GLSL Programming/OpenGL ES 2.0 Pipeline

The OpenGL ES 2.0 pipeline is important for GLSL shaders in OpenGL ES 2.0 and WebGL. It is also very similar to the OpenGL 2.0 pipeline without many of the features that were deprecated in newer versions of OpenGL. Therefore, the OpenGL ES 2.0 pipeline is not only highly relevant for programmers of mobile graphics using OpenGL ES 2.0 and web-based 3D graphics using WebGL, but also a very good starting point to learn about desktop-based 3D graphics using OpenGL, including 3D graphics in game engines such as Blender, Unity and Torque 3D.

Parallelism in the OpenGL Pipeline

GPUs are highly parallel processors. This is the main reason for their performance. In fact, they implement two kinds of parallelism: vertical and horizontal parallelism:

- **Vertical parallelism** describes parallel processing at different **stages of a pipeline**. This concept was also crucial in the development of the assembly line at Ford Motor Company: many workers can work in parallel on rather simple tasks. This made mass production (and therefore mass consumption) possible. In the context of processing units in GPUs, the simple tasks correspond to less complex processing units, which save costs and power consumption.
- **Horizontal parallelism** describes the possibility to process work in **multiple pipelines**. This allows for even more parallelism than the vertical parallelism in a single pipeline. Again, the concept was also employed at Ford Motor Company and in many other industries. In the context of GPUs, horizontal parallelism of the graphics pipeline was an important feature to achieve the performance of modern GPUs.

Ford assembly line, 1913.

Assembly plant of the Bell Aircraft Corporation with multiple parallel assembly lines, ca. 1944.

The following diagram shows an illustration of vertical parallelism (processing in stages represented by boxes) and horizontal parallelism (multiple processing units for each stage represented by multiple arrows between boxes).

Vertex Data	e.g. triangle meshes provided by 3D modeling tools
↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓	many vertices are processed in parallel
Vertex Shader	a small program in GLSL is applied to each vertex
↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓	

Primitive Assembly	setup of primitives, e.g. triangles, lines, and points
↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓	many primitives are processed in parallel
Rasterization	interpolation of data for all pixels covered by the primitive (e.g. triangle)
↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓	many fragments (corresponding to pixels) are processed in parallel
Fragment Shader	a small program in GLSL is applied to each fragment (i.e. covered pixel)
↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓	
Per-Fragment Operations	configurable operations on each fragment (i.e. covered pixel)
↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓	results of many fragments are written in parallel to the framebuffer
Framebuffer	array of pixels in which the computed fragment colors are stored

In the following diagrams, there is only one arrow between any two stages. However, it should be understood that GPUs usually implement the graphics pipeline with massive horizontal parallelism. Only software implementations of OpenGL, e.g. Mesa 3D (see the Wikipedia entry), usually implement a single pipeline.

Programmable and Fixed-Function Stages

The pipelines of OpenGL ES 1.x and core OpenGL 1.x are configurable fixed-function pipelines, i.e. there is no possibility to include programs in these pipelines. In OpenGL (ES) 2.0 two stages (the vertex shader and the fragment shader stage) of the pipeline are programmable, i.e. small programs (shaders) written in GLSL are applied in these stages. In the following diagram, programmable stages are represented by green boxes, fixed-function stages are represented by gray boxes, and data is represented by blue boxes.

Vertex Data	e.g. triangle meshes provided by 3D modeling tools
↓	
Vertex Shader	a small program in GLSL is applied to each vertex

↓	
Primitive Assembly	setup of primitives, e.g. triangles, lines, and points
↓	
Rasterization	interpolation of data (e.g. color) for all pixels covered by the primitive
↓	
Fragment Shader	a small program in GLSL is applied to each fragment (i.e. covered pixel)
↓	
Per-Fragment Operations	configurable operations on each fragment (i.e. covered pixel)
↓	
Framebuffer	array of pixels in which the computed fragment colors are stored

