

Real Time Rendering Engine

**Real – Time Graphics
with a modern graphics approach**

Flow Render Engine

Final Thesis

**Master's degree final project
Master computing engineering
High performance computing area**

**Xavier Figuera Alberich
December 2019
xfiguera@uoc.edu**

Real Time Rendering Engine

**Real – Time Graphics
with a modern graphics approach**

Flow Render Engine

Final Thesis

**Master's degree final project
Master computing engineering
High performance computing area**

Xavier Figuera Alberich

December 2019

xfiguera@uoc.edu

[*http://www.flowrenderengine.com/*](http://www.flowrenderengine.com/)

Tutors:

Ester Arroyo Garriguez

Josep Jorba Esteve



Document license

This document, Real-time rendering engine, real-time graphics with a modern graphics approach, Flow Render Engine, is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 international licence.



You can receive a copy of the license in the following link:

[*https://creativecommons.org/licenses/by-nc-sa/4.0/*](https://creativecommons.org/licenses/by-nc-sa/4.0/)

Contents

1 Introduction.....	1
1.1 What is real time rendering?.....	1
1.2 What is a real-time rendering engine?.....	3
1.3 Graphics Hardware.....	4
1.3.1 Desktop GPUs.....	6
1.3.2 Workstations GPUs.....	6
1.3.3 Mobile GPUs.....	6
1.4 Graphics APIs.....	6
1.4.1 OpenGL.....	6
1.4.2 Direct3D.....	7
1.4.3 Vulkan.....	8
1.5 Graphic effects shaders.....	8
1.5.1 History of shaders.....	8
1.5.2 Modern shader languages.....	10
1.6 Render engines today.....	10
1.6.1 Real-Time render engines.....	10
1.6.1.1 Ogre3D.....	10
1.6.1.2 Open Scene Graph (OSG).....	10
1.6.1.3 Irrlicht.....	11
1.6.1.4 Magnum.....	11
1.6.1.5 Horde3D.....	11
1.6.2 Ray-Tracing render engines.....	11
1.6.2.1 RenderMan.....	11
1.6.2.3 POV-Ray.....	12
1.7 Project goals.....	12
1.8 Document structure.....	13
2 Basics of real-time rendering.....	13
2.1 The rendering pipeline.....	13
2.2 GPU Programming.....	15
2.2.1 Vertex Shaders.....	16
2.2.2 Geometry Shaders.....	16
2.2.3 Pixel Shaders.....	16
2.3 Scene graph and spatial data structures.....	17
3 Engine architecture and features.....	18
3.1 Introduction.....	18
3.2 Engine architecture.....	19
3.2 Rendering interface approach.....	20
3.3 Fluent interface approach.....	21
3.4 Design patterns used.....	22
3.4.1 Decorator.....	22
3.4.2 Abstract Factory.....	22
3.4.3 Singleton.....	22
3.5 Engine features.....	22
4 Engine modules definition and implementation.....	23
4.1 Core module.....	23
4.1.1 Engine submodule.....	24
4.1.1.1 Application context.....	24
4.1.1.2 Singleton template class.....	25
4.1.2 Math submodule.....	25
4.1.2.1 Basic functions.....	25
4.1.2.2 Vectors.....	26
4.1.2.3 Matrices.....	28

4.1.2.4 Vectors and matrices unit tests.....	30
4.1.2.7 2D and 3D geometric primitives.....	33
4.1.2.8 Geometric transformations.....	34
4.1.2.8.1 Model Matrix.....	35
4.1.2.8.1.1 Translation.....	35
4.1.2.8.1.2 Scaling.....	36
4.1.2.8.1.3 Rotation.....	38
4.1.2.8.2 Viewing Transformations.....	40
4.1.2.8.2.1 Viewport transform.....	41
4.1.2.8.2.2 Projection matrix.....	42
4.1.2.8.2.2.1 Orthographic Projection.....	42
4.1.2.8.2.2.2 Perspective Projection.....	44
4.1.2.8.2.3 View matrix (lookAt).....	46
4.1.2.8.2.4 Rotation matrix (yawPitchRoll).....	47
4.1.2.9 Transformations unit tests.....	49
4.1.2.10 Normal calculation.....	50
4.1.2.11 Tangents and bi-tangents calculation for bump mapping.....	52
4.1.3 Platform sub-module.....	56
4.1.4 Utility sub-module.....	57
4.1.4.1 Dynamically loaded C++ Objects.....	58
4.1.4.1.1 How does it work?.....	58
4.1.4.1.2 Implementation within the engine.....	61
4.1.4.2 File system.....	62
4.1.4.3 Log system.....	63
4.1.4.4 Timer.....	63
4.2 Graphics module.....	63
4.2.1 Resources.....	65
4.2.1.1 Vertex Declaration.....	65
4.2.1.2 Vertex Format.....	66
4.2.1.3 Vertex Element.....	66
4.2.1.4 Buffers.....	68
4.2.1.4.1 Vertex Buffers.....	69
4.2.1.4.2 Index Buffers.....	71
4.2.1.5 Textures 2D.....	72
4.2.2 Utility.....	75
4.2.2.1 Managers.....	75
4.2.2.1.1 Buffer Managers.....	76
4.2.2.1.2 Texture Manager.....	76
4.2.2.1.3 Shader Manager.....	76
4.2.2.1.4 Render Effect Manager.....	76
4.2.2.1.5 Material Manager.....	77
4.2.2.1.6 Mesh Manager.....	77
4.2.2.2 Model importers.....	77
4.2.2.2.1 Wavefront OBJ importer.....	77
4.2.3 Windowed application.....	86
4.2.3.1 Handling keyboard and mouse events.....	88
4.2.4 Scene Graph.....	89
4.2.4.1 Renderable objects and meshes.....	91
4.2.4.2 Render Transaction.....	92
4.2.4.3 Renderable object set and scene handler.....	93
4.2.4.4 Camera.....	94
4.2.4.5 Lighting.....	97
4.2.4.5.1 Directional light.....	101
4.2.4.5.2 Point light.....	102
4.2.4.5.3 Spot light.....	103
4.2.4.6 Materials.....	105

4.2.5 Renderer.....	106
4.2.5.1 Render states.....	108
4.2.5.1.1 Wire frame state.....	109
4.2.5.1.2 Depth test state.....	110
4.2.6 Effects.....	111
4.2.6.1 Shaders.....	111
4.2.6.1.1 Shader parameters data.....	114
4.2.6.1.2 Render pass.....	116
4.2.6.1.3 Render technique.....	117
4.2.6.2 Local Effects.....	118
4.2.6.2.1 Render effect.....	118
4.2.6.2.2 Renderable Effect.....	120
4.2.6.2.3 Effects implemented within the engine.....	121
4.2.7 Data Types.....	122
4.2.7.1 Transform class.....	122
4.2.7.2 Color class.....	124
4.3 GraphicsOGL3 module.....	124
4.3.1 Reference to the OpenGL functions used.....	125
5 Applications over the engine.....	127
5.1 Applications implemented to test the engine.....	129
6 Summary.....	129
6.1 Future Works.....	130
7 Bibliography and resources.....	131
7.1 Section 1.....	131
7.2 Section 2.....	132
7.3 Section 3.....	132
7.4 Section 4.....	133
7.5 Other resources.....	134

1 Introduction

1.1 What is real time rendering?

The rendering concept, should be seen as an automatic process generation of a photorealistic or non-photorealistic image of a geometric data source, this data source can come from different sources, such as parsed files with a data export tool, these data represent a model to render, the models can be in 3D or 2D, although in a real-time rendering, three-dimensional objects are usually rendered.

On the other hand, the concept in real time in this case, should be understood as that the images are generated online, and the generation rate is fast enough so that the sequence of images looks like an animation that simulates something. This is the most highly interactive area of computer graphics. The rendering cycle occurs at a speed fast enough so that the viewer does not see individual images, but immerses himself in a dynamic process.

The rate at which images are displayed is measured in frames per second (FPS) or Hertz (Hz). At around 6 FPS, a sense of interactivity starts to grow, 24 FPS might be acceptable and is certainly real-time, but a higher rate is important for minimizing response time. Video games aim for 30, 60, 72, or higher FPS. From about 72 FPS and up, the human eye cannot detect any differences in display rate. For this rates, the interactivity sense is total, so a good performance is to achieve 15 milliseconds of temporal delay between frames.

Not only speed is the only criterion in the real-time rendering concept, as has been said, real-time rendering usually involves producing three-dimensional images, therefore, if interactivity is combined with three-dimensional rendering, there are sufficient conditions for obtain real-time rendering, but there is a third element that, if combined with the concepts explained, completes the definition: the graphics accelerator hardware, see **section 1.3** for more details.

This type of hardware dedicated to three-dimensional graphics has been available in professional workstation for many years, but at the consumer level, the use of this hardware is relatively recent and with the rapid evolution in this area, every computer, tablet, video game console and mobile phone, actually comes with a built-in graphics card processor, this evolution in recent years in graphics hardware has powered a widely research in the field of interactive computer graphics. Some examples of real-time rendering performed with hardware accelerators over the years are shown in **figure 1.1**, **figure 1.2**, **figure 1.3** and **figure 1.4**.

Popular application areas of real-time rendering are video games, scientific computation visualization systems, CAD systems, flight simulation, industry software simulation, virtual reality, architecture, among others.

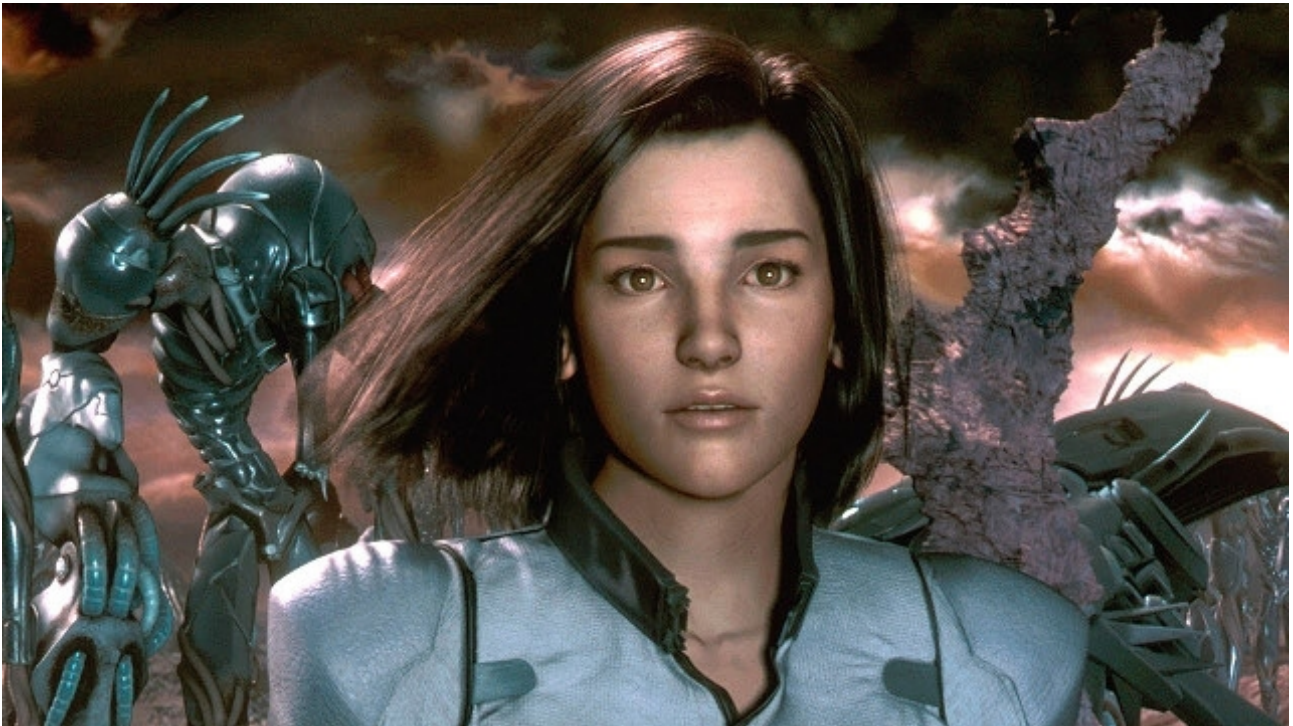


Figure 1.1. a shot from *Final Fantasy: The Spirits Within* film, Copyright by Square Company 2001 All rights reserved.



Figure 1.2. a shot from *Toy Shop* demo, Copyright by Natalya Tatarchuk, ATI Research Inc. 2005 All rights reserved.

1.2 What is a real-time rendering engine?

A real-time rendering engine should be seen as a middleware that plays a fundamental role in different real-time or interactive graphics applications, such as video games, scientific computation visualization systems, CAD systems, flight simulation, industrial software simulation, among others.

The engine takes 3D graphics primitives as input and generates real-time images as output. The real-time concept in this case, should be understood as that the images are generated online and the generation rate is fast enough so that the sequence images looks like an animation that simulates something as described in **section 1.1**.

Middleware users are mainly application developers. For application developers, a rendering engine is a software development kit. More precisely, a rendering engine consists of a set of reusable modules, such as static or dynamic link libraries. By using these libraries, developers can focus on the business logic of the application, without diverting attention to rather complicated graphics rendering issues.

In most cases, a professional rendering engine, usually does rendering tasks better than the programs written by application developers who are not computer graphics professionals. Meanwhile, adopting a good rendering engine in application development projects, can reduce the development period, since lots of complex works are done by the rendering engine and, consequently, development costs and risks are alleviated.

There are various rendering engines, that are available as commercial packages and open-source projects as well, in **section 1.5** are presented some of them.



Figure 1.3. a shot from Metal Gear Solid 4: Guns Of The Patriots, Copyright Konami Digital Entertainment 2008 All rights reserved.



Figure 1.4. a shot from Gran Turismo Sport, Copyright Sony Interactive Entertainment and Polyphony Digital 2017 All rights reserved.

1.3 Graphics Hardware

The real-time rendering is available on consumer-level from 1996 when appear 3Dfx Voodoo 1 graphics accelerator[3DfxVd1] on market, before that, the graphic data management to generate images had to be carried out by the CPU. From 1996 until now, the graphic accelerators has evolved a lot in performance terms, and during this evolve many task has been moved from the CPU to GPU (Graphics Processing Unit), the modern GPUs are a heterogeneous chip multi-processor highly tuned for graphics. Nowadays the CPU is mainly used to prepare graphic data to be sent to GPU and is responsible for user interaction.

This evolution drives to that nowadays exist a lot powerful commodity graphic hardware, where the hardware target within the computer graphics field, is focused to synthesize or render 2D raster images, from data that represent 3D scenes, these data contain information related with the scene geometry, which are projected to 2D surface, simulating a camera, see **section 4.1.2.8.2** Viewing Transformations and **section 4.2.4.4** camera, and to enhance the realism, lighting, materials and textures are involved.

There are different graphics hardware architectures for different systems, such as personal computers, video game consoles, tablets or mobile phones, in a modern personal computer normally a graphic system is composed by a CPU and a graphic card with a GPU and memory, and between them, there is a communication bus. As already mentioned, the CPU only processes the graphic data in advance to transfer it, to the main memory and then they sent to the graphic card video memory, trough data bus, in a modern personal computers, is used a PCI-Express expansion bus for this task, in the older computers, the AGP expansion bus was used. After that, the graphic data is efficiently processed by GPU through pipeline stages and finally the results are displayed onto the screen, see **section 2.1** the rendering pipeline.

However, the architecture described above, is not the only one, for example in a modern video game consoles like Sony PlayStation 3 or 4 or Microsoft Xbox 360 or Xbox One, the CPU can communicate directly with the GPU without any additional communication bus. See [\[RTGraRenEn\]](#) [\[PS4Arch\]](#) [\[GEngArchPS4\]](#).

The smartphones GPUs are typically designed with limited power ceiling of less than 1 watt [\[GProceUnits\]](#). As a result, the mobile GPU usually has fewer cores, lower memory bandwidth, and variant architecture when compared to the desktop GPUs.

A GPU inside a mobile device is typically integrated into the application processor system-on-a-chip (SoC) which also consists of one or several CPUs, DSP, and other application-specific accelerators. Instead of having its own graphics memory, an embedded GPU shares the system bus with other computing cores to access the external memory and therefore has much lower memory bandwidth than those of laptop and high-performance desktop systems [\[GProceUnits\]](#).

Mobile GPUs are usually designed with emphasis of lower power consumption rather than performance. Reducing the traffic between the GPU and the memory is one of the key techniques to reduce the power consumption in the architecture level design [\[GProceUnits\]](#).

For reducing this consumption, different techniques are implemented on-chip, mainly the techniques are focused to avoid unnecessary memory access, to reduce system memory transactions during the rendering process caches for pixel, texture and vertex are implemented this provides better performance since cache access has less latency than off-chip memory [\[iPackMan\]](#). another technique used to reduce the memory number transactions, is store the compressed data (e.g. compressed textures, vertex and frame buffers), for then to be decompressed on-the-fly by GPU cores before processing [\[iPackMan\]](#).

Another interesting technique implemented is culling stage before rendering in a rendering pipeline thus culling remain as fixed function like NVIDIA Tegra GPU, see [\[BHEG\]](#), traditional graphics pipeline usually renders all the polygons including the occluded ones, and then culling technique display the polygons according the depths of the polygons, thus minimize unnecessary memory access and helps to lower the power consumption.

This scenario with different kinds of hardware, makes necessary that exists only one way to communicate with different hardware from the programming perspective, the graphic APIs solve this, see **section 1.4**, although in its early days, computer graphics had no standard programming models, vendors provided a low-level interface to their hardware and each programmer or development group create their own approach for create a screen display, obviously this is not very efficient or portable.

At the beginning, the GPUs did not have much flexibility, because the pipeline are fixed see **section 2.1** the rendering pipeline, the GPUs have increased the power and flexibility, because they have increased their programmability in pipeline stages, obviously, exists other factors that has done evolve the GPUs as well. Today, not only rendering tasks are executed in a GPU, another tasks can be executed on them, for example physics simulation or collision detection.

On the other hand, in the last years, the GPU computing power has been higher than advanced multicore processors, for this reason, the GPU has become very popular for the general-purpose computing algorithms and not just for graphics tasks like real-time rendering. The general purpose computing over GPU is known as GPGPU (General-Purpose Computing on Graphics Processing Unit), obviously this is another history, that is outside the scope of this work.

1.3.1 Desktop GPUs

The most popular desktop GPUs vendors have been along the years, Nvidia, ATI that was bought by AMD in 2006, and Intel, obviously there are other vendors, but here, only a small reference are made to the most popular. The most popular desktop GPUs are at the beginning, Riva series following by GeForce series manufactured by Nvidia, ATI starts with Rage series and following by Radeon series until nowadays. Intel has manufactured different generations of GPUs starts at the first generation until nowadays with the generation 11.

1.3.2 Workstations GPUs

The workstation GPUs there are different series for each vendor, Nvidia has the Quadro series and Tesla series mainly, in the case of AMD, has different series such as Fire, Radeon PRO series, finally, Intel GPUs are shared with the same architecture between desktop and workstation GPUs and are embedded within the CPU processor.

