Analysis of Hardware-Accelerated Pixel and Vertex Shader Tools

Gerhard Lorenz
Gerhard.Lorenz@gmx.net

Munich University of Applied Sciences
Fachhochschule München
Presentation Overview

• Title and Terms
• Graphics Pipeline
• High-Level Shading Languages
• Shader Examples
Introduction

Analysis of Hardware-Accelerated Pixel and Vertex Shader Tools

Gerhard Lorenz
Gerhard.Lorenz@gmx.net

Munich University of Applied Sciences
Fachhochschule München
Introduction

Analysis of Hardware-Accelerated Pixel and Vertex Shader Tools

Gerhard Lorenz
Gerhard.Lorenz@gmx.net

Munich University of Applied Sciences
Fachhochschule München
Shader Tools

- Shader Languages (Low-Level vs. High-Level)
- Shader IDEs (Integrated Development Environment)
- Shader Plugins for DCC (Digital Content Creation)
Introduction

Analysis of Hardware-Accelerated

Pixel and Vertex Shader Tools

Gerhard Lorenz
Gerhard.Lorenz@gmx.net

Munich University of Applied Sciences
Fachhochschule München
Data Elements

- Pixel
- Fragment
- Vertex
- Primitive
Presentation Overview

• Title and Terms
• Graphics Pipeline
• High-Level Shading Languages
• Shader Examples
Programmable Vertex Pipeline

Programmable Functions

- vertex transformation
- normal transformation and normalization
- texture coordinate generation and transformation
- per-vertex lighting
- color material application
Rasterizer (Non-Programmable)
Programmable Fragment Processor

Fragment Pipeline $\Rightarrow$ Programmable Fragment Processor $+$ Raster Operations

Texture Unit

**Programmable Functions**

- operations on interpolated values
- texturing
- color sum
- fog
Shader Concept

- Vertex Shader
- Fragment Shader
- execution on single data elements
- replacement for fixed-functions
Presentation Overview

• Title and Terms
• Graphics Pipeline
• High-Level Shading Languages
• Shader Examples
Why High-Level Shading Languages?

Assembler

```assembly
RSQR R0.x, R0.x;
MULR R0.xyz, R0.xxxx, R4.xyzz;
MOVR R5.xyz, -R0.xxyz;
MOVR R3.xyz, -R3.xxyz;
DP3R R3.x, R0.xyzz, R3.xyzz;
SLTR R4.x, R3.x, {0.000000}.x;
ADDR R3.x, {1.000000}.x, -R4.x;
MULR R3.xyz, R3.xxxx, R5.xyzz;
MULR R0.xyz, R0.xxyz, R4.xxxx;
ADDR R0.xyz, R0.xxyz, R3.xyzz;
DP3R R1.x, R0.xyzz, R1.xxyz;
MAXR R1.x, {0.000000}.x, R1.x;
LG2R R1.x, R1.x;
MULR R1.x, {10.000000}.x, R1.x;
EX2R R1.x, R1.x;
MOVR R1.xyz, R1.xxxx;
MULR R1.xyz, {0.900000, 0.800000, 1.000000}.xyzz, R1.xxyz;
DP3R R0.x, R0.xyyy, R2.xyzz;
MAXR R0.x, {0.000000}.x, R0.x;
MOVR R0.xyz, R0.xxxx;
ADDR R0.xyz, {0.100000, 0.100000, 0.100000}.xyzz, R0.xxyz;
MULR R0.xyz, {1.000000, 0.800000, 0.800000}.xyzz, R0.xxyz;
ADDR R1.xyz, R0.xxyz, R1.xxyz;
```

C for Graphics (Cg)

```c
float3 cSpec = 
    pow(max(0, dot(Nf, H)), phongExp).xxx;
float3 cPlastic = 
    Cd * (cAmbi + cDiff) + Cs * cSpec;
...
```

