Chapter 10

GPU, Shaders, and Shading Languages

Programmable Graphics Hardware: History

- 1984 : Cook's Shade trees
  - RenderMan Shading Language in late 80's
- 1980 ~ 1998 : (Hardware) Pixel-Planes/PixelFlow at UNC
- 1992 : OpenGL 1.1 standard
- 1998 : Multi-texturing
- 1999 : NVIDIA's GeForce256 -
- 1999 : Register Combiners
- 1999 : Stanford's Real-Time Programmable Shading Project
  - Implement programmable shading operation in real time via multiple rendering passes
  - Quake III

Programmable Graphics Hardware: History

- **2000 :**
  - **Peercy et al. proposed a system that translated RenderMan shaders to run in multiple passes on graphics hardware**
  - **They found that GPU lacked two features that would make this approach very general**
    - **Dependent texture reads – use computation results as texture coordinates**
    - **Data types with extended range and precision in textures and color buffers**
- **2001 : NVIDIA GeForce 3**
  - **DirectX 8.0, 8.1 (Shader Model 1.1, 1.2~1.4)**
  - **Vertex shader**
    - **Programmed in assembly-like language**
  - **Pixel shader**
    - **Allows very limited "programs" (DirectorX 8.0); become better (8.1)**
      - **Limited in length (12 instructions or less)**
      - **Lacked dependent texture read and float data**
      - **No flow control**

