Some Abbreviations and their Definitions used in Computer Graphics

Abbreviation Definition

CAD Computer-Aided Design

CAM Computer-Aided Manufacturing

API **Application Programming Interface**

/* Although it is possible for the OpenGL API to be implemented entirely in software, in OpenGL it is designed to be implemented mostly or entirely in hardware. */

OpenGL **Open Graphics Library**

GPU **Graphics Processing Unit**

GLUT **Graphics Library Utility Tool**

/* library of utilities for OpenGL programs, which primarily perform system-level I/O with the host operating system. Functions performed include window definition, window control, and monitoring of keyboard and mouse input. Routines for drawing a number of geometric primitives (both in solid and wireframe mode) are also provided, including cubes, spheres and the Utah teapot. GLUT also has some limited support for creating pop-up menus. */

GLU **OpenGL Utility Library**

/*It consists of a number of functions that use the base OpenGL library to provide higher-level drawing routines from the more primitive routines that OpenGL provides. */

GLEW **The OpenGL Extension Wrangler Library (GLEW)**

/* It is a cross-platform C/C++ library that helps in querying and loading OpenGL extensions. GLEW provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform. */

GLUI **OpenGL User Interface Library**

/* It is a C++ user interface library based on the OpenGL Utility Toolkit (GLUT) which provides controls such as buttons, checkboxes, radio buttons, and spinners to OpenGL applications. */

GLES **OpenGL Library for Embedded Systems**

Supersampling is a spatial anti-aliasing method, i.e. a method used to remove aliasing (jagged and pixelated edges, colloquially known as "jaggies") from images rendered in computer games or other computer programs that generate imagery. Aliasing occurs because unlike real-world objects, which have continuous smooth curves and lines, a computer screen shows the viewer a large number of small squares. These 'pixels' are all the same size, and each one has a single color. A line can only be shown as a collection of pixels, and therefore appears jagged unless it is perfectly horizontal or vertical. The aim of supersampling is to reduce this effect. Color samples are taken at several instances inside the pixel (not just at the center as normal), and an average color value is calculated. This is achieved by rendering the image at a much higher resolution than the one being displayed, then shrinking it to the desired size, using the extra pixels for calculation. The result is a downsampled image with smoother transitions from one line of pixels to another along the edges of objects.

The number of samples determines the quality of the output.

CUDA is a parallel computing platform and application programming interface (API) model created by NVIDIA. It allows software developers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing – an approach known as GPGPU. The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements. The CUDA platform is designed to work with programming languages such as C, C++ and Fortran.

2D Shaders act on digital images, also called textures in computer graphics work. They modify attributes of pixels.

3D Shaders act on 3D models or other geometry but may also access the colors and textures used to draw the model or mesh. Vertex shaders are the oldest type of 3d shader, generally modifying on a per-vertex basis.

Vertex shaders are the most established and common kind of 3d shader and are run once for each vertex given to the graphics processor. The purpose is to transform each vertex's 3D position in virtual space to the 2D coordinate at which it appears on the screen (as well as a depth value for the Z-buffer). Vertex shaders can manipulate properties such as position, color and texture coordinate, but cannot create new vertices.

Geometry shaders are a relatively new type of shader, introduced in Direct3D 10 and OpenGL 3.2; formerly available in OpenGL 2.0+ with the use of extensions. This type of shader can generate new graphics primitives, such as points, lines, and triangles, from those primitives that were sent to the beginning of the graphics pipeline. Geometry shader programs are executed after vertex shaders.

Tessellation shaders

As of OpenGL 4.0 and Direct3D 11, a new shader class called a tessellation shader has been added. It adds two new shader stages to the traditional model. Tessellation control shaders (also known as hull shaders) and tessellation evaluation shaders (also known as Domain Shaders), which together allow for simpler meshes to be subdivided into finer meshes at run-time according to a mathematical function.

Shaders and Parallel Programming

Shaders are written to apply transformations to a large set of elements at a time, for example, to each pixel in an area of the screen, or for every vertex of a model. This is well suited to parallel processing, and most modern GPUs have multiple shader pipelines to facilitate this, vastly improving computation throughput.