1.3.3 Mobile GPUs

The major SoCs and the mobile GPUs available in the market include Qualcomm's Snapdragon SoC with the Adreno 200 GPU[Qualcomm], TI's OMAP3 SoC with the PowerVR SGX 530/535[TiOmap3], and Nvidia's Tegra2 SoC with its own ultra-low-power(ULP) version of GeForce GPU[BHEG].

For more information see the list of GPUs for different vendors[ListNvidia][ListAMD][ListIntel].

1.4 Graphics APIs

An Application Programming Interface (API) defines the way that applications interact with components of a computer system. In case of the graphics APIs, this interface is typically implemented by driver software that is written by graphics hardware vendors.

By having a standard API, applications can be written so that they work with many different kinds of graphics hardware see **section 1.3**. For example, the same app will run on a determinate device, no matter which vendor supplied its GPU design, because the graphics drivers for each type of hardware expose the same API to the applications, although the underlying GPU hardware architecture is very different.

The most popular graphics APIs, for many years have been OpenGL and Microsoft's Direct3D, which is part of DirectX, DirectX is more than just a graphics API. DirectX contains tools to deal with sound, input, networking, and multimedia. Finally, recently appeared a new generation API called Vulkan. The following sections explains these graphic APIs in more detail.

1.4.1 OpenGL

OpenGL was originally introduced by Silicon Graphics in 1992. It is important to highlight, that OpenGL is not a stand-alone library, is only a specification and its implementation, it depends on the platform where developing for, so, OpenGL makes no hardware support assumptions, the specification only says what should be done, but does not say how it should happen, or how fast it should work.

OpenGL has been supported by the most operating systems available until nowadays. This makes it, the first choice for developing portable graphics applications, OpenGL is a pure state machine

that contains different switches working in a binary state (on/off). These states are used to build dependency mapping, in the vendor driver, to manage resources and control them in an optimal way, to yield maximum performance. Graphics hardware vendors and other graphics related companies have organized themselves as the OpenGL Architecture Review Board (ARB), which leads and defines the OpenGL interface specification. OpenGL, uses the extension concept for early integration of new features provided by the graphics accelerators. Since 2006 OpenGL has been managed by the non-profit technology consortium Khronos Group[khg].

In 2004 OpenGL 2.0 was introduced, in this version, the functionality in the graphics pipeline was fixed, this means, there were a fixed operations set hard-wired in the graphics hardware, and it was impossible to modify the graphics pipeline. However, in this version the shader objects were introduced for the first time, that enabled to do changes in the graphics pipeline by the programmers, through special programs called shaders, which were written in a special language called OpenGL shading language (GLSL) see **section 1.5**.

In 2008 OpenGL 3.0 was introduced, in OpenGL 3.x the major drastic changes in the OpenGL history has been made, from this version starts the modern manner to program computer graphics, and is the approach used in this project.

Two profiles, the core profile and the compatibility profile exist in OpenGL 3.x. The core profile basically contains all of the non-deprecated functionality, whereas the compatibility profile retains deprecated functionality for backwards compatibility. The last OpenGL 3 version is 3.3, released in 2010. After, other versions has been appeared going through version 4 onwards, however, the changes introduced are not as drastic. The current OpenGL version is 4.6 released in 2017, at the time of writing this document.

In 2003 was released OpenGL ES 1.0, specially designed for embedded systems like smartphones and tablets. OpenGL ES is a subset of widespread adopted OpenGL standard used in desktop systems and video game consoles. This subset removes some redundancy from OpenGL API, such as multiple methods that perform the same operation, the most useful method was adopted and redundant methods was removed. At the same time, new features was introduced to address specific constraints of handled devices focused for example to reduce the power consumption and increase the shaders performance[OpenGLES].

1.4.2 Direct3D

Direct3D is OpenGL main competitor, and its follow-on Vulkan, nowadays Direct3D is developed by Microsoft in cooperation with some graphics card vendors like NVIDIA or AMD, unlike OpenGL, Direct3D works under Windows only, and new functionalities are exposed through API changes on top of that, so Microsoft change the API several times. Otherwise, the new functionalities in OpenGL are introduced firstly in ARB extensions, and later on are introduced in OpenGL core, for this reason, the core changes slowly than Direct3D.

Direct3D dates back to 1995 when Microsoft was working on a new operating system called Windows 95, in those times MS-DOS was the game programming platform, MS-DOS allowed direct access any part of the system, such as graphic cards, mouse, keyboards and sound devices, but Windows 95 restricted this access, so, it was needed a way to have access to this devices thought Windows 95. In February 1995, Microsoft bought Render Morphics company, which developed a 3D graphics API named Reality Lab which was used in medical imaging and CAD software, then Microsoft starts to develop a 3D graphics engine for Windows 95. The first version of Direct3D was released at June 2, 1996 shipped in DirectX 2.0. and then followed DirectX 3.0 at September 26, 1996. DirectX is a collection of APIs for handling tasks related to

multimedia, such as video games among others, see DirectX version history[DirectXhis]. Since DirectX 8 released in November 2000, DirectX3D has superseded the Direct Draw framework and also taken responsibility for the 2D graphics rendering as well. Microsoft strives to continually update DirectX3D to support the latest technology available on 3D graphics cards, actually the latest DirectX version is 12.0, DirectX 12.0 is a new generation graphic API and is a direct competitor of Vulkan. The language shaders used are HLSL (High-Level Shader Language) that developed by Microsoft for DirectX 9 API.

1.4.3 Vulkan

Released in 2016 by Khronos group[khg], Vulkan API is next-generation 3D graphics API like DirectX12.0, however, exist competitors such as Microsoft's DirectX 12 and Appel's Metal, nevertheless, DirectX is limited to its Windows variants and Metal to Mac (OS X and iOS). Vulkan like OpenGL is cross-platform and supports almost all the available OS platforms, this list includes Windows 7, 8, and 10, Linux, Tizen, SteamOS and Android.

The original Vulkan project, was designed and developed by AMD, based on their proprietary Mantle API. Mantle displayed cutting-edge capabilities through several games, thereby testing its revolutionary approach and fulfilling all the competitive demands of the industry. AMD made their code open source and donated it to Khronos group[khg], that together other vendors made collaborative efforts to release Vulkan.

Vulkan API has a new architecture, that takes full advantage of modern graphics processor units to produce high-performance graphics and general-purpose algorithms calculation. As been said, Vulkan is often referred to as the next generation graphics and compute API for modern GPUs. It is an open standard, that aims to address the traditional APIs inefficiencies such as OpenGL, which were designed for single-core processors and does not fit well to modern hardware[FixOpenGL].

Vulkan aversely, was designed with multi-threading support in mind, multiple threads work asynchronously, feeding the GPU in an efficient manner. This is achieved in Vulkan by having no global state, jointly with separating work generation from work submission, and no synchronizations in the driver. The other Vulkan characteristic key, is that it provides a much lower-level fine-grained control over the GPU, enabling developers to maximize performance across many platforms[VulkanBench].

1.5 Graphic effects shaders

The shaders are used widely in several computer graphic applications, to produce a very wide effects range. simple lighting models are generated with shaders and more complex uses, like alter the hue, saturation brightness or contrast of an image, other effects can be image blurring, light bloom, volumetric lighting, normal mapping for depth effects, bokeh, cel shading, posterization, bump mapping, distortion, chroma keying (so-called "bluescreen/greenscreen" effects), edge detection and motion detection, psychedelic effects, and many others.

1.5.1 History of shaders

The shaders are relatively recent phenomenon, but the history of effects on computer graphics goes back to 1977 when "Star Wars Episode IV: A New Hope" was filmed, this movie did use some computer graphics, mainly vector-based effects, even though what it did was well below the capabilities of that time. Then in 1980, the computer division of Lucasfilm was created for image processing in 2D and 3D graphics rendering by hardware[GraphicShaders].

In 1983 Lucasfilm separates the 2D and 3D into their own company and the 2D group was called Pixar and this was sold to Steve Jobs in 1986. The 2D group created a hardware device called Pixar Image Computer (PIC)[PIC] to perform image processing. The PIC used 4-way SIMD (single instruction multiple data) operations to perform image processing on all four RGBA components simultaneously, the actual OpenGL GLSL language, uses the evolution of the PIC SIMD paradigm[GraphicShaders].

However, Pixar abandoned the project to focus on 3D rendering by hardware and created the prototype REYES system hardware rendering[Reyes], at the end, the hardware idea was abandoned, in favour of a general-purpose software solution, which became the package called *PhotoRealistic RenderMan (PRMan)* rendering engine[GraphicShaders], this software is used by Pixar for their own films among other things[RenderMan], however, *RenderMan* is not a real-time rendering engine, see **section 1.6.2** ray-tracing render engines.

It is important to highlight, that the modern shaders use, was introduced by Pixar with their *RenderMan* interface specification in Version 3.0, originally published in May 1988. Before this, in 1984 Rob L. Cook from Pixar and co-creator of the *RenderMan* published "Shade Trees" paper[ShadeTrees] in which he showed, how rendering process could be modified by user writing a "scripts", and inserting them in a suitable places in the rendering pipeline, this concept is still valid today. This concept allowed to create a lot of effects without having to constantly be adding new code permanently into the render. Quickly, this concept was used for commercial purposes, in 1985 this was used in the movie *Young Sherlock Holmes*, which created the Stained Glass Knight shown in **figure 1.5** and [sgk]. Other works was done until today, some of them are [slgh][rtps].



Figure 1.5. a shot from *Stained Glass Knight*, *Young Sherlock Holmes* movie, copyright Amblin entertainment/ILM/Paramount pictures 1985, All rights reserved.

1.5.2 Modern shader languages

By earlier 2000s, the graphic hardware had evolved enough for needed a flexible shading capability that Rob L. Cook described in 1984 in his paper "Shade Trees"[[ShadeTrees](#)]. This fact, carried out that the first implementations of modern shading languages appeared, this implementations were Cg developed by NVIDIA[[SysProgGrH](#)][[CgTutorial](#)] and HLSL(High Level Shader Language)[[DirectX9intro](#)] developed by Microsoft as part of its Direct3D graphics API, and shortly after, GLSL (OpenGL Shading Language) was created by the OpenGL Architecture Review Board (ARB) as part of OpenGL graphics API. Cg and HLSL were developed at the same time but are separate products. All this languages, have the same functionality, vertex, geometry and fragment or pixel shaders with a C-like language, and with them can be access to key data values within the graphics pipeline.

1.6 Render engines today

In **section 1.1** has been explained the concept of real-time rendering, but the rendering concept can be split up into two main categories, real-time rendering treated in this project and pre-rendering. The real-time rendering is also known as online rendering and pre-rendering as offline rendering, offline rendering is used to create realistic images and movies where each frame can take hours or days to complete. In this section, some of the current engines that use any of these techniques will be listed, separated them by technique, in case of pre-rendering, some ray-tracing engines will be cited, the engines, can be open-source or proprietary and some of them may be are not only a pure real-time rendering engines, some of them can be game engines, since game engines have a real-time rendering engine built-in.

Obviously a lot of engines exist nowadays, and is impossible to refer to all, therefore only some them will be cited see[[OffLineRenderersList](#)] for offline rendering and[[GameEngineList](#)] for game engines lists for more information.

1.6.1 Real-Time render engines

1.6.1.1 Ogre3D

Ogre (Object-Oriented Graphics Rendering Engine) is a real-time rendering engine, is not a game engine the software only is real-time renderer, is written in C++ and the initial release was in February 2005. Implements the following graphic APIs, Direct3D 9 and 11, OpenGL included ES2, ES3 and OGL3+ and WebGL(Emscripten), is platform independent and support Windows all major versions and WinRT, Linux, Mac OSX, Android and iOS. This software has been use in professional video games development, the game torchlight of the Runic Games is an example of this.

License: MIT

For more information: <https://www.ogre3d.org/about/features>

1.6.1.2 Open Scene Graph (OSG)

A powerful rendering middleware based on the theory of scene graph see **section 4.2.4** scene graph, is written in C++, OpenGL 1.0 to OpenGL 4.2 and OpenGL ES 1.1 and 2.0 are supported, is cross platform running on small devices such as embedded graphics platforms to phones, tablets, laptops and desktops and dedicated image generator clusters used in full scale simulators and immersive 3D displays. This product is mainly used in the following fields: Visual simulation, virtual and augmented reality, medical and scientific visualization, to education and games.

License: OpenSceneGraph Public License, is a relaxation of the Less GNU Public License (LGPL)
For more information: <http://www.openscenegraph.org/index.php/about/features>

1.6.1.3 Irrlicht

Is another real-time rendering engine, is written in C++ and the initial release was in 2003, is cross platform and runs on Windows, Linux, OSX, Solaris/SPARC and have implemented several graphics APIs: Direct3D 9.0, OpenGL1.2 to 4.x, the engine has its own software render layer implemented as well.

License: zlib license
For more information: http://irrlicht.sourceforge.net/?page_id=45

1.6.1.4 Magnum

Lightweight and modular C++11/C++14 graphics middleware for games and data visualization.

License: MIT/Expat
For more information: <https://doc.magnum.graphics/magnum/>

1.6.1.5 Horde3D

Horde3D is a small open source 3D rendering engine. It is written in an effort to create a graphics engine that offers the stunning visual effects expected in next-generation games, while at the same time being as lightweight and conceptually clean as possible. This engine is supported by the University of Augsburg. The engine only implement OpenGL rendering API layer, hence is cross-platform compatible written in C++.

License: Eclipse Public License v1.0 (EPL)
For more information: <http://www.horde3d.org/features.html>

There are others real-time rendering engines, although are not a pure real-time rendering engines, since they are also game engines, some of them are, Urho3D, Godot, Unity, Unreal Engine, cryEngine, so forth. Some of them are open-source and others are proprietary, for a non complete game engine list, see [\[GameEngineList\]](#).

1.6.2 Ray-Tracing render engines

1.6.2.1 RenderMan

This framework has been developed by Pixar for the last 30 years, see [section 1.5.1](#) and [\[RenderMan\]](#). RenderMan is a high performance renderer, is the state-of-the-art ray tracing framework. Generates high-quality 2D images from 3D scene information typically created with a 3D design software such as Autodesk Maya, Katana or Houdini. This software send data to RenderMan interface for rendering.

RenderMan has much in common with OpenGL, despite the two APIs being targeted to different sets of users, OpenGL to real-time hardware-assisted rendering and RenderMan to photorealistic off-line rendering with ray-tracing techniques. Both APIs take the form of a stack-based state machine with conceptually immediate rendering of geometric primitives.

Licence: Proprietary

For more information: <https://rmanwiki.pixar.com/display/REN22/RenderMan>

1.6.2.2 Mental-Ray

This engine was developed by mental images in Germany, the initial release was in 1989, in 2007 the company was acquired by NVIDIA and was rebranded as NVIDIA Advanced Rendering Center (ARC). The company provides rendering and 3D modelling technology for entertainment, computer-aided design, scientific visualization and architecture. However, Mental Ray has been discontinued since 2017 by NVIDIA.

Like RenderMan works via plug-ins with a 3D design software such as Autodesk Maya, 3ds Max, despite in this case, can work as stand-alone as well in remote or local machines.

Licence: Proprietary

For more information: <http://www.nvidia-arc.com/index.php>

https://en.wikipedia.org/wiki/Mental_Images

1.6.2.3 POV-Ray

Is a ray-tracing engine which generates images from a text-based scene description. It was originally based on DKBTrace[DKBT] programmed by Commodore Amiga. The initial release was in 1991 and is a cross-platform engine written in C++. Pov-Ray is an open-source software.

Licence: AGPLv3.

For more information: <https://www.povray.org/>

There are other rendering engines such as, V-Ray, Arnold, LuxRender so forth. for a non complete list see[OffLineRenderersList].

1.7 Project goals

The main objective of the project is to build a real-time rendering engine with a modern graphics approach completely decoupled from the graphic API and cross-platform, to achieve this approach, object-oriented programming and software engineering in general will be used, defining different layers and an interface that will be implemented with a specific graphic API, in this way, the implementation of the graphic API is decoupled from the rest of the engine.

So, the project focuses on developing the initial pillars of a real-time rendering engine maintaining a scalable structure so must be possible to expand or add more features to the engine in the future without changing the main engine structure. Due to implementing a rendering engine is a relatively big and complex software project, this work aims to highlight the initial steps to build such software from scratch from a point of view as simplified as possible.

Obviously there are good books and other documentation about this topic, but mostly are more complex, however, should be consulted for more information. Throughout the project reference will be made to any of this documentation, see **section 7** bibliography and resources, and see **section 3** engine architecture and features.

1.8 Document structure

The document is structured as follows, the **section 1** introduces the real-time concept and the real-time rendering engine, jointly with a graphics hardware review from the beginning, the section also introduces the different graphics APIs, which can be used to program the different graphic hardware present nowadays in a common manner. The history of the effects that is very related to shaders programming is also discussed. Finally a small revision of some current real-time rendering engines and ray-tracing engines is made, jointly with the project goals. The **section 2** review the basics of real time rendering, such as the render pipeline, the GPU architecture and how to program it. The **section 3** exposes the engine architecture and their features, to finally discuss in **section 4** all the design and implementation of the engine. **Section 5** shown with a simple example, how a specific application that uses the engine should be structured. **Section 6** discuss the project summary and future works and **section 7** collect the bibliography and documentation used for this project.

2 Basics of real-time rendering

2.1 The rendering pipeline

Real-time rendering engines perform different steps repeatedly, displaying rendered images at a rate of 30, 50 or 60 frames per second to provide the illusion of motion, to get this, a 3D virtual scene is described and a virtual camera is positioned and oriented to produce the desired view, then various light sources are defined jointly with the visual properties of the surfaces with materials and textures, finally all this is taken to the screen of the device making a rasterization of triangles, all this can be seen like the steps of rendering. This rendering steps are implemented using a software/hardware architecture known as a pipeline or rendering pipeline.

The pipeline is just an ordered chain of stages where each stage has specific purpose, operating on a stream of input data items and producing a stream of output data. The pipeline has a parallel architecture, so each stage can typically operate independently of the other stages. This parallelization occurs both within each stage and globally. This implies that the pipeline speed is determined by the slowest pipeline stage.

The render pipeline is accommodate by the GPU for rendering and describes what steps a graphics system needs to perform to render a 3D scene to a 2D raster images onto display screen, the steps to achieve the 2D images onto display screen depend on the software and hardware used, so, the render pipeline refers to the state of the art methods used for rendering, and consequently does not exist a universal render pipeline, each graphic APIs tends to unify similar steps to abstract the underlying hardware, and the most of the render pipeline steps are implemented in graphic accelerator hardware. Hence, the render pipeline has two different levels, one is the software API level, such as OpenGL, Direct3D or Vulkan, that provide a logical frameworks of how the 3D scene must be rendered, and the other is the real hardware implementation level that it depends the underlying hardware in the system, so, the render pipeline performance is strongly linked to the hardware architecture.