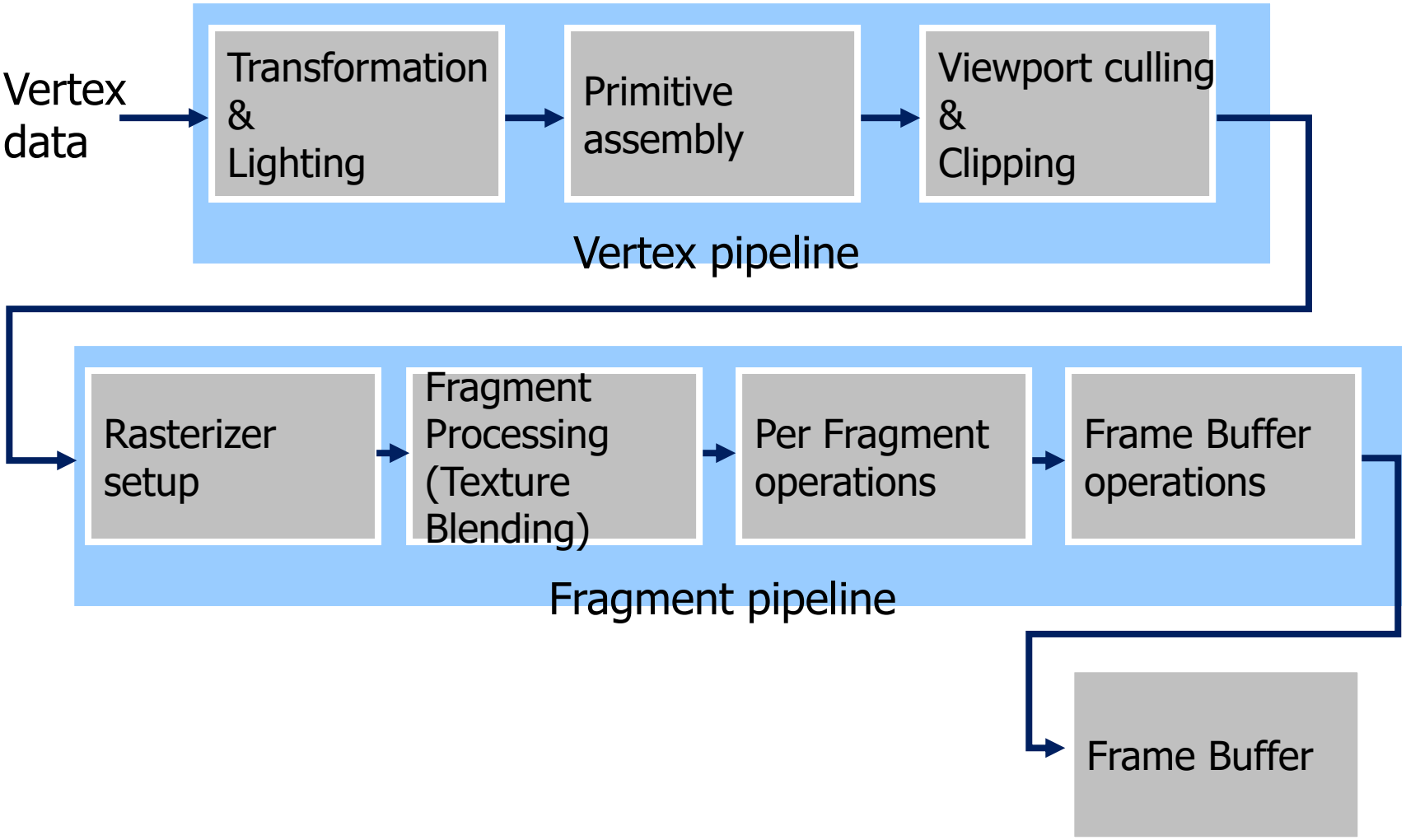
Programmable Graphics Hardware: History

- **2002**
  - **DirectX 9.0 (Shader Model 2.0)**
  - **Fully programmable vertex and pixel shaders**
  - **Support for arbitrary dependent texture reads and 16-bit float**
  - **Support for flow control**
  - **Shader programming language: HLSL, GLSL, Cg**
- **2004**
  - **Shader 3.0**
    - **An incremental improvement, turning optional features into requirements**
    - **Adding dependent texture reads in vertex shader**
- **2007**
  - **Shader 4.0 – DirectX 10 – support only high-level languages**
    - **Unified shader architecture**
    - **Geometry shader, Stream output**

Fixed Pipeline

- **Vertex**
  - **Attributes of vertex are fixed**
    - **Position, normal, RGBA, texture coordinate**
  - **Projection transformation is fixed**
  - **Lighting function is fixed**
    - **Phone model (isotropic BRDF)**
- **Fragment**
  - **How to generate final pixel color is fixed**
    - **Texture access and blending with multitexture**

Fixed Pipeline



Fixed Pipeline

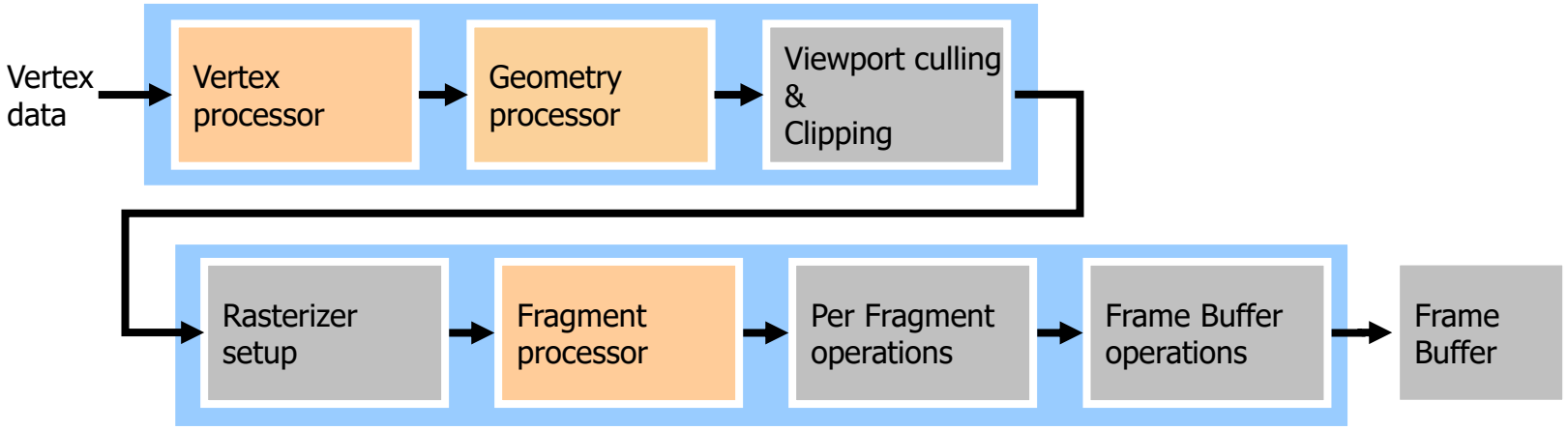
- **Transformation & lighting**
  - **Model, view, projection transformation**
  - **Lighting computation**
- **Primitive assembly**
  - **Assembly vertices to primitives such as line segment and triangles**
- **Viewport culling and clipping**
  - **Must be done on a primitive by primitive basis**
  - **Back-face culling**
  - **View volume culling**

Fixed Pipeline

- **Rasterization or scan conversion**
  - **Determines fragments inside the primitive**
    - **Output a set of fragments for each primitives with information such as color, depth, texture coordinate**
- **Texture blending or fragment processing**
  - **Texture accessing and blending**
  - **Fog**
- **Per fragment operations**
  - **Stencil and depth test**
  - **Fragments passing all tests are written to frame-buffer (with alpha blending)**

