Introduction to Shaders

Marco Benvegnù hiforce@gmx.it

5ª Scuola Estiva di
VISUALIZZAZIONE
SCIENTIFICA
E GRAFICA
INTERATTIVA 3D

CINECA
Consorzio Interuniversitario

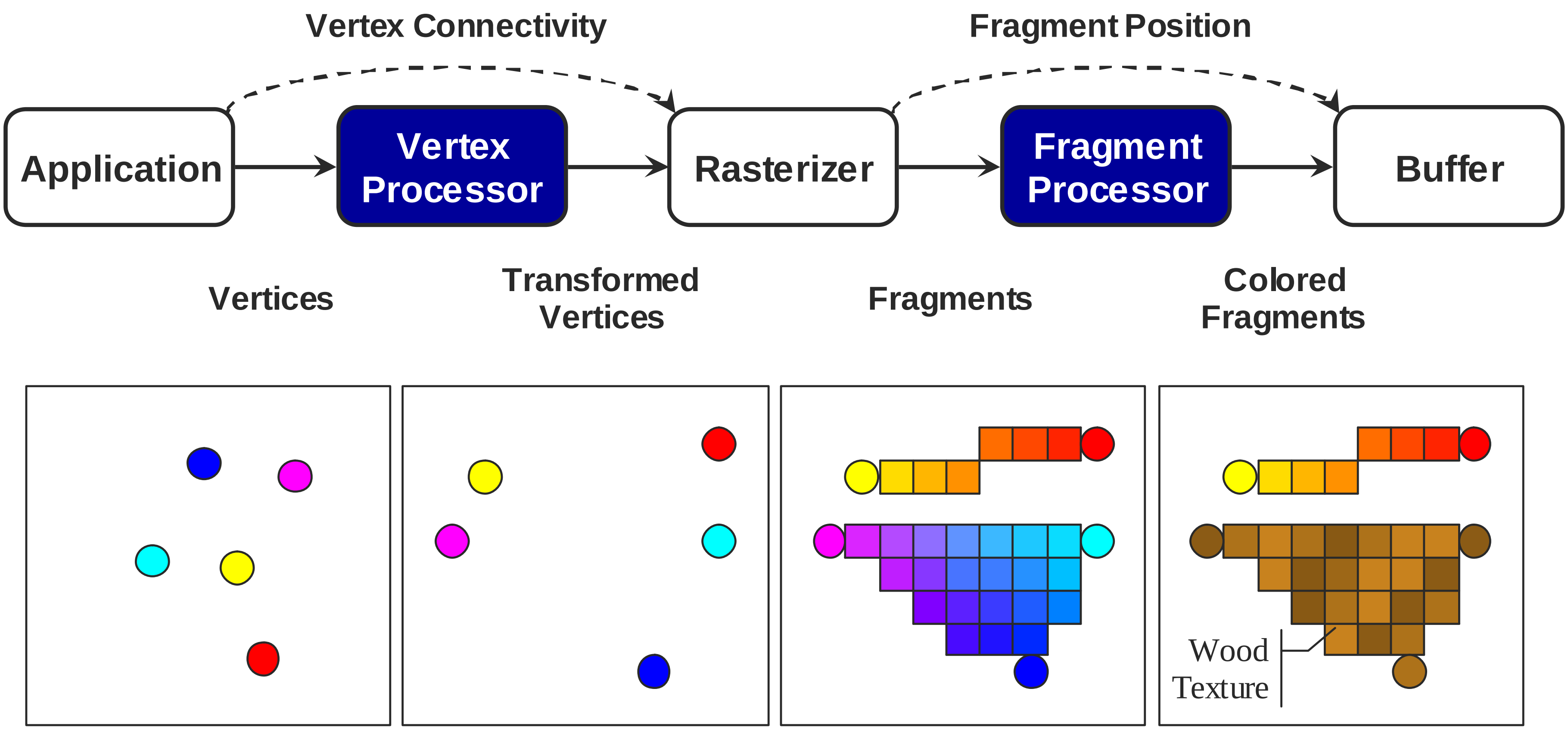
www.benve.org

Summer 2005

Overview

- Rendering pipeline
- Shaders concepts
- Shading Languages
- Shading Tools
- Effects showcase
- Setup of a Shader in OpenGL
- Introduction to GLSL

