Fondamenti di Grafica Tridimensionale

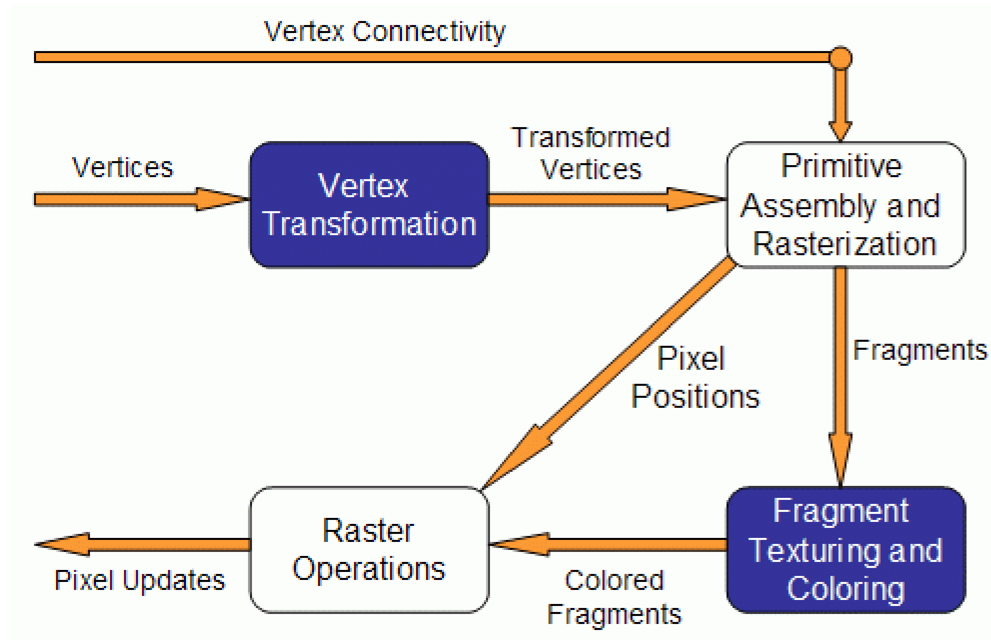
Paolo Cignoni
p.cignoni@isti.cnr.it
http://vcg.isti.cnr.it/~cignoni

Introduzione

- Riferimenti
  - Opengl 2.0 http://www.opengl.org/documentation/opengl_current_versio
  - GLSL official specs http://www.opengl.org/documentation/oglsl.html
  - GLSL stands for GL Shading Language
- Tutorials
  - http://www.lighthouse3d.com/opengl/glsl/index.php?intro
  - Il sito dell'orange book http://www.3dshaders.com/
  - Shader Designer http://www.typhoonlabs.com/index.php?action=developer.htr
  - Render Monkey http://www.ati.com/developer/rendermonkey/index.html

