CENG 3547
COMPUTER GRAPHICS
HW2
FIXED FUNCTION PIPELINE VS PROGRAMMABLE SHADERS

Gizem PESEN
pesengizem@gmail.com
github : **https://github.com/gizempesen/Ceng3547Shaders**

Tuesday 22nd December, 2020

1 Fixed function pipeline details

1.0.1 Fixed function pipeline

Fixed function pipeline

A graphics processing pipeline with a fixed set of processing stages that cannot be modified by a programmer.
Data for an image passes through a sequence of processing stages, with the image as the end product. The sequence is called a **"pipeline."** With a fixed-function pipeline, the programmer can enable and disable stages and set options that control the processing but cannot add to the functionality.

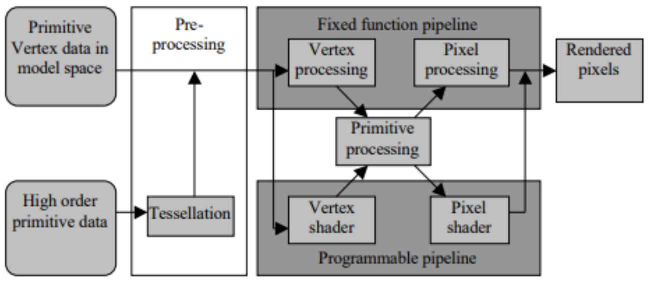


Figure 6. The Direct3D fixed function pipeline.

Figure 1: Fixed Function Pipeline

1.0.2 Traditional processing

In software engineering, a pipeline consists of a chain of processing elements (processes, threads, coroutines, functions, etc.), arranged so that the output of each element is the input of the next; the name is by analogy to a physical pipeline. Usually some amount of buffering is provided between consecutive elements. The information that flows in these pipelines is often a stream of records, bytes, or bits, and the elements of a pipeline may be called filters; this is also called the pipes and filters design pattern. Connecting elements into a pipeline is analogous to function composition.

1.0.3 Vertex arrays

VAO

A **Vertex Array Object** (or VAO) is an object that describes how the vertex attributes are stored in a Vertex Buffer Object (or VBO). This means that the VAO is not the actual object storing the vertex data, but the descriptor of the vertex data.

Vertex attributes can be described by the glVertexAttribPointer function and its two sister functions **glVertexAttribIPointer** and **glVertexAttribLPointer**, the first of which we'll explore below.

1.0.4 Shading

shader

A program to be executed at some stage of the rendering pipeline. OpenGL shaders are written in the GLSL programming languages. For WebGL, only vertex shaders and fragment shaders are supported for demo.

The shading pipeline is the list of rendering passes performed at the camera level to produce an image. Graphical data are attached to the camera itself. Therefore, we 'render' the contents of a camera, and the produced result is positioned in the window that is hosting the camera. The viewpoint render list to which the camera has been attached defines the viewport information of the camera in the window.

The way geometries (meshes, lines, texts, etc...) are rendered on screen is defined by the contents of the material applied to the shapes. Each material can use a number of rendering passes, as shown by the schema below:

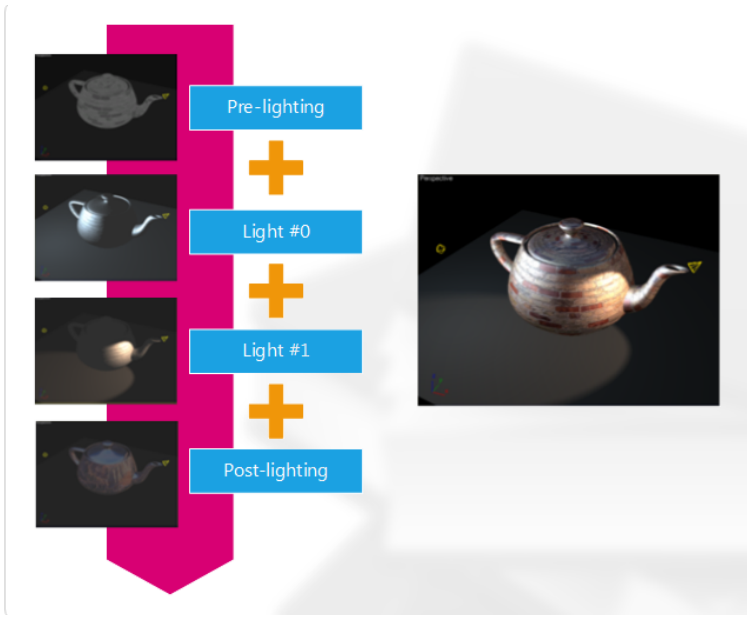


Figure 2: Shading

1.0.5 Texturing

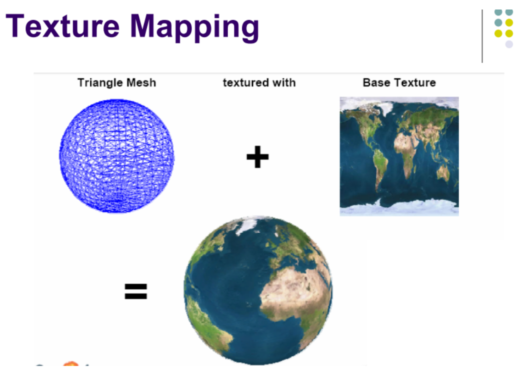


Figure 3: There are a few texture targets : `GL_TEXTURE_1D`,`GL_TEXTURE_2D`,`GL_TEXTURE_3D`, `GL_TEXTURE_CUBE_MAP`.

.

2 Programmable shaders details

2.0.1 History of shaders

OpenGL is **25** years old! Since the first release in **1992**, a lot has happened (and is still happening actually, with the newly released **Vulkan API** and the 4.6 GL release) and consequently, before diving into the book, it is important to understand OpenGL API evolution over the years. If the first API (1.xx) has not changed too much in the first twelve years, a big change occurred in 2004 with the introduction of the dynamic pipeline (OpenGL 2.x), i.e. the use of shaders that allow to have direct access to the GPU. Before this version, OpenGL was using a **fixed pipeline** that made it easy to rapidly prototype some ideas. It was simple but not very powerful because the user had not much control over the **graphic pipeline**. This is the reason why it has been deprecated more than ten years ago and you don't want to use it today. Problem is that there are a lot of tutorials online that still use this fixed pipeline and because most of them were written before **modern GL**, they're not even aware (and cannot) that they use a deprecated API.