Programming

The language in which shaders are programmed depends on the target environment. The official OpenGL and OpenGL ES shading language is OpenGL Shading Language, also known as GLSL, and the official Direct3D shading language is High Level Shader Language, also known as HLSL.

However, Cg is a third-party shading language developed by Nvidia that outputs both OpenGL and Direct3D shaders. Apple released its own shading language called Metal Shading Language as part of the Metal framework.

OpenGL Shading Language (abbreviated: **GLSL** or **GLslang**), is a high-level shading language based on the syntax of the C programming language.

The **High-Level Shader Language** or **High-Level Shading Language** (**HLSL**) is a proprietary shading language developed by Microsoft for the Direct3D 9 API to augment the shader assembly language, and went on to become the required shading language for the unified shader model of Direct3D 10 and higher.

HLSL is analogous to the GLSL shading language used with the OpenGL standard. It is very similar to the Nvidia Cg shading language, as it was developed alongside it. HLSL shaders can enable profound speed and detail increases as well as many special effects in both 2d and 3d computer graphics.

Rasterization is the task of taking an image described in a vector graphics format (shapes) and converting it into a raster image (pixels or dots) for output on a video display or printer, or for storage in a bitmap file format.

Shader Examples

A sample trivial GLSL vertex shader

This transforms the input vertex the same way the fixed-function pipeline would.

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

Note that ftransform() is no longer available since GLSL 1.40 and GLSL ES 1.0. Instead, the programmer has to manage the projection and modelview matrices explicitly in order to comply with the new OpenGL 3.1 standard.

```
#version 140

uniform Transformation {
	mat4 projection_matrix;
	mat4 modelview_matrix;
};

in vec3 vertex;

void main(void) {
	gl_Position = projection_matrix * modelview_matrix * vec4(vertex, 1.0);
}
```

A sample trivial GLSL tessellation shader

This is a simple pass-through Tessellation Control Shader for the position.

```
#version 400

layout(vertices=3) out;

void main(void) {
	gl_out[gl_InvocationID].gl_Position = gl_in[gl_InvocationID].gl_Position;

	gl_TessLevelOuter[0] = 1.0;
	gl_TessLevelOuter[1] = 1.0;
	gl_TessLevelOuter[2] = 1.0;
	gl_TessLevelInner[0] = 1.0;
	gl_TessLevelInner[1] = 1.0;
}
```

This is a simple pass-through Tessellation Evaluation Shader for the position.

```
#version 400

layout(triangles,equal_spacing) in;

void main(void) {
	vec4 p0 = gl_in[0].gl_Position;
	vec4 p1 = gl_in[1].gl_Position;
	vec4 p2 = gl_in[2].gl_Position;

	vec3 p = gl_TessCoord.xyz;

	gl_Position = p0*p.x + p1*p.y + p2*p.z;
}
```

A sample trivial GLSL geometry shader

This is a simple pass-through shader for the color and position.

```
#version 120
#extension GL_EXT_geometry_shader4 : enable

void main(void) {
	for (int i = 0; i < gl_VerticesIn; ++i) {
		gl_FrontColor = gl_FrontColorIn[i];
		gl_Position = gl_PositionIn[i];
		EmitVertex();
	}
}
```

Since OpenGL 3.2 with GLSL 1.50 geometry shaders were adopted into core functionality which means there is no need to use extensions. However, the syntax is a bit different. This is a simple version 1.50 pass-through shader for vertex positions (of triangle primitives):

```
#version 150

layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;

void main(void) {
	for (int i = 0; i < gl_in.length(); ++i) {
		gl_Position = gl_in[i].gl_Position;
		EmitVertex();
	}
	EndPrimitive();
```

```
}
```

A sample trivial GLSL fragment shader

This produces a red fragment.

```
#version 120

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

In GLSL 1.30 and later you can do

```
glBindFragDataLocation(Program, 0, "MyFragColor");
```

where:

- Program – your shader program's handle;
- 0 – color buffer number, associated with the variable; if you are not using multiple render targets, you must write zero;
- "MyFragColor" – name of the output variable in the shader program, which is associated with the given buffer.

```
#version 150

out vec4 MyFragColor;

void main(void) {
	MyFragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```