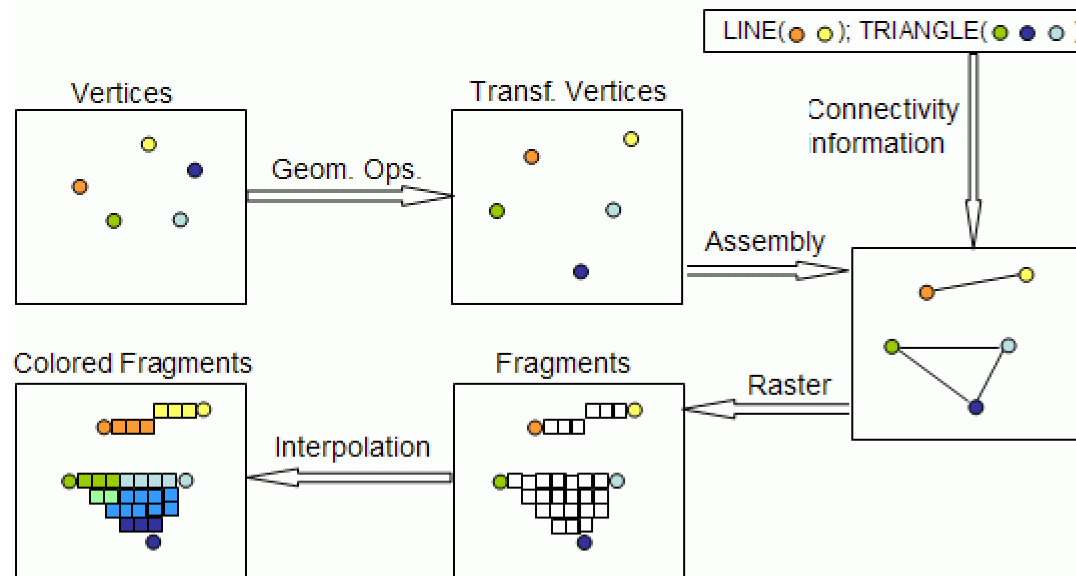
Pipeline Again

Molto semplificata



Pipeline Again

- Molto semplificata
-



Vertex Transformation

- Input
  - Vertex attributes (pos norm color ecc)
- Operation Performed
  - Vertex position transformation
  - Lighting computations per vertex
  - Generation and transformation of texture coordinates

Primitive Assembly e Raster

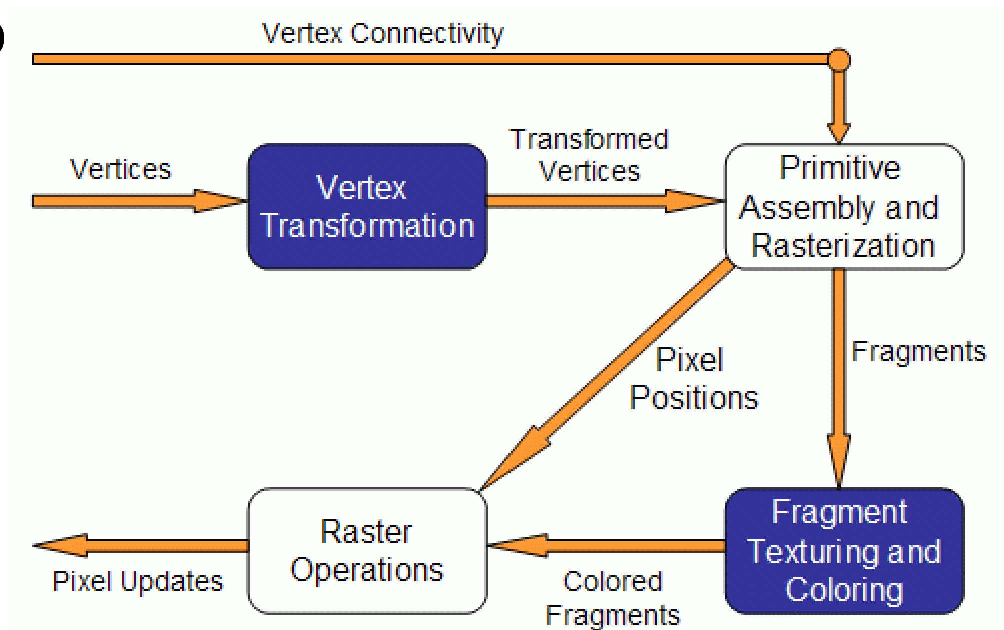
- Input
  - Transformed vertexes
  - Connectivity info
- Operation Performed
  - Clipping and backface culling
  - Determinazione posizione frammenti
  - Generazione per ogni frammento attributi interpolati

Fragment Texturing e Coloring

- Input
  - The pixels location
  - The fragments depth and color values
  - Tex coord arrivano gia' preparate...
- The common end result of this stage per fragment is a color value and a depth for the fragment

Shaders

- In pratica si rimpiazza alcune funzionalita
  - Vertex shaders may be written for the Vertex Transformation stage.
  - Fragment shaders replace the Fragment Texturing and Co



Vertex Processor

- The vertex processor is responsible for running the vertex shaders.
- The input for a vertex shader is the vertex data:
  - namely its position, color, normals, etc, depending on what the OpenGL application sends.

Vertex Processor

- vertex shader tasks:
  - Vertex position transformation using the modelview and projection matrices
  - Normal transformation, and if required its normalization
  - Texture coordinate generation and transformation
  - Lighting per vertex or computing values for lighting per pixel
  - Color computation

Vertex Processor

- No requirement to perform all the operations above,
  - When a vertex shader is used it becomes responsible for replacing all the needed functionality of this stage of the pipeline.