=> Phong Shader
High-Level Shading Languages Design Goals

- **replacement** for assembler

- **similarity** to existing high-level languages (C, C++, Java)

- **simplicity** in hardware access

- **portability** across various platforms and chipsets

- **extensibility** for new hardware generations

- support for **non-shading algorithms**
# Language Comparison

## Cg - HLSL - GLSL

<table>
<thead>
<tr>
<th></th>
<th>C for Graphics (Cg)</th>
<th>High-Level Shader Language (HLSL)</th>
<th>OpenGL Shading Language (GLSL)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Logo</strong></td>
<td><img src="logo_cg.png" alt="Cg Logo" /></td>
<td><img src="logo_hlsl.png" alt="HLSL Logo" /></td>
<td><img src="logo_glsl.png" alt="GLSL Logo" /></td>
</tr>
<tr>
<td><strong>Authority</strong></td>
<td>NVIDIA</td>
<td>Microsoft</td>
<td>OpenGL ARB</td>
</tr>
<tr>
<td><strong>Operating System</strong></td>
<td>Windows, Linux, MacOS</td>
<td>Windows</td>
<td>Windows, Linux, MacOS</td>
</tr>
<tr>
<td><strong>Graphics API</strong></td>
<td>OpenGL, DirectX</td>
<td>DirectX</td>
<td>OpenGL</td>
</tr>
<tr>
<td><strong>Function Support</strong></td>
<td>Cg runtime</td>
<td>DirectX runtime</td>
<td>OpenGL 1.5 extensions</td>
</tr>
<tr>
<td><strong>Compatibility</strong></td>
<td>HLSL</td>
<td>Cg</td>
<td></td>
</tr>
<tr>
<td><strong>Data to Hardware (App-&gt;VP-&gt;FP)</strong></td>
<td>GL/DX profiles/semantics hardware-oriented</td>
<td>DX profiles/semantics hardware-oriented</td>
<td>built-in/user-defined var's high-level-oriented</td>
</tr>
<tr>
<td><strong>Configuration</strong></td>
<td>profile-dependent</td>
<td>profile-dependent</td>
<td>hardware-dependent</td>
</tr>
<tr>
<td><strong>Optimization</strong></td>
<td>profiles, compiler, driver</td>
<td>profiles, compiler, driver</td>
<td>driver (compiler)</td>
</tr>
<tr>
<td><strong>Compiler Integration</strong></td>
<td>external compiler</td>
<td>external compiler</td>
<td>driver environment</td>
</tr>
<tr>
<td><strong>Compilation Process</strong></td>
<td>simple, comfortable function usage</td>
<td>simple, comfortable function usage</td>
<td>similar to standard compilation and linking</td>
</tr>
<tr>
<td><strong>Handling</strong></td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td><strong>Examples</strong></td>
<td>+</td>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td><strong>Features</strong></td>
<td>+</td>
<td>0</td>
<td>+</td>
</tr>
<tr>
<td><strong>Stability/Performance</strong></td>
<td>+</td>
<td>+</td>
<td>0</td>
</tr>
</tbody>
</table>
Compilation Process - Cg

- no driver dependency
- configuration through profiles
- ASM code accessible

3D Application

 shader source code

Core Cg Runtime

 Shader Context

 Program Object

OpenGL Cg Runtime

Direct3D Cg Runtime

3D API: OpenGL or Direct3D

Graphics Card Driver (Linker)

executable code

Graphics Processing Unit (GPU)
Compilation Process - GLSL

- compiler/linker part of driver
- compiler and linker supplied by vendors
- high performance

3D Application

shader source code

OpenGL API

OpenGL Driver

Compiler ↔ Shader Object
compiled code

Linker ↔ Program Object
executable machine code

Graphics Processing Unit (GPU)
Data Types and Structures

- **data types** (*Cg*): bool, int, float, half, fixed

- **data structures**: vector, 1 to 4 elements  e.g. float4
  matrix, up to 16 elements  e.g. float4x4
  texture, for texture maps  e.g. sampler2D
Vector Components and Operations