Modern Programmable Pipeline

- **Three programmable processors to replace components in the fixed rendering pipeline**
  - **Vertex processor**
  - **Geometry processor**
  - **Fragment (pixel) processor**

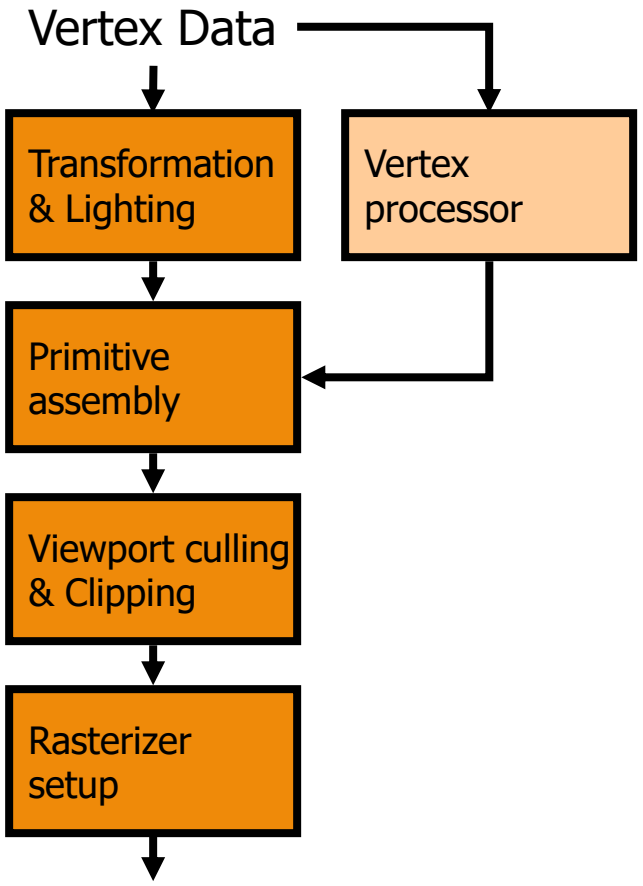


Why Programmable Pipeline?

- **Vertex processor**
  - **Modify/create/ignore attributes of vertex, such as position, color, normal, texture coordinates**
    - **Deformation, transformation**
  - **Support different lighting models**
- **Geometry processor**
  - **Allows GPU to create and destroy geometric primitives (points, lines, triangles) on the fly.**
- **Fragment processor**
  - **Per-pixel lighting, allowing complex shading equation to be evaluated per pixel.**

Vertex Processor

- **Programmability in vertex level.**
  - **Replace the Transformation & Lighting unit.**
    - **Vertex transformation**
    - **Normal transformation & normalization**
    - **Texture coordinate generation**
    - **Texture coordinate transformation**
    - **Per-vertex lighting**



Vertex Processor

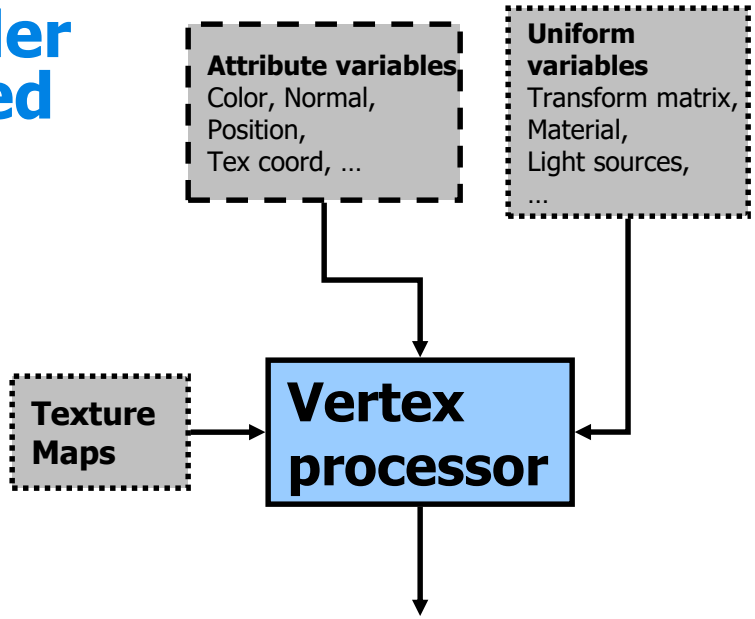
- **What does vertex processor do**
  - **Each vertex is associated with a vertex processor.**
  - **Provides a way to modify attributes associated with each vertex.**
    - **Position, Normal, Color, Texture coordinates…**
    - **In OpenGL, the vertex is assigned using glVertex*().**
  - **Performs the lighting computation**
  - **Shaders to be run on the vertex processor are called vertex shaders.**

