E V O L U T I O N  O F  T H E  O P E N G L  G R A P H I C S  S Y S T E M

Daniele Giannetti
Introduction
The OpenGL Pipeline

- OpenGL is a software interface towards the graphical hardware, a particular resource managed by the OS.
- It allows drawing of complex 3D objects using a sequence of operations performed (if possible) using specialized hardware components.
- This sequence of operations is often called **pipeline**.

Pipeline before OpenGL version 1.5:
OpenGL 1.5 (2003) introduced a set of extension to allow usage of shaders.

A shader is a program written in a specific programming language, compiled and executed on a particular pipeline stage instead of the default stage operations.

With current hardware, shaders are usually executed on GPU cores instead of normal CPU execution (which is considered a software fallback condition).

Benefits of shaders:
1. Improved flexibility during rendering (not only relying on predefined functions).
2. Improved performance (because the programmer is forced to work in a much more “hardware friendly” way).

GLSL is the OpenGL companion shading language, but it’s not the only one permitted in OpenGL (see also Cg).
OpenGL Evolution
OpenGL version 2.0 and 2.1

• The default pipeline in OpenGL 1.5 did not change from the previous versions.
• With version 2.0 (2004) the shader-related extensions were included in the core OpenGL specification (they were not extensions anymore).
• We will not discuss here the changes introduced with version 2.1 (2006), the default pipeline operations did not change.

NOTE:
The introduction of shaders and their inclusion in the core OpenGL specification slowly moved the fixed OpenGL pipeline (represented in slide 1) towards a flexible *shader-programmable structure*.
As the graphical API specifications (OpenGL and Direct3D) strongly influence the video card architecture engineered by manufacturers, nowadays we are moving towards *highly programmable parallel processors*.
This version released in 2008 introduced a major change known as the Deprecation Model.

**Deprecation Model**

All functions related to the fixed-function OpenGL pipeline were declared deprecated, to be removed in future versions. This effectively means that the fixed-function pipeline is being removed from the API.

Which features were deprecated?

- All direct draw calls (e.g. `glBegin()`, `glVertex3f()`, ..., `glEnd()`)
- Precompiled lists of OpenGL calls (a.k.a. display lists)
- Explicit lighting (e.g. `glLightfv()`, ...)
- Fixed-function pipeline effects (e.g. `glFogfv()`, ...)
- Functions used to handle matrices and transformation (e.g. `glRotatef()`, ...)
- ...
To write forward compatible OpenGL 3.0 code, all drawing must happen using non-deprecated functions:

1. Store vertices data *Vertex Buffer Objects (VBO)*: buffers owned and managed by the GL implementation and usually stored in video memory.
2. Always use shaders (at least one vertex shader and one fragment shader) to specify the GL pipeline operations (the default pipeline operations are deprecated).

To avoid usage of deprecated OpenGL calls, the programmer may request to the operating system a *forward compatible* context when creating the OpenGL 3.0 context. A forward compatible context will not support deprecated features.

Programming in a forward compatible way introduces more *flexibility* (because of shaders) and better *performances* (because of the reduced number of OpenGL calls during drawing), but the performance will ultimately depend upon the OpenGL implementation (part of the display drivers).
• Released March 24th, 2009.
• As predicted in the Deprecation Model, all deprecated features were removed from the core OpenGL specification and moved in an extension (the GL_ARB_compatibility extension).
• The GL_ARB_compatibility extension should only be used to maintain compatibility with old applications.

The OpenGL pipeline must now be programmed using shaders:
OpenGL Evolution
OpenGL version 3.2 and 3.3 [1]

• Version 3.2 was released August 3rd, 2009.
• Fixed-function pipeline features were re-introduced in the OpenGL core specification, but in a specific profile.

OpenGL Profile
An OpenGL profile is a subset of OpenGL features to be used for specific application domains.

Two profiles were specified (the used one is chosen at context creation):
• **Core Profile**
  Supports all non-deprecated features and will include all future changes (recommended for new applications).
• **Compatibility Profile**
  Supports all deprecated and non-deprecated features, support for this profile is *optional* in OpenGL 3.2 implementations (used to maintain compatibility with old applications).
OpenGL Evolution
OpenGL version 3.2 and 3.3 [2]

- **Geometry shaders** were included in the core OpenGL specification, these shaders can modify primitives assembled by the primitive assembly pipeline stage by creating and moving vertices.
- Before version 3.2, these objects were only available as extensions.
- Version 3.3 (release in March 2010) maintains the profile separation, we will not examine other features introduced in this update.

OpenGL 3.3 is now supported by all graphics card drivers from major manufacturers (provided they are being updated regularly).

Newer versions (OpenGL 4.0 and 4.1) are only supported on recent hardware devices because they introduce some features that need explicit hardware support in order to run efficiently.

