Assignment 2

CENG 3547 Introduction to Computer Graphics

Ahmet Oral

ahmetoral@posta.mu.edu.tr

December 2020

Abstract

The aim of this paper is to compare evolution of shaders with fixed function pipeline while explaining relevent terms and softwares used in computer graphics. I started from the fixed function pipeline and talked about important developments along the time. I tried to explain every subject assigned to me while being careful to not breaking time line and subject integrity.

Contents:

1) Fixed Function Pipeline

a)What is Fixed Function Pipeline?

A pipeline is just a series of processes that ocur in order, like: Read File -> Process Data -> Print Results. A pipeline can have any number of processes within it.The point is data from one process is fed into the next process, like an assembly line. Each process Works independently and whenever one process has data ready for the next, it forwards it on.

A fixed function pipeline (Figure1) is a simple programming abstraction with a series of well-defined and specifically named pipeline stages. Data throughput in a pipeline is fast in comparison to the traditional processing. The hardware was wired and narrowly specialized so that it performed some standard operations on the data, and because it was wired that way, it was so much faster than doing them on your processor. This kind of hardware was popular from the early 1980s to the late 1990s. OpenGL and DirectX provided an easy to use graphics API to provide abstraction.

On the other hand, fixed function pipeline is like an machine with a lot of switches/values to configure. One cannot change how the function is implemented as well as the order of execution. So we can say it is not flexible at all. You could change fog color, lightning etc. but If you ever wanted to draw anything complex, the CPU simply wasn't fast enough, and submitting your vertices to the GPU was the only choice.