- **vector component**: \((x, y, z, w)\) or \((r, g, b, a)\), letter set mixing not allowed
  
  ```cpp
  int4 vector1 = \{1, 2, 3, 4\};  // vector1.z = (3)
  ```

- **swizzling**: read-access to arbitrary vector components via 
  ```cpp
  int4 vector1 = int4(1, 2, 3, 4);  // vector1=(1, 2, 3, 4)
  int4 vector2 = vector1.wzyx;  // vector2=(4, 3, 2, 1)
  ```

- **write masking**: restricted write access to arbitrary vector components via 
  ```cpp
  vector2.ragb = vector1;  // vector2=(1, 4, 2, 3)
  vector2.aagb = vector1;  // error! multiple assignment
  ```
## C-Like Feature Comparison

<table>
<thead>
<tr>
<th>Features</th>
<th>C/C++</th>
<th>Cg</th>
<th>GLSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++ style comments</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>preprocessor</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>type conversion/casting</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>arrays</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>structures</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>char/string</td>
<td>yes</td>
<td>no (reserved)</td>
<td>no (reserved)</td>
</tr>
<tr>
<td>enum/union/class</td>
<td>yes</td>
<td>no (reserved)</td>
<td>no (reserved)</td>
</tr>
<tr>
<td>vector, matrix constructor</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>swizzle operator (.)</td>
<td>no</td>
<td>yes</td>
<td>yes, additionally (s, t, p, q)</td>
</tr>
<tr>
<td>write mask operator (.)</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>arithmetic operators</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>boolean/comparison operator</td>
<td>yes</td>
<td>&amp;&amp;,</td>
<td></td>
</tr>
<tr>
<td>control flow/conditional operator</td>
<td>yes</td>
<td>yes (restricted)</td>
<td>yes (restricted)</td>
</tr>
<tr>
<td>function definition</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>function overloading</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>function recursion</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>pointer</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
Language Recommendation
Cg - HLSL - GLSL

• operating-system-independency => Cg, GLSL
• cross-graphics-API development => Cg
• vendor-independency => GLSL
Cg Experiences

- integration in applications
- compiler errors
- hardware access
- shader debugging
- support (www.shadertech.com)
High-Level Shading Languages

Conclusion

• allow C-like programming

• enable comprehensible shader code

• reduce development time

• future-proof through extensibility

• enable unique real-time effects
Presentation Overview

• Title and Terms
• Graphics Pipeline
• High-Level Shading Languages
• Shader Examples
High-Level Shading Languages
Shader Examples

- Phong Shading
- Bump Mapping
- Framework "ogl_cg"
- Shader load balancing
- Bump Mapping comparison in different spaces
Phong Shading
=Per-Fragment Shading

- lighting computation at each fragment (Fragment Shader)
- color applied to each fragment
- slow, but highest quality possible
- specular highlights also occur within polygons
Phong Shading - Vertex Shader

```c
void main(
    // vertex position in object space
    in float4 os_VertexPosition : POSITION,
    // vertex normal in object space
    in float4 os_VertexNormal : NORMAL,

    // vertex position in clip space
    out float4 cs_VertexPosition : POSITION,
    // vertex position in eye space
    out float4 es_VertexPosition : TEXCOORD0,
    // vertex normal in eye space
    out float4 es_VertexNormal : TEXCOORD1,

    // transformation matrices
    uniform float4x4 modelViewProj,
    uniform float4x4 modelView,
    uniform float4x4 modelViewIT
)
{
    // vertices to clip space for primitive assembly
    cs_VertexPosition = mul(modelViewProj, os_VertexPosition);
    // vertex position to eye space
    es_VertexPosition = mul(modelView, os_VertexPosition);
    // vertex normal to eye space
    es_VertexNormal = mul(modelViewIT, os_VertexNormal);
}
```
void main(
    // fragment position in eye space
    in  float4 es_FragmentPosition : TEXCOORD0,
    // fragment normal in eye space
    in  float4 es_FragmentNormal : TEXCOORD1,

