

A Survey Paper on OpenGL ES

Ms. Warsha M.Choudhari

Professor, Information Technology
Datta Meghe Institute of Engineering, Technology &
Research, Wardha, India
warshaswaradhni@gmail.com

Ms. Rinku Rajankar

Professor, Computer Science & Engineering
ITM College of Engineering, Nagpur, India
rinkurajankar@gmail.com

Abstract—In this survey paper introduce the OpenGL (Graphics Library) Es graphics system, which is very useful for embedded system. It means that software interface to graphics hardware. It allows you to create interactive programs that produce color images of moving, three dimensional objects. Using OpenGL, we can control computer graphics technology to produce realistic pictures, or ones that depart from reality in imaginative ways.

Keywords: *Embedded System, OpenGL Es*

1. INTRODUCTION

OpenGL ES is an application programming interface (API) for advanced 3D graphics targeted at handheld and embedded devices such as cell phones, personal digital assistants (PDAs), consoles, appliances, vehicles, and avionics.

There are two standard 3D APIs, DirectX and OpenGL.

DirectX is the de facto standard 3D API for any system running the Microsoft Windows operating system and is used by the majority of 3D games on that platform.

OpenGL is a cross-platform standard 3D API for desktop systems running Linux, various flavors of UNIX, Mac OS X, and Microsoft Windows. The API is used by user interfaces as in Mac OS X, workstation computer-aided design (CAD) applications like CATIA, and digital content creation applications such as Maya and Soft Image XSI.

It is mostly used for games such as the Doom and Quake series. Due to the widespread adoption of OpenGL as a 3D API, it made sense to start with the desktop OpenGL API in developing an open standard 3D API for handheld and embedded devices and modifying it to meet the needs and constraints of the handheld and embedded device space. The device constraints that OpenGL ES addresses are:

- very limited processing capabilities and memory availability
- low memory bandwidth, sensitivity to power consumption, and
- lack of floating-point hardware.

The working group used the following criteria in the definition of the OpenGL ES specification: [1]

- The OpenGL API is very large and complex and the goal of the OpenGL ES working group was to create an API suitable for constrained devices. To achieve this goal, the working group removed any redundancy from the OpenGL API. In any case where there was more than one way of performing the same operation, the most useful

method was taken and the redundant techniques were removed. Example is specifying geometry, where in OpenGL an application can use immediate mode, display lists, or vertex arrays. In OpenGL ES, only vertex arrays exist and immediate mode and display lists were removed.

- Removing redundancy was an important goal, but maintaining compatibility with OpenGL was also important. As much as possible, OpenGL ES was designed such that applications that were written to the embedded subset of functionality in OpenGL would also run on OpenGL ES. The reason this was an important goal is it allows developers to leverage both APIs and develop applications and tools that use the common subset of functionality. Although this was an important goal, there are cases where it has deviated, especially with OpenGL ES 2.0.

- New features were introduced to address specific constraints of handheld and embedded devices. For example, to reduce the power consumption and increase the performance of shaders, precision qualifiers were introduced to the shading language.

- The designers of OpenGL ES aimed to ensure a minimum set of features for image quality. Most handheld devices have limited screen sizes, making it essential that the quality of the pixels drawn on the screen is as good as possible.

- The OpenGL ES working group wanted to ensure that any OpenGL ES implementation would meet certain acceptable and agreed-on standards for image quality, correctness, and robustness. This is done by developing appropriate conformance tests that an OpenGL ES implementation must pass to be considered compliant.

2. OPEN GL ES 1.0 vs OpenGL ES 2.0

OpenGL ES 1.0 uses a fixed pipeline, which is a fancy way of saying we use built-in functions to set lights, vertexes, colors, cameras, and more.

OpenGL ES 2.0 uses a programmable pipeline, which is a fancy way of saying all those built-in functions go away, and we have to write everything yourself.

3. OpenGL ES 2.0

OpenGL ES 2.0 implements a graphics pipeline with programmable shading and consists of two specifications:

There are two types of shaders:

- (i) OpenGL ES 2.0 API specification
- (ii) OpenGL ES Shading Language Specification (OpenGL ES SL).

Figure shows the OpenGL ES 2.0 graphics pipeline. The shaded boxes in Figure indicate the programmable stages of the pipeline in OpenGL ES

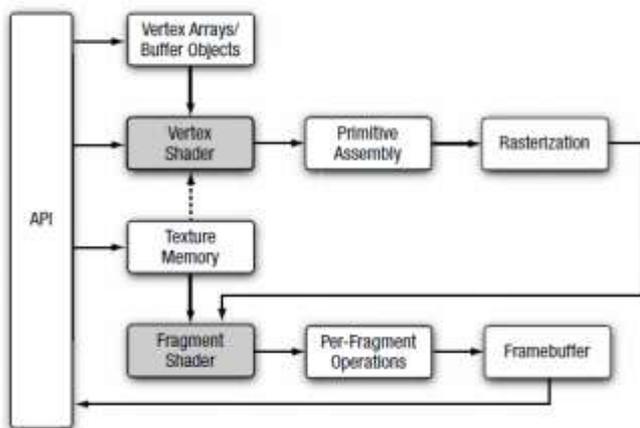


Fig 3.1 OpenGL ES 2.0 graphics pipeline

3.1 Vertex Shader

Vertex shaders are programs that get called once per vertex in your scene. So if you are rendering a simple scene with a single square, with one vertex at each corner, this would be called four times. Its job is to perform some calculations such as lighting, geometry transforms, etc., figure out the final position of the vertex, and also pass on some data to the fragment shader.