Simplified Overwiev of Fixed Function Pipeline

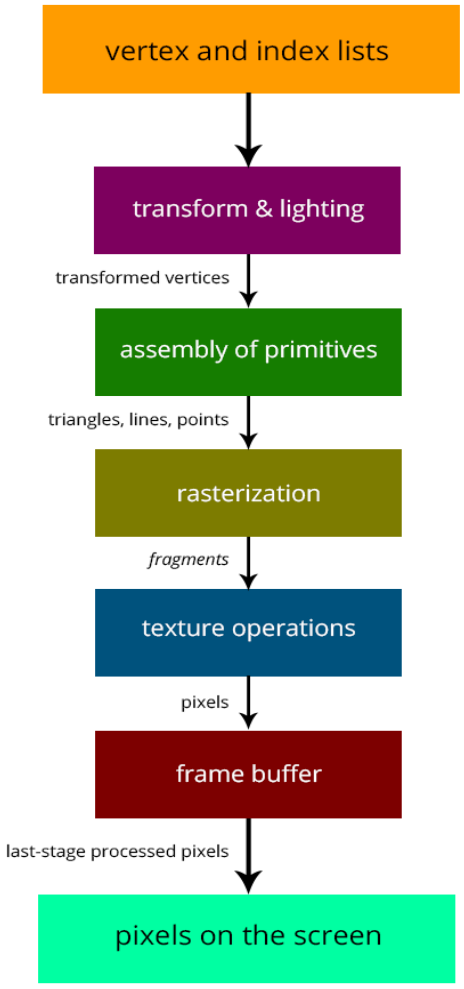
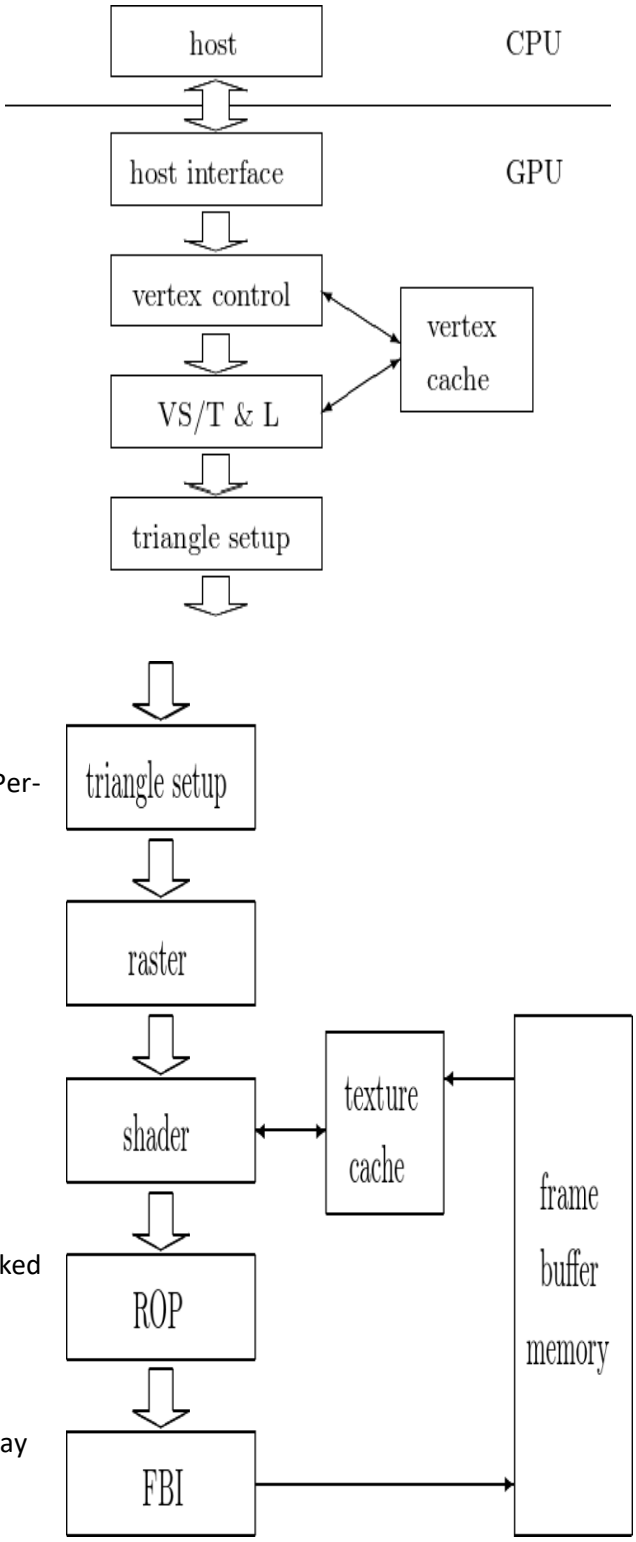


Figure 1

b) How Fixed Function Pipeline Operates?

Now lets take a closer look on the steps of Fixed Function Pipeline. Each stage as followed in the figure on right.

The stages of the pipeline are as follows:



1)Vertex Control

This stage receives parametrized triangle data from the CPU. The data gets converted and placed into the vertex cache.

2)VS/T & L (vertex shading, transform, and lighting)

The VS/T & L stage transforms vertices and assigns per-vertex values, e.g.: colors, normals, texture coordinates, tangents. The vertex shader can assign a color to each vertex, but color is not applied to triangle pixels until later.

3)Triangle Setup

Edge equations are used to interpolate colors and other per-vertex data across the pixels touched by the triangle.

4)Raster

The raster determines which pixels are contained in each triangle. Per-vertex values necessary for shading are interpolated.

5)Shader

The shader determines the final color of each pixel as a combined effect of interpolation of vertex colors, texture mapping, per-pixel lighting, reflections, etc.

6)ROP (Raster Operation)

The final raster operations blend the color of overlapping/adjacent objects for transparency and antialiasing effects. For a given viewpoint, visible objects are determined and occluded pixels (blocked from view by other objects) are discarded.