Vertex Processor

- **What vertex processor can not do**
  - **It does not replace graphics operations for assembling the primitives.**
    - **You can not ask vertex processor to change the primitive type or change the order of vertices form the primitives.**
  - **No vertex can be created or deleted.**
  - **Results of a vertex can not be passed to another vertex.**
    - **Parallel computing**

Vertex Processor

- **Dataflow**
  - **The inputs of a vertex shader are the attributes associated on the vertex (vertex posotion, color, normal, texture coordinates…).**
    - **These values are assigned by glVertex(), glNormal(), glColor(), … in OpenGL.**
  - **The outputs of a vertex shader are the attributes of the vertex, also.**



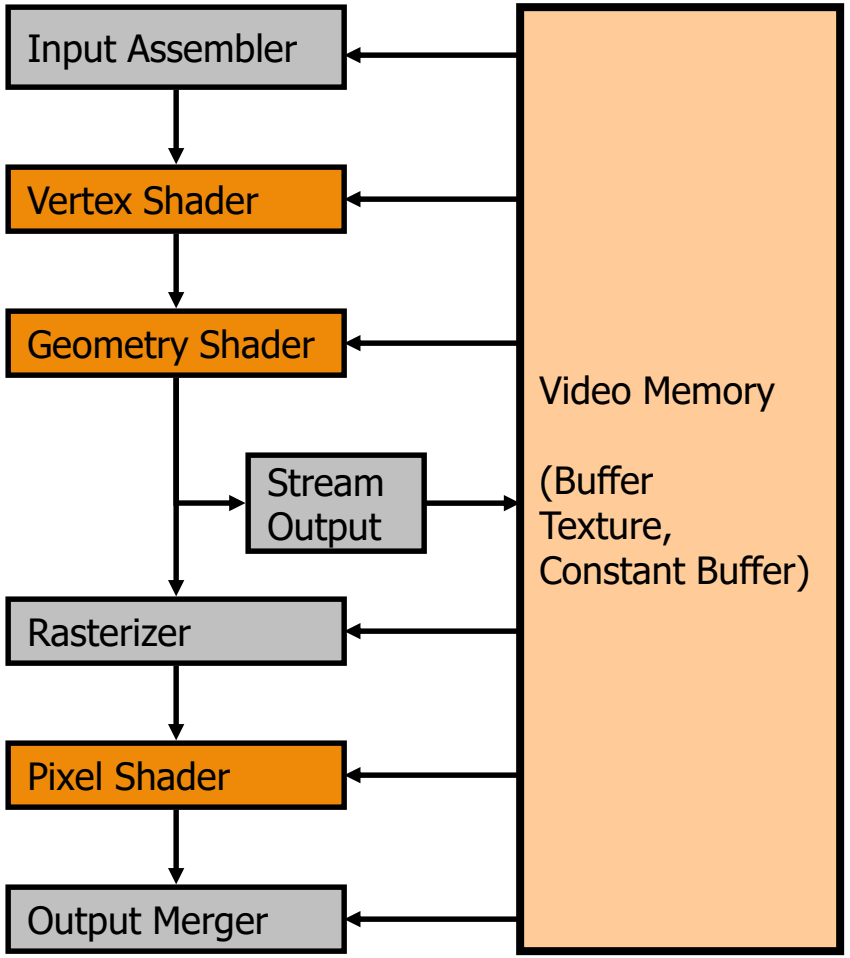
Vertex Processor
Fixed vs. programmable

- **Without programmable vertex shaders**
  - **Change of vertex attributes must be done by CPU, and the result is sent to the pipeline for every frame.**
  - **No way to change the vertex lighting**

- **With programmable vertex shader**
  - **All changes of vertex attributes are done by vertex shaders**
  - **Different vertex lightings can be performed**