The inputs to the vertex shader consist of the following:

- Attributes—Per-vertex data supplied using vertex arrays.
- Uniforms—Constant data used by the vertex shader.
- Samplers—A specific type of uniforms that represent textures used by the vertex shader.
- Shader program—Vertex shader program source code or executable that describes the operations that will be performed on the vertex.

The outputs of the vertex shader are called varying variables. In the primitive rasterization stage, the varying values are calculated for each generated fragment and are passed in as inputs to the fragment shader. The mechanism used to generate a varying value for each fragment from the varying values assigned to each vertex of the primitive is

called interpolation. The outputs of the vertex shader are called varying variables. In the primitive rasterization stage, the varying values are calculated for each generated fragment and are passed in as inputs to the fragment shader. The inputs and outputs of the vertex shader are shown in Figure.

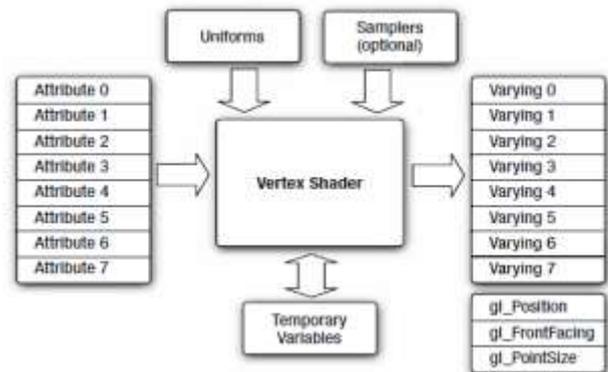


Fig 3.2 Inputs and outputs of the vertex shader

Vertex shaders can be used for traditional vertex-based operations such as transforming the position by a matrix, computing the lighting equation to generate a per-vertex color, and generating or transforming texture coordinates. Alternately, because the vertex shader is specified by the application, vertex shaders can be used to do custom vertex transformations.

Primitive Assembly

After the vertex shader, the next stage in the pipeline is primitive assembly. A primitive is a geometric object that can be drawn using appropriate drawing commands in OpenGL ES. These drawing commands specify a set of vertex attributes that describes the primitive's geometry and a primitive type. Each vertex is described with a set of vertex attributes. These vertex attributes contain information that the vertex shader uses to calculate a position and other information that can be passed to the fragment shader such as its color and texture coordinates.

In the primitive assembly stage, the shaded vertices are assembled into individual geometric primitives that can be drawn such as a triangle, line, or point-sprite. For each primitive, it must be determined whether the primitive lies within the view frustum (the region of 3D space that is visible on the screen). If the primitive is not completely inside the view frustum, the primitive might need to be clipped to the view frustum. If the primitive is completely outside, it is discarded. After clipping, the vertex position is converted to screen coordinates. A culling operation can also be performed that discards primitives based on whether they face forward or backward. After clipping and culling, the primitive is ready to be passed to the next stage of the pipeline, which is the rasterization stage.

Rasterization

The next stage, shown in Figure is the rasterization phase where the appropriate primitive (point-sprite, line, or triangle) is drawn. Rasterization is the process that converts primitives into a set of two dimensional fragments, which are processed by the fragment shader. These two-dimensional fragment represent pixel s that can be drawn on the screen.

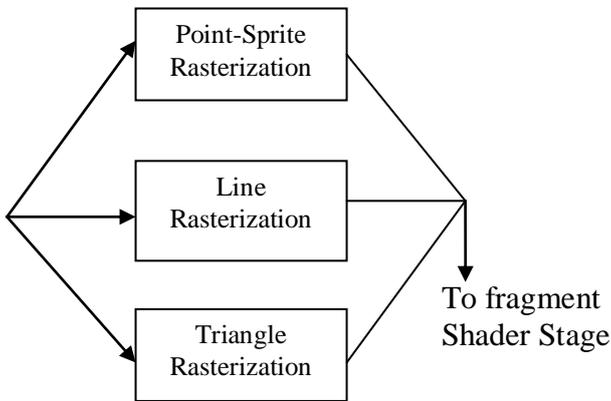


Fig 3.3 Rasterization phase

3.2 Fragment Shader

Fragment shaders are programs that get called once per pixel in your scene. So if you're rendering that same simple scene with a single square, it will be called once for each pixel that the square covers. Fragment shaders can also perform lighting calculations, etc, but their most important job is to set the final color for the pixel.