Here, is shown a conceptual stages of the render pipeline without reference to any specific graphic API or specific hardware, to explain how the rendering pipeline transforms data to achieve a 2D raster scene from a conceptual point of view.

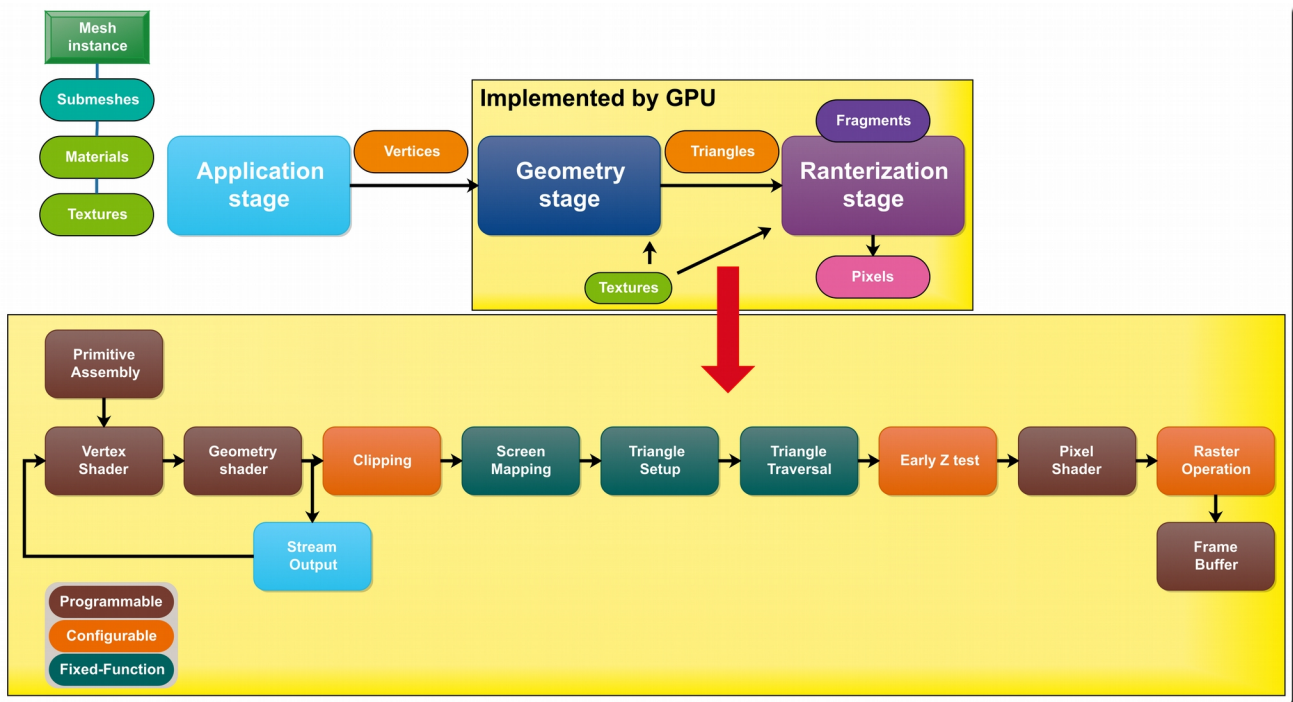


Figure 2.1 depicts how the geometric data changes when it passes through the various stages of the rendering pipeline

The previous steps to the application stage are **offline tools** used to create the materials, textures, models and scenes, that will be rendered by the engine. This tools can be different design applications like Blender, Maya, 3ds Max, Gimp, Photoshop and so on.

The **application stage** is responsible to feed the geometry processing stage, discussed below, the application stage occurs in CPU level and is possible to identify three main roles, the visible mesh are identified and send to the GPU, hence only the visible mesh are rendered. The geometry are send to the geometry stage via some API rendering call command like **gl*Draw*** in case of OpenGL. Finally this stage makes a shader parameter and render states control, hence the uniform parameters passed to the shader are configured by the application stage to ensure that each geometry is rendered properly.

The **geometry stage** break down the mesh into individual vertices, which are processed largely in parallel. This stage occurs in GPU and implements the following stages.

- **Vertex shader**, this stage is responsible for transformation and shading/lighting of individual vertices, the lighting applied in this stage is called **Gouraud shading**. On modern GPUs the vertex shader has full access to texture data. In this stage is applied the type of projection, among others. This stage is programmable.
- **Geometry shader**, is optional stage programmable that operates on entire primitives such as triangles, lines and points in homogeneous clip space.
- **Stream output stage** is present in some GPUs, and permit amazing visual effects to be achieved without the aid of CPU, an example of this can be the hair rendering.
- **Clipping stage**, this stage clipping the primitives are partial or totally outside the visual volume defined inside the frustum, the frustum is a shape in the form of a pyramid with a cut off top. When the primitive is totally outside is discarded with frustum culling, when the primitive are partially outside is clipped, identifying vertices that lie outside the frustum and then finding the intersection of the triangle's edges with the planes of the frustum. These

intersection points become new vertices that define one or more clipped triangles. This stage is configurable.

- **Screen mapping stage**, scales and shifts the vertices from homogeneous clip space into screen space. This stage is fixed and non-configurable.
- **Triangle set-up stage**, this stage is non-configurable and the rasterization hardware is started for convert the triangles into fragments.
- **Triangle traversal stage**, each triangle is broken into fragments and usually each fragment correspond to a pixel, but this depends the antialiasing techniques since multiple fragments may be created per pixel. This stage is fixed and non-configurable.
- **Early z-Test stage**, in the older GPU designs the z-test was done along with alpha testing, after pixel shader, but in modern approaches is done before pixel shader and this is the reason that is called early z-test. This stage checks the depth of the fragment and is discarded it, if it is being occluded by the pixel already in the frame buffer. This stage is configurable.
- **Pixel shader**, this stage has different jobs, for example, apply the light in each fragment run the per-pixel lighting called **Phong shading**, determine the fragment's color, can also discard transparent fragments. This stage is programmable.
- **Raster Operation stage**, this stage is configurable, mainly this stage converts triangles into fragments that are shaded, passed through various tests (z-test, alpha test, stencil test, so forth.) and finally blended into the frame buffer.

For more information, see [\[GengArchPipeLine\]](#).

2.2 GPU Programming

A GPU is designed specifically to work with a high degree of parallelism to perform data-parallel computations on very large datasets. In recent years, all GPUs employ the general principles of SIMT(single instruction multiple thread) parallelism in all architecture designs. The SIMT classification was formulated by NVIDIA and has been added to Flynn's taxonomy to refer to the design of graphics processing units (GPUs). However, the design is not unique to NVIDIA GPUs, other GPU vendors also apply SIMT, although the specifics of GPU designs vary from vendor to vendor and from product line to product line in significant ways.

The SIMT is basically a combination of SIMD parallelism with vectorized ALUs with MIMD parallelism. The parallelism within the GPU, occurs so that the elements are processed in any order to obtain the final result.

The SIMD vectorization perform data-parallel computations, so, is possible to depict an example of this with a two potentially very large arrays of input vectors, that they produce an output array containing the scalar dot products of those vectors. See the following code snippet, this code has been extracted from [\[GengArchGPUprog\]](#).

```
void DotArrays_ref(unsigned count, float r[], const float a[], const float b[])
{
    for(unsigned i = 0; i < count; i++) {
        //treat each block of four floats as a
        //single four-element vector
        const unsigned j = i * 4;
        r[i] = a[j+0]*b[j+0] //ax*bx
        + a[j+1]*b[j+1]     //ay*by
        + a[j+2]*b[j+2]     //az*bz
        + a[j+3]*b[j+3];    //aw*bw
    }
}
```

This code when it is executed with a SIMD parallelism, the computation performed by each iteration of a loop is independent of the computations performed by the other iterations, so that the computations occurs in any order instead of performing the computations one by one. So with a SIMD approach, it is possible to make different computations simultaneously, so they if done, four, eight or sixteen computations simultaneously, then we are reducing the iteration count by a factor of four, eight or sixteen, respectively. This concept in a GPUs are carried to the extreme adding a large number of computations simultaneously, thus if we had a GPU with 2048 lanes and the input array in the code snippet example contained 2048 elements or fewer, would be possible literally execute the entire loop in a single iteration.

A GPUs contains many of these SIMD units, instead of 2048 lanes, each SIMD unit has typically eight or sixteen lanes, so, a modern GPU is capable of processing literally thousands of data elements in parallel, hence, this architecture makes the GPUs very suitable to compute graphics data, since a GPU must deal with millions of pixels when a pixel shader is applied or with hundreds of thousands or even millions of 3D mesh vertices when the vertex shader or the geometric shader comes into play, in each frame at 30 or 60 FPS. At the same time, modern GPUs also expose their computational power for a general-purpose use, where one of the main pillars is the vectorization, this phenomenon is known as GPGPU (General-Purpose Computing on Graphics Processing Unit).

In the previous section it has been explained the GPU pipeline, and up to this point, it has been explained the common architecture of GPUs nowadays, along its computational power, appropriate to manage graphics data. Hence, the programmable shaders comes into play, when we want to program a GPU for graphics, this programs modify the behaviour of the pipeline stages, see [figure 2.1](#), the programmable stages concretely, and [section 1.5.2](#) modern shader languages.

The following sections introduces the basics of the three types of shader programs, available in modern graphic approaches, which have been illustrated in [figure 2.1](#), mainly, a shader takes a single element of input data and transforms it into zero or more elements of output data.

2.2.1 Vertex Shaders

The vertex shader is executed per each vertex, so a vertex shader can only access the vertex that it is managing, so, fetching the data of another model vertex is not possible. At the vertex shader input there is a vertex with its attributes, these are computed, to get a vertex transformed or illuminate, if the lighting is computed within the vertex shader. Finally this type of shader can not create new vertices.

2.2.2 Geometry Shaders

The geometry shader is capable of generate new primitives including new vertices, so this type of shader generate new geometry, so that is possible to convert a determinate type of primitive into another. At the geometry shader input there is a single n -vertex primitive, so $n=1$ will be a point, $n=2$ will be a line segment, and $n=3$ will be a triangle. The output could be zero or more primitives, thus it could convert points into two-triangle quads, or it could transform triangles into triangles but optionally discard some triangles and so on.

2.2.3 Pixel Shaders

The pixel shaders are applied to each fragment, this fragment have been interpolated from the three vertices of the triangle from which it came during the rasterization process. The output of a pixel shader is the final pixel color that will be written into the frame buffer, it is worth pointing out

that a pixel shader is also capable of discarding fragments explicitly, where in which case it produces no output.

2.3 Scene graph and spatial data structures

A rendering engine is a complex piece of software and there is no standard form of design. However, there are some fundamental design philosophies that are largely linked to the design of the underlying 3D hardware. A common and efficient approach is to use a layered architecture, as will be seen in **section 3.2** engine architecture. The layer that implement the graphic API independent rendering interface provided by the underlying layer of the rendering engine is simply focused on representing a collection of primitives efficiently, however, this layer does not take into account that parts of the scene are visible, the visible parts of a scene are those that are inside the camera frustum, the responsibility of determining which geometries are visible during the rendering process, is the responsibility of the top layer, where they combine different techniques to analyse which parts of the scene are visible during the rendering process, to achieve efficient rendering, since rendering parts are not visible, it makes no sense since it is totally inefficient.

As seen in **section 2.1** the rendering pipeline, in clipping stage, the render pipeline contains configurable stages that allow an explicit sacrifice of all objects does not lie within the camera frustum, this is known as frustum culling, however, this is not enough since making an explicit sacrifice of all objects in complex scenes are usually an incredible waste of resources. This leads to the need to have a data structure that manages all the geometry of the scene in an upper layer that decides which objects are visible to send them to the lower layer to be rendered, thus increasing the rendering efficiency put that only visible objects are rendered.

This data structure is known as a scene graph, with it, is possible to quickly and efficiently discard large parts of geometry that are nowhere near the camera frustum prior to performing detailed frustum culling. This structure is also used to order the geometry of the scene.

Normally, a scene graph in a real-time rendering engine is structured with a tree-like graph, using spatial data structures. There are different types of spatial structures such as quadtrees and octrees, BSP trees, kd trees and spatial hashing techniques, this leads to different types of scene graphs, the structure of data to be used will depend on the nature of the scenes to render.

3 Engine architecture and features

3.1 Introduction

The engine development approach is totally decoupled of the graphic API, this decoupling allows to be able to implement the rendering behaviour with different existing graphic APIs, without modifying anything of the common underlying layers, such as are **Core** layer and **Graphics** layer. To get this, both the **Core** and **Graphics** layer mainly declares an interface that finally is implemented in the **GraphicsOGL3** layer with a determinate graphic API.

The graphic API implementation, has been done with OpenGL 3.3 only, due to the engine has a modern graphics approach. Along the document, all will be written with the same approach, that is to say, independent of the graphic API used, hence, no reference will be made to anything that is specific to OpenGL, thus avoiding writing an OpenGL tutorial, since, this is not the objective of this work. Nevertheless, in the **section 4.3.1** the OpenGL functions used are referenced, however no explanation will be made in this regard. The project target is to design a modular rendering engine totally decoupled of the graphic API used in a simplest manner as possible, to get an engine with a good scalability.

Obviously, the design shown in this project is not the only possible and only is the principle that how can design a real-time rendering engine from scratch. The project, try to simplify the scenario to the fullest without losing a good scalability, and some techniques that will show here should be improved to get a better efficiency as will seen later.

At the same time, the proposed engine, is not the only neither the better, simply is a possible manner to front facing this kind of project from scratch, in a simplified way without losing scalability being important to reach more efficiency in the future, and other people, sure they could be contribute with a lot good ideas.

The project embraces different topics to get the completely product, mainly covers the essential architecture of the engine to store and manage geometric data, the geometric transformations needed to ordering the objects in a scene, jointly with the viewing transformations to build a camera to handling a 3D viewing. A system effects is proposed working with GLSL 3.3 and an OBJ file format parser. Finally the engine implements an own mathematical library.

All the real-time simulations such as games, or other graphics applications require a camera from the point of view of which the 3D perspective scene is rendered. The engine implements two camera models, perspective and orthographic, although it uses perspective projection camera model by default.

The effects are an important feature in a real-time rendering engine, because they make the rendered scenes more realistic and consequently closer to reality, although sometimes simply make them more spectacular.

The effects involves different things related such as textures, materials and lighting model, all this is implemented within the engine from an essential point of view, being scalable if applicable, is worth pointing out, that in modern graphics approach all this is ruled via shaders, so the shaders has an important role here, consequently the shader system proposed, supports the effects implemented with GLSL3.3.

At the same time, due to the constant technological advances together with the constant changes that often happen in a software project, becomes necessary mainly carefully architect the

rendering interface and obviously, this also affects the way of the new visual effects are added to the engine, so that the impact of changes must be minimal and limited to a small section of the engine when a new effect is added.

The mechanism proposed to manage the effects inside the engine is discussed in the **section 4.2.6** effects, the mechanism proposed here is an adapted and simplified version of the mechanism proposed by the **Wild Magic 5.17** engine, see [geomTools], though **Wild Magic** does not work with GLSL and the mechanism proposed by the engine is much more complex. It is important to highlight, that the mechanism implemented within the **Wild Magic 5.17** is not explained nowhere I know, although exists a book [3DGEArch], where is explained an older version of the engine, but the effects system is quite different to the implemented in the **Wild Magic 5.17**, so, this project document would be a document where is explained a simplified effect system based on **Wild Magic 5.17** simplified and adapted to GLSL (OpenGL Shading Language). At the same time, this book [3DGEArch] has been useful as well, during all the project development as a source of inspiration in some things, among other sources, see **section 7** bibliography and resources.

3.2 Engine architecture

This section introduces the real-time rendering engine architecture main parts, the project approach seeks an implementation of a rendering engine with an essential structure with an object orientation paradigm, the term essential in this case, must be understood as the initial pillars of the engine. The **figure 3.1** shows this essential structure. Due to this implementation target, some render engine important components have been omitted initially, to simplify the implementation, however, are not completely omitted since the basic components are implemented leaving it ready for future works, so, they will be present along design and in the future they can be expanded with minimal impact on the engine architecture, the engine implementation will be discussed in **section 4** on this document.

Mainly, the parts that have been omitted are the scene graph and spatial data structures management, these are used to organize the scene objects in a hierarchical way via tree-like structure, with acyclic graphs usually to render scenes in efficient manner, since when rendering a lot of objects in graphical application in a complex scene a linear method like an array or a vector to store renderable objects becomes less useful.

However initially, a linear method with a vector to store renderable objects is implemented in conjunction with the assumption that all objects are visible initially, since neither geometry discard techniques are implemented, this linear method should look like a render queue, that in the future, will be replaced with a complete scene graph implementation, with the geometry discard techniques corresponding to improve the rendering efficiency.

The engine is mainly organized with three layers or modules, see **figure 3.1**, each module has a name that defines the global functionality in the engine, these modules are defined as follows: **Core**, **Graphics** and **GraphicsOGL3**. These modules are related between them starting from the bottom layer to the top layer, in this case the **Core** layer is related to the **Graphics** layer and the **Graphics** layer is related to the **GraphicsOGL3** layer, and so, all the modules build the engine. Henceforth and during all this project both terms module and layer will be used, referring to the same concept. The reason to split the engine with modules pursues the goal to group the functionality of the rendering engine by topics and software level, at the same time, to get more definition granularity the modules are split into submodules internally using namespaces.

Below is a brief description of each module that will be expanded in **section 4** engine modules definition and implementation.

The **Core** module, is the lowest software level of the engine, maintains the system functions a general application framework and common utility functions like mathematics library especially designed and implemented for this render engine, with the common functionalities for computer 3D graphics, all is separated in submodules.

The **Graphics** module, is build on top of the **Core** module, is a platform-independent module that defines different submodules with different data structures and mechanism to store and manage the geometry data in computer memory, before being sent to the graphic card memory to be rendered.

The **GraphicsOGL3** module, implements the interface defined in the **Graphics** module, this implementation builds the bridge between the rendering interface and OpenGL in this case, this module is platform-dependent and could be implemented with different manner with other graphic API like Vulkan or Direct3D as well, making different modules for each graphic API, but in this project is implemented with a modern computer graphics paradigm with OpenGL 3.3 only.

Each module and submodule is defined as a name space, with the same name as the module and submodule, and all name spaces are inside another name space called "**Flwre::**" this acronym defines the rendering engine name "**Flow Render Engine**".

3.2 Rendering interface approach

The rendering interface main goal, is to provide an easy-to-use entry for coding graphics both in 2D and 3D. This approach must to enable the developer should not bother about graphics API specific details. Therefore the interface abstracts graphic device features in a graphics-API-independent way.

The rendering interface itself, provides all common rendering operations like creating, vertex and index buffers, textures, shaders, with a factory approach, finally this interface is implemented in the **GraphicsOGL3** module with the OpenGL graphic API, which can be loaded and switched at runtime, therefore this module must be seen as a plug-in. The interface concept makes the **Graphics** module completely independent from the underlying hardware and any graphics API like OpenGL, Direct3D or Vulkan.