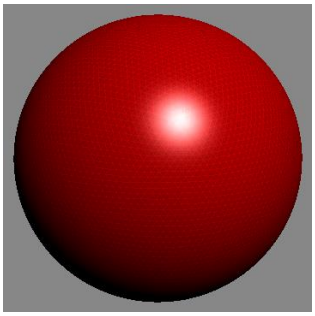
7)FBI (Frame Buffer Interface)

The FBI stages manages memory reads from and writes to the display frame buffer memory.

2) PROGRAMMABLE SHADERS

a)What is a Shader?

In computer graphics, shading refers to the process adding a decpiction of depht by altering the color of a model in the 3D scene, based on things like the surface's angle to lights, its distance from lights, its angle to the camera, material properties and etc. to create a photorealistic effect. Shading is performed during the rendering process by a program called a shader, and a shader is a piece of code that is executed on the Graphics Processing Unit (GPU), found on a graphics card, to manipulate an image before it is drawn to the screen.



Gouraud shading, named after Henri Gouraud, is an interpolation method used in computer graphics to produce continuous shading of surfaces represented by polygon meshes. In practice, Gouraud shading is most often used to achieve continuous lighting on Triangle meshes by computing the lighting at the corners of each triangle and linearly interpolating the resulting colours for each pixel covered by the triangle. Gouraud first published the technique in 1971 and it was one of the first shading techniques developed for computer graphics.

The term "shader" was first introduced to the public in May 1988 by Pixar with version 3.0 of their RenderMan Interface Specification, which is an open API developed by Pixar Animation Studios to describe three-dimensional scenes and turn them into digital photorealistic images. It includes the RenderMan Shading Language

At first, the term shader was only used to refer to “pixel shaders”, but soon enough new uses of shaders such as vertex and geometry shaders were introduced, making the term shaders more general.

b)Intoduction of Programmable Shaders.

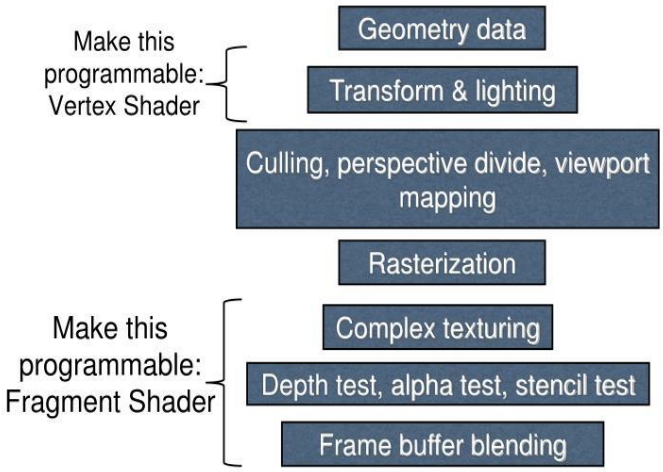
The graphics world was changing rapidly. While the fixed-function pipeline provided some nice features, it’s limitations were becoming evident. So, In 2001, nVidia intruduced GeForce3 video card, which was the first programmable GPU.

With this development, the Programmable Pipeline has been introduced to the graphics world. After this intoduction, the computer graphics exploded. With the ability to program what exactly happened to vertices, fragments, textures, and so on, and to do it fast, provided nearly endless possibilities.

Now, the developers could control the outputs of two stages in the pipeline; the Vertex Processing and Fragment Processing stages. This makes the vertex- and pixel processing much more flexible. These stages are controlled through GPU programs called "**Shaders**." and were first written in a shader specific assembly language.

The GPU program that controls the Vertex Processing stage is known as a **Vertex Shader**.

Whereas, the program that controls the Fragment Processing stage is known as a **Fragment Shader**.



c)Comparing Programmable and Fixed Function Pipeline

The Table below compares the Programmable Pipeline and Fixed Function Pipeline for flexibility, application complexity, learnability and deployability.