ShaderGen

- Un tool per generare glsl shaders che ricalchino esattamente il comportamento della fixed pipleine di opengl in un dato setup
- http://developer.3dlabs.com/downloads/s
  - Nota l'ultima ver richiede opengl2.0 drivers...

Vertex processor

- The vertex processor has no information regarding connectivity,
  - operations that require topological knowledge can't be performed in here.
  - not possible for a vertex shader to perform back face culling, since it operates on vertices and not on faces.
- The vertex processor processes vertices of a triangle individually
  - no clue of the remaining vertices.

Vertex Processor

- The vertex shader is responsible for at least writing a variable: gl_Position, usually transforming the vertex with the modelview and projection matrices.

Vertex Processor

- Access to OpenGL state,
  - so it can perform operations that involve lighting for instance, and use materials.
- It can also access textures
  - only available in the newest hardware
- No access to the frame buffer.

Fragment Processor

- The fragment processor is where the fragment shaders run:
  - Computing colors, and texture coordinates per pixel
  - Texture application
  - Fog computation
  - Computing normals if you want lighting per pixel

Fragment Processor

- Inputs
  - interpolated values computed in the previous stage of the pipeline such as vertex positions, colors, normals, etc...
- In the vertex shader these values are computed for each vertex.
- As in the vertex processor, fragment shader it replaces all the fixed functionality.
  - not possible to have a fragment shader texturing the fragment and leave the fog for the fixed functionality.

Fragment Processor

- The fragment processor operates on single fragments:
  - i.e. it has no clue about the neighboring fragments.
  - The shader has access to OpenGL state, similar to the vertex shaders, and therefore it can access for instance the fog color specified in an OpenGL application.

Fragment Processor

- Fragment shader can't change the pixel coordinate:
  - Recall that in the vertex processor the modelview and projection matrices can be used to transform the vertex.
- The fragment shader has access to the pixels location on screen but it can't change it
  - Si sa dove si va a finire ma non ci si puo' fare molto...

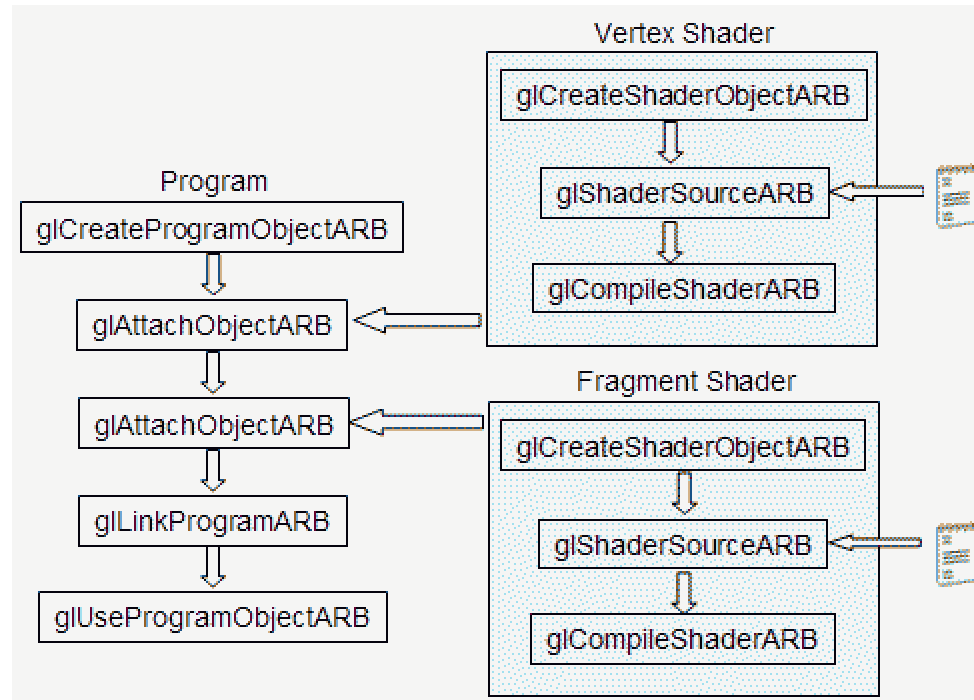
Fragment

- A fragment shader has two output options:
  - discard fragment, hence outputting nothing
  - to compute
    - gl_FragColor (the final color of the fragment), or
    - gl_FragData when rendering to multiple targets.
  - Depth can also be written although it is not required since the previous stage already has computed it.
- Notice that the fragment shader has no access to the frame buffer.
  - blending occur only after the fragment shader has run.

