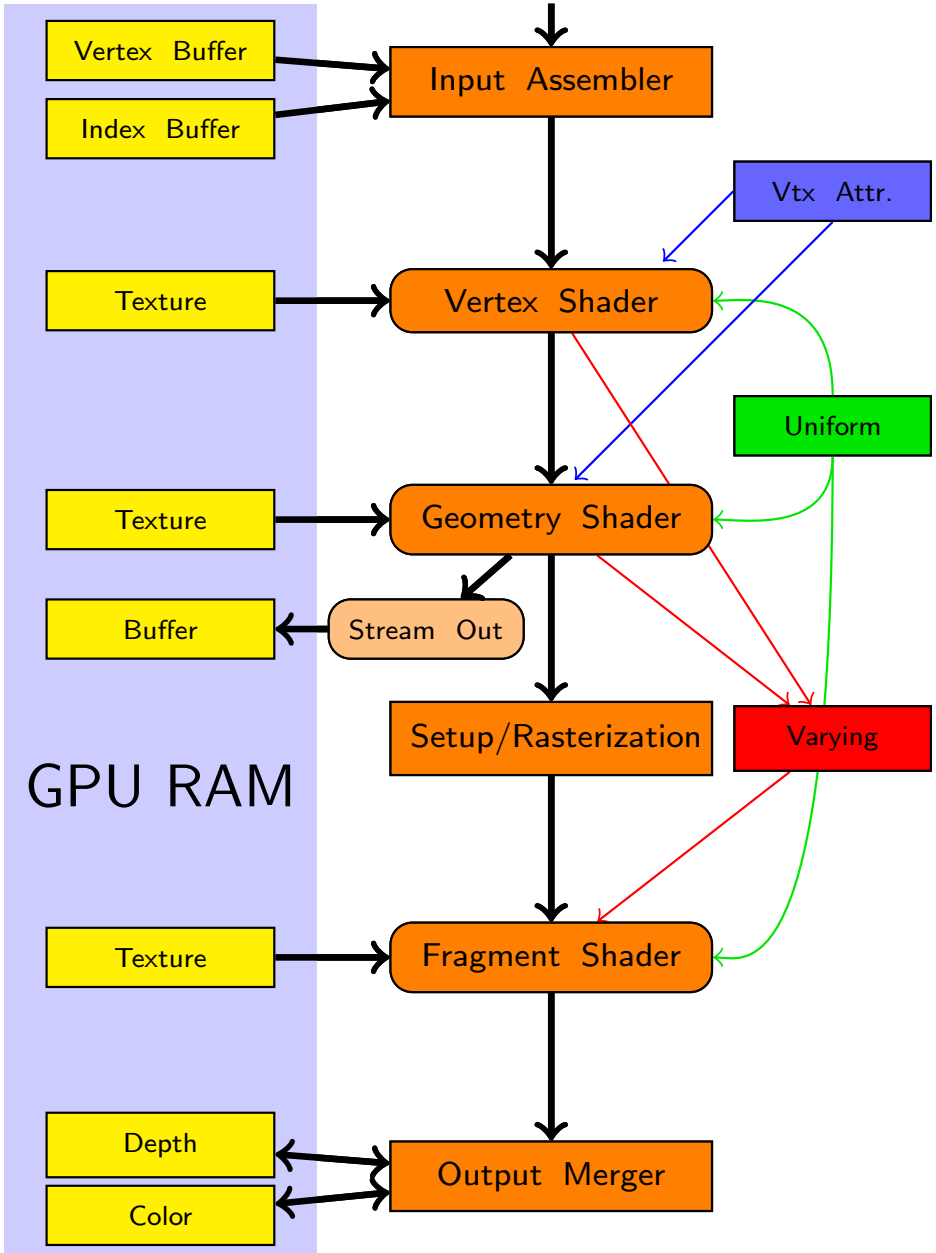
Communication in the pipeline

- Uniform variables are set by the CPU and are accessible in all shaders
- Vertex attributes are encoded in the geometry stream and accessible in {vertex,geometry} shaders
- Varying variables are set by {vertex,geometry}-shaders, interpolated across primitive and read in fragment shader.



Communication in the pipeline

- Uniform variables are set by the CPU and are accessible in all shaders
- Vertex attributes are encoded in the geometry stream and accessible in {vertex,geometry} shaders
- Varying variables are set by {vertex,geometry}-shaders, interpolated across primitive and read in fragment shader.



Communication in the pipeline

- Uniform variables are set by the CPU and are accessible in all shaders
- Vertex attributes are encoded in the geometry stream and accessible in {vertex,geometry} shaders
- Varying variables are set by {vertex,geometry}-shaders, interpolated across primitive and read in fragment shader.