	Programmable Pipeline	Fixed Function Pipeline
Flexibility	+implement various algorithms in shader programs	-limited customization capability
For Simple Application	-must configure the whole pipeline	+performs simple task with less configurations
For Complex Application	+can achieve various effects	-most advanced effects are impossible
Learning Curve	-one must have full knowledge of the pipeline and GLSL before writing an application	+can create simple applications without much knowledge
Deploy	-should consider all graphics driver environment	+works in most graphics driver environment

We can see that while fixed function pipeline is fast and easier to use, it doesn't have much flexibility. So creating complex applications with it is not suitable because it doesn't have advanced effects. On the other hand, programmable pipeline is harder to learn and requires a stronger system, but it has much more flexibility and it can achieve various effects.

d)Fragment, Vertex, Geometry and Tesselation Shaders

Fragment Shader (Pixel Shader in Direcx) is the Shader stage that will process a Fragment generated by the Rasterization into a set of colors and a single depth value.The output of a fragment shader is a depth value, a possible stencil value , and zero or more color values to be potentially written to the buffers in the current framebuffers. Fragment shaders take a single fragment as input and produce a single fragment as output.The image on the left has various applied effects such as rim lightning and normal mapping

Vertex shaders are capable of altering properties such as position, color, and texture coordinates, but cannot create new vertices like geometry shaders can. The output of the vertex shader goes to the next stage in the pipeline, which is either a geometry shader if present, or the pixel shader and rasterizer otherwise. Vertex shaders can enable powerful control over the details of position, movement, lighting, and color in any scene involving 3D models. To the left an example is shown where the vertices are moved along their normal.

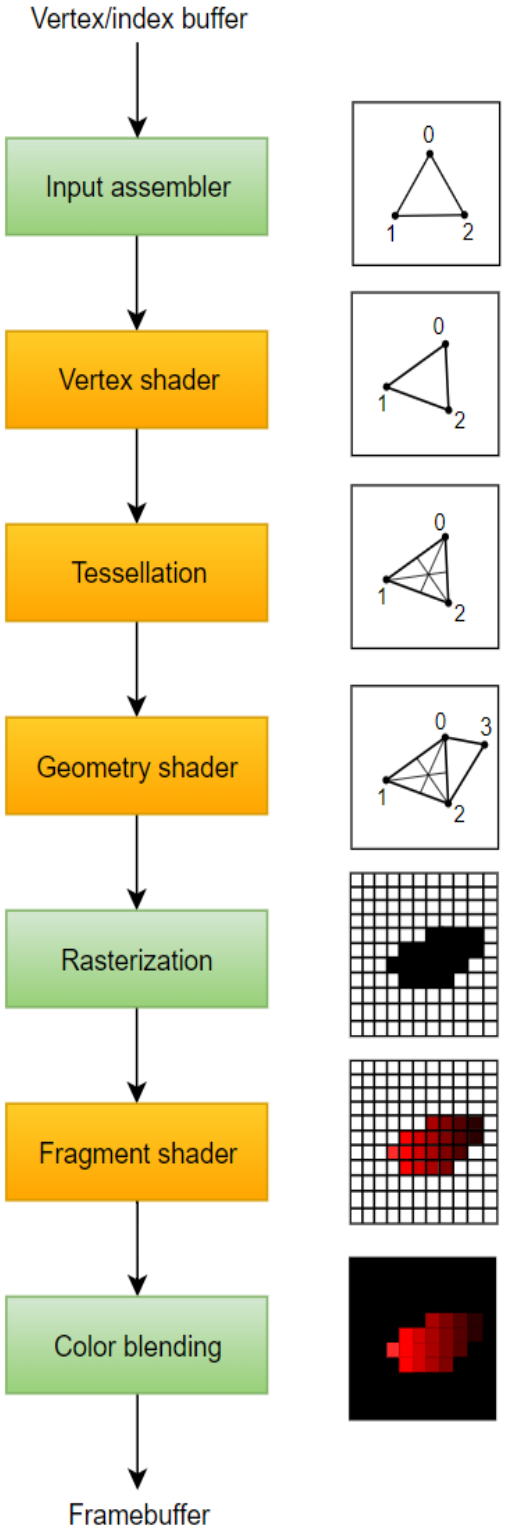
Geometry shader programs are executed after vertex shaders. They take as input a whole primitive, possibly with adjacency information. For example, when operating on triangles, the three vertices are the geometry shader's input. The shader can then emit zero or more primitives, which are rasterized and their fragments ultimately passed to a pixel shader. An example is shown on the left, where the bunny has polygonal fur generated by the geometry shader