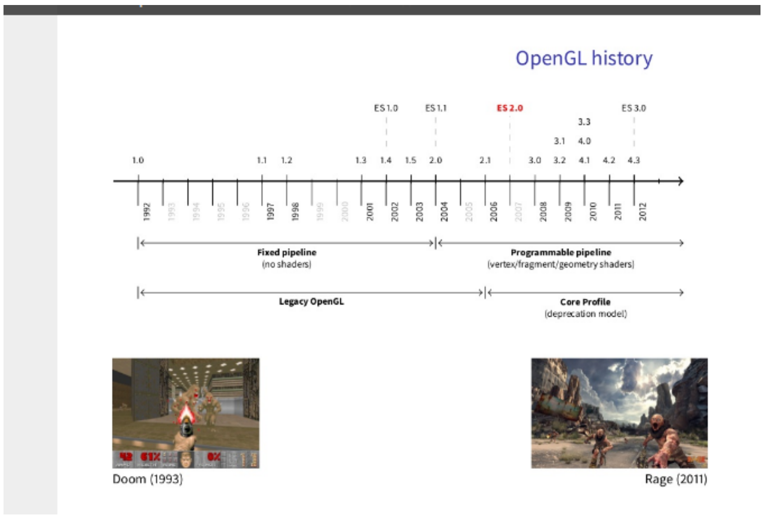
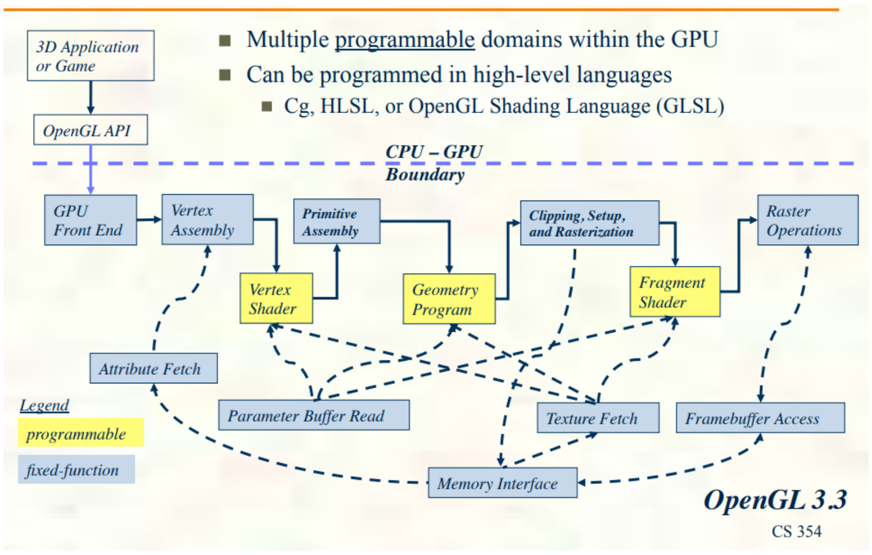


Figure 4: History



- Multiple programmable domains within the GPU
- Can be programmed in high-level languages
  - Cg, HLSL, or OpenGL Shading Language (GLSL)

Figure 5: Programming Shaders in GPU

2.0.2 GPU and performance

The pipeline, at the very highest level, can be broken into two parts: the CPU and the GPU. Although CPU optimization is a critical part of optimizing your application, it will not be the focus of this chapter, because much of this optimization has little to do with the graphics pipeline. Figure shows that within the GPU, there are a number of functional units operating in parallel, which essentially act as separate special-purpose processors, and a number of spots where a bottleneck can occur. These include vertex and index fetching, vertex shading (transform and lighting, or TL), fragment shading, and raster operations (ROP).

2.0.3 Vertex, fragment, tesellation, geometry shaders

Programmable Vertex Processor
The vertex processor is a programmable unit that operates on incoming vertex attributes, such as position, color, texture coordinates, and so on. The vertex processor is intended to perform traditional graphics operations such as vertex transformation, normal transformation/normalization, texture coordinate generation, and texture coordinate transformation.
The vertex processor only has one vertex as input and only writes one vertex as output. Topological information of the vertices is not available.

Programmable Geometry Processor
The greometry processor allows access to the geometry (lines, triangles, quads etc.). It is even possible to create new geometry. However, the geometry shader is not part of the OpenGL Shading Language specification. It is a multivendor extension and currently available (for developers) on NVidia GeForce 8 series graphics cards. Because this is a very important extension to the OpenGL Shading Language it is mentioned here and in some tutorials.

Programmable Fragment Processor
The fragment processor is intended to perform traditional graphics operations such as operations on interpolated values, texture access, texture application, fog, and color sum.

Tesellation:

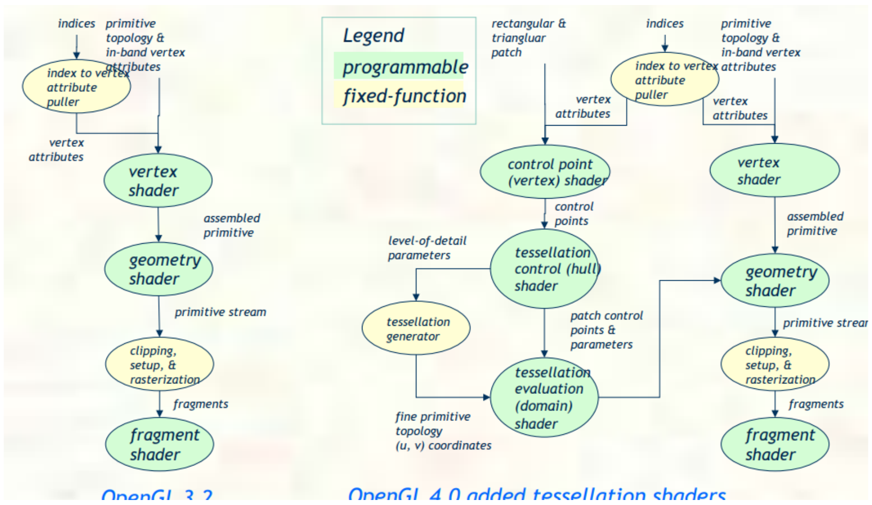

indices
primitive topology & in-band vertex attributes
index to vertex attribute puller
vertex attributes
vertex shader
assembled primitive
geometry shader
primitive stream
clipping, setup, & rasterization
fragments
fragment shader
OpenGL 3.2
Legend
programmable
fixed-function
rectangular & triangluar patch
indices
primitive topology & in-band vertex attributes
index to vertex attribute puller
vertex attributes
vertex attributes
control point (vertex) shader
vertex shader
control points
assembled primitive
level-of-detail parameters
tessellation control (hull) shader
geometry shader
tessellation generator
patch control points & parameters
primitive stream
clipping, setup, & rasterization
tessellation evaluation (domain) shader
fine primitive topology (u, v) coordinates
fragments
fragment shader
OpenGL 4.0 added tessellation shaders

Figure 6: Programable Tesellation Shaders

Vertex Shader

A shader program that will be executed once for each vertex in a primitive. A vertex shader must compute the vertex coordinates in the clip coordinate system. It can also compute other properties, such as color.

.

Fragment Shader

A shader program that will be executed once for each pixel in a primitive. A fragment shader must compute a color for the pixel, or discard it. Fragment shaders are also called pixel shaders.

2.0.4 GLSL

Glsl

OpenGL Shading Language (GLSL) is a high-level shading language with a syntax based on the C programming language. It was created by the OpenGL ARB (OpenGL Architecture Review Board) to give developers more direct control of the graphics pipeline without having to use ARB assembly language or hardware-specific languages. for demo.

2.0.5 Shading and Texturing

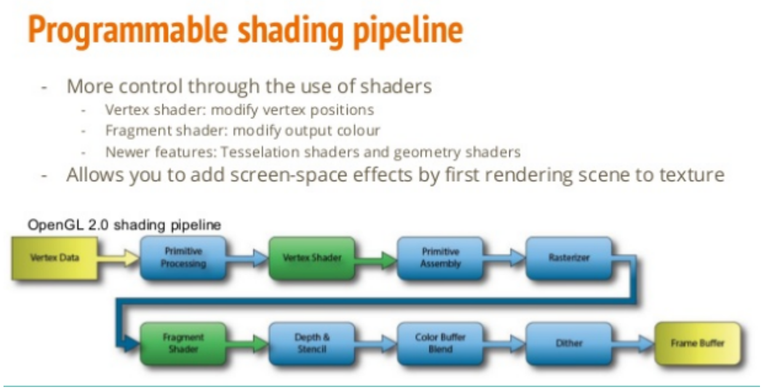
The fixed-function pipeline is as the name suggests the functionality is fixed. So someone wrote a list of different ways you'd be permitted to transform and rasterise geometry, and that's everything available. In broad terms, you can do linear transformations and then rasterise by texturing, interpolate a colour across a face, or by combinations and permutations of those things. But more than that, the fixed pipeline enshrines certain deficiencies.

For example, it was obvious at the time of design that there wasn't going to be enough power to compute lighting per pixel. So lighting is computed at vertices and linearly interpolated across the face.