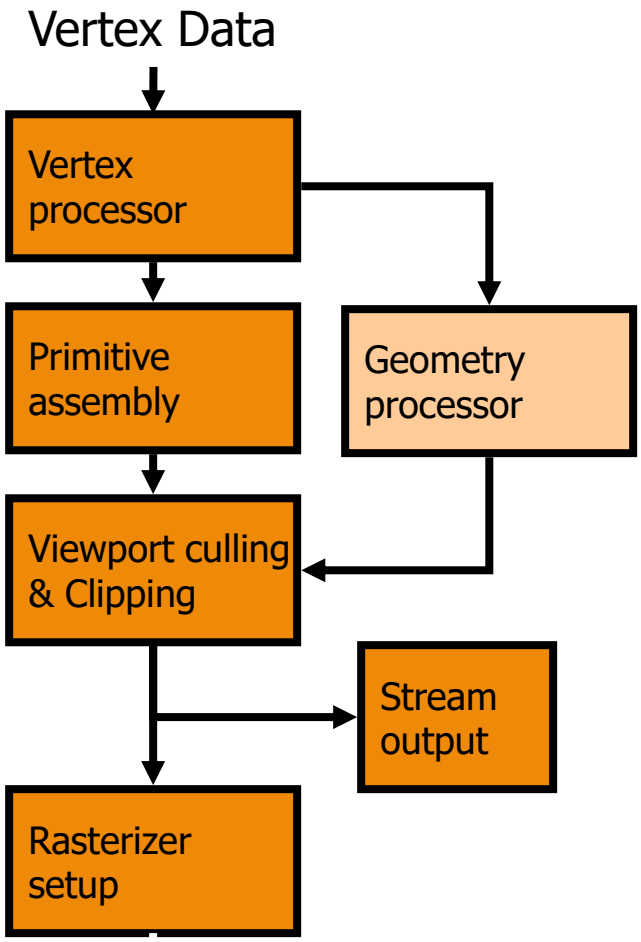
GPU in DX10 (in late 2006)

- **Direct X 10 features**
  - **Unified shader architecture**
  - **Geometry shader**
  - **Stream out**



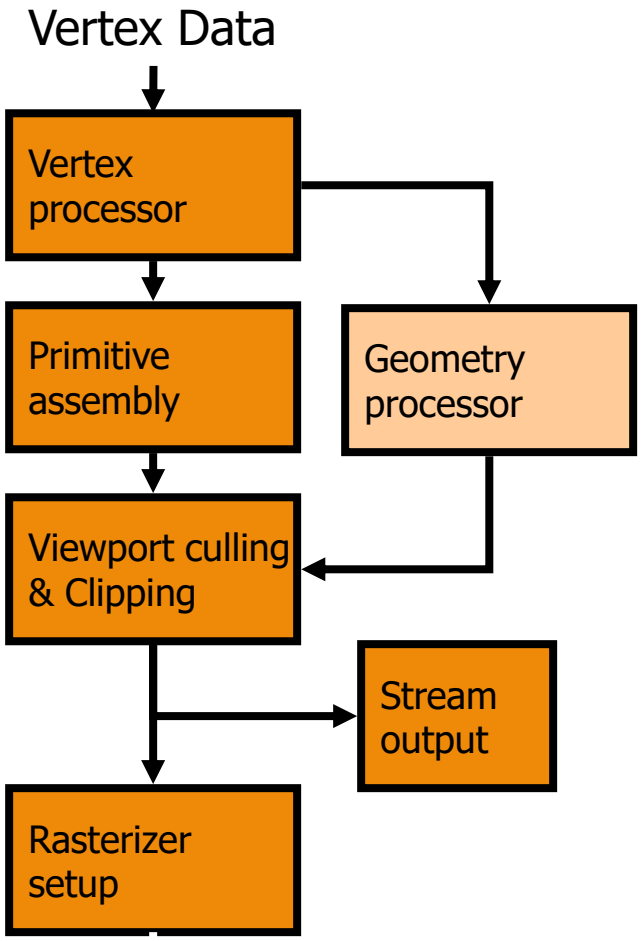
Geometry Shader

- **Programmability for primitive assembly**
  - **Replace the Primitive Assembly unit. Right after vertex shader.**
  - **Input**
    - **Primitives**
      - **A point, A line, A triangle**
      - **Line with adjacency (2 adjacent vertices on the polyline)**
      - **Triangle with adjacency (3 vertices outside of the triangle)**
  - **Output**
    - **Zero or more primitives**
      - **Points, Polylines, Triangle strips**
      - **May be in different type**
        - » **Input triangles, output their centroids**