No Tesselation vs Tesselation

Tesselation is used to manage datasets of vertex sets presenting objects in a scene and divide them into suitable structures for rendering. Especially for real-time rendering, data is tessellated into triangles, Tesselation shader allows to subdivide geometry based on certain rules to increase the mesh quality. This is often used to make surfaces like brick walls and staircases look less flat when they are nearby (Figure on the left)

e)Stages of Programmable Pipeline

We can see the steps of the Programmable Pipeline. Figure on the right is a Programmable pipeline configured to draw a triangle. Green color indicates that stage is fixed-function while the orange colored stages are programmable.



The *input assembler* collects the raw vertex data from the buffers you specify and may also use an index buffer to repeat certain elements without having to duplicate the vertex data itself.

The *vertex shader* is run for every vertex and generally applies transformations to turn vertex positions from model space to screen space. It also passes per-vertex data down the pipeline.

The *tessellation shaders* allow you to subdivide geometry based on certain rules to increase the mesh quality. This is often used to make surfaces like brick walls and staircases look less flat when they are nearby.

The *geometry shader* is run on every primitive (triangle, line, point) and can discard it or output more primitives than came in. This is similar to the tessellation shader, but much more flexible. However, it is not used much in today's applications because the performance is not that good on most graphics cards except for Intel's integrated GPUs.

The *rasterization* stage discretizes the primitives into *fragments*. These are the pixel elements that they fill on the framebuffer. Any fragments that fall outside the screen are discarded and the attributes outputted by the vertex shader are interpolated across the fragments, as shown in the figure. Usually the fragments that are behind other primitive fragments are also discarded here because of depth testing.

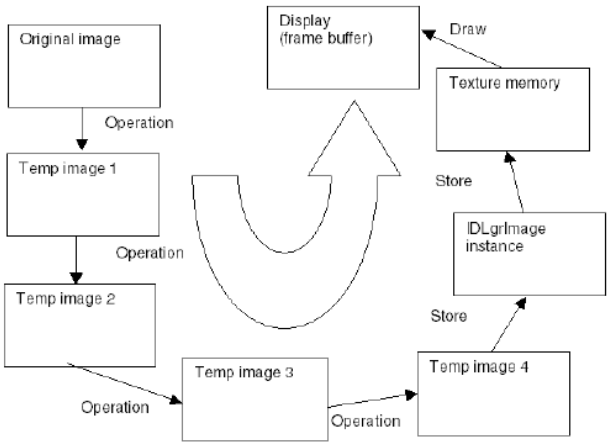
The *fragment shader* is invoked for every fragment that survives and determines which framebuffer(s) the fragments are written to and with which color and depth values. It can do this using the interpolated data from the vertex shader, which can include things like texture coordinates and normals for lighting.

The *color blending* stage applies operations to mix different fragments that map to the same pixel in the framebuffer. Fragments can simply overwrite each other, add up or be mixed based upon transparency.