Concretely in this project the interface is implemented with OpenGL 3.3 only, however, the interface approach leaves ready the engine to be implemented with other graphics APIs like Vulkan or Direct3D, but in case to be implemented it, with Direct3D, the engine and the applications would not be portable to other platforms for obvious reasons, since Direct3D is a graphic API for Microsoft Windows only.

The following figure, depicts the structure discussed above, and how the applications use the rendering engine in conceptual manner, the diagram shows **GraphicsVK** and **GraphicsDX** modules, but this kept in grey color, due to not implemented in this project, however it shows for better understanding.

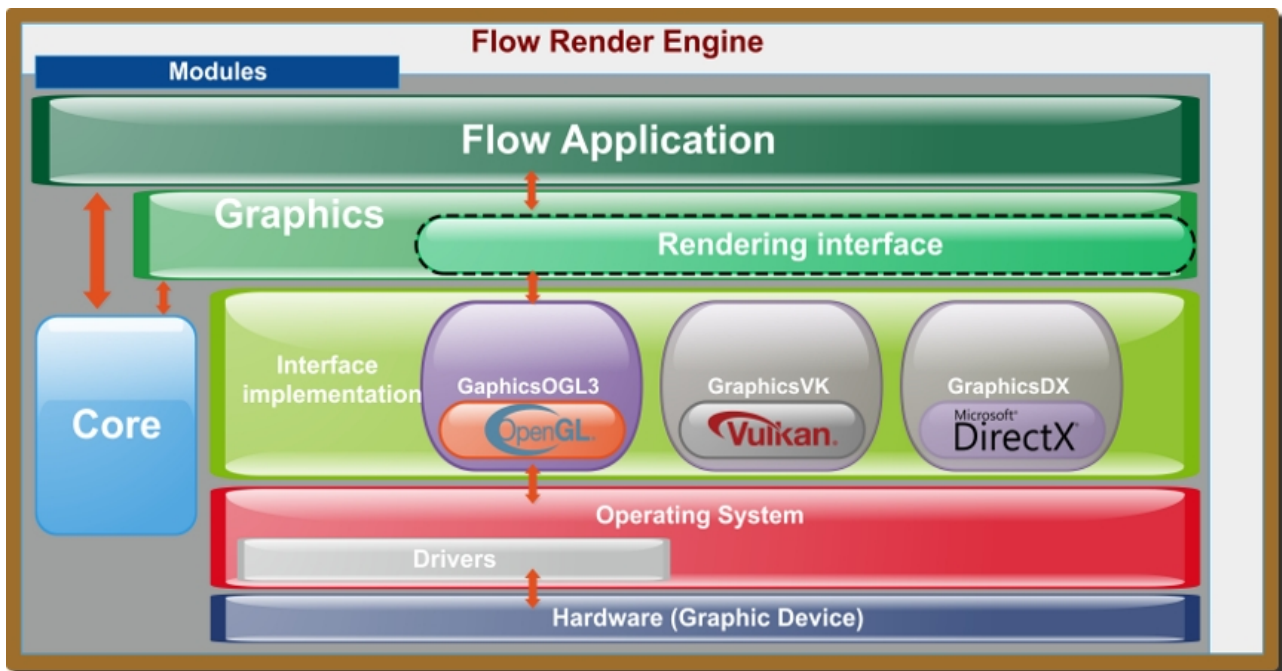


Figure 3.1 depicts the engine architecture and how the application using the engine in conceptual manner.

3.3 Fluent interface approach

The fluent interface approach, is a method for designing object orientated APIs based on method chaining to the purpose of making the source code more readable close to that of ordinary written prose, this approach creates a domain-specific language within the interface implemented, see [\[fluentInterface\]](#).

A fluent interface is implemented, using method chaining to implement method cascading in languages that do not natively support cascading like C++, concretely by having each method return a pointer to itself.

The engine is implemented with C++ with this approach, so, the API engine allows write code like this:

```
triangle = new Flwre::Graphics::SceneGraph::PrimitiveMeshShapes();
triangle->Triangle().create();
```

instead of:

```
triangle = new Flwre::Graphics::SceneGraph::PrimitiveMeshShapes();
triangle->Triangle();
triangle->create();
```

However, the latter would also be valid, but the engine is implemented with fluent interface approach, so more readable and elegant code is achieved.

3.4 Design patterns used

Design patterns in the context of software engineering solve specific design problems and make object-oriented designs more flexible, elegant and ultimately reusable, since they reuse solutions that have worked in the past in different scenarios. See [\[dPatterns1\]](#)[\[dPatterns2\]](#)[\[dPatterns3\]](#) for more details on the design patterns used in this project.

In the render engine development, have been used design patterns to solve some scenarios to keep the code clean and focused on a singular purpose. Along this document is explained where have been used the following design patterns.

3.4.1 Decorator

A decorator is defined as an entity that encapsulates and hides the underlying complexity of another entity by means of well-defined interfaces. Mainly a wrapper fulfills the need of a simplified and specific programming interface. An example of this can be a C++ interface that acts as a wrapper around a C-language interface. The decorators, in this project have been applied to the managers see **section 4.2.2.1** managers.

3.4.2 Abstract Factory

This pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. To achieve this, an interface is provided known as abstract factory, which can be used by clients to create a determinate objects families related to abstract factory type. To create these objects, the clients don't know about concrete classes which get instantiated, so, the clients see only the created object interfaces, abstract products. The concrete factories implement the abstract factory interface, so, only the concrete factories know which classes should instantiate, these are known as concrete products, see **section 4.2.2.1** managers to see how this pattern is applied to the engine.

3.4.3 Singleton

When one class instance is allowed within a system only, then the singleton pattern comes into play. The singleton pattern ensures, that a class has only one instance and provides a unique access point to the class. The main characteristics are: the class constructor is always declared as a private to ensure no one else can make an instance of it. The pattern itself is responsible to provide a global access point to its instance, to accomplish this, the class provides a method which creates the global instance. This pattern is applied in different engine parts, see **sections 4.2.2.1** managers.

3.5 Engine features

- Written in pure C++ and totally object oriented.
- API with fluent interface approach for more readable code.
- Real-Time 2D and 3D rendering using OpenGL 3.3 only.
- No third-party mathematics libraries used, own implementation mathematical library.
- Platform independent. Initially runs on Windows only.
- Builds on various compilers, such as MSVC and GCC 5.1+ under windows.
- Vertex and Fragment shader support written in GLSL 3.3, with a certain structure, which the engine will recognize.
- Direct import mesh file format: Wavefront OBJ with own parser with triangulate faces loaded supported only, no third-party libraries used.

- Direct textures import in several image file formats: JPG, PNG, TGA, BMP, PSD, GIF, HDR, PIC via third-party library to import images, see `[stdImage]` `std_image`.
- Camera models supported: perspective and orthographic projection.
- Textures 2D supported.
 - Single texturing and multi-texturing supported.
- Sky-boxes support.
- Local Effects.
 - Lighting.
 - Materials.
 - Normal Maps.

4 Engine modules definition and implementation

This section presents the completely engine architecture and the modules definition in conjunction with its implementation. Along the document will be explained the complete engine implementation, showing the design and the modules implementation.

It is important to bright out that each module can be seen like a layer as well, and at the same time each of them becomes in a system library if the engine layers are compiled as dynamic libraries manner.

4.1 Core module

The **Core** module defined within the namespace `Flwre::Core`, works at the lowest software level of the engine, this module defines different sub-modules to manage system functions as follows:

Submodule	Short description
<i>Engine</i>	Defines a common application context and singleton template.
<i>Math</i>	A set of headers that will define the engine math library with the essential necessary mathematical functions related with computer graphics.
<i>Platform</i>	A set of headers that will define OS-dependent data types and macros to define symbol visibility in the dynamic shared objects (DSO) for different platforms.
<i>Utility</i>	Defines timer, filesystem access, engine log and dynamic loader mechanism to allows runtime dynamic libraries loading.

The following sections exposes more deeply the sub-modules presented above.

4.1.1 Engine submodule

This sub-module defines a base application context abstract class and a singleton class template, jointly are presented all components defined within the utility sub-module, see [figure 4.1](#) UML diagram. The utility sub-module will be explained in the [section 4.1.4](#).

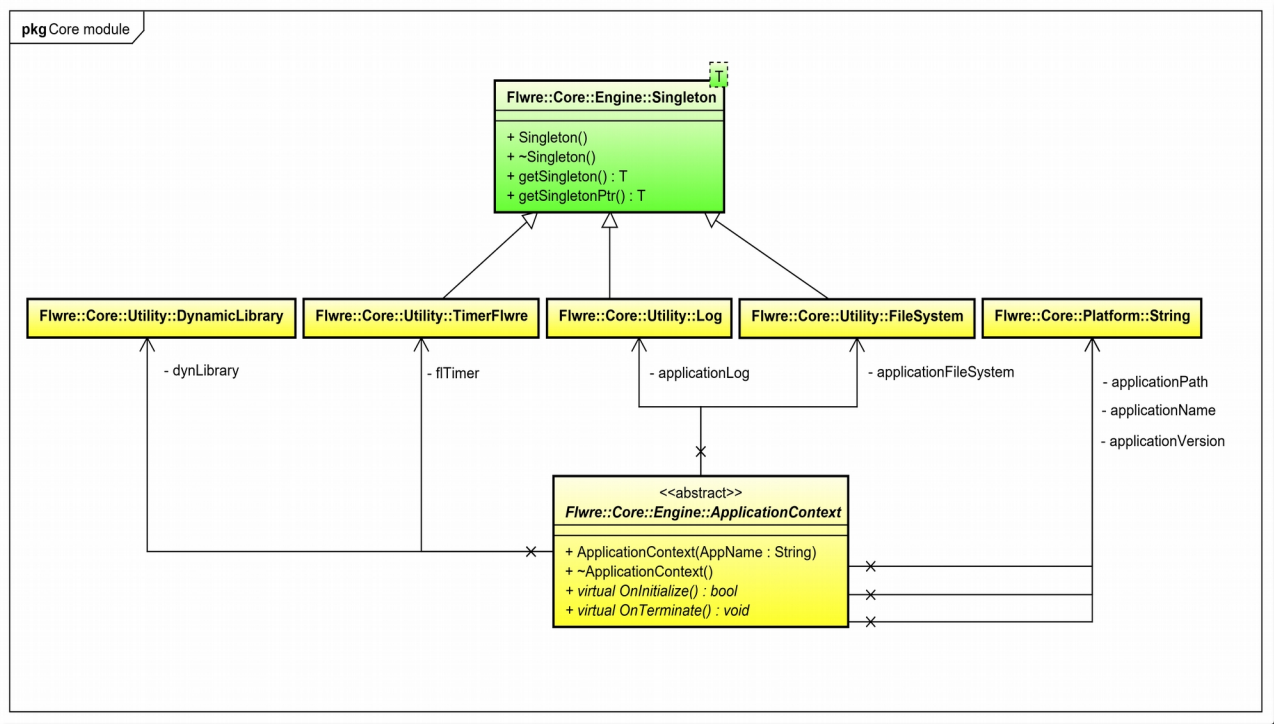


Figure 4.1 depicts the engine sub-module in an UML diagram with its associations in a simplified manner.

4.1.1.1 Application context

The engine is designed to handle a single application and the application context class defines the low-level common pointers used in the application, this class contains the minimum support for all applications types, so this class is the base class of the any application that it use the engine, initially it contains a minimum but sufficient definition for the current implementation.

For the time being windowed applications for displaying the rendering results are supported only, see [section 4.2.3](#) windowed application sub-module, however with this approach could be possible implement another applications type inside the engine, for example a console application that do not requires a window for displaying results since the engine could has different tools implemented.

The application context class is abstract and defines an interface with two pure virtual methods, **OnInitialize()** and **OnTerminate()** without committing to a particular implementation since the behaviour depends in certain manner to the type of application specialization and the behaviour implemented in derived classes, so this methods are overridden in the application that use the engine to initialize and terminate the resources used in the application itself and at the same time also those of the engine as well, when the application is started see example in [section 5](#) applications over the engine.

At the same time, has the common pointers to **File System**, **Logging**, **Timer** associations and string variables to store application name, application path and application version.

4.1.1.2 Singleton template class

Defines a template for creating single-instance to global classes, in this way all classes derived of singleton template, can be accessed from anywhere of the engine via public pointer to the singleton instance, different engine components can be accessed with this technique, see [figure 4.1](#) and [section 4.2.2.1.1](#) buffer managers.

The following code snippet shows an example how to access the class `TimerFlwre`'s `showFrameRate()` method with a singleton manner approach.

```
Flwre::Core::Utility::TimerFlwre::getSingletonPtr()->showFrameRate();
```

4.1.2 Math submodule

The name space `Flwre::Core::Math` holds the mathematics library, the library has been custom designed for the engine, and it provides different linear algebra related types like vectors and matrices jointly with the common functions to generate the common geometric and viewing transformation matrices, since that the render engine is only implemented with OpenGL for the moment, has been implemented a right-handed system in the transformation matrices only. The library is designed with a template classes approach and is a C++ header-only library and thus does not need to be compiled, basically the library is a set of headers.

The template implementation focusing allows to become independent of any particular data type, so vector and matrix classes are molds with a generic methods implemented, so there is a single definition for vectors and matrices and can be defined in this case with different kinds of numerical data types like integers, floats and so on, for instance: `vector<int>`, `vector<float>`, `matrix4<float>`, `matrix4<int>` among others are possible, see [table 1.1](#).

4.1.2.1 Basic functions

The basic constants and functions implemented are:

- $\pi = 3.141592653589793238463$
- $\pi \cdot 2 = 6.283185307$
- $\left(\frac{\pi}{2}\right) = 1.5707963267949$
- Function to conversion radians into degrees:
 - $\text{Degrees} = \left(\frac{180^\circ}{\pi}\right) \cdot \text{radians}$
- Function to conversion degrees into radians:
 - $\text{Radians} = \left(\frac{\pi}{180^\circ}\right) \cdot \text{degrees}$
- `max` and `min` functions, with the names **flmax** and **flmin**.

The `max` and `min` functions are re-implemented to avoid the interferences created with windows header file and its `max` `min` definition macros, reimplement the functions is not the better solution but for the moment it stays like this.

4.1.2.2 Vectors

Vectors are of fundamental importance in any 3D engine, since are used to represent points in space, such as the location of objects or the vertices of a triangle mesh for instance, although are used as well to represent directions such as the orientation of the camera or the surface normals of a triangle mesh as will be seen later.

The vectors are represented in various types in the computer graphics, usually are represented by two-dimensional, three-dimensional, or four-dimensional components. Although more abstract definitions are possible, here the definition is restricted to vectors defined by n -components of real or integer numbers mainly, where n is typically 2, 3, or 4 as been explained above, see **section 4.1.3** platform submodule **table 1.1**.

An n -dimensional vector V can be written as:

$$V = \langle V_1, V_2, \dots, V_n \rangle,$$

where the numbers V_i are called the components of the vector V . This is a conceptual definition of a vector, but usually the components be labelled with the name of value types that contains, for instance, the components of a three-dimensional point P could be written as P_x, P_y and P_z . Down below is explained how this scenario has been implemented within the engine.

The vectors are implemented with a hierarchical specialization, the base class vector implements the common operations related with vectors and the specializations defines vectors with two-dimensional, three-dimensional, or four-dimensional quantities of components with its related operations, see UML diagram below.

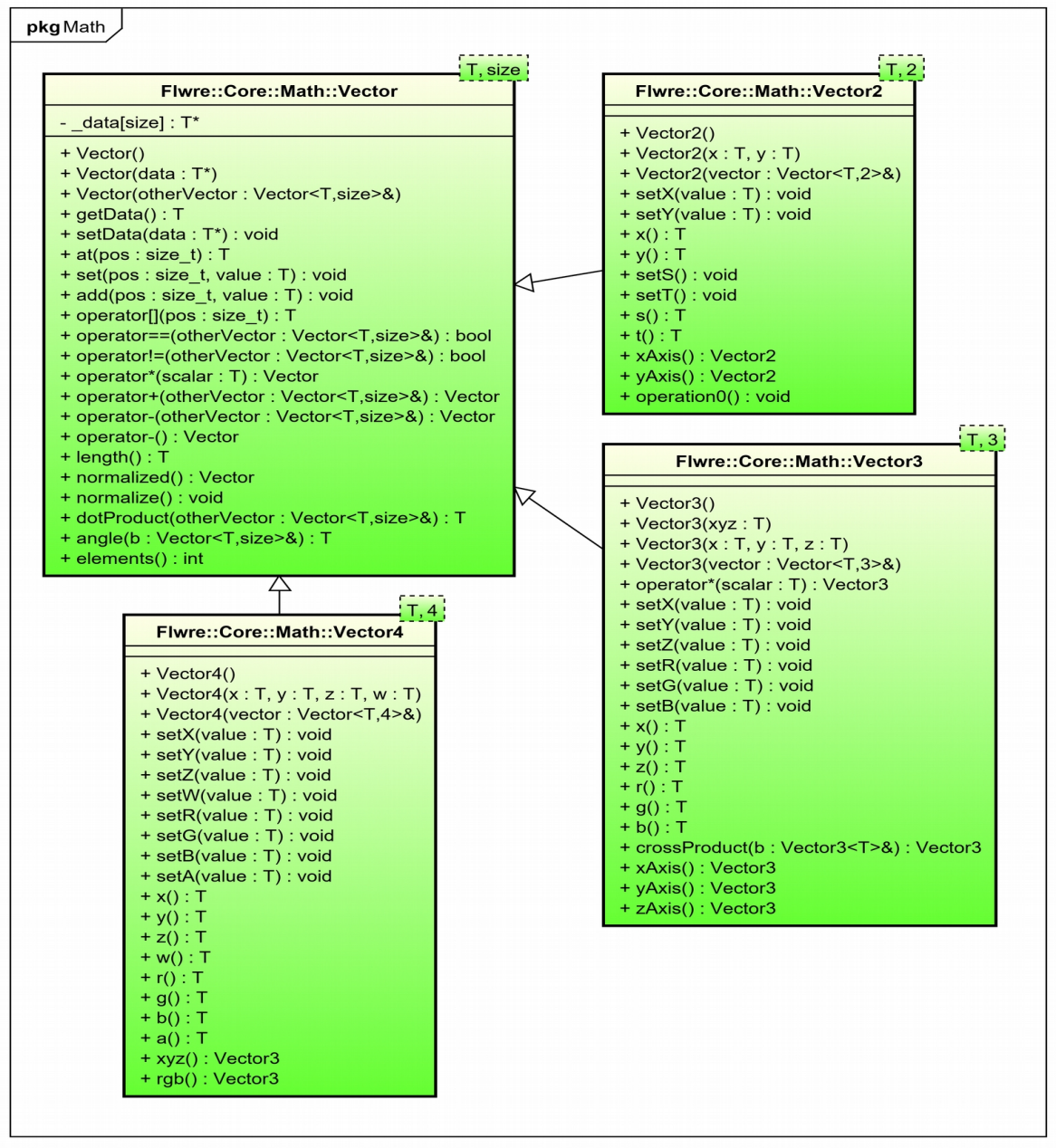


Figure 4.2 depicts the vectors template classes with an UML diagram.