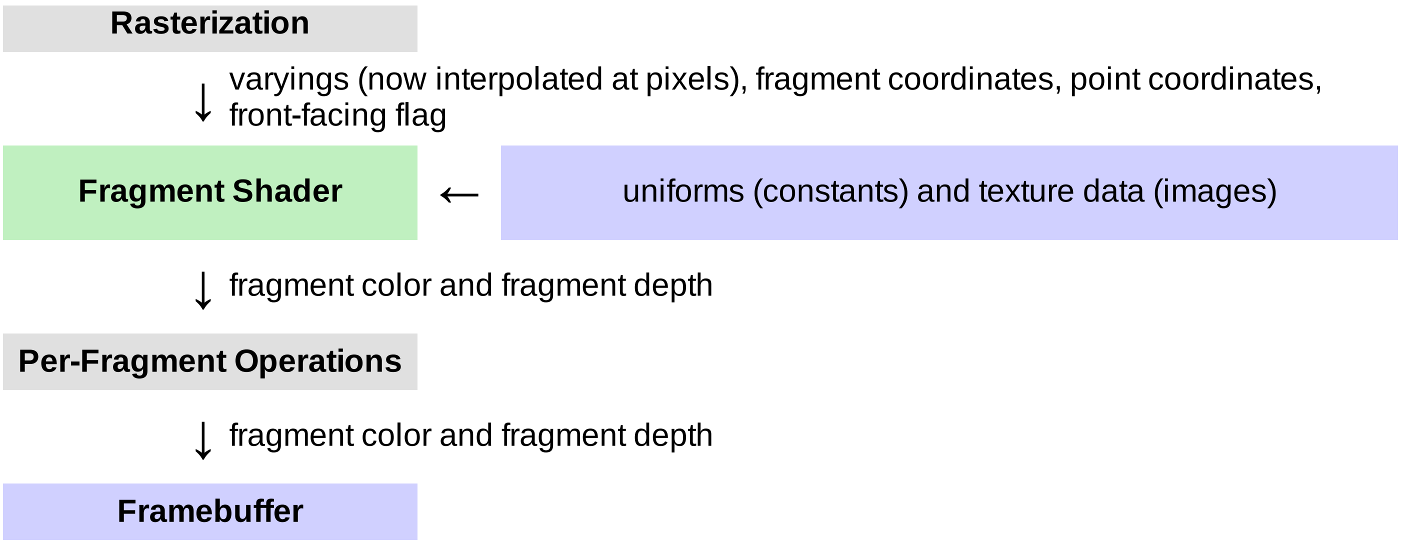
The vertex shader and fragment shader stages are discussed in more detail in the platform-specific tutorials. The rasterization stage is discussed in Section “Rasterization” and the per-fragment operations in Section “Per-Fragment Operations”.

The primitive assembly stage mainly consists of clipping primitives to the view frustum (the part of space that is visible on the screen) and optional culling of front-facing and/or back-facing primitives. These possibilities are discussed in more detail in the platform-specific tutorials.

Data Flow

In order to program GLSL vertex and fragment shaders, it is important to understand the input and ouput of each shader. To this end, it is also useful to understand how data is communicated between all stages of the OpenGL pipeline. This is illustrated in the next diagram:

Vertex Data		
↓ attributes (usually position, color, normal vector, and texture coordinates)		
Vertex Shader	←	uniforms (constants); (optionally texture data but not in most OpenGL ES 2.0 implementations)
↓ varyings (at vertices), vertex position, and point size		
Primitive Assembly		
↓ varyings (at vertices), vertex position, and point size		


Rasterization
varyings (now interpolated at pixels), fragment coordinates, point coordinates, front-facing flag
Fragment Shader
uniforms (constants) and texture data (images)
fragment color and fragment depth
Per-Fragment Operations
fragment color and fragment depth
Framebuffer

Attributes (or vertex attributes, or attribute variables) are defined based on the vertex data. The vertex position in an attribute is in object coordinates, i.e. this is the position as specified in a 3D modeling tool.

Uniforms (or uniform variables) have the same value for all vertex shaders and all fragment shaders that are executed when rendering a specific primitive (e.g. a triangle). However, they can be changed for other primitives. Typically, vertex transformations, specifications of light sources and materials, etc. are specified as uniforms.

Varyings (or varying variables) have to be consistently defined by the vertex shader and the fragment shader (i.e. the vertex shader has to define the same varying variables as the fragment shader). Typically, varyings are defined for colors, normal vectors, and/or texture coordinates.

Texture data include a uniform sampler, which specifies the texture sampling unit, which in turn specifies the texture image from which colors are fetched.

Other data is described in the tutorials for specific platforms.

Further Reading

The OpenGL ES 2.0 pipeline is defined in full detail in the “OpenGL ES 2.0.x Specification” and the “OpenGL ES Shading Language 1.0.x Specification” available at the “Khronos OpenGL ES API Registry” (http://www.khronos.org/registry/gles/).

A more accessible description of the OpenGL ES 2.0 pipeline is given in Chapter 1 of the book “OpenGL ES 2.0 Programming Guide” by Aaftab Munshi, Dan Ginsburg and Dave Shreiner published by Addison-Wesley (see its web site (http://www.opengles-book.com/)).

< GLSL Programming