Using Shaders

- Ricetta per usare gli shader
- Two extensions are required:
  - GL_ARB_fragment_shader
  - GL_ARB_vertex_shader
- Similar to write a C program.
  - Each shader is like a C module, and it must be compiled separately, as in C.
  - The set of compiled shaders, is then linked into a program, exactly as in C.

Using Shaders



Using Shaders

1) Creating an object which will act as a shader container.

- GLhandleARB glCreateShaderObjectARB(GLenum shaderType);
    - shaderType - GL_VERTEX_SHADER_ARB or GL_FRAGMENT_SHADER_ARB.
- as many shaders as you want to add to a program,
    - only ONE main function for the set of vertex shaders and ONE main function for the set of fragment shaders in each single program.

Using Shaders2

2) add some source code.

The source code for a shader is a string array (can be reused)

- void glShaderSourceARB(GLhandleARB shader, int numOfStrings, const char **strings, int *lenOfStrings);
  - shader - the handler to the shader.
  - numOfStrings - the number of strings in the array.
  - strings - the array of strings.
  - lenOfStrings - an array with the length of each string, or NULL, meaning that the strings are NULL terminated

Using Shaders3

- Finally, the shader must be compiled. The function to achieve this is:
  - void glCompileShaderARB(GLhandleARB program);
- Parameters:
  - program - the handler to the program

Using Shader Program

1) creating an object which will act as a program container.

The function available for this purpose returns a handle for the container.

- GLhandleARB glCreateProgramObjectARB(void);

- *As many programs as you want.*
  - On rendering, you can switch from program to program, and even go back to fixed functionality during a single frame.
    - For instance you may want to draw a teapot with refraction and reflection shaders, while having a cube map displayed for background using OpenGL's fixed functionality.

Using Shader Programs

2) To attach a shader to a program use the function:

- void glAttachObjectARB(GLhandleARB program, GLhandleARB shader);
  - program - the handler to the program.
  - shader - the handler to the shader to attach.
- If you have a pair vertex/fragment of shaders you'll need to attach both to the program.
  - many shaders of the same type (vertex or fragment) attached to the same program
  - for each shader type only be one main function
  - the same vertex shader in several programs.

Using Shader Programs

3) Link the compiled shaders into program

- void glLinkProgramARB(GLhandleARB program);
- After the link operation the shader's source can be modified, and the shaders recompiled without affecting the program.

- To actually load and use the program
  - glUseProgramObjectARB(GLhandleARB prg);
    - prg - the handler to the program you want to use, or zero to return to fixed functionality
    - Each program is assigned an handler,
    - you can have as many programs linked and ready to use as you want (and your hardware allows).

Example

```
void setShaders() {

v = glCreateShaderObjectARB(GL_VERTEX_SHADER_ARB);
f = glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);

char *vs = textFileRead("toon.vert");
char *fs = textFileRead("toon.frag");

const char * vv = vs;
const char * ff = fs;

glShaderSourceARB(v, 1, &vv,NULL);
glShaderSourceARB(f, 1, &ff,NULL);

free(vs);free(fs);

glCompileShaderARB(v);
glCompileShaderARB(f);

p = glCreateProgramObjectARB();

glAttachObjectARB(p,v);
glAttachObjectARB(p,f);

glLinkProgramARB(p);
glUseProgramObjectARB(p);
}
```

Cleaning Up

- void glDetachObjectARB(GLhandleARB program, GLhandleARB shader);
  - program - The program to detach from.
  - shader - The shader to detach.
- Only shaders that are not attached can be deleted so this operation is not irrelevant.
  - void glDeleteObjectARB(GLhandleARB id);
  - In the case of a shader that is still attached , it is not deleted, but marked for deletion.
    - The delete operation will only be concluded when the shader is no longer attached to any program, (i.e.detached from all programs it was attached to)