The vector base class implements the following operations:

- Constructors to build an empty vector and copying one vector to another vector passing it as parameter.
- Data access method to retrieve the pointer where starts the array that represents the vector, see **getData** method.
- Setters and getters for a given vector value position, see **at** and **set** methods.
- Addition method for a value in a given position in the vector, see **add** method.
- Basic arithmetic operations: product by a scalar jointly addition and subtraction with another vector. Although in computer graphics the common operation is the product.
- Equality operators between vectors.

- Dot product, this is useful to find out the angle between two vectors and determine whether two vectors are perpendicular.
- Vector normalization.
- Vector magnitude, see length method.
- Method to retrieve the number of elements that contains a given vector, a vector3 has three elements and so on.

The specializations defines access methods wrapped in a conceptual names definition, so the methods like **x()**, **y()**, **z()** and **r()**, **g()**, **b()** or **s()**, **t()**, **setX()**, **setR()**, **setW()** or **w()** which manipulate homogenous coordinates can be found among others in different vector specializations classes. However this is not the better scenario since some this methods are global and could be implemented in the base class, but for the moment stays like this.

Finally in the vector3 specialization class the cross product is implemented and in the vector4 exist methods to convert a vector4 to vector3 see **xyz()** and **rgb()** methods.

4.1.2.3 Matrices

In 3D graphics programming the matrices are very important, since in a 3D graphics engine calculations can be performed in a multitude of different Cartesian coordinates spaces, so moving from one coordinate space to another requires the use of transformation matrices, the transformations are explained in **section 4.1.2.8**, and to begin with here is explained the matrix container to be used within the engine to achieve the transform calculations.

An $n \times m$ matrix M is an array of numbers having n rows an m columns. If $n = m$, then is said that the matrix M is square and to refer to a determinate entry of M is written M_{ij} , where that refers to at the i -th row of the j -th column. As an example, suppose that M is a 4×4 matrix, then could be write:

$$M = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

The entries for which $i = j$ are called the main diagonal entries of the matrix. A square matrix whose only non zero entries appear on the main diagonal is called a diagonal matrix, when there are only ones on the main diagonal and zeros elsewhere of the matrix, then is called identity matrix I_n or elementary matrix, this matrix is common used when we are working with transformations as will seen later.

$$I_n = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}$$

Down below is explained how this scenario has been implemented within the engine. The matrix class template is implemented with the same approach as vector template class, however for the moment matrix class temple is not specialized yet, in the **section 4.1.2.8** will be specialized to implement the common transformations matrices with 4×4 matrices, see UML diagram below.

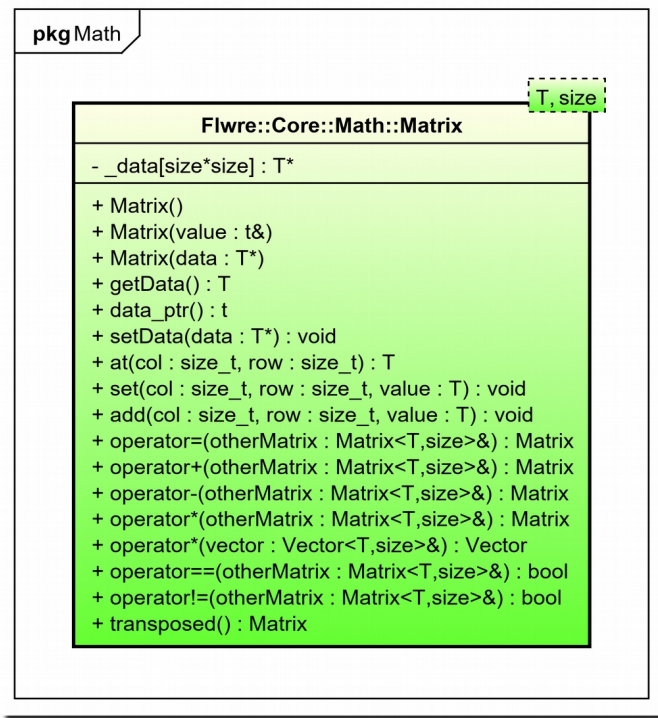


Figure 4.3 depicts the matrix template base class with an UML diagram.

The matrix base class implements the following operations:

- Constructors to build an empty matrix, an identity matrix and copying one matrix to another matrix passing it as parameter.
- Data access methods to retrieve a pointer where starts the array that represents the matrix, see **getData** method.
- Setters and getters for a given matrix value position by column and row, see **at** and **set** methods.
- Addition method for a value in a given position in the matrix getting it by column and row, see **add** method.
- Basic arithmetic operations: product by a vector or another matrix jointly addition and subtraction with another matrix.
- Equality operators between matrices.
- Matrix assignment operator.
- Matrix transposed.

Although has been implemented different arithmetic operations, both in vectors and matrices is important to highlight that the major common operation in graphical applications is the product or multiplication.

Both the vector base class and the matrix base class, overloading the operator << in friend manner, the insertion operator << is used to output streams and in this case is overloaded to display the data contained in a determinate vector or matrix instance, this is used in the unit tests for the data visualization on the standard output via **cout** stream object, defined to access to the standard output.

The key word **friend** defines the operator outside the class scope, for this reason is not shown in UML diagrams but has access all the class members, even if are private or protected, so can get

display the vector data or matrix data. The operator must be overloaded to recognize an ostream on the left and a vector or matrix on the right.

See the code snippet example implemented in a vector template class:

```

/** @brief Print vector on standard output.
 * @param o on the left
 * @param reference to a vector on the right
 * @return the vector send to the standard output
 */
inline friend std::ostream& operator<<(std::ostream& o, const Vector<T, size>& vector)
{
    std::cout.precision(9);
    o << "(";
    for (size_t pos = 0; pos != size; pos++) {
        if ((pos + 1) < size) {
            o << vector.at(pos) << ", ";
        }
        else {
            o << vector.at(pos);
        }
    }
    o << ")" << std::endl;
    return o;
}

```

All methods in mathematical library, has been implemented with **inline** manner, to get more efficiency when used in the engine, for more details see source code implemented.

4.1.2.4 Vectors and matrices unit tests

To verify the correct mathematical library behaviour, has been implemented some unit tests to determine whether the library is fit for use. Basically the unit tests try different kind of calculations with vectors and matrices that will be used later in the engine.

Below is shown the unit test for the vectors.

```

PS D:\Release> .\UnitTest_Vector.exe --log_level=all
Running 1 test case...
Entering test module "boost_test_macro_overview"
D:\nUnitTest_Vector.cpp(33): Entering test case "Unit_Test_Math_Library_FlowRenderEngine"

*****
* Unit Test for Vector class template -- Math library Flow Render Engine -- author: Xavier Figuera - 22/10/2019 - : *
* Vector3 v1 = (1.0f, 2.0f, 3.0f) *
* Vector3 v2 = (2.0f, 2.0f, 3.0f) *
* Vector3 v3 = (1.0f, 2.0f, 3.0f) *
* Vector3 v8 = (4.0f, 6.0f, 8.0f) *
* Vector3 v9 = (2.0f, 4.0f, 6.0f) *
*****
v0 = (8, 6)
v1 = (1, 2, 3)
v1.set(0, 14.0f) -- v1 = (14, 2, 3)
v1.add(1, 2.0f) -- v1 = (14, 4, 3)
v1.at(3) -- v1 = 3
v1[0] = 14
v2.set(0, v1[2]) = (3, 0, 0)
v1 = (14, 4, 3)
v3 = (1, 2, 3)
D:\UnitTest_Vector.cpp(85): error: in "Unit_Test_Math_Library_FlowRenderEngine": check v1 == v3 has failed
D:\UnitTest_Vector.cpp(86): info: check v1 != v3 has passed
D:\UnitTest_Vector.cpp(87): info: check v1 != v2 has passed
v1.set(0, 1.0f) and 1.set(1, 2.0f) -- v1 = (1, 2, 3)
D:\UnitTest_Vector.cpp(92): info: check v1 == v3 has passed

```

```
D:/UnitTest_Vector.cpp(93): error: in "Unit_Test_Math_Library_FlowRenderEngine": check v1 != v3 has failed
v1 = (1, 2, 3)
v4 = v1 * 5.0f -- v4 = (5, 10, 15)
v5 = v4 / 5.0f -- v5 = (1, 2, 3)
v6 = v5 + v5 -- v6 = (2, 4, 6)
v7 = v5 - v1 -- v7 = (0, 0, 0)
v4.length() = 18.7083
D:/UnitTest_Vector.cpp(110): error: in "Unit_Test_Math_Library_FlowRenderEngine": check v4.length() == sqrtf(pow(5.0f, 2) + pow(10
.0f, 2) + pow(15.0f, 2)) has failed
-v4 = (-5, -10, -15)
Normalized vector *****
v8 = (4, 6, 8)
v8.normalized() = (0.371390671, 0.557085991, 0.742781341)
v8 = (4, 6, 8)
v8.normalized().length() = 1
D:/UnitTest_Vector.cpp(122): error: in "Unit_Test_Math_Library_FlowRenderEngine": check v8.normalized().length() == 1.0f has fail
ed
v8.normalize() = (0.371390671, 0.557085991, 0.742781341)
End normalized vector *****
Dot product *****
v1 = (1, 2, 3)
v9 = (2, 4, 6)
v1.dotProduct(v9) = 28
D:/UnitTest_Vector.cpp(136): info: check v1.dotProduct(v9) == 28.0f has passed
D:/UnitTest/UnitTest_Vector.cpp(137): error: in "Unit_Test_Math_Library_FlowRenderEngine": check v1.dotProduct(v9) == 28.5f has
failed
Dot product precision
D:/UnitTest_Vector.cpp(139): error: in "Unit_Test_Math_Library_FlowRenderEngine": difference{0.00357144} between v1.dotProduct(v9)
{28} and 28.1f{28.1000004} exceeds 1e-009%
D:/UnitTest_Vector.cpp(140): error: in "Unit_Test_Math_Library_FlowRenderEngine": difference{3.56947e-005} between v1.dotProduct(v
9){28} and 28.001f{28.0009995} exceeds 1e-009%
D:/UnitTest_Vector.cpp(141): info: difference{} between v1.dotProduct(v9){28} and 28.000000001f{28} doesn't exceed 1e-009%
D:/UnitTest_Vector.cpp(142): info: difference{} between v1.dotProduct(v9){28} and 28.000000001f{28} doesn't exceed 1e-009%
D:/UnitTest_Vector.cpp(143): error: in "Unit_Test_Math_Library_FlowRenderEngine": difference{0.00357144} between v1.dotProduct(v9)
{28} and 28.1f{28.1000004} exceeds 1.19209e-007%
D:/UnitTest_Vector.cpp(144): info: difference{} between v1.dotProduct(v9){28} and 28.00000001f{28} doesn't exceed 1.19209e-007%
D:/UnitTest_Vector.cpp(145): error: in "Unit_Test_Math_Library_FlowRenderEngine": difference{6.81196e-008} between v1.dotProduct(v
9){28} and 28.000001f{28.0000019} exceeds 1.19209e-007%
D:/UnitTest_Vector.cpp(146): error: in "Unit_Test_Math_Library_FlowRenderEngine": difference{3.40598e-007} between v1.dotProduct(v
9){28} and 28.00001f{28.0000095} exceeds 1.19209e-007%
D:/UnitTest_Vector.cpp(147): error: in "Unit_Test_Math_Library_FlowRenderEngine": difference{3.54222e-006} between v1.dotProduct(v
9){28} and 28.0001f{28.0000992} exceeds 1.19209e-007%
Vector x = (1, 3, -5)
Vector y = (4, -2, -1)
x.dotProduct(y) = 3
D:/UnitTest_Vector.cpp(160): info: check x.dotProduct(y) == 3.0f has passed
Angle in radians between x and y vectors is 1.45991 radians
Angle in degrees between x and y vectors is 83.6468 degrees
End dot product *****
v1s.xAxis() = (1, 0, 0)
v1s.yAxis() = (0, 1, 0)
v1s.zAxis() = (0, 0, 1)
v1s.zAxis() = (0, 0, 0)
Angle between A = (1, 0, 0) and B = (0, 1, 0) are 90 degrees
D:/UnitTest_Vector.cpp(176): error: in "Unit_Test_Math_Library_FlowRenderEngine": check Flwre::Core::Math::RadiansIntoDegrees(v1s.
xAxis().angle(v1s.yAxis())) == 90.0f has failed
Angle between A = (1, 0, 0) and B = (0, 0, 1) are 90 degrees
D:/UnitTest_Vector.cpp(178): error: in "Unit_Test_Math_Library_FlowRenderEngine": check Flwre::Core::Math::RadiansIntoDegrees(v1s.
xAxis().angle(v1s.zAxis())) == 90.0f has failed
Angle between A = (1, 0, 0) and B = (0, 0, 1) are 1.5708 radians
D:/UnitTest_Vector.cpp(180): error: in "Unit_Test_Math_Library_FlowRenderEngine": check v1s.xAxis().angle(v1s.zAxis()) == 1.5708f
has failed
D:/UnitTest_Vector.cpp(181): error: in "Unit_Test_Math_Library_FlowRenderEngine": check v1s.xAxis().angle(v1s.zAxis()) == v1s.xAxi
s().angle(v1s.yAxis()) has failed
Cross product *****
a = (1, 2, 3)
b = (-7, -8, -6)
Cross product a=(1, 2, 3) b=(-7, -8, -6) c = a x b (12, -15, 6)
c = (12, -15, 6)
D:/UnitTest_Vector.cpp(193): info: check a.crossProduct(b) == c has passed
Cross product a=(3, -3, 1) b=(4, 9, 2) c = a x b (-15, -2, 39)
Orthogonal to the vectors 'a' and 'b' verification:
dot product a.c = 0
dot product b.c = 0
End Cross product *****
v1s4c = (0, 0, 0, 0)
v1s4c = (4, 4, 4, 0)
v2s4c = (4, 4, 35.5, 2)
D:/UnitTest_Vector.cpp(227): info: check v10 == v11 has passed
v10 = (4, 4, 35.5)
```

```
v11 = (4, 4, 35.5)
rgb() = (4, 4, 35.5)
v3s4 = (4, 4, 35.5, 2)
elements in v3s4 = 4
D:\TFM\src\Renderers\Flowrenderenginepac2\Core\Math\UnitTest\UnitTest_Vector.cpp(33): Leaving test case
"Unit_Test_Math_Library_FlowRenderEngine"; testing time: 245ms
Leaving test module "boost_test_macro_overview"; testing time: 252ms
```

Below is shown the unit test for the matrices:

```
PS D:\Release> .\UnitTest_Matrix.exe --log_level=all
Running 1 test case...
Entering test module "boost_test_macro_overview"
D:\UnitTest_Matrix.cpp(29): Entering test case "Unit_Test_Math_Library_FlowRenderEngine"

**** Matrix 2x2 product ****
A =
[1 2
 3 4]
B =
[5 6
 7 8]
A * B =
[19 22
 43 50]
A * B =
[19 22
 43 50]
D:\UnitTest_Matrix.cpp(100): info: check (A * B) == E has passed
D:\UnitTest_Matrix.cpp(101): error: in "Unit_Test_Math_Library_FlowRenderEngine": check (A * B) != E has failed
E =
[19 22
 43 50]
**** Matrix 2x2 transposed ****
(E)t =
[19 43
 22 50]
D:\UnitTest_Matrix.cpp(107): info: check E.transposed() == E.transposed() has passed

**** Matrix 3x3 product ****
D:\UnitTest_Matrix.cpp(161): info: check (F * G) == FxG has passed
F * G =
[46 52 61
 109 124 145
 172 196 229]
D:\UnitTest_Matrix.cpp(164): info: check (F * F) == (F * F) has passed
F * F =
[30 36 42
 66 81 96
 102 126 150]
**** Test with identity matrices ****
matrix2x2 identity created:
[1 0
 0 1]
_identityExpected2x2:
[1 0
 0 1]
D:\UnitTest_Matrix.cpp(185): info: check matrix2x2 == _identityExpected2x2 has passed
_identityFake2x2:
[1 0
 2 1]
D:\UnitTest_Matrix.cpp(187): error: in "Unit_Test_Math_Library_FlowRenderEngine": check matrix2x2 == _identityFake2x2 has failed
D:\UnitTest_Matrix.cpp(188): info: check matrix2x2 != _identityFake2x2 has passed
matrix3x3 identity created:
[1 0 0
 0 1 0
 0 0 1]
_identityExpected3x3:
[1 0 0
 0 1 0
 0 0 1]
D:\UnitTest_Matrix.cpp(209): info: check matrix3x3 == _identityExpected3x3 has passed
_identityFake3x3:
[1 0 1
 0 1 0
 2 0 1]
D:\UnitTest_Matrix.cpp(211): error: in "Unit_Test_Math_Library_FlowRenderEngine": check matrix3x3 == _identityFake3x3 has failed
D:\UnitTest_Matrix.cpp(212): info: check matrix3x3 != _identityFake3x3 has passed
matrix4x4 identity created:
[1 0 0 0
 0 1 0 0
 0 0 1 0
 0 0 0 1]
```

```

_identityExpected4x4:
[1 0 0 0
 0 1 0 0
 0 0 1 0
 0 0 0 1]

D:/UnitTest_Matrix.cpp(237): info: check matrix4x4 == _identityExpected4x4 has passed
_identityFake4x4:
[1 0 1 0
 0 1 0 0
 4 0 1 0
 2 0 0 1]

D:/UnitTest_Matrix.cpp(239): error: in "Unit_Test_Math_Library_FlowRenderEngine": check matrix4x4 == _identityFake4x4 has failed
D:/UnitTest_Matrix.cpp(240): info: check matrix4x4 != _identityFake4x4 has passed
D:/UnitTest_Matrix.cpp(29): Leaving test case "Unit_Test_Math_Library_FlowRenderEngine"; testing time: 307ms
Leaving test module "boost_test_macro_overview"; testing time: 312ms

```

4.1.2.7 2D and 3D geometric primitives

The geometric primitives implemented within the engine, are on the one hand 2D primitives and its meshes can be generated with procedural manner, this are defined in the methods declared in **PrimitiveMeshShapes** class and the objects 2D defined are triangles, circles, ellipses, pentagons, hexagons, octagons, quads among others, at the same time this class defines methods to generate some surfaces of revolution in a three-dimensional space procedurally as well, the surfaces of revolution are Torus and Sphere jointly with a cube.

Nevertheless, the explanation how the 2D objects and the surfaces of revolution are implemented jointly with the mathematical concepts related are outside the scope of this project, see the code implementation for more information and see [\[Torus\]](#) [\[Sphere\]](#).

The following **figure 4.4** shows the **PrimitiveMeshShapes** class and its class hierarchy with a UML diagram.