Rendering Pipeline



Application

- Output:
  - Vertices attributes (position, texture coordinates, color, normal...)
  - Connectivity information
- This snippet sends the position and color of 2 vertices:

```
glBegin(GL_LINES);
   glColor3f(1.0, 0.9, 0.0);
   glVertex3f(1.0, 4.0, 2.0);
   glColor3f(1.0, 0.1, 0.0);
   glVertex3f(2.0, 1.0, 2.0);
glEnd();
```

Vertex Processor

- Input: Vertex attributes
- Predefined operations:
  - Vertex position transformation
  - Lighting computations per vertex
  - Generation or transformation of texture coordinates
- Output: Transformed vertex attributes

Rasterizer

- Input:
  - Transformed vertices
  - Connectivity information
- Operations:
  - Primitives assembly (and clipping)
  - Rasterization (determines the pixels covered by the primitive)
  - Interpolation of vertices attributes along each primitive
- Output:
  - Position of the *fragments* in the frame buffer
  - Interpolated attributes for each fragment

Fragment Processor

- Input: Interpolated fragment attributes
- Predefined operations:
  - Texturing
  - Fog
- Output: Final color and depth of the fragment

Buffer

- Inputs:
  - Fragment color and depth
  - Fragment position
- Operations:
  - Scissor test
  - Alpha test and blending
  - Stencil test
  - Depth test

What are Shaders?

- Shaders are programs executed by vertex and fragment processors in the graphics hardware

- Vertex Shaders fully replace the “T&L Unit”
- Pixel Shaders fully replace the “Texturing Unit”

Why the need?

- Before 2001 the graphics hardware wasn't programmable and offered limited control on the rendering pipeline
- But the graphics industry is mostly driven to create new and newer effects
  - Programmers started to perform multi-pass rendering and spend more and more time to tweak the render state for tasks beyond the original scope of design
  - GPU vendors started to implement custom extensions
- Shaders opened the door to new exciting rendering effects and tecniques

Vertex Shader

- Input: Vertex attributes
- Common tasks:
  - Vertex position transformation
  - Per vertex lighting
  - Normal transformation
  - Texture coordinates transformation or generation
  - Vertex color computation
  - Geometry skinning
- Cannot access any vertex other than the current one
- Minimal output: Vertex position (in the clip space)

Fragment Shader

- Input: Interpolation of the vertex shader outputs
- Common tasks:
  - Texturing (even procedural)
  - Per pixel lighting
  - Fragment color computation
- Cannot access neighboring fragments
- Typical output: Fragment color