Communicating

- La parte interessante
  - Come si passa le informazioni tra l'applicazione il vertex shader e il fragment shader
  - ONE WAY
    - the only output from a shader is to render to some targets, usually the color and depth buffers.
  - Opengl state
    - shader has access to part of the OpenGL state
- In this context, GLSL has two types of variable qualifiers
  - Uniform
  - Attribute

Uniform variables

- A uniform variable can have its value changed by primitive only, i.e.,
  - its value can't be changed between a glBegin / glEnd pair.
  - it can't be used for vertices attributes
  - for values that remain constant along a primitive, frame, or even the whole scene.
  - Uniform variables can be read (but not written) in both vertex and fragment shaders

Uniform Variables

- Get the memory location of the variable.
  - Note that this information is only available after you link the program.
- GLint glGetUniformLocationARB(GLhandleARB program, const char *name);
  - program - the handler to the program
  - name - the name of the variable.
  - The return value is the location of the variable, which can then be used to assign values to it:
    - void glUniform1fARB(GLint location, GLfloat v0);
    - GLint glUniform{1,2,3,4}fvARB(GLint location, GLsizei count, GLfloat *v);
    - GLint glUniformMatrix{2,3,4}fvARB(GLint location, GLsizei count, GLboolean transpose, GLfloat *v);
    -

Attribute Variables

- Set variables per vertex
- Attribute variables can only be read (not written) in a vertex shader
  - GLint *glGetAttribLocationARB*(GLhandleARB program,char *name);
  - void *glVertexAttrib2fARB*(GLint location, GLfloat v0, GLfloat v1);
  - void *glEnableVertexAttribArrayARB*(GLint loc);
  - void *glVertexAttribPointerARB*(GLint loc, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const void *pointer);

Data Types

- float bool int
- vec{2,3,4} bvec{2,3,4} ivec{2,3,4}
- mat2 mat3 mat4
- sampler1D, sampler2D, sampler3D samplerCube

- Note
  - No auto type cast
  - Many access modes
    - vec4 a = vec4(1.0,2.0,3.0,4.0);
    - float posX = a.x;
    - float posY = a[1];
    - vec2 posXY = a.xy;
    - float depth = a.w

Functions

- Simili al in c
  - No return of array
  - No recursion
  - Overload solo se lista param differente
- Qualifiers of parameters of a function:
  - in - for input parameters
  - out - for outputs of the function. The return statement is also an option for sending the result of a function.
  - inout - for parameters that are both input and output of a function

Varying Variables

- Defined in the vertex shader
- Are received linearly interpolated in the fragment shader
  - eg. to have a per-fragment normal
  -

Hello GLSL

Vertex shader

```
void main()
{
    gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix *
      gl_Vertex;
}
```

Fragment Shader

```
void main()
{
    gl_FragColor = vec4(0.4,0.4,0.8,1.0);
}
```

Hello GLSL

Vertex shader

```
void main()
{
    gl_Position = ftransform();
}
```

Fragment Shader

```
void main()
{
    gl_FragColor = vec4(0.4,0.4,0.8,1.0);
}
```

Hello GLSL

Vertex shader

```
attribute vec4 gl_Color;
varying vec4 gl_FrontColor; // writable on the vertex shader
varying vec4 gl_BackColor; // writable on the vertex shadervoid
void main()
{
    gl_FrontColor = gl_Color;
    gl_Position = ftransform();
}
```

Fragment Shader

```
varying vec4 gl_Color; // readable on the fragment shader
void main()
{
    gl_FragColor = gl_Color;
}
```

Toon shading minimo

Vertex shader

```
attribute vec4 gl_Color;
varying vec4 gl_FrontColor; // writable on the vertex shader
varying vec4 gl_BackColor; // writable on the vertex shadervoid
void main()
{
    gl_FrontColor = gl_Color;
    gl_Position = ftransform();
}
```

Fragment Shader

```
varying float intensity;
void main()
{
vec4 color;
if (intensity > 0.95) color = vec4(1.0,0.5,0.5,1.0);
else if (intensity > 0.5) color = vec4(0.6,0.3,0.3,1.0);
else if (intensity > 0.25)color = vec4(0.4,0.2,0.2,1.0);
else color = vec4(0.2,0.1,0.1,1.0);
gl_FragColor = color;
}
```