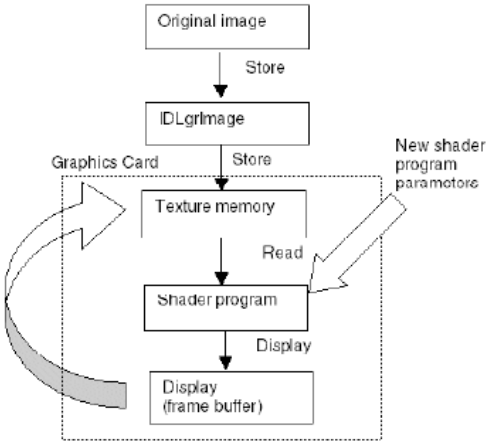
3)How shaders enhance performance

Using a shader lets you take advantage of the processing power of the graphics card processing unit (GPU) instead of relying solely on the system CPU. Also, the GPU can operate on multiple data streams simultaneously. For example, some GPUs can execute a fragment shader on up to 24 fragments (pixels) simultaneously, which provides a significant performance advantage over a CPU which can only process one pixel at a time.

As an example, a typical image processing application may apply several operations to a set of images, store the results in an IDLgrImage object, then displays the image. The figure on righy illustrates this process. The application performs several image operations, creating intermediate images that may be reused. This process requires a significant amount of computation and data movement before the final image is copied into the image object and the graphic device's texture memory. Additionally, all or most of this process must be repeated any time the parameters of an operation change, reducing interactive performance.



If we move the same image processing application to a shader program, IDL accomplishes the majority of the processing cycle on the graphics card. Without a shader program and suitable hardware, updating an image may require several tenths of a second or more. Noticeable display updates may occur with CPU processing.
With a shader program, the display rate with the same amount of processing can be hundreds of frames per second. Display updates will be smooth with GPU processing. Display rates of hundreds of frames per second are not always useful, but when lower rates are used, more CPU resources are available for other operations.



4) From assembly to high level shader languages.

In December 2002, with the release of DirectX 9.0, Microsoft introduced a simpler shader language called HLSL (High Level Shader Language). HLSL is a C like language, which, compared to shader assembly, is claimed to be easier to write, more readable and easier to maintain. Since HLSL code is actually compiled into shader assembly code , there is no actual difference in performance between writing shaders in HLSL or shader assembly.

Another language to write shaders is GLSL (GLslang) which is a short term for the official OpenGL Shading Language. GLSL is a C/C++ similar high level programming language for several parts of the graphic card. It covers most of the features that are expected in a language. With GLSLS programmers could code shaders which are executed on the CPU.

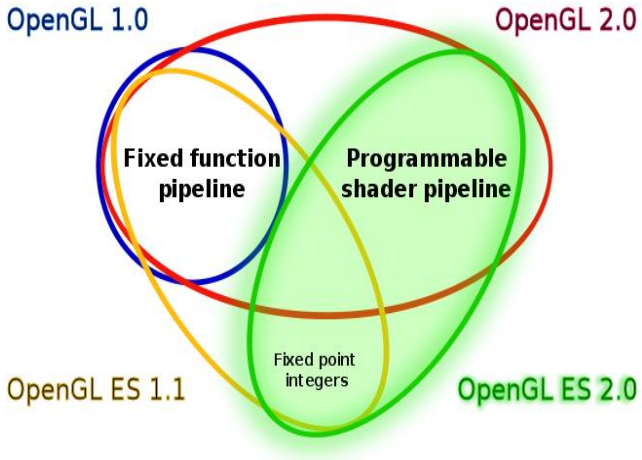
Later nVidia intruduced another language called Cg. This language was supported by both DirectX and OpenGL. It is also slower due to high level of integration.

With these technologies, developers started to write all kinds of shaders like, arallax mapping, custom lighting models, refraction and etc. Later, even completely custom lighting systems emerged, such as deferred shading and light pre-pass, and you could see complex post-processing effects such as screen space ambient occlusion and horizon based ambient occlusion.