Geometry Shader

- Primitive as input
  - Adjacency optional
  - New OpenGL primitive types
    - `GL_LINES_ADJACENCY_EXT`
    - `GL_LINES_STRIP_ADJACENCY_EXT`
    - `GL_TRIANGLES_ADJACENCY_EXT`
    - `GL_TRIANGLES_STRIP_ADJACENCY_EXT`
- Outputs zero or more primitives
  - 1024 scalars out max
- Output primitive(s) need not be connected

Motivation

Logical graphics pipeline

Shading languages

Conclusion

Shading Languages

We can program the vertex, geometry and fragment processors by using Shading Languages

Offline Rendering (Production Shaders)

- Shading Languages initially arose in Pixar's RenderMan
- Designed to allow for extremely high-quality images and movies
- No demand for production shaders to be executed in real time
- Current real-time shading languages are inspired by Renderman
- Shader are executed on CPU

Real Time Rendering

- Arose since fixed function pipeline was not flexible enough
- Strict real-time requirement (games run at 40-100FPS)
- Shaders are executed on GPU

Real-Time Shading Languages

Different shading languages exist for consumer hardware:

1. Nvidia Cg (OpenGL/DirectX/WGF)
2. Microsoft HLSL (DirectX/WGF)
3. OpenGL Shading Language (OpenGL)

They all provide (almost) the same functionality.

The OpenGL Shading Language

- Based on C/C++
- Shader is stored as source code
- Compile - Link - Run cycle
- Compiler is embedded in driver
- Hardware independent

GLSL at a glance

- C/C++-like syntax

Examples

- looping `for`, `while`, `do-while`
- selection `if`, `if-else`
- basic datatypes `bool`, `int`, `float`, `struct`, arrays
- functions (call by value)
- no pointers $\longrightarrow$ no bitwise operators
- no strings
- no type promotion `float f = 0.0;`

GLSL at a glance

- C/C++-like syntax
- Built-in Vector (2d, 3d, 4d)

Examples

- `vec4 v = vec4(1.0, 0.0, -3.0, 2.0 );`
- Swizzling `vec3 w = v.yzx; // (0.0, -3.0, 1.0)`
- Indexing
  `float f = v[2]; // -3.0`
- Component-wise operation
  `w.x = v.x + f;`
- Component-wise mathops
  `vec4 v = v+w;`

GLSL at a glance

- C/C++-like syntax
- Built-in Vector (2d, 3d, 4d)
- Built-in Matrix ($2 \times 2, 3 \times 3, 4 \times 4$)

Examples

- `mat2 m = mat2(1.0, 2.0, 3.0, 4.0 ); //` $\begin{bmatrix} 1.0 & 3.0 \\ 2.0 & 4.0 \end{bmatrix}$
  - `mat4 n = mat4(1.0);`
    `// n = diag(1.0)`
  - `vec2 v = m[1];`
    `//` $v = (1.0, 2.0)^T$
  - `float f = m[2][0];`

GLSL at a glance

- C/C++-like syntax
- Built-in Vector (2d, 3d, 4d)
- Built-in Matrix ($2 \times 2, 3 \times 3, 4 \times 4$)
- Built-in standard library of functions

Examples

- Trigonometric `sin, tan, atan...`
- Common `abs, sign, min, max...`
- Geometric `length, dot, cross...`
- Vector relational

GLSL at a glance

- C/C++-like syntax
- Built-in Vector (2d, 3d, 4d)
- Built-in Matrix $(2 \times 2, 3 \times 3, 4 \times 4)$
- Built-in standard library of functions
- Variable qualifiers

Examples

- `const` qualifier
- `attribute` qualifier
- `uniform` qualifier
- `varying` qualifier

GLSL at a glance

- C/C++-like syntax
- Built-in Vector (2d, 3d, 4d)
- Built-in Matrix $(2 \times 2, 3 \times 3, 4 \times 4)$
- Built-in standard library of functions
- Variable qualifiers
- Built in variables

Examples

- Uniforms: gl_ModelViewMatrix
- Attributes gl_color, gl_vertex, gl_normal
- Varyings: gl_FrontColor, gl_BackColor
- Special: gl_Position, gl_FragDepth, gl_FragColor

GLSL at a glance

- C/C++-like syntax
- Built-in Vector (2d, 3d, 4d)
- Built-in Matrix $(2 \times 2, 3 \times 3, 4 \times 4)$
- Built-in standard library of functions
- Variable qualifiers
- Built in variables
- GS special functions

Examples

- `EmitVertex()`
- `EmitPrimitive()`

Motivation

Logical graphics pipeline

Shading languages

Conclusion

Programming model

In spite of emerging tools:

> *GPUs are the only type of parallel processor that has ever seen widespread success – (Matt Pharr)*

- . . . because developers generally don't know they are parallel!
  - Pipeline approach might actually be useful for more than just graphics
- Parallel programming is notoriously difficult
  - Heterogeneous processors don't make it easier

Next five years: A second revolution

- Continued GPU innovation
- CPUs finally providing FLOPS as well
  - Don't require as much parellelism
- Shared memory/high bandwidth interconnect enable flexible computation model
- Future role of todays graphics APIs unclear

Thats all folks

Questions?

Thanks to the following:

- Matt Pharr – Many thoughts are stolen from his GH06 talk
- David Luebke – Our inside man at Nvidia
- Mike Mantor – Our inside man at AMD
- Christopher Dyken – For always answering questions