    // final fragment color
    out float4 color : COLOR,

    // light colors
    uniform  float4 lcD,       // light color diffuse
    uniform  float4 lcS,       // light color specular
    // material colors
    uniform  float4 mcD,       // material color diffuse
    uniform  float4 mcS,       // material color specular
    // effects
    uniform  float shininess,   // highlight control
    // positions
    uniform  float3 es_LightPosition
)
{
    ...
}
Phong Shading - Fragment Shader (II)

{ // fragment position in eye space
  float3 P = es_FragmentPosition.xyz;
  // denormalization by interpolation!
  float3 N = normalize(es_FragmentNormal.xyz);
  // local lightsource
  float3 L = normalize(es_LightPosition.xyz - P.xyz);
  // local viewer
  float3 V = normalize(-P); // view point at (0,0,0)
  // half-angle vector of Phong-Blinn lighting model
  float3 H = normalize(L + V);

  // compute the intensity of diffuse light
  float diffuseLight = max(dot(L, N), 0); 
  float4 diffuse = mcD * lcD * diffuseLight;
  // compute the intensity of specular light
  float specularLight = pow(max(dot(H, N), 0), shininess);
  if (diffuseLight == 0) specularLight = 0;
  float4 specular = mcS * lcS * specularLight;

  // add all color components
  color.rgb = diffuse.rgb + specular.rgb;
  color.a = 1.0; // set alpha value to 1.0
}
Parameter Passing

Uniform Variables

float es_lightpos[]={0.0, 1.0, 1.0};
// light position in eye space
CGparameter cg_LightPosition=
    cgGetNamedParameter(cg.fProgram, "es_LightPosition");
cgGLSetParameter3fv(cg_LightPosition, es_lightpos);

Vertex Arrays

// pass along pointer to vertex position array
CGparameter cg_VertexPosition=
    cgGetNamedParameter(cg.vProgram, "os_VertexPosition");
cgGLEnableClientState(cg_VertexPosition);
cgGLSetParameterPointer(cg_VertexPosition, 3, GL_FLOAT, 0,
    vertexDataPtr);
Advanced Per-Fragment Shading Bump Mapping

Bump Mapping Principle

Geometry Normals + Bump Normals ⇒ Addition = Perturbed and Normalized Normals

Bump Mapping Process
Advanced Per-Fragment Shading Bump Mapping

Correct Bump Mapping

\[ P_0 = (x_0, y_0, z_0, u_0, v_0) \]
\[ P_1 = (x_1, y_1, z_1, u_1, v_1) \]
\[ P_2 = (x_2, y_2, z_2, u_2, v_2) \]

TBN Rotation Matrix

\[
\begin{align*}
\mathbf{T} &= \begin{vmatrix}
\frac{\partial x}{\partial u} & \frac{\partial x}{\partial v} \\
\frac{\partial y}{\partial u} & \frac{\partial y}{\partial v} \\
\frac{\partial z}{\partial u} & \frac{\partial z}{\partial v}
\end{vmatrix} \\
\mathbf{B} &= \begin{vmatrix}
\frac{\partial x}{\partial u} & \frac{\partial x}{\partial v} \\
\frac{\partial y}{\partial u} & \frac{\partial y}{\partial v} \\
\frac{\partial z}{\partial u} & \frac{\partial z}{\partial v}
\end{vmatrix}
\end{align*}
\]

\[
\mathbf{TBN} = \begin{bmatrix}
T_x & B_x & N_x \\
T_y & B_y & N_y \\
T_z & B_z & N_z
\end{bmatrix}
\]
Shader Demos
Questions ???