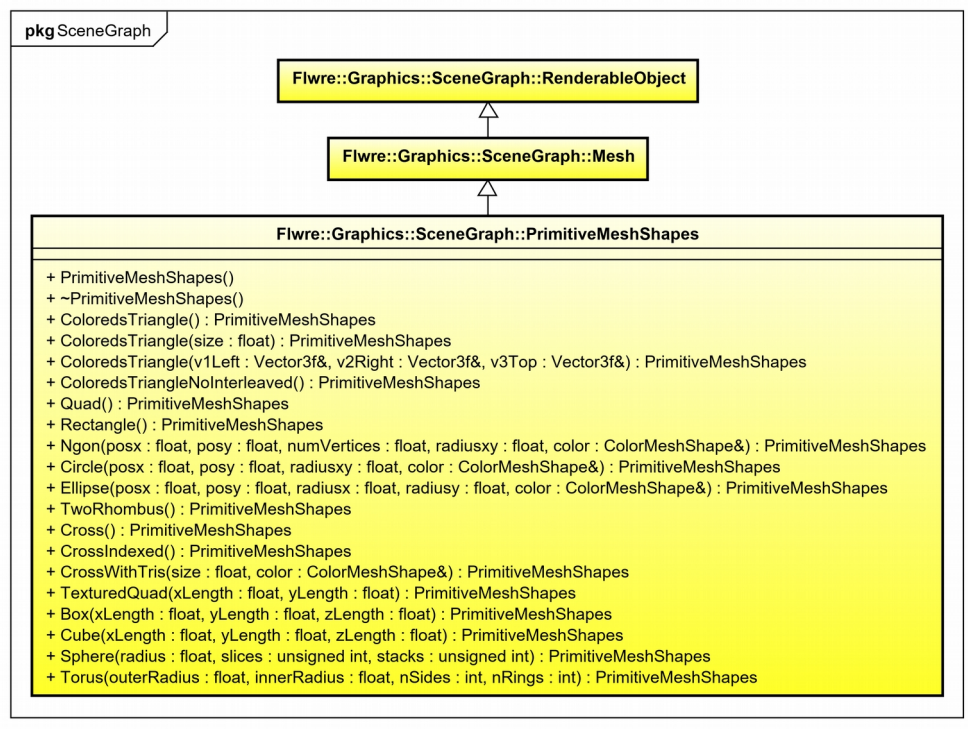


Figure 4.4 depicts the **PrimitiveMeshShapes** class and its class hierarchy with an UML diagram.

4.1.2.8 Geometric transformations

Linear algebra is the basic notation for transforms and can be used to express many of the operations required to arrange objects in 3D scene, allowing for instance to mount a camera for handling 3D viewing and get the objects onto the screen. Geometric transformations like the rotation, translation, scaling, and projection, can be accomplished with matrix multiplication and the transformation matrices used to do this, are the subject of this section.

This notation actually simplifies the mathematical descriptions and manipulations of linear models and its possible to solve systems of linear equations such as this:

$$\begin{aligned}x' &= U_1x + V_1y + W_1z + T_1 \\y' &= U_2x + V_2y + W_2z + T_2 \\z' &= U_3x + V_3y + W_3z + T_3\end{aligned}\quad (4.1)$$

A 3D point $P(x, y, z)$ can be transformed into $P'(x', y', z')$ using the definition 4.1. But nevertheless, the linear transformations are also possible to express them as matrices, which provide certain advantages for viewing the transform and for interfacing to various types of computer graphics hardware, hence the definition 4.1 can be written in matrix form as follows.

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} U_1 & V_1 & W_1 \\ U_2 & V_2 & W_2 \\ U_3 & V_3 & W_3 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} T_1 \\ T_2 \\ T_3 \end{bmatrix}\quad (4.2)$$

Nevertheless, the transformations inside the engine are represented with 4×4 square matrices, since they manipulate homogeneous coordinates, this coordinates introduce an extra coordinate w , which is usually set to a value of 1.0 basically. Homogeneous coordinates provides a compact and elegant way to represent the transformations within a single mathematical entity and are extension of Cartesian space adding an additional dimension to explain the projective space, so Cartesian space is just one of many planes in the projective space, hence actually Euclidean geometry is a subset of projective geometry, see [\[math3DCG\]\[songho\]](#) for more information, this concept will be expanded in the **section 4.1.2.8.2.3** view matrix(lookAt). To extend the matrix of the definition 4.2 to four dimensions and setting its fourth coordinate, which is w coordinate as been explained above, is needed construct a 4×4 transformation matrix M corresponding to the 3×3 matrix and the 3D translation T shown in 4.2 definition as follows.

$$M = \left[\begin{array}{ccc|c} U_1 & V_1 & W_1 & T_1 \\ U_2 & V_2 & W_2 & T_2 \\ U_3 & V_3 & W_3 & T_3 \\ \hline 0 & 0 & 0 & 1 \end{array} \right] = \begin{bmatrix} U_1 & V_1 & W_1 & T_1 \\ U_2 & V_2 & W_2 & T_2 \\ U_3 & V_3 & W_3 & T_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}\quad (4.3)$$

This is only an example to explain how the transformations are represented and how are manipulated inside the engine, translations and others transformations will be explained in the following sub-sections more deeply.

The definition 4.3 stores 16 numbers arranged into 4 rows and 4 columns where generally this numbers are stored as floats, is important to highlight that any graphics hardware is heavily tailor made towards performing operations on 4 components vectors, hence making them also ideal for computing matrices in this case the columns are the vectors U , V and W . that they represents a determinate coordinate system.

So within the engine has defined a new class template, that encapsulates the transformations, this class is called Matrix4 and is specialized from matrix template base class and implements the methods that generates the common transformations matrices such as translations, rotations, scaling and projection matrices, all this is used within the engine to accomplish different targets as will explained below, see [figure 4.5](#) Matrix4 class template with an UML diagram.

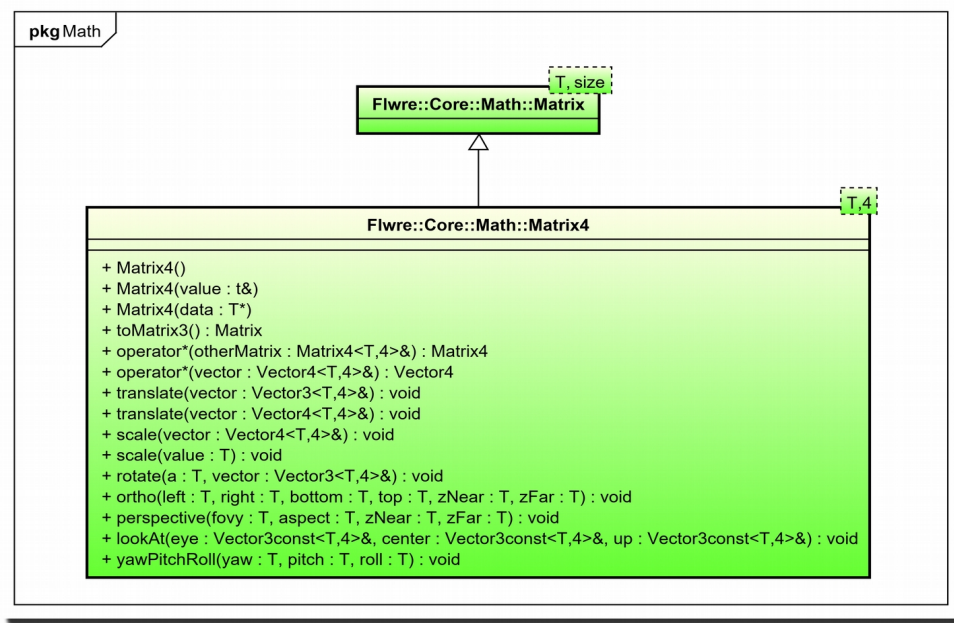


Figure 4.5 depicts the matrix4 class template with an UML diagram.

4.1.2.8.1 Model Matrix

The model matrix is used to convert from local space to world space, the coordinate values stored in the vertex buffers see [section 4.2.1.4.1](#) vertex buffers, this data stored in the graphic card buffers, defines a determinate mesh represented in local space coordinates. The world space are the global coordinate system that determines where objects are in relation to each other in the scene.

All vertices of a determinate mesh will be transformed by the same model matrix by the vertex shader, to achieve this the model matrix can contain any combination of translation, rotation and scale, then multiplying together this transform matrices is possible to build a model matrix to translate, rotate and scale the objects to wherever is wanted into world space, see [section 4.2.7.1](#) transform class and **getLocalTransform** method in the renderable object, see [section 4.2.4.1](#) renderable object and meshes. In short, the model matrix helps to push the objects into the world space.

In the following sub-sections are explained the geometric transformations mentioned above, such as the translation, rotation and scale, in conjunction with the viewing transformations involved in the camera location and the projection of the objects onto display screen.

4.1.2.8.1.1 Translation

The definition 4.1 constitutes a linear transformation from the coordinate system C to a second coordinate system C' , where the coordinates in this system are $\langle x', y', z' \rangle$ and can be expressed as linear functions of coordinates $\langle x, y, z \rangle$ in C , the definition 4.2 written this in matrix form.

The translation is used to move or translate an object by shifting all its points the same amount. The transform form has been shown in definition 4.2, however this definition in homogeneous coordinates just like is managed within the engine can be shown as follows.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (4.4)$$

So, for example with the definition 4.4 could be possible translate the point $P(x, y, z)$ of a mesh in local space to a world space position defined by the translation (Tx, Ty, Tz) to get the new point $P'(x', y', z')$, this represented in a 3D space coordinate system, could be seen as follows in the [figure 4.6](#). The translation are implemented within the method translate in matrix4 class, see [figure 4.5](#). The following figure shows the translation in conceptual manner.

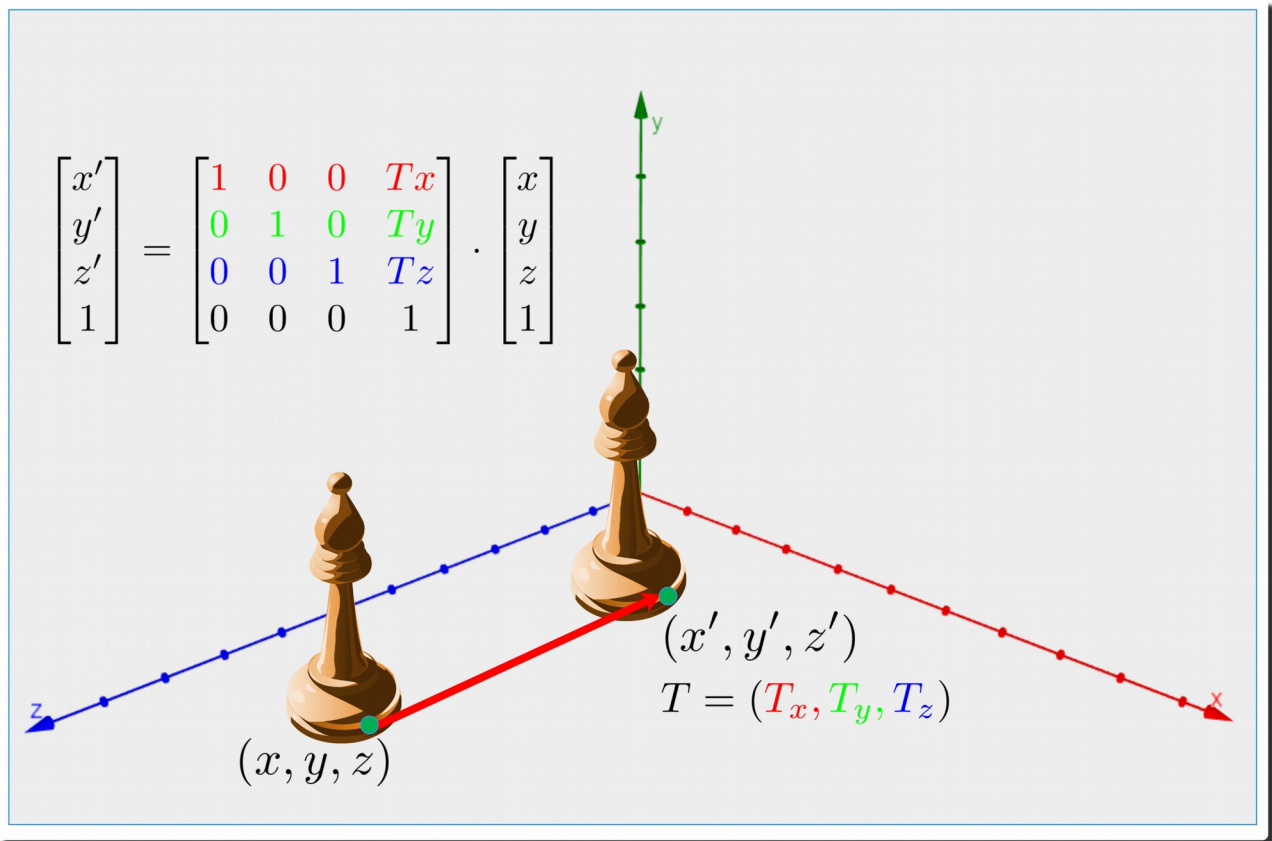


Figure 4.6 depicts an object translation in conceptual manner.

4.1.2.8.1.2 Scaling

To scale a vector P by a factor of S is needed simply calculate $P' = SP$, then with three dimensions, this operation can be expressed as the matrix product such as follows.

$$\begin{bmatrix} S & 0 & 0 \\ 0 & S & 0 \\ 0 & 0 & S \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} \quad (4.5)$$

This is called a uniform scale. But is also possible to scale a vector by different amounts along the x , y and z -axes, here the matrix defined in 4.5 definition, changes since whose diagonal entries are not necessarily all equal, this is called non uniform scale and the matrix product can be expressed as follows.

$$\begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} \quad (4.6)$$

The engine implements the uniform and non uniform scale and the matrix definition in homogeneous coordinates just like is managed within the engine can be shown as follows.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (4.7)$$

Where the variables (S_x, S_y, S_z) are scale factors to define scaling matrix using a three component vector (x, y, z) , see methods scale with different signatures in [figure 4.5](#).

The following [figure 4.7](#) shows scaling in conceptual manner. (the object non translate only scale, hence does not move from its original position when is scaled, although in the image below seems to be translated but is only a conceptual manner to depicts it.)

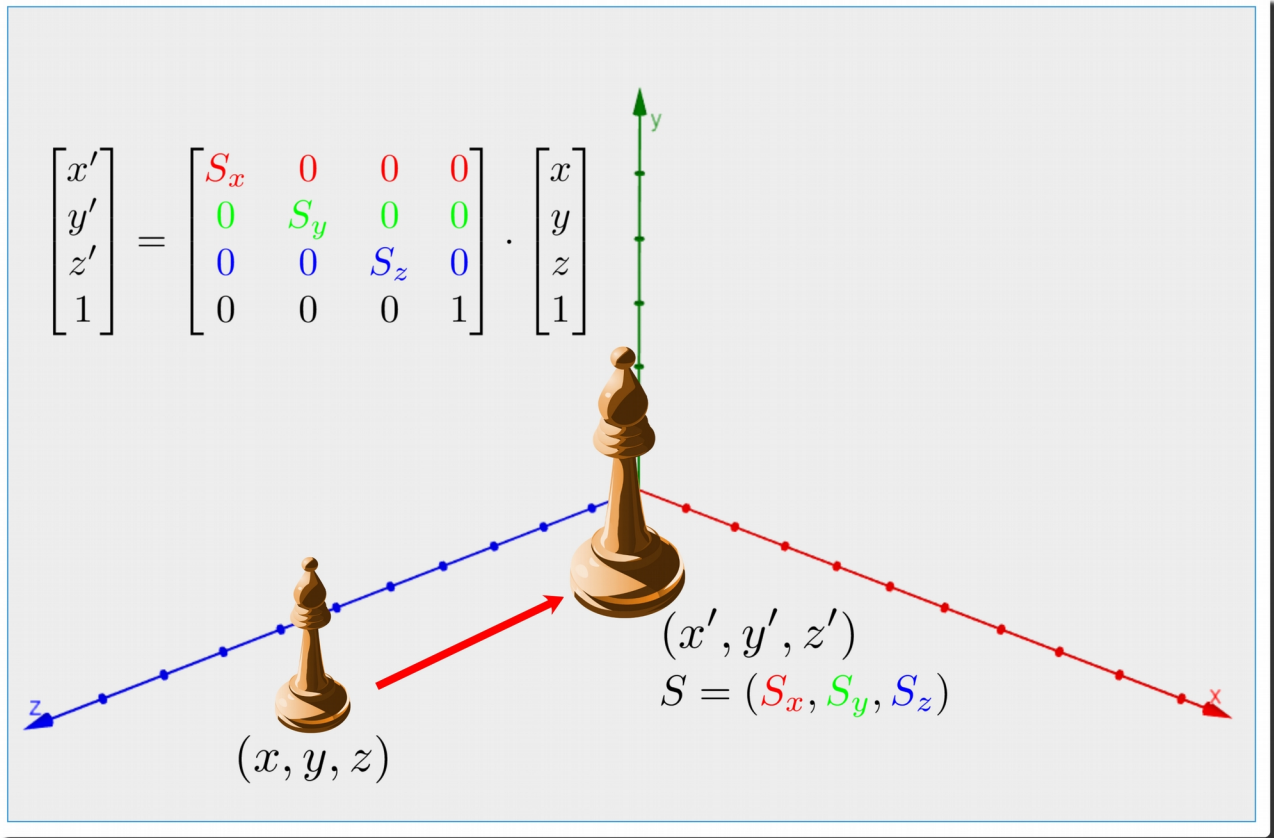


Figure 4.7 depicts how an object is scaled in conceptual manner.

4.1.2.8.1.3 Rotation

The translations and scaling are relatively easy to understand, however rotations are a bit more complicated. In 3D space an object is rotated about an axis, whether it be the x , y or z -axis, or some arbitrary axis, this rotations are called Euler rotations by the Swiss mathematician Leonhard Euler (1707–1783).

To rotate a vertex about z -axis the following matrix $R_z(\theta)$ that perform this rotation through the angle θ over z -axis can be written as follows.

$$R_z(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \cos(\theta)x - \sin(\theta)y \\ \sin(\theta)x + \cos(\theta)y \\ z \end{bmatrix} \quad (4.8)$$

and similarly the matrix $R_x(\theta)$ and $R_y(\theta)$ that perform rotations thought an angle θ about the x and y -axes respectively and can be written as follows.

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x \\ \cos(\theta)y - \sin(\theta)z \\ \sin(\theta)y + \cos(\theta)z \end{bmatrix} \quad (4.9)$$

$$R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \cos(\theta)x + \sin(\theta)z \\ y \\ -\sin(\theta)x + \cos(\theta)z \end{bmatrix} \quad (4.10)$$

Then the matrix $R_z(\theta)$ can be visualized as rotating a point $P(x, y, z)$ on a plane parallel with the xy -plane as depicts the **figure 4.8**.

