Real-Time Volume Graphics

[02] GPU Programming
Graphics Processor

Vertex Processors

Fragment Processors

Memory Access
Z-Compare and Blending

Example
NVidia GeForce6
PC Architecture

- CPU
  - 6.4 GB/s
- North Bridge
- System Memory
  - 6.4 GB/s
- South Bridge
- GPU
  - up to 8 GB/s
- Video Memory
  - up to 35 GB/s
What can the hardware do?

- Rasterization
  - Decomposition into fragments
What can the hardware do?

- **Rasterization**
  - Decomposition into fragments
  - Interpolation of color
What can the hardware do?

- Rasterization
  - Decomposition into fragments
  - Interpolation of color
  - Texturing
    - Interpolation/Filtering
    - Fragment Shading
What can the hardware do?

- **Rasterization**
  - Decomposition into fragments
  - Interpolation of color
  - Texturing
    - Interpolation/Filtering
    - Fragment Shading
What can the hardware do?

- **Rasterization**
  - Decomposition into fragments
  - Interpolation of color
  - Texturing
    - Interpolation/Filtering
    - Fragment Shading

- **Fragment Operations**
  - Depth Test (Z-Test)
  - Alpha Blending (Compositing)
Rasterization

Geometry Processing → Rasterization → Fragment Operations

- Polygon Rasterization
  - Decomposition of primitives into fragments

- Texture Fetch
  - Interpolation of texture coordinates
  - Filtering of texture color

- Texture Application
  - Combination of primary color with texture color

Primitives → Fragments
Fragment Operations

- Geometry Processing
- Rasterization
- Fragment Operations
  - Alpha Test: Discard all fragments within a certain alpha range
  - Stencil Test: Discard a fragment if the stencil buffer is set
  - Depth Test: Discard all occluded fragments
  - Alpha Blending: Combine the fragment color with the color already in the frame buffer

Fragments

Real-time Volume Graphics
Christof Rezk Salama
Computer Graphics and Multimedia Group, University of Siegen, Germany
Eurographics 2006
Graphics Hardware

Scene Description

Programmable Pipeline

Vertex Shader

Fragment Shader

Fragment Operations

Vertices

Primitives

Fragments

Pixels

Raster Image

REAL-TIME VOLUME GRAPHICS
Christof Rezk Salama
Computer Graphics and Multimedia Group, University of Siegen, Germany

Eurographics 2006
Vertex Shader

**Important Features:**

- Vertex Shader has information about one single Vertex only (no topological information)!

- For each set of vertex-attributes, the vertex shader generates exactly one vertex
  - The vertex shader cannot create additional vertices!
  - The vertex shader cannot discard vertices from the stream!

- The term „shader“ is somehow misleading, since the vertex shader can change the geometry!
Vertex Shader Instructions

- Assembly-Language, such as
  - ABS: absolute value
  - ADD: addition
  - DP3: scalar product (dot product)
  - DP4: scalar product 4-components
  - DST: distance vector
  - LIT: illumination terms
  - MUL: multiplication
  - MAD: multiply and add
  - SUB: subtraction
  - XPD: cross product

- Most commands are vector commands (4 components)
High-Level Shading Languages

Who wants to write assembler code?

- Stanford Shading Language
  - Cg (developed by Nvidia) for OpenGL and DirectX
  - DirectX 9.0 HLSL (DirectX only, Syntax similar to Cg)
  - GLSL (OpenGL shading language)

- Syntax similar to C
- plus vector variables and vector instructions:
  - `float4 v1; // same as float v1[4] in C`
  - `int3 v2; // same as int v2[3] in C`
- Swizzling: `float4 v3 = v1.xzzy;`
Programmable Vertex Processor

Begin Vertex

Input-Registers

copy vertex attributes to input registers
Programmable Vertex Processor

1. Begin
2. Vertex
3. Vertex Program Instructions
4. Input-Registers
5. Temporary Registers

- Copy vertex attributes to input registers
- Fetch next instruction
- Read input- or temporary registers
Programmable Vertex Processor

Begin Vertex
- Copy vertex attributes to input registers

Vertex Program Instructions

Input-Registers

Temporary Registers

Output-Registers

Fetch next instruction

Read input- or temporary registers

Mapping: Negation Swizzling

Execute command

Write to output or temp. registers
Programmable Vertex Processor

1. Begin Vertex
   - Copy vertex attributes to input registers

2. Vertex Program Instructions
3. Input Registers
4. Temporary Registers
5. Output Registers

6. Fetch next instruction
7. Read input- or temporary registers
8. Mapping: Negation Swizzling
9. Execute command
10. Write to output or temp. registers

Flowchart:
- If finished? (no)
  - Repeat
- If finished? (yes)
  - Emit Vertex
Fragment Processor

Begin Fragment

copy fragment attributes to Input register

Input-Registers
Fragment Processor

Begin Fragment

copy fragment attributes to Input register

Fetch next instruction

Fragment Program Instructions

Input Registers
Fragmnet Processor

Begin Fragment

- Fragment Program
  - Instructions
- Input Registers
- Temporary Registers

Copy fragment attributes to Input register

Fetch next instruction

Read input of temporary registers
Fragment Processor

Begin Fragment

Copy fragment attributes to Input register

Fetch next instruction

Read input of temporary registers

Mapping: Negation Swizzling

Texture Instruction?

execute instruction

Write to output or temporary registers

Fragment Program Instructions

Input-Registers

Temporary Registers

Output Registers
Fragment Processor

Begin Fragment
- copy fragment attributes to input register

Fragment Program Instructions
- Input-Registers
- Temporary Registers
- Texture Memory
- Output Registers

Fetch next instruction
Read input of temporary registers
Mapping: Negation Swizzling
Texture Instruction?
- yes
  - Texture Address and Sample Texture
  - Interpolate Texel Color
- no
  - Execute Instruction
  - Write to output or temporary registers
Phong Shading

Per-Pixel Lighting: Local illumination in a fragment shader

```c
void main(float4 position : TEXCOORD0,
          float3 normal : TEXCOORD1,
          out float4 oColor : COLOR,
          uniform float3 ambientCol,
          uniform float3 lightCol,
          uniform float3 lightPos,
          uniform float3 eyePos,
          uniform float3 Ka,
          uniform float3 Kd,
          uniform float3 Ks,
          uniform float shiny)
{
```

Phong Shading

Per-Pixel Lighting: Local illumination in a fragment shader

```c
float3 P = position.xyz;
float3 N = normal;
float3 V = normalize(eyePosition - P);
float3 H = normalize(L + V);
float3 ambient = Ka * ambientCol;
float3 L = normalize(lightPos - P);
float diffLight = max(dot(L, N), 0);
float3 diffuse = Kd * lightCol * diffLight;
float specLight = pow(max(dot(H, N), 0), shiny);
float3 specular = Ks * lightCol * specLight;

oColor.xyz = ambient + diffuse + specular;
oColor.w = 1;
```