There were some intermediate extensions related to specific effects — dot3 plus cubemaps for per-pixel lighting from a single source, for example — but the programmable pipeline lets you do whatever you want at each stage, giving you complete flexibility.

In the first place that allowed better lighting, then better general special effects (ripples on reflective water, imperfect glass, etc), and more recently has been used for things like deferred rendering that flip the pipeline on its end.

All support for the fixed-functionality pipeline is implemented by programming the programmable pipeline on hardware of the last decade or so. The programmable pipeline is an advance on its predecessor, afforded by hardware improvements.


Programmable shading pipeline

- More control through the use of shaders
  - Vertex shader: modify vertex positions
  - Fragment shader: modify output colour
  - Newer features: Tesselation shaders and geometry shaders
- Allows you to add screen-space effects by first rendering scene to texture

Figure 7: Shading

2.0.6 Scientific visualization

Although OpenGL is widely known for its use in games, it has also many other applications. One of these is the visualization of scientific data. Technically, there is not a great difference between drawing datasets and drawing game graphics, but the emphasis is different. Instead of a perspective view of our data, the scientist usually wants an orthographic view. Instead of specular highlights, reflections and shadow, scientific data is usually presented with primary colors and just a bit of smooth shading.

It may sound like only simple OpenGL features are used, but in return a scientist wants the data rendered with a high accuracy, without any artifacts, and without arbitrary clipping of geometry or lighting. Also, raw data might need a lot of transformation before it can be rendered, and these transformations cannot always be implemented as matrix multiplications. Before the advent of the programmable shaders, scientific visualisation was lot harder to do on graphics cards.

Last: .

Last:

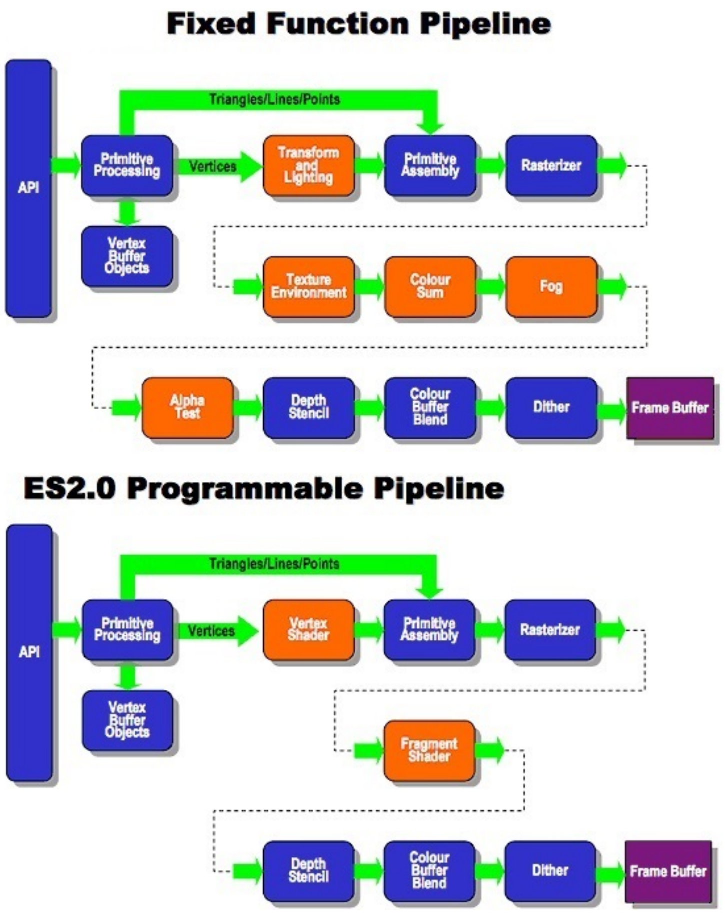


Figure 8: vs

Figure 9: OpenGL ES

3 OpenGL ES

OpenGL ES

OpenGL ES is an **"embeddable subset"** of OpenGL. It slims down the rather large OpenGL API to the bare essentials, so that it can be implemented on devices with simpler, cheaper hardware, and above all, low enough power requirements to run on batteries. For example, it is available as standard on smartphones running both Apple's IOS and Google's Android operating system.