New version of the OpenGL API are usually released with new specifications for the GLSL shading language.
OpenGL 3.2 and 3.3 pipeline (high level of detail):
• OpenGL 4.0 was released together with version 3.3 of the API.
• Introduced double-precision floating-point numbers to be used in shaders, this can be very useful in some particular shading algorithm.
• Introduced two new programmable pipeline stages, where two new types of shaders are executed:
  These two shader are used to
  1. **Tessellation control shader**
  2. **Tessellation evaluation shader**
These two shader are used to create new primitives during rendering (tessellation) in an efficient way. This could also be done using geometry shaders, but in a much less efficient way.
• OpenGL 4.1 was released July 25\textsuperscript{th}, 2010.
• We will not examine the features that have been introduced in this OpenGL version.
A Simple OGL 3.3 application
Drawing a blue rectangle [1]

- Because all simple direct draw calls are unavailable, this trivial application becomes not so trivial.

1. Create shaders to be used during rendering (GLSL 3.30):

```glsl
// Vertex Shader
#version 330 core
uniform mat4 mat;
in vec4 mPosition;
in vec4 mColor;
out vec4 fColor;
void main(void) {
    fColor = mColor;
    gl_Position = mat * mPosition;
}
```

```glsl
// Fragment Shader
#version 330 core
in vec4 fColor;
out vec4 outColor;
void main(void) {
    outColor = fColor;
}
```
A Simple OGL 3.3 application
Drawing a blue rectangle [2]

2. Load and compile the shaders:

```c
<string is a const char* containing the vertex shader source>
GLint ret;
GLuint verShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(verShader, 1, (const GLchar**)&string, NULL);
glCompileShader(verShader);
glGetShaderiv(verShader, GL_COMPILE_STATUS, &ret);
if(ret == GL_FALSE) { <error handling> }
</Similar procedure for the fragment shader>
GLuint programID = glCreateProgram();
glAttachShader(programID, verShader);
glAttachShader(programID, fragShader);
glLinkProgram(programID);
glGetProgramiv(programID, GL_LINK_STATUS, &ret);
if(ret == GL_FALSE) { <error handling> }
```
A Simple OGL 3.3 application
Drawing a blue rectangle [3]

3. Set attribute index and get uniform locations:

```cpp
// Bind shader attributes indices
glBindAttribLocation(programID, INDEX_POS, "mPosition");
glBindAttribLocation(programID, INDEX_COL, "mColor");
// Get shader uniform locations
GLint matLoc = glGetUniformLocation(programID, "mat");
// Bind fragment shader output index
glBindFragDataLocation(programID, 0 /* first color attachment */, "outColor");
```

- The last call will bind the output to the first color attachment of the default framebuffer, that is the screen.
- `INDEX_POS` and `INDEX_COL` will be used as indices where to connect the VBOs to before rendering.
- `matLoc` will be used to load the uniform value before rendering.
4. Prepare data that will be stored in the VBOs.

```c
GLfloat vertices[16] = {
    0.5f, -0.5f, 0.0f, 1.0f,
    0.5f, 0.5f, 0.0f, 1.0f,
    -0.5f, -0.5f, 0.0f, 1.0f,
    -0.5f, 0.5f, 0.0f, 1.0f
};

GLfloat colors[16] = {
    0.0f,0.0f,1.0f,1.0f,
    0.0f,0.0f,1.0f,1.0f,
    0.0f,0.0f,1.0f,1.0f,
    0.0f,0.0f,1.0f,1.0f
};
```
5. Generate and fill VBOs, and group them in a VAO (*Vertex Array Object*):

```c
GLuint verarray; GLuint buffer[2]; // VAO and VBOs identifiers
glGenVertexArrays(1, &verarray); // VAO
glBindVertexArray(verarray);
glGenBuffers(2, buffer); // VBOs (positions and colors)
// Fill and activate a VBO for the vertex positions
glBindBuffer(GL_ARRAY_BUFFER, buffer[0]);
gleBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat)*4*4, 
             vertices, GL_STATIC_DRAW);
gleVertexAttribPointer(INDEX_POS, 4, GL_FLOAT, GL_FALSE, 0, 0);
gleEnableVertexAttribArray(INDEX_POS);
// Fill and activate a VBO for the vertex colors
glBindBuffer(GL_ARRAY_BUFFER, buffer[1]);
gleBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat)*4*4, 
             colors, GL_STATIC_DRAW);
gleVertexAttribPointer(INDEX_COL, 4, GL_FLOAT, GL_FALSE, 0, 0);
gleEnableVertexAttribArray(INDEX_COL);
```
A Simple OGL 3.3 application
Drawing a blue rectangle [6]

6. Render the rectangle:

```c
// Inside the rendering cycle
<framebuffer cleanup>
GLfloat matrix[] = {
    1.0f, 0.0f, 0.0f, 0.0f,
    0.0f, 1.0f, 0.0f, 0.0f,
    0.0f, 0.0f, 1.0f, 0.0f,
    0.0f, 0.0f, 0.0f, 1.0f
};
glUseProgram(programID);
glBindVertexArray(verarray);
glUniformMatrix4fv(matLoc, 1, GL_FALSE, matrix);
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4); // single draw call
<swap buffers>
```
A Simple OGL 3.3 application
Drawing a blue rectangle [7]
References

• The OpenGL registry (http://www.opengl.org/registry/)
Thank you!