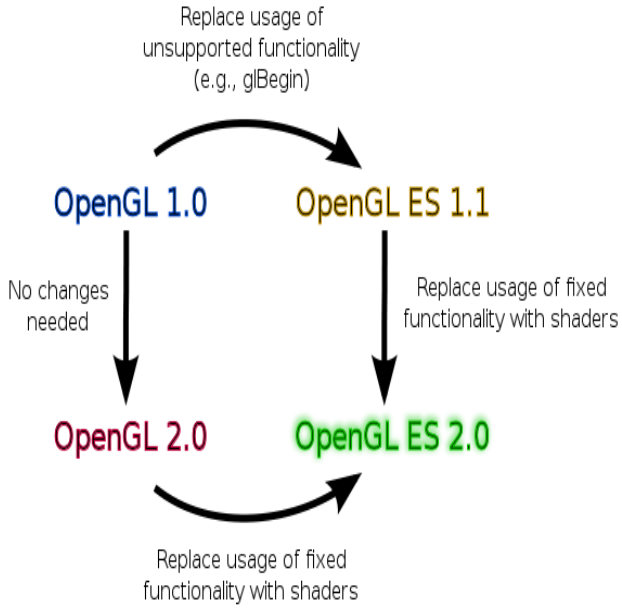
5) OpenGL ES

In the 1980s developing computer graphics was very hard. Every hardware needed it's own custom software. To this end OpenGL were introduced in 1992. OpenGL provided an easy to use graphics API to provide (hardware) abstraction. Then OpenGL ES 1.0 was released publicly July 28, 2003. It was based on the original OpenGL 1.3 API.

OpenGL for Embedded Systems (OpenGL ES or GLES) is a subset of the OpenGL computer graphics rendering application programming interface (API) for rendering 2D and 3D computer graphics such as those used by video games, typically hardware-accelerated using a graphics processing unit (GPU). It is designed for embedded systems like smartphones, tablet computers, video game consoles and PDAs. OpenGL ES is the most widely deployed 3D graphics API in history. The API is cross-language and multi-platform. The libraries GLUT and GLU are not available for OpenGL ES. OpenGL ES is managed by the non-profit technology consortium Khronos Group. Vulkan, a next-generation API from Khronos, is made for simpler high performance drivers for mobile and desktop devices.



All versions and variants of OpenGL are not directly compatible, so a certain amount of changes are needed when moving applications from one variant to another. The diagram on right summarizes these changes for major OpenGL versions. OpenGL 1.0 applications can be ported to OpenGL ES 1.1, but need changes if they are using some of the removed APIs. OpenGL 2.0 application that only uses programmable shaders is possible to port OpenGL ES 2.0, but may still need some work due to differences in the shading language versions. Lastly, porting OpenGL 1.0 or OpenGL ES 1.1 applications to OpenGL ES 2.0 requires a rewrite to replace fixed function API usage with programmable shaders.



6)WebGL

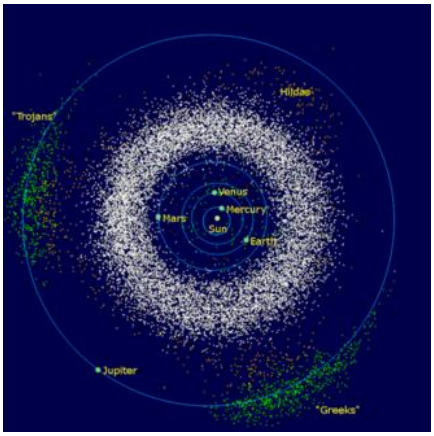
WebGL is a graphics application programming interface (API) created for use in web applications. It is based off the open graphics language (OpenGL) embedded standard (ES).WebGL is used by developers to provide a platform-independent means of creating interactive graphical applications on the web.

WebGL is not only used to draw the graphics of 2D and 3D games, but also to accelerate the functions of web based image editors and their effects, as well as physics simulations. Although WebGL is functionally based off OpenGL ES, it is partly written in JavaScript. WebGL is used to render interactive 2D and 3D graphics in compatible web browsers. The API allows users to experience interactive content on webpages, with GPU acceleration, without having to first download or install any plug-ins. For developers, WebGL provides low-level access to hardware with the familiar code structure of OpenGL ES.