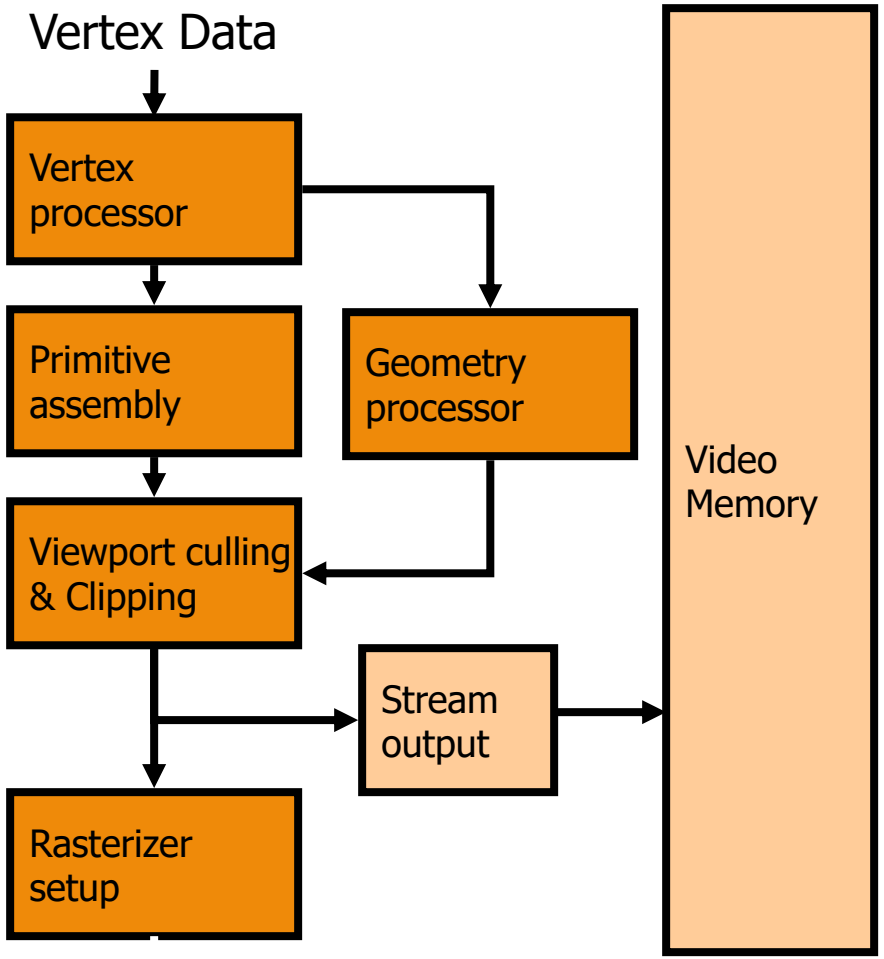
Geometry Shader

- **Programmability for primitive assembly**
  - **A mesh can be modified by editing vertices, adding new primitives, and remove others.**
  - **To provide additional capability to generate complex primitives.**
    - **Applications**
      - **Silhouette detection and extrusion**
      - **Access to mesh topology**
      - **Mesh tessellation**



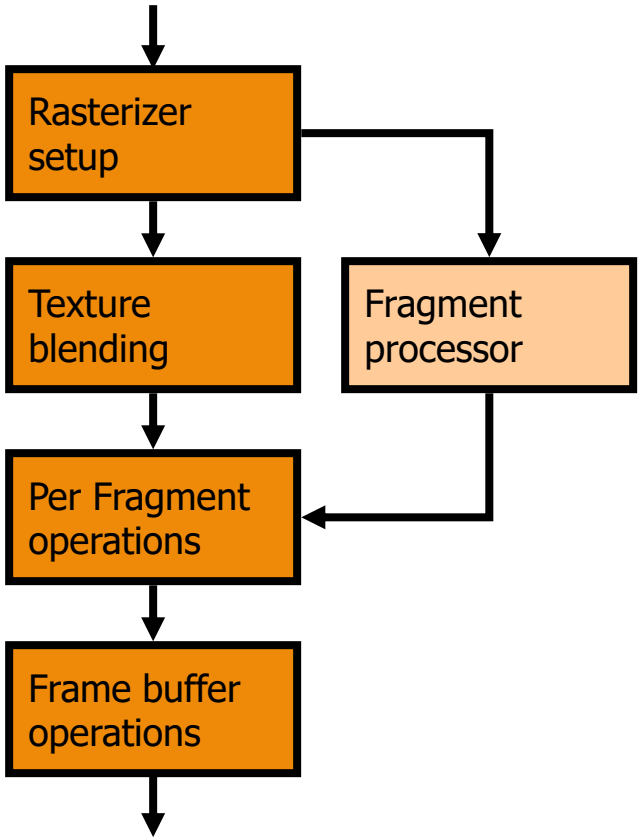
Stream Output

- **After vertices are processed by vertex shader and, optionally, geometry shader, these can be output in a stream in addition to being sent on to the rasterization stage.**
  - **Rasterization can be turned off entirely**
  - **Non-graphical stream processor**
  - **Data can be sent back through pipeline**
    - **Allow interactive processing**
    - **Useful for simulating particle effects**