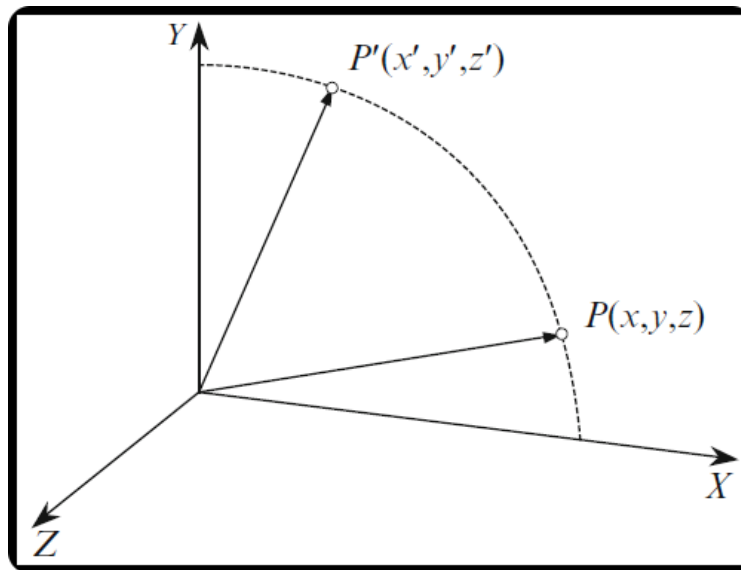


Figure 4.8 depicts rotating the point P over the z -axis to get the new point $P'(x', y', z')$

Then the definition 4.8, 4.9 and 4.10 with homogenous coordinates can be written as follows.

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ \cos(\theta) \cdot y - \sin(\theta) \cdot z \\ \sin(\theta) \cdot y + \cos(\theta) \cdot z \\ 1 \end{bmatrix} \quad (4.11)$$

$$R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) \cdot x + \sin(\theta) \cdot z \\ y \\ -\sin(\theta) \cdot x + \cos(\theta) \cdot z \\ 1 \end{bmatrix} \quad (4.12)$$

$$R_z(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) \cdot x - \sin(\theta) \cdot y \\ \sin(\theta) \cdot x + \cos(\theta) \cdot y \\ z \\ 1 \end{bmatrix} \quad (4.13)$$

Then is possible combination the matrices shown above to rotate around x, y and z -axes, first rotating about x then y and finally z -axis, however this scenario introduces a problem called Gimbal lock, this is a major weaknesses of Euler angles, basically Gimbal lock occurs if during rotation one of the three rotation axes is by accident aligned with another, thereby reducing by one the number of available degrees of freedom. For more information about it, see [\[fundaComGra\]](#) [\[MathComGra\]](#).

The definitive solution to prevent Gimbal locks is implement the rotations using Quaternions [\[MathComGra1\]](#), however this will not implemented in this project. Here an intermediate solution is implemented that does not completely prevent Gimbal locks although it gets a lot harder that occurs this scenario.

The intermediate solution is rotate around an arbitrary unit axis (R_x, R_y, R_z) called unit vector, then instead of combining the rotation matrices shown in definition 4.11, 4.12 and 4.13, is used a single matrix combining all matrices instead. The matrix that combine all rotation matrices can be written in the following manner.

$$\begin{bmatrix} \cos(\theta) + R_x^2(1 - \cos(\theta)) & R_x R_y(1 - \cos(\theta)) - R_z \sin(\theta) & R_x R_z(1 - \cos(\theta)) + R_y \sin(\theta) & 0 \\ R_y R_x(1 - \cos(\theta)) + R_z \sin(\theta) & \cos(\theta) + R_y^2(1 - \cos(\theta)) & R_y R_z(1 - \cos(\theta)) - R_x \sin(\theta) & 0 \\ R_z R_x(1 - \cos(\theta)) - R_y \sin(\theta) & R_z R_y(1 - \cos(\theta)) + R_x \sin(\theta) & \cos(\theta) + R_z^2(1 - \cos(\theta)) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This matrix is implemented within the engine to rotate the objects. See method rotate in [figure 4.5](#) matrix4 class template with an UML diagram depicts above. The following [figure 4.9](#) shows the rotation in conceptual manner.

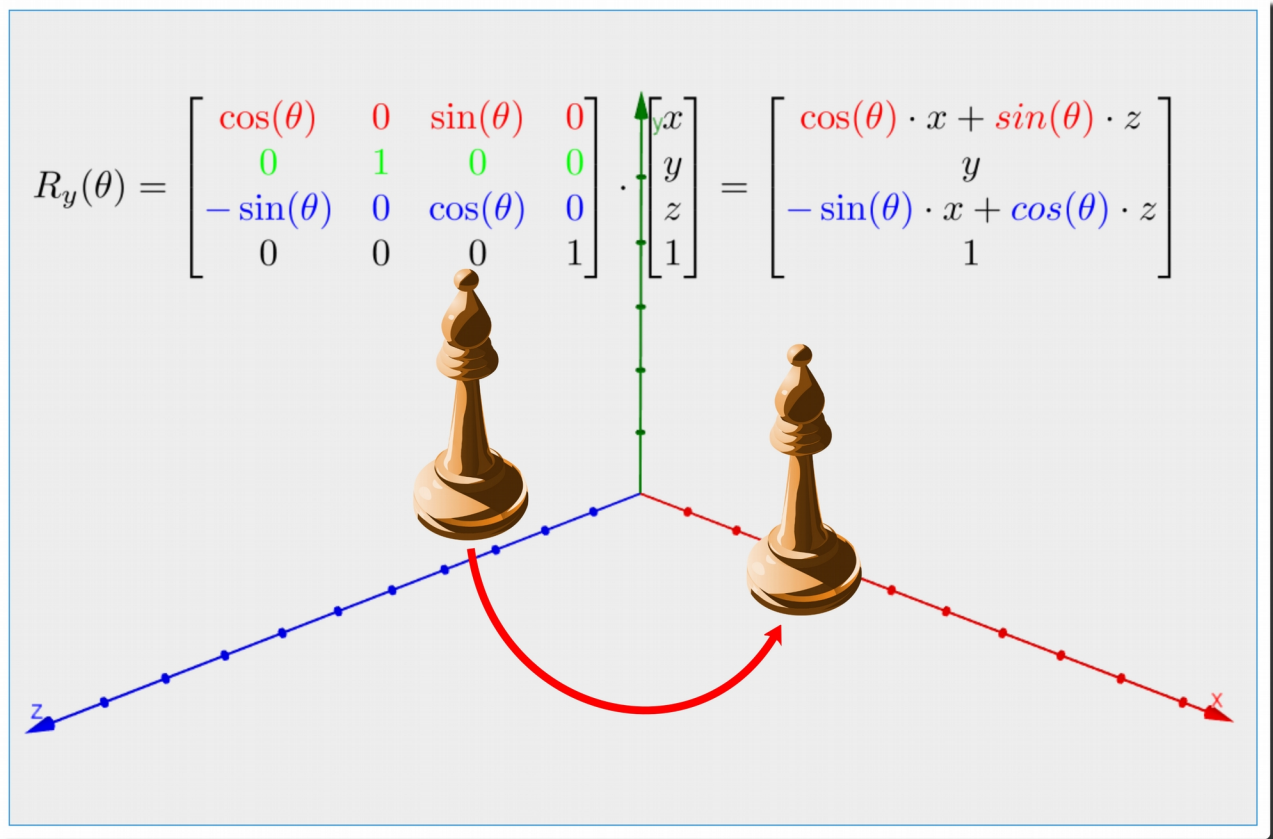


Figure 4.9 depicts a rotation object about y-axis in conceptual manner.

4.1.2.8.2 Viewing Transformations

In the previous **section 4.1.2.8** geometric transformations, and concretely in the **section 4.1.2.8.1** model matrix and their sub-sections, has been discussed the geometric transforms as a tool to organize geometric objects in a 3D scene, translating, rotating and scaling the objects. In a real-time rendering engine, all this is visualized in a 2D view in screen space from a 3D world, to move the objects between their 3D locations to their positions to a 2D view are used geometric transformations as well, but in this case, these 3D mapping to 2D space are called viewing transformations, and plays an important role in the object-order rendering, in which is need to rapidly find the image-space location in 2D of each object in the 3D scene.

This viewing transformations express a determinate projection type like orthographic or perspective projection, they projecting the 3D points in the scene in world space to a 2D points in the image space, the 3D points are represented as (x, y, z) coordinates in the canonical coordinate system and the points in the screen space are expressed in units of pixels, all of this it depends of different factors and this include the camera position and camera orientation, the type of projection jointly the field of view and finally the resolution of the image set on the display screen.

By canonical term, it should be understood as NDC coordinates (normalized coordinate system) which is what the graphic API expects to receive in the final stage in the sequence of transformations before to be mapped to the screen space in device coordinates as pixels, see **figure 4.10** below.

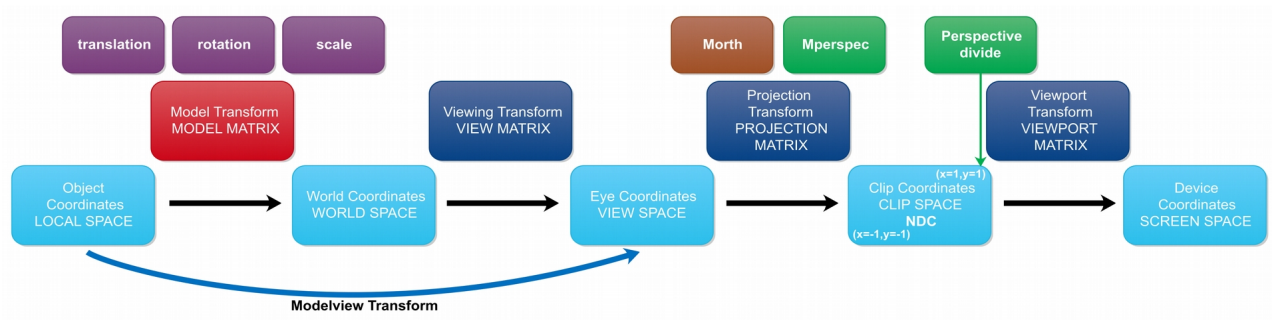


Figure 4.10 depicts the sequence of transformations explained in the section 4.1.2.8 and sub-sections.

The **figure 4.10** depicts the complete sequence of transformations needed to organize and project the objects from local space to the screen space, in the following sections will be explained the viewing transformations with separately manner, as the product of several simpler transformations, the viewing transformations are marked in blue navy, the explanation is made in backwards manner, namely from the viewport transform on the right to the viewing transform on the left, though are applied in the direction as shown in the **figure 4.10**. It is worth pointing out, that not all transformations are implemented within the engine, the transformations marked in red and navy blue are implemented within the engine in the `Matrix4` class within the mathematical library, excepting the viewport transform and the perspective divide that are not implemented within the engine, since are applied by the graphic API within the render pipeline, see [\[pipelineOGL3\]](#) [\[persDivide\]](#) for more information. In the engine, the transformations implemented on it, are passed to the vertex shader within the `updateUniformsConstants` method, see **section 4.2.6.1.1** shader parameter data.

4.1.2.8.2.1 Viewport transform

This transformation is not implemented within the engine but is explained to achieve better compression of the projection process because it has an important role. This is implemented within the render pipeline within the vertex post-processing stage, see [\[pipelineOGL3\]](#) section pipeline.

In fact this transformation is the simplest of all the explained in the following sections, at the same time will be reused for any transformation applied previously before of this in any viewing condition.

This transformation is the last applied in the sequence of product operations of the viewing transformations that explained here, so after this transformation the output is treated in other stages within the render pipeline to finally to be drawn onto display screen.

To explain this, we imagine the geometry projected in a clip coordinates (clip space) with NDC coordinates, containing the visible data where are contained all 3D points whose Cartesian coordinates are normalized and are between -1 and $+1$, that is $(x, y, z) \in [-1, 1]$ then the projection are made onto screen where there is $x = -1$ to the left side of the screen, $x = +1$ to the right side of the screen, $y = -1$ to the bottom of the screen, and $y = +1$ to the top of the screen see **figure 4.10** clip space.

The matrix to transform from the 3D points in NDC coordinates to the device coordinates x and y can be seen like this:

$$\begin{bmatrix} x_{screen} \\ y_{screen} \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{n_x}{2} & 0 & \frac{n_x-1}{2} \\ 0 & \frac{n_y}{2} & \frac{n_y-1}{2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{ndc} \\ y_{ndc} \\ 1 \end{bmatrix} \quad (4.14)$$

The definition 4.14 ignores the z -coordinate of the points in the NDC clip space due to a point of distance along the projection direction does not affect where that point projects in the image. Nevertheless, the viewport matrix in homogeneous coordinates has to contemplate the z -coordinate since it can be used to make closer surfaces hide more distant surfaces, see **section 4.2.5.1.2** depth test state (z-buffer algorithms). The following definition shows the viewport transform matrix M_{vp} .

$$M_{vp} = \begin{bmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x-1}{2} \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y-1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4.15)$$

4.1.2.8.2.2 Projection matrix

The projection matrix takes the world coordinates from the eye coordinates and mapping to the view plane or clip space, this space has been explained in the above section and stretches from -1 to 1 on each axis, then any object past those values will either be culled, if the entire object is outside of this range, or clipped if part of the object is still inside the values. This matrix is part of the projection process to do the projection of the 3D world defined via transformation matrices (model matrix see **section 4.1.2.8.1**) and vertex coordinates, onto the view plane or clip space, to finally be drawn onto the display screen as explained in the section above viewport transformation.

The projection matrix gives a sense of perspective to the scene depending on how the projection matrix values are calculated, basically this can be grouped into two basic types of calculation explained in the two following subsections, this is an orthographic projection and perspective projection, see the boxes **Mortho** in brown and **Mperspec** in green in the **figure 4.10** above.

This calculation flavour is implemented within the engine mathematical library, see **ortho** and **perspective** methods, both are projection matrices, see **figure 4.5** above.

4.1.2.8.2.2.1 Orthographic Projection

In orthographic projection the lines are parallel and perpendicular to the image plane, then the resulting views are called orthographic since when using this perspective type each of the vertex coordinates are directly mapped to clip space, without any perspective division since always $w = 1$ and so the perspective has no effect, this causes that objects farther away do not seem smaller.

This projection is often used for mechanical and architectural drawings because they keep parallel lines parallel and preserve the size and shape, hence not have vertices distorted by perspective being more useful for mechanical and architectural drawings.

This perspective creates a cuboid parallel view volume area around the origin using the following values to determine the maximum viewing.

$x = left \equiv$ left plane,
 $x = right \equiv$ right plane,
 $y = bottom \equiv$ bottom plane,
 $y = top \equiv$ top plane,

$z = z_{near} \equiv$ near plane,
 $z = z_{far} \equiv$ far plane.

then the orthographic matrix implemented within the engine is defined as follows.

$$M_{orth} = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{-2}{z_{far}-z_{near}} & -\frac{z_{far}+z_{near}}{z_{far}-z_{near}} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4.16)$$

See **ortho** method in [figure 4.5](#) above. It is worth pointing out, that this matrix depicts a right handed coordinate system used in the engine, see [\[coordSys\]](#), for more information about it. In a left-handed system, the matrix would be the same except column 2 row 2 that would be written in positive the numerator.

The matrices combination M_{vp} (4.15) and M_{orth} (4.16) do the projection to the screen coordinates x and y , (screen space) initially the z -coordinate point is ignored unless it is used z -buffering algorithms.

$$\begin{bmatrix} x_{pixel} \\ y_{pixel} \\ z_{ndc} \\ 1 \end{bmatrix} = (M_{vp}M_{orth}) \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (4.17)$$

The [figure 4.11](#) shows the orthographic view volume:

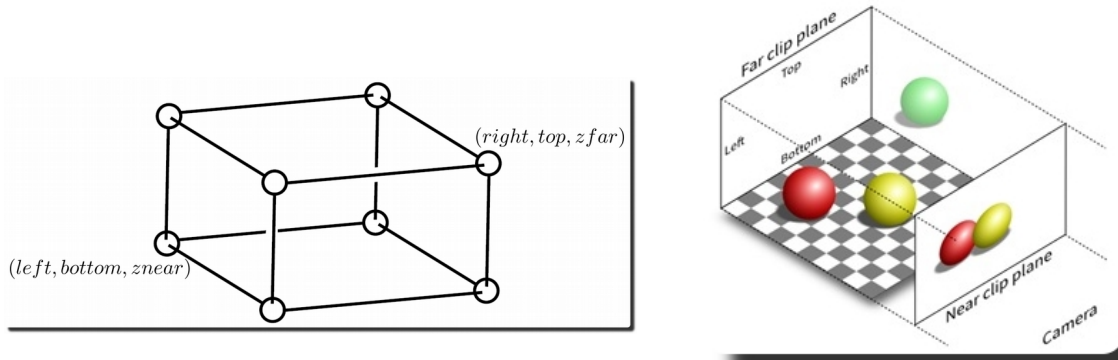


Figure 4.11 depicts the cuboid view volume in an orthographic projection

4.1.2.8.2.2 Perspective Projection

Unlike the orthographic projection, the perspective projection lines are not parallel nor perpendicular to the image plane adding perspective forshortening, then makes the objects get bigger and smaller as they get closer and farther away from the viewpoint. This scenario makes that parallel lines extending into the distance and appear to converge at their vanishing point.

This type of perspective is common used in 3D video games, though also used in other applications that they want to simulate a projection close to reality.

The perspective projection matrix implemented within the engine is as follows.

$$M_{perspec} = \begin{bmatrix} \frac{1}{aspect \cdot \tan(\frac{fovy}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{fovy}{2})} & 0 & 0 \\ 0 & 0 & -\frac{z_{far} + z_{near}}{z_{far} - z_{near}} & -\frac{2 \cdot z_{far} \cdot z_{near}}{z_{far} - z_{near}} \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad (4.18)$$

See **perspective** method in [figure 4.5](#) above. The aspect is the screen width / screen height, fovy is the vertical field of view, the angle of vision should be indicates in degrees, so the larger this value more objects are visible.

The perspective-view or view frustum in this case is a pyramid as follows:

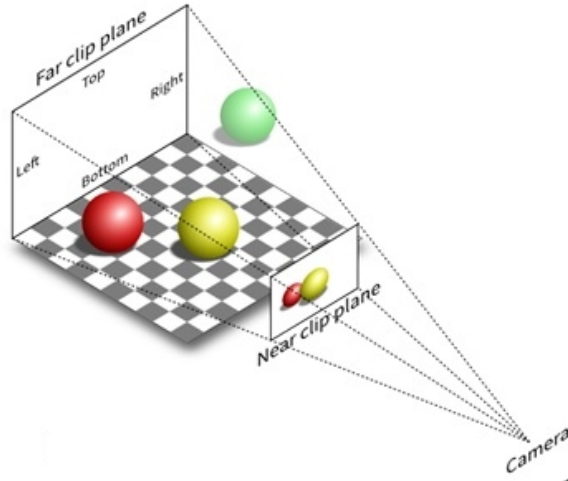


Figure 4.12 view frustum perspective projection.