OpenGL ES Versus Regular OpenGL

- No Begin/End grouping and associated calls for individually specifying vertex info: instead, you must use the xxxPointer calls to pass entire buffers of coordinates at a time, and then draw them with DrawArrays (selecting contiguous subarrays) or DrawElements (selecting individual array elements by index).
- Only 2D textures, no 3D or 1D.
- No support for polygons other than triangles.
- ES adds the option to specify coordinates etc as fixed-point values (calls with an x suffix) instead of floating-point (f suffix).
- On some platforms (e.g. Android GLES 2.0) there is no OpenGL error queue storage, only last error is stored.

4 WebGL

WebGL

WebGL (Web Graphics Library) is a JavaScript API for rendering interactive 2D and 3D graphics within any compatible web browser without the use of plug-ins.

Figure 10: webgl

WebGL is fully integrated with other web standards, allowing GPU-accelerated usage of physics and image processing and effects as part of the web page canvas. WebGL elements can be mixed with other HTML elements and composited with other parts of the page or page background. WebGL programs consist of control code written in JavaScript and shader code that is written in **OpenGL ES Shading Language (GLSL ES)**, a language similar to C or C++, and is executed on a computer's graphics processing unit (GPU).

5 References

sec:1.0.1

- `https://www.sciencedirect.com/topics/computer-science/fixed-function-pipeline`
- `https://www.khronos.org/opengl/wiki/Fixed_Function_Pipeline`
- `https://www.sciencedirect.com/topics/computer-science/fixed-function-pipeline`

sec:1.0.2

- `https://en.wikipedia.org/wiki/Pipeline_(software)`

sec:1.0.3

- `http://www.songho.ca/opengl/gl_vertexarray.html`
- `https://www.cs.utexas.edu/~theshark/courses/cs354/lectures/cs354-11.pdf`

sec:1.0.4

- `http://math.hws.edu/graphicsbook/c6/s1.html`
- `http://www.downloads.redway3d.com/downloads/public/documentation/bk_bm_the_shading_pipeline.html`

sec:1.0.5

- `http://web.cse.ohio-state.edu/~shen.94/781/Site/Slides_files/pipeline.pdf`
- `https://stackoverflow.com/questions/30476888/fixed-pipeline-texture-rendering`

sec:2.0.1

- https://www.labri.fr/perso/nrougier/python-opengl/

sec:2.0.2

- https://www.eecg.utoronto.ca/~moshovos/ACA05/read/GPU-micro.pdf
- https://developer.download.nvidia.com/books/HTML/gpugems/gpugems_ch28.html
- http://graphics.stanford.edu/courses/cs148-10-fall/lectures/programmable.pdf
- http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.518.899&rep=rep1&type=pdf

sec:2.0.4

- https://en.wikipedia.org/wiki/Shader

sec:2.0.5

- https://en.wikipedia.org/wiki/OpenGL_Shading_Language
- https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/By_example/Hello_GLSL
- https://stackoverflow.com/questions/18950395/
- https://castle-engine.io/x3d_implementation_shaders.php#section_uniforms_tex
- https://sites.google.com/site/face2manoj/opengl/pipeline

sec:2.0.7

- https://en.wikibooks.org/wiki/OpenGL_Programming/Scientific_OpenGL_Tutorial_01

sec:2.0.8

- https://www.slideshare.net/NicolasRougier1/opengl-scientific-visualization

sec:3

- https://www.khronos.org/opengl/wiki/OpenGL_ES

sec:4

- https://en.wikipedia.org/wiki/WebGL
- https://www.khronos.org/webgl/

sec:5

- https://community.khronos.org/t/fixed-function-pipeline-vs-shaders/74117
- https://stackoverflow.com/questions/30476888/fixed-pipeline-texture-rendering