Fragment Processor

- **Programmability in fragment level.**
  - **Replace the following operations**
    - **Operations on interpolated values**
    - **Texture access**
    - **Texture applications**
    - **Fog**
    - **Color sum**
    - **Alpha test (new for Shader Model 4.0)**



Fragment Processor

- **What does fragment processor do**
  - **Each pixel is associated with a fragment processor.**
  - **Provides a way to compute the final pixel color and depth value base on the input data.**
    - **The input data are the result of the rasterization stage, and can not be controlled by user explicitly.**
  - **Shaders to be run on the fragment processor are called fragment shaders or pixel shaders.**

Fragment Processor

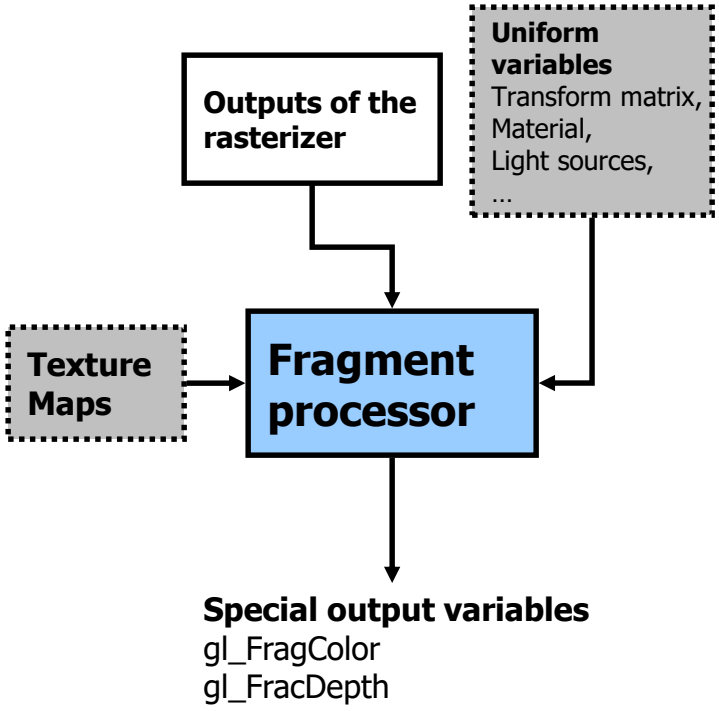
- **What does fragment processor do (cont.)**
  - **Dependent texture read**
    - **Access the texture content base on the texture coordinate computed on the fragment.**
    - **Early version of vertex shader can not perform the dependent texture read - Nvidia's Geforce 6 series have.**
  - **Multiple render targets (MRT)**
    - **Multiple vectors can be generated for each fragment and saved to different buffers**
      - **a single pass can generate several images, instead of one pass per output buffer.**
      - **Blending can be performed on multiple buffers**
        - » **DirectX 10.1 allows different blending operations on each MRT buffer; previous versions only allow same blending operation.**

Fragment Processor

- **What fragment processor can not do**
  - **Frame buffer operations (depth test, alpha test, stencil test…)**
  - **Alpha blending, pixel ownership test, plane masking…**
  - **The fragment processor can not update the pixel that does not associate to it.**
  - **Results of a pixel can not be used by another pixel in the same frame.**

Fragment Processor

- **Dataflow**
  - **The inputs of a fragment shader are the attributes associated on the fragment (depth value, color, normal, texture coordinates…).**
    - **These values are interpolated in the rasterization stage.**
    - **Can not be assigned by user directly.**
  - **The outputs of a fragment shader are the color and the depth value of the pixel.**



Notes on GPU programming

- **CPU**
  - Access main memory.
  - Can be accessed directly.
  - General purpose.
  - Unrestricted input and output.
  - One (usually).
- **GPU (Graphics Processing Unit)**
  - Access texture memory.
  - Can only be accessed through graphics API.
  - Specific purpose vector processor.
  - Input and output are restricted.
  - Several (can be used as parallel computing).