The fragment shader implements a general-purpose programmable method for operating on fragments. The fragment shader, as shown in Figure is executed for each generated fragment by the rasterization stage and takes the following inputs: Varying variables—Outputs of the vertex shader that are generated by the rasterization unit for each fragment using interpolation.

- Uniforms—Constant data used by the fragment shader.
- Samplers—A specific type of uniforms that represent textures used by the fragment shader.
- Shader program—Fragment shader program source code or executable that describes the operations that will be performed on the fragment. The fragment shader can either discard the fragment or generate a color value referred to as `gl_FragColor`. The color, depth, stencil, and screen coordinate location (xw, yw) generated by the rasterization stage become inputs to the per-fragment operations stage of the OpenGL ES 2.0 pipeline.[1]

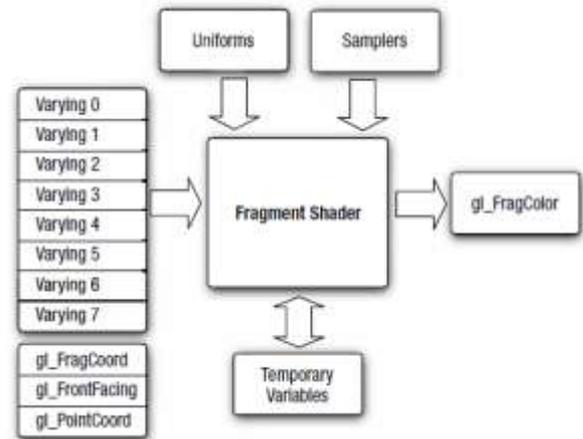
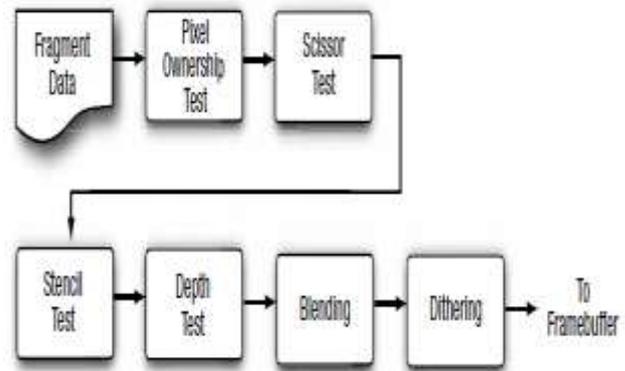


Fig 3.4 Fragment shader

Per-Fragment Operations

After the fragment shader, the next stage is per-fragment operations. A fragment produced by rasterization with (xw, yw) screen coordinates can only modify the pixel at location (xw, yw) in the frame buffer. Figure describes the OpenGL ES 2.0 per-fragment operations stage.



3.5 Per-fragment operations

The per-fragment operations stage performs the following functions (and tests) on each fragment:

- Pixel ownership test—This test determines if the pixel at location (xw, yw) in the frame buffer is currently owned by OpenGL ES. This test allows the window system to control which pixels in the frame buffer belong to the current OpenGL ES context. For example, if a window displaying the OpenGL ES frame buffer window is obscured by another window, the windowing system may determine that the obscured pixels are not owned by the OpenGL ES context and therefore might not be displayed at all.
- Scissor test—The scissor test determines if (xw, yw) lies within the scissor rectangle defined as part of the OpenGL ES state. If the fragment is outside the scissor region, the fragment is discarded.
- Stencil and depth tests—These perform tests on the stencil and depth value of the incoming fragment to determine if the fragment should be rejected or not.

- Blending—Blending combines the newly generated fragment color value with the color values stored in the frame buffer at location (xw, yw).

- Dithering—Dithering can be used to minimize the artifacts that can occur from using limited precision to store color values in the frame buffer.

At the end of the per-fragment stage, either the fragment is rejected or a fragment color, depth, or stencil value is written to the frame buffer at location (xw, yw). The fragment color, depth, and stencil values are written depending on whether the appropriate write masks are enabled or not. Write masks allow finer control over the color, depth, and stencil values written into the appropriate buffers. For example, the write mask for the color buffer could be set such that no red values get written into the color buffer.

In addition, OpenGL ES 2.0 also provides an interface to read back the pixels from the frame buffer. Only pixels can be read back from the color buffer. The depth and stencil values cannot be read back.

4. CONCLUSION

It allows you to create interactive programs that produce color images of moving, three dimensional objects.

The iPhone 3GS brings OpenGL ES 2.0—programmable hardware into millions of consumer's hands.

5. REFERENCES

- [1] Aftab Munshi, DanGinsburg, Dave Shreiner, "OpenGL Es 2.0 Programming Guide", ISBN-13: 978-0-321-50279-7.
- [2] Nakhoon Baek and KwanHee Yoo "Providing Direct3D Features over the Desktop OpenGL", International Journal of Smart Home Vol. 5, No. 4, October, 2011