Simple sample

```
void main() // Vertex shader
{
        gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}


void main() // Fragment shader
{
        gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Shaders concepts

- Shaders have access to Textures and to the Render state (parameters, matrices, lights, materials…)
- A *Pass* is the rendering of a 3D Model with a Vertex and Pixel Shader pair
- An effect can require multiple Passes
  - Each pass can use a different Model and/or Shader pair
  - A Pass can render to a texture (to be used by another Pass)
- Think to the “fixed functionality” as to the default Shader

Shading Languages

HLSL (Microsoft, 2002)	Cg (nVidia, 2002)	GLSL (ARB, 2003)

ASM Shading Languages (2001)

Direct3D (Microsoft, 1995)	OpenGL (ARB, 1992)

Other Languages

- *Sh* is a meta-language embedded in C++, allowing the integration of Shaders in the application source code
- *BrookGPU* is a scientific computing language for GPUs, allowing general-purpose computations on powerful graphics hardware

High level languages

- C-like sintax
- Data types:
  - Vectors (1 to 4 floats, integers, booleans)
  - Matrices (2x2, 3x3, 4x4)
  - Arrays and Textures
- Conditions, loops, functions
- Vector-Matrix algebra
- Special instructions (Trigonometry, Exponentials, Geometry, Interpolation...)

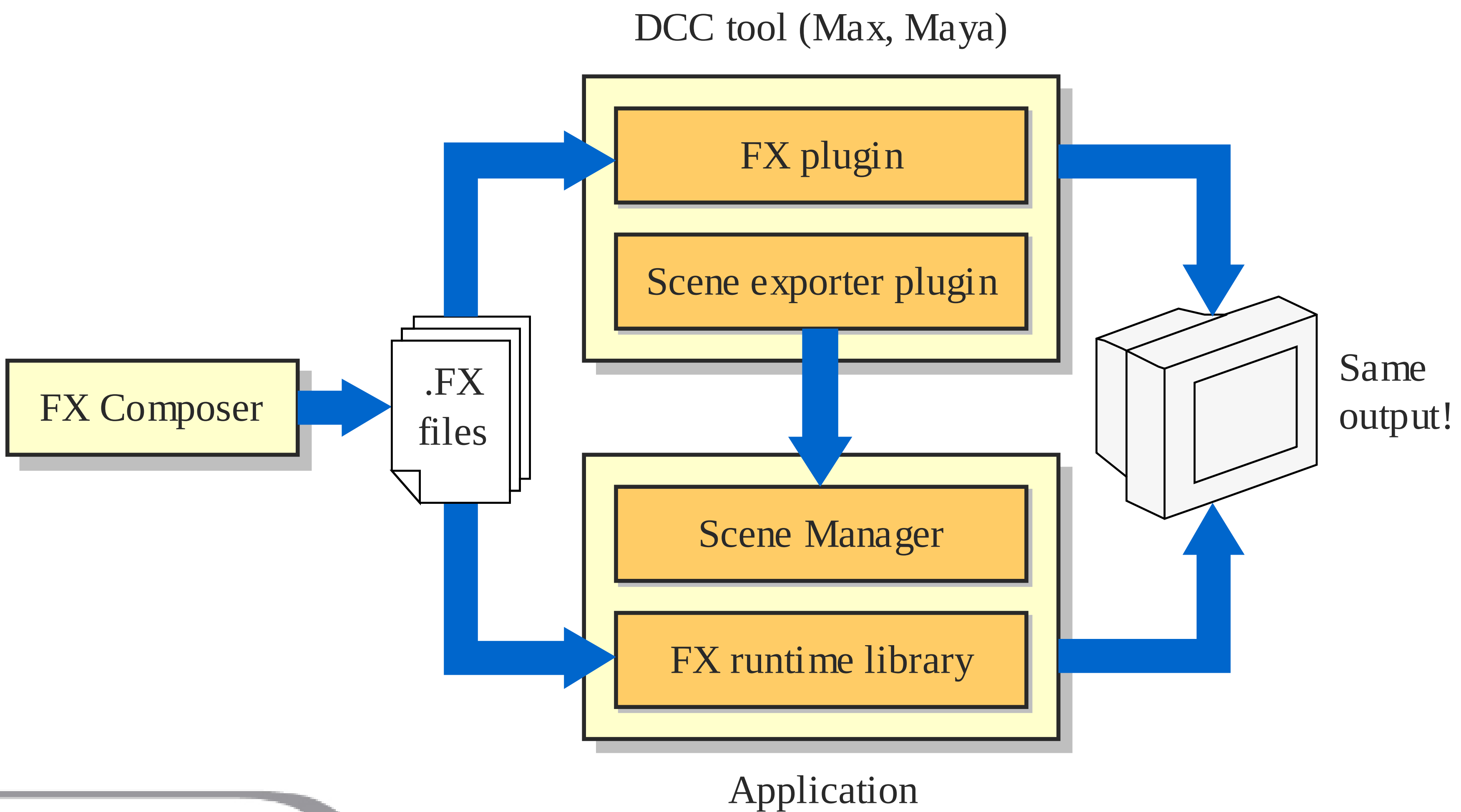
Runtime compilation

- Shaders will be compiled when the 3D application starts
- They will be validated and optimized for the current hardware
  - Direct3D natively compiles HLSL Shaders
  - OpenGL natively compiles GLSL Shaders
  - The *Cg Runtime Library* allows to compile Cg Shaders in both D3D and OGL applications
- Still, high level Shaders can be compiled down to ASM to be fully optimized at hand (for several hardware profiles)

Shading tools

- RenderMonkey (*ATI / 3Dlabs*)
  - Intuitive IDE
  - Supports GLSL
- Cg Toolkit (*nVidia*)
  - All you need to start with Cg
- FX Composer (*Microsoft / nVidia*)
  - FX files editor (best interchange format)
  - Best profiling and debugging tools
- Maya/Max plugins to test and tune shaders in your scene

Workflow


DCC tool (Max, Maya)
FX plugin
Scene exporter plugin
FX Composer
.FX files
Scene Manager
FX runtime library
Same output!
Application

How to learn Shaders

1. Start by using existing Shaders in your applications (no need to learn a shading language for this)
2. Try to tweak some parameters and equations, to see their meaning (*RenderMonkey* is perfect for this)
3. Don’t reinvent the weel: many effects are already implemented! Instead learn how to adapt them to your needs
4. Learn linear algebra and lighting models

References

- www.ati.com/developer
- developer.nvidia.com
- www.shadertech.com

- Cg Toolkit: User's Manual
- The OpenGL Shading Language
- The OpenGL Graphics System: A Specification