Notes on GPU programming

- **GPU is not a general purpose processor**
  - **It is part of the rendering pipeline.**
  - **Carefully study the spec. of GPU before using it.**
  - **Only several programs are suitable to be run on GPU.**
- *** New API for non-graphics applications**
  - **Nvidia's CUDA**
- **Cooperate between vertex and fragment shaders.**
  - **In most cases, the vertex and fragment shaders are used together to solve the problem.**

Notes on GPU programming

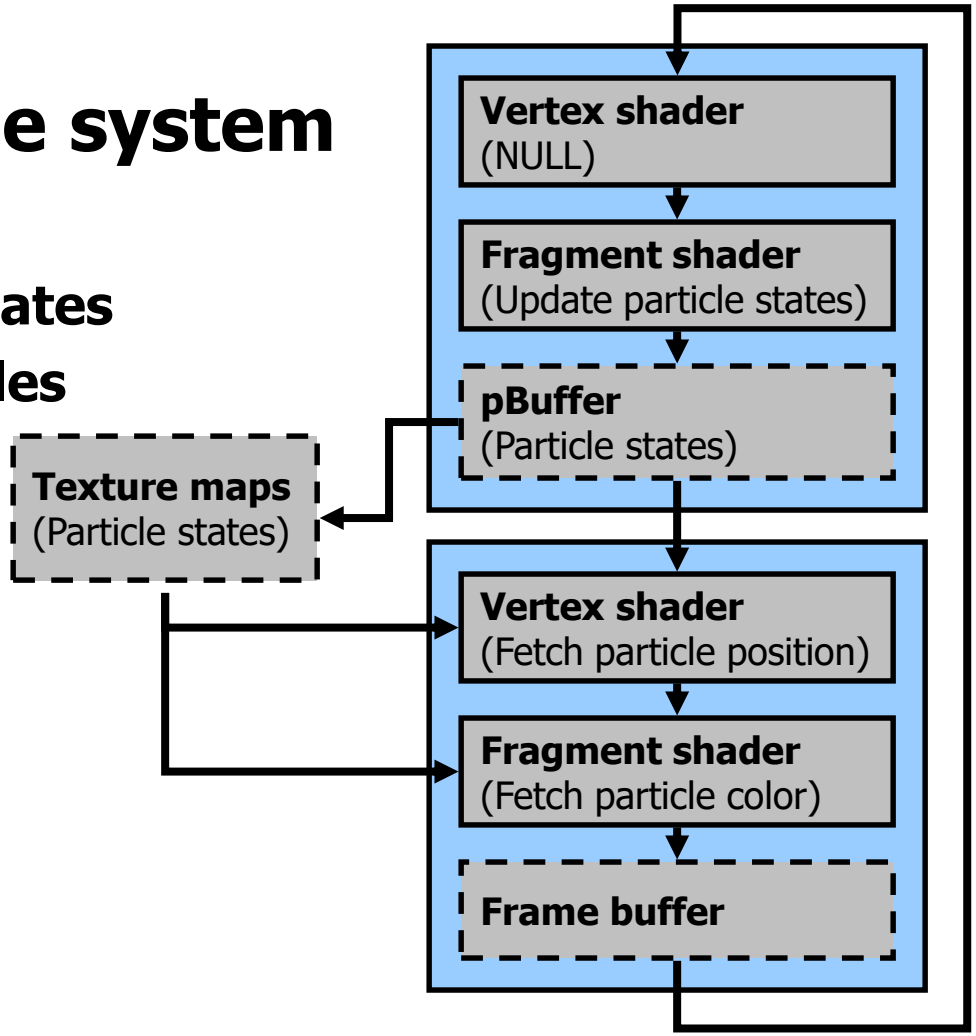
- **Multi-pass rendering**
  - **Several programs require multiple rendering passes to solve the problems.**
    - **Multiple shaders.**
    - **The result of current rendering pass will be used in the next pass.**
      - **Use pBuffer or frame buffer object (FBO) to store the result.**

Shading languages

- **Shaders are programmed using C-like shading languages**
  - **HLSL, Cg, GLSL**
  - **Compiled to a machine-independent assembly language, which is converted to the actual machine language in a separate step.**

Notes on GPU programming

- **Example : Particle system**
  - **Two pass**
    - **Update particle states**
    - **Render the particles**



Extending OpenGL

- **OpenGL version**
  - **OpenGL 1.0 ~ 1.5 (in about 10 years)**
    - **Use OpenGL extension to support new programmable pipeline features**
    - **Use assembly-like codes, which is loaded to shaders by OpenGL extensions**
  - **OpenGL 2.0 in 2004**
    - **Include OpenGL Shading Language (GLSL)**
      - **C-like**
      - **Add more language features and data types that make easier to program shaders**
      - **Simpler (than Cg) to develop OpenGL shaders using GLSL**

Extending OpenGL

- **Cg**
  - **Similar to GLSL, but has different targeted users**
    - **Supports shaders that are portable across multiple platforms, including OpenGL and Microsoft's DirectX**
    - **Virtually identical to Microsoft's High Level Shading Language (HLSL)**
    - **Interface between Cg and OpenGL is more sophisticated than the interface between GLSL and OpenGL.**