Basics of GPU-Based Programming

Daniel Weiskopf
Institute of Visualization and Interactive Systems
University of Stuttgart
Overview

• Rendering pipeline on current GPUs
• Low-level languages
• High-level shading languages

• Also see tutorial
  „GPGPU: General-Purpose Computation on Graphics Processors”
Graphics Pipeline

Scene description

Vertices → Primitives → Fragments → Pixels

- Geometry processing
- Rasterization
- Fragment operations
- Lighting, Transformations
- Texturing, Blending
**Programmable Pipeline**

- 3D application
- 3D API
- GPU front end
- GPU
- Shaders
- vertex processor
- fragment processor
- frame buffer
- raster operations
- rasterization
- primitive assembly
- CPU
- CPU
**Issues**

- How are vertex and pixel shaders specified?
  - Low-level, assembler-like
  - High-level language

- Data flow between components
  - Per-vertex data (for vertex shader)
  - Per-fragment data (for pixel shader)
  - Uniform (constant) data: e.g. model-view matrix, material parameters
What Are Low-Level APIs?

• Current low-level APIs:
  – OpenGL extensions: GL_ARB_vertex_program, GL_ARB_fragment_program
  – DirectX 9: Vertex Shader 2.0, Pixel Shader 2.0, Vertex Shader 3.0, Pixel Shader 3.0

• Older low-level APIs:
  – DirectX 8.x: Vertex Shader 1.x, Pixel Shader 1.x
  – OpenGL extensions: GL_ATI_fragment_shader, GL_NV_vertex_program, …
Applications of Vertex Programming

• Customized computation of vertex attributes
• Computation of anything that can be interpolated linearly between vertices

Limitations:
  – Vertices can neither be generated nor destroyed
  – No information about topology or ordering of vertices is available
Applications of Fragment Shaders

- Customized computation of fragment attributes
- Computation of anything that should be computed per pixel

→ Basis for texture-based flow visualization

- Limitations:
  - Fragments cannot be generated
  - Position of fragments cannot be changed
  - No information about geometric primitive is available
**Fragment Shader**

- Circumvents the traditional fragment pipeline
- What is not replaced?
  - Fragment tests (alpha, stencil, and depth tests)
  - Blending
**Fragment Shader: Programming Model**

- **Input registers**
- **Shader**
  - **Textures**
  - **Constants**
  - **Registers**
- **Output registers**
**DirectX 9: Pixel Shader 2.0**

- Pixel Shader 2.0 introduced in DirectX 9.0
  - Syntax of Pixel Shader 3.0 is closely related
- Instruction set
  - Numerical operations
  - Texture operations (lookup)
- Similar functionality and limitations in GL_ARB_fragment_program
  - Similar registers and syntax
**DirectX 9: Pixel Shader 2.0**

- Declaration of texture samplers

  ```
  dcl_type s*
  ```

  - Examples:

    ```
    dcl_2d s0
    dcl_volume s1
    ```

- Declaration of input color and texture coordinate

  ```
  dcl v*[,mask]
  dcl t*[,mask]
  ```
**Pixel Shader 2.0: Instructions**

- **Instruction set**
  - Operate on floating-point scalars or 4-vectors
  - Basic syntax:
    ```
    op destination [, source1 [, source2 [, source3]]]
    ```
  - Example:
    ```
    mov oC0, v0; // sets resulting color oC0
    ```

- **Texture sampling**
  - Syntax:
    ```
    op destination, source, sn
    ```
  - Example:
    ```
    texld r2, t1, s0;
    ```
Pixel Shader 2.0: Simple Example

```shadertex
ps_2_0

dcl_2d s0
dcl t0.xy
texld r1, t0, s0
mov oC0, r1
```
High-Level Shading Languages

Why?
– Avoids programming, debugging, and maintenance of long assembly shaders
– Easy to read
– Easier to modify existing shaders
– Automatic code optimization
– Wide range of platforms
Assembler vs. High-Level Language

Assembler

\[
\begin{align*}
& \text{dp3 r0, r0, r1} \\
& \text{max r1.x, c5.x, r0.x} \\
& \text{pow r0.x, r1.x, c4.x} \\
& \text{mul r0, c3.x, r0.x} \\
& \text{mov r1, c2} \\
& \text{add r1, c1, r1} \\
& \text{mad r0, c0.x, r1, r0} \\
& \ldots
\end{align*}
\]

High-level language

\[
\begin{align*}
& \text{float4 cSpec} = \text{pow(max(0, dot(Nf, H)), phongExp)}.xxx; \\
& \text{float4 cPlastic} = \text{Cd} \times (\text{cAmbi} + \text{cDiff}) + \text{Cs} \times \text{cSpec}; \\
& \ldots
\end{align*}
\]

Blinn-Phong shader expressed in both assembly and high-level language
**Data Flow through Pipeline**

- Vertex shader program
- Fragment shader program
- Connectors

![Diagram showing data flow through a pipeline with 3D application, vertex program, fragment program, frame buffer, and connectors]
High-Level Shading Languages

• Cg
  – “C for Graphics”
  – By NVIDIA

• HLSL
  – “High-level shading language”
  – Part of DirectX 9 (Microsoft)

• OpenGL 2.0 Shading Language
**Cg**

- Typical concepts for a high-level shading language
- Language is (almost) identical to DirectX HLSL
- Syntax, operators, functions from C/C++
- Conditionals and flow control
**Cg: Pixel Shader**

- Connectors: What kind of data is transferred to / from pixel program?
- Actual pixel shader
**Cg: Pixel Shader Example**

```cg
// vertex shader to pixel shader
struct vertexOut {
    float4 Position : POSITION;
    float4 Diffuse : COLOR0;
    float4 TexCoord : TEXCOORD0;
};

// final pixel output: data from pixel shader to frame buffer
struct pixelOut {
    float4 Col : COLOR;
};

// pixel shader
pixelOut mainPS(
    vertexOut IN, // input from vertex shader
    uniform float param // constant parameter
) {
    pixelOut OUT;
    ...
    // more code
    OUT.Col = IN.Diffuse;
    return OUT; // output of pixel shader
}
```
What’s Next?

• Usage of GPU and CPU for flow visualization