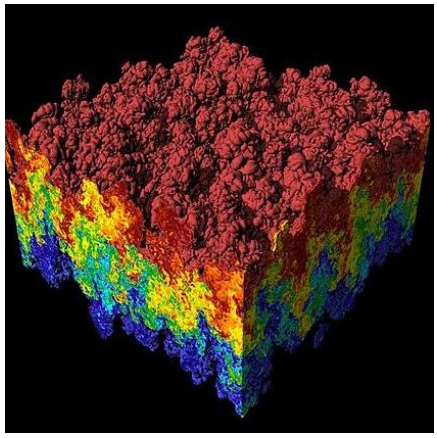
7)Using Shaders to Enhance Scientific Visualizations

Scientific visualization refers to the process of representing raw, scientific data as images, providing an external aid to improve scientists' interpretations of large data sets and to gain insights that may be overlooked by statistical methods alone. Displaying complex data in visual form provides a clear and intuitive approach. With developing technology in computer graphics, science is using these new Techniques to impove our undestanding of the universe.

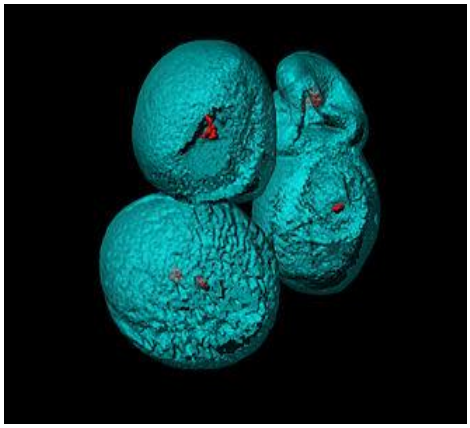
Solar system image of the main asteroid belt and the Trojan asteroids.



Scientific visualization of a simulation of a Rayleigh–Taylor instability caused by two mixing fluids



Surface rendering of Arabidopsis thaliana pollen grains with confocal microscope.



GPU shaders aren't just for cool special effects. They can be used in the drive to understand large, complex data sets and even create simulations for scientific research purposes. Techniques like contour plots, color coding constant value surface rendering and custom shapes are used to present scientific data. With this applications we could see the world in a perspective that we would never see otherwise. There is no limit for where computer graphics can be applied. They can be used in health care, mathematics, physics, biology, geogrophy and the list just goes on…

8)Conclusion

As we can see, computer graphics has come a long vay since the fixed function pipelines. At first, graphics were designed on a hardware that wired in a specific way that is not changeable and hard coded. As deman for better realism increases, with the new technological advancements, first graphic cards were introduced with programmable shaders. This was a huge step in realistic graphics and a milestone in computer graphics world. After the programmable pipeline, progress accelerated as developers had more control and flexibility in their hands. Naturally, with this step and new powerfull GPU's, hard coded fixed function pipeline passed the torch to the new programmable shaders. In 2007 OpenGL ES 2.0, released and removed fixed-function pipeline in favor of a programmable one. This paper includes some important developments and technologies that were invented on the way. It's really fascinating how the computer graphics evolved in such a (relatively) short time.

References:

Performance evaluation of the fixed function pipeline and the programmable pipeline Markus Holmåker, Magnus Woxblom

Graphics shaders Mike Hergaarden January 2011, VU Amsterdam

CSCI 420 Computer Graphics Lecture 4 - Jernej Barbic - University of Southern California

A Brief History of Shaders – Oregon State University – Mike Bailet

TOM McREYNOLDS, DAVID BLYTHE, in Advanced Graphics Programming Using OpenGL, 2005

Ron Fosner, in Real-Time Shader Programming, 2003

Andy Jonson's CS 488 Course Notes

Foley, Van Dam, Feiner and Hughes, "Computer Graphics – Principles and Practice"

vulkan-tutorial.com

nVidia.com

non copytrighted wikipedia images