But in really the perspective projection does not create the 3D effect in perspective, for that is needed to do something called the perspective divide, this task is carried out by the graphic API, and are involved the homogeneous coordinate w adding an additional dimension to explain the projective space inside the Cartesian space as explained in [section 4.1.2.8](#) geometric transformations, see[\[songho\]](#).

The perspective divide, occurs before the viewing transformation as can be seen in the [figure 4.10](#), the projection matrix defined in 4.18 definition, sets things up so that after multiplying with it the eye coordinates in view space, the coordinate w will increase the further away the object is, the

graphical API will apply the perspective divide, so then the further away something is, the more it will be pulled towards the center of the screen.

This explanation can be depicted in the following example of perspective divide.

Assuming this projection matrix that looks as follows.

$$M_{perspec} = \begin{bmatrix} 1.5 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1.2 & -2.2 \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad (4.19)$$

Then this matrix will transform eye coordinates as follows.

$$\begin{aligned} \begin{bmatrix} 1.5 \\ 1 \\ -1 \\ \color{red}{1} \end{bmatrix} &= \begin{bmatrix} 1.5 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1.2 & -2.2 \\ 0 & 0 & -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \\ -1 \\ 1 \end{bmatrix} \\ \begin{bmatrix} 1.5 \\ 1 \\ 0.2 \\ \color{red}{2} \end{bmatrix} &= \begin{bmatrix} 1.5 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1.2 & -2.2 \\ 0 & 0 & -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \\ -2 \\ 1 \end{bmatrix} \\ \begin{bmatrix} 3 \\ 2 \\ 0.2 \\ \color{red}{2} \end{bmatrix} &= \begin{bmatrix} 1.5 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1.2 & -2.2 \\ 0 & 0 & -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 2 \\ -2 \\ 1 \end{bmatrix} \end{aligned} \quad (4.20)$$

Here in 4.20 definition in the result obtained marked in red, is possible to see how the projection matrix sets up the w coordinate, then the perspective divide realized by the graphic API does the perspective effect as follows obtaining the NDC coordinates.

$$\begin{aligned} \begin{bmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \end{bmatrix} &= \begin{bmatrix} x_c / \color{red}{w} \\ y_c / \color{red}{w} \\ z_c / \color{red}{w} \end{bmatrix} \\ \begin{bmatrix} 1.5 \\ 1 \\ -1 \end{bmatrix} &= \begin{bmatrix} 1.5 / \color{red}{1} \\ 1 / \color{red}{1} \\ -1 / \color{red}{1} \end{bmatrix}, \quad \begin{bmatrix} 0.75 \\ \frac{1}{2} \\ 0.1 \end{bmatrix} = \begin{bmatrix} 1.5 / \color{red}{2} \\ 1 / \color{red}{2} \\ 0.2 / \color{red}{2} \end{bmatrix}, \quad \begin{bmatrix} 1.5 \\ 1 \\ 0.1 \end{bmatrix} = \begin{bmatrix} 3 / \color{red}{2} \\ 2 / \color{red}{2} \\ 0.2 / \color{red}{2} \end{bmatrix} \end{aligned} \quad (4.21)$$

Finally the viewport transformation do the projection to the device coordinates screen space similarly that explained in 4.17 definition.

For more information about perspective divide see [\[persDivide1\]](#)

4.1.2.8.2.3 View matrix (lookAt)

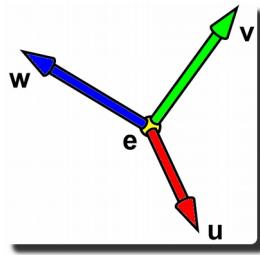
The graphic APIs no has a camera, the camera is simulated by another transformation called viewing transform, and is build with the view matrix, applying this transformation is possible to move all objects in the scene, obtaining the illusion of a camera.

More concretely the viewing transform, generates a view space where all vertex coordinates are seen from the perspective of the camera as the origin of the scene, so the view matrix transforms all the world coordinates from the world space into view coordinates that are relative to the camera direction and position. Hence this transformation is added to the sequence of transformations depicted in the [figure 4.10](#) concretely to the product of the viewport and projection transformations, so that it converts the incoming points from world space to camera coordinates in view space before they are projected.

To specify the camera position and orientation, can be used the following convention.

The eye position e , the looking direction g and the view-up vector t . With these vectors is possible to set up a coordinate system with three perpendicular unit axes u, v, w with the position of the camera as the origin e . To build the coordinate system is possible to do the following.

$$w = -\frac{g}{\|g\|}, u = \frac{t \times w}{\|t \times w\|}, v = w \times u \quad (4.22)$$



With these three perpendicular axes, is possible to create a matrix with this three axes plus a translation vector with the position of the camera, so will be possible transform any vector to that coordinate space by multiplying it with this matrix. This matrix is implemented within the engine applying the explained above, within the method lookAt in the matrix4 class defined in [figure 4.5](#) with a UML diagram.

The view matrix is as follows.

$$M_{lookAt} = \begin{bmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad (4.23)$$

So, is possible to view this transformation as first moving e to the origin and then aligning u, v, w to x, y, z . So, the x_u, y_u, z_u are the direction cosines of the x -axis, x_v, y_v, z_v are the direction cosines of the y -axis, and finally x_w, y_w, z_w are the direction cosines of the z -axis.

4.1.2.8.2.4 Rotation matrix (yawPitchRoll)

The transform in the definition 4.23 relates points in world space to camera space (eye coordinates), but another approach for locating the camera involves Euler angles, with them is possible to define the frame of camera reference within the world space, but how as discussed in the **section 4.1.2.8.1.3** rotation, Euler angles suffer Gimbal locks. However if the camera is located in world space using Euler angles, the transform relating world coordinates to camera coordinates can be derived from the inverse operations discussed in the **section 4.1.2.8.1.3** rotation, concretely the definitions 4.11, 4.12 and 4.13.

The rotations defined there also can be known as *yaw*, *pitch* and *roll*, nevertheless, this angles sometimes when are referred in technical papers or books are different, because a left-handed system of axes is used rather than a right-handed system, then the vertical axis may be the *y*-axis or the *z*-axis, consequently the matrices that represents the rotations are different, here the Cartesian coordinate system represented is the right-handed system.

- *yaw* is the angle of rotation about the *y*-axis.
- *pitch* is the angle of rotation about the *x*-axis.
- *roll* is the angle of rotation about the *z*-axis.

The inverse matrix rotations can be represented with the same matrices shown in definitions 4.11, 4.12 and 4.13 in the **section 4.1.2.8.1.3** rotation, as the transposed of the original rows and columns, where in this case firstly θ is replaced by *yaw*, *pitch* and *roll* in each rotation matrix and is inverted, consequently the matrices can be represented as follows.

To rotate about *y*-axis

$$R_y(-yaw) = \begin{bmatrix} \cos(yaw) & 0 & -\sin(yaw) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(yaw) & 0 & \cos(yaw) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4.24)$$

To rotate about *x*-axis

$$R_x(-pitch) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(pitch) & \sin(pitch) & 0 \\ 0 & -\sin(pitch) & \cos(pitch) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4.25)$$

To rotate about *z*-axis

$$R_z(-roll) = \begin{bmatrix} \cos(roll) & \sin(roll) & 0 & 0 \\ -\sin(roll) & \cos(roll) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4.26)$$

Then this matrices can be represented by a single homogeneous matrix such as has been implemented within the engine inside the **yawPitchRoll** method shown in the **figure 4.5**. Below is shown the implemented matrix.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} T_{11} & T_{12} & T_{13} & T_{14} \\ T_{21} & T_{22} & T_{23} & T_{24} \\ T_{31} & T_{32} & T_{33} & T_{34} \\ T_{41} & T_{42} & T_{43} & T_{44} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (4.27)$$

where

$$\begin{aligned} T_{11} &= \cos(yaw) \cos(roll) + \sin(yaw) \sin(pitch) \sin(roll) \\ T_{12} &= \sin(roll) \cos(pitch) \\ T_{13} &= -\sin(yaw) \cos(roll) + \cos(yaw) \sin(pitch) \sin(roll) \\ T_{14} &= 0 \\ T_{21} &= -\cos(yaw) \sin(roll) + \sin(yaw) \sin(pitch) \cos(roll) \\ T_{22} &= \cos(roll) \cos(pitch) \\ T_{23} &= \sin(roll) \sin(yaw) + \cos(yaw) \sin(pitch) \cos(roll) \\ T_{24} &= 0 \\ T_{31} &= \sin(yaw) \cos(pitch) \\ T_{32} &= -\sin(pitch) \\ T_{33} &= \cos(yaw) \cos(pitch) \\ T_{34} &= 0 \\ T_{41} &= T_{42} = T_{43} = 0 \\ T_{44} &= 1 \end{aligned}$$

With this matrix is possible position and orientated the camera as well within the world space, where $T_{14} = t_x$, $T_{24} = t_y$ and $T_{34} = t_z$ being the camera translation.

4.1.2.9 Transformations unit tests

To verify the correct transformation behaviour, has been implemented some unit tests to determine whether the library is fit for use. Basically the unit tests first try the geometric transformations explained above such as the translation, rotation and scale for later will be used in the engine.

Below is shown the unit tests for translation, scale and rotation.

```

D:\bin\Release> .\UnitTest_Matrix4.exe --log_level=all
Running 1 test case...
Entering test module "boost_test_macro_overview"
d:/unittest_matrix4.cpp(43): Entering test case "Unit_Test_Math_Library_FlowRenderEngine"
Translation *****
1 - identity matrix:
matrix4_1t =
[1 0 0 0
 0 1 0 0
 0 0 1 0
 0 0 0 1]

Translation - (x, y, z, w) = (2, 2, 0, 1)
matrix4_1t =
[1 0 0 2
 0 1 0 2
 0 0 1 0
 0 0 0 1]

Translation again - (x, y, z, w) = (2, 2, 0, 1)
matrix4_1t =
[1 0 0 4
 0 1 0 4
 0 0 1 0
 0 0 0 1]

Translation again - (x, y, z, w) = (1, 2, 3, 1)
matrix4_1t =
[1 0 0 5
 0 1 0 6
 0 0 1 3
 0 0 0 1]

Translation *****
Translation from origin - (x, y, z, w) = (1, 2, 3, 1)
--> matrix4_2t =
[1 0 0 1
 0 1 0 2
 0 0 1 3
 0 0 0 1]

Testing equality
d:/unittest_matrix4.cpp(81): error: in "Unit_Test_Math_Library_FlowRenderEngine": check matrix4_1t == matrix4_2t has failed
d:/unittest_matrix4.cpp(82): info: check matrix4_1t == matrix4_1t has passed
d:/unittest_matrix4.cpp(83): info: check matrix4_1t != matrix4_2t has passed
d:/unittest_matrix4.cpp(84): error: in "Unit_Test_Math_Library_FlowRenderEngine": check matrix4_1t != matrix4_1t has failed
End Translation *****

Scaling *****
--> matrix_a uniform scale - (x, y, z, w) = (0.5, 0.5, 0.5, 1)
--> matrix_a =
[0.5 0 0 0
 0 0.5 0 0
 0 0 0.5 0
 0 0 0 1]

--> matrix_a uniform scale - (x, y, z, 1) = (2.0)
--> matrix_a =
[1 0 0 0
 0 1 0 0
 0 0 1 0
 0 0 0 1]

--> matrix_a non uniform scale - (x, y, z, w) = (1, 2, 3, 1)
--> matrix_a =
[1 0 0 0
 0 2 0 0
 0 0 3 0
 0 0 0 1]

--> matrix_a uniform scale - (x, y, z, 1) = (8.0)
--> matrix_a =
[8 0 0 0
 0 16 0 0
 0 0 24 0
 0 0 0 1]

End Scaling *****
Other testing *****
Test - 1 *****
--> matrix4_1t =
[1 0 0 5
 0 1 0 6
 0 0 1 3
 0 0 0 1]

```

```
--> matrix4_1t translated and scaled - (x, y, z, w) = (1, 2, 3, 1)
--> matrix4_1t =
[1 0 0 5
 0 2 0 6
 0 0 3 3
 0 0 0 1]
Test - 2 *****
--> Mt is translated twice from origin to (2, 2, 0, 1)
--> and Mt is translated again to (1, 2, 3, 1)
--> then Mt =
[1 0 0 5
 0 1 0 6
 0 0 1 3
 0 0 0 1]
--> Ms is non uniform scaled with (1, 2, 3, 1)
--> then Ms =
[1 0 0 0
 0 2 0 0
 0 0 3 0
 0 0 0 1]
--> then if Mt * Ms =
[1 0 0 5
 0 2 0 6
 0 0 3 3
 0 0 0 1]
End Other testing *****
Rotate *****
--> Mr is rotated 90 degrees with the arbitrary unit axis, the unit vector is (Rx,Ry,Rz) = (1, 0, 0)
MrExpected =
[1 0 0 0
 0 -4.37113883e-08 -1 0
 0 1 -4.37113883e-08 0
 0 0 0 1]
Result Mr =
[0.99999994 0 0 0
 0 -4.37113883e-08 -1 0
 0 1 -4.37113883e-08 0
 0 0 0 1]
d:/unittest_matrix4.cpp(170): info: check Mr == MrExpected has passed
End Rotate *****
D:\bin\Release>
```

As it can be seen in the table, the geometric transformations works correctly.

4.1.2.10 Normal calculation

In modern graphic approaches, the lighting calculations are made into the shaders, but in order to be able to simulate the effects of light bouncing off to the meshes surfaces, is necessary determinate the direction the surfaces facing in, to achieve this is necessary calculate the normal of each mesh surface, the normal is a normalized direction vector pointing away from the surface.

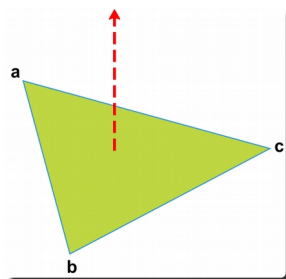


Figure 4.13 normalized direction vector pointing away from the surface.

The normals are treated like vertex attributes, see **section 4.2.1.3** vertex element. All meshes are formed by polygons that are triangles, and is necessary calculate the surface normal per each triangle of the mesh, to achieve this the cross product comes into play. For a given two vectors a

and b the cross product produces a vector that is orthogonal to both a and b , being the surface normal of a triangle, see [figure 4.13](#).

$$surface\ normal = ||b - a \times c - a||$$

To calculate the normal, is needed takes one of the triangle's vertices jointly with the other two vertices are used to generate the normal via cross product, finally the surface normal are normalized. The direction of the normal that pointing away from the surface is determined by the order how the vertices are took, so the cross product of vectors a and b is the inverse of the cross product of vectors b and a .

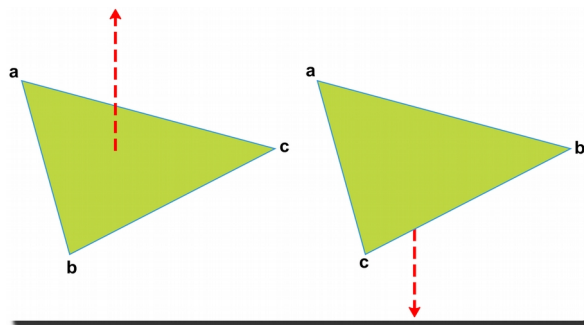


Figure 4.14 anti-clockwise vertex winding and clockwise vertex winding.

The left image shows anti-clockwise vertex winding and the right image shows the clockwise vertex winding.

The normals calculation within the engine initially is implemented within a method called **generateNormals** within the mesh class, so the normals must generated calling this method manually, otherwise the mesh will not has normals calculated, and the light will no affect to the meshes. Nevertheless the method implementation could be improved.

The explained above how the normal is calculated, when the mesh is indexed can not applied since does not works, due to a single vertex is part of multiple faces and each of which faces is in a wildly differing direction, so a given vertex can be part of different faces, in front of this situations a common solution is to use the normalised sum of the normals for each face the vertex is used in, with this the normals that get interpolated by the vertex shader gives as a result a normal that approximately represents each surface.

Finally, to preserve the direction of the normals when a transformation is applied to a mesh since obviously that is the vertex is transformed by the model matrix see [section 4.1.2.8.1](#) consequently the vertices normal should be transformed too, hence is needed a matrix to achieve this, that in this case is not exactly the model matrix.

Depending the transformation type applied maybe enough use the upper 3×3 section of the model matrix, but in case that the transformation applied in a model matrix is a non uniform scale this will not work, to solve this the common solution is apply the inverse transpose of the model matrix, this will preserve the rotations in the model matrix inverting the scales and achieving a normals in the correct direction.

Due to the matrix inversion is quite costly, if the model has a uniform scale, then using only the model matrix is correct, otherwise if the model has a non uniform scale applied, a matrix model inverse transpose is needed, but calculate the inverse transpose in the vertex shader all the time

with the GPU rather than in the CPU is a good general purpose solution, though is possible calculate it in CPU too, but the engine does not have it implemented.

4.1.2.11 Tangents and bi-tangents calculation for bump mapping

The illumination at each pixel rendered is determined by the normal vector used during the evaluation of the lighting formula, in the previous section, it has been discussed the calculation of the normals of a surface to apply diffuse and specular lighting, but this is not enough to portray reality. Surfaces are seldom completely flat, so it must be found a way to simulate the roughness of a surfaces. To simulate this roughness, the bump mapping comes into play. The bump mapping is way of storing the roughness of a surface.

Hence, with the bump maps is possible to represent more accurately the materials lighting adding greater detail, closer them to the reality much more, a bump map stores a normal per texel, where each of which points in a varying direction in accordance with the improprieties of the material that they are simulating. using a texture map to distort the normal vector at each pixel. The following figure shows the explained above.

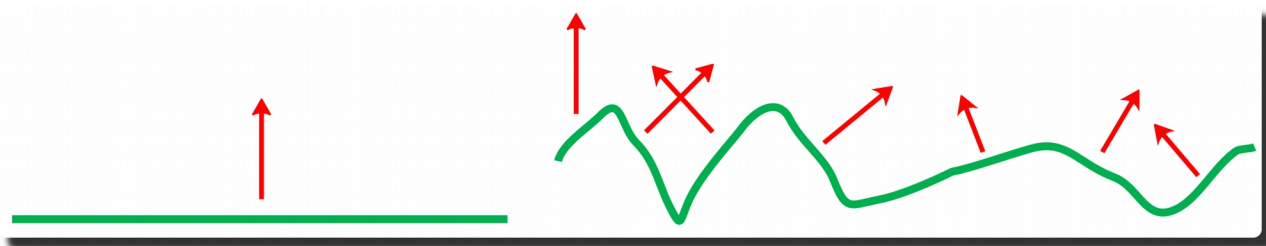


Figure 4.15 on the left are depicted a surface with its normal, on the right a surface with normals derived from a bump map.

The bump maps are stored like any other texture, and the components of each normal vector are encoded as a *RGB* color, where each of them, storing a component of normalized direction vector. With this scenario, it is possible to have a unique per-fragment normals. The following figure, shows an example of a texture and the corresponding bump map, in the middle, in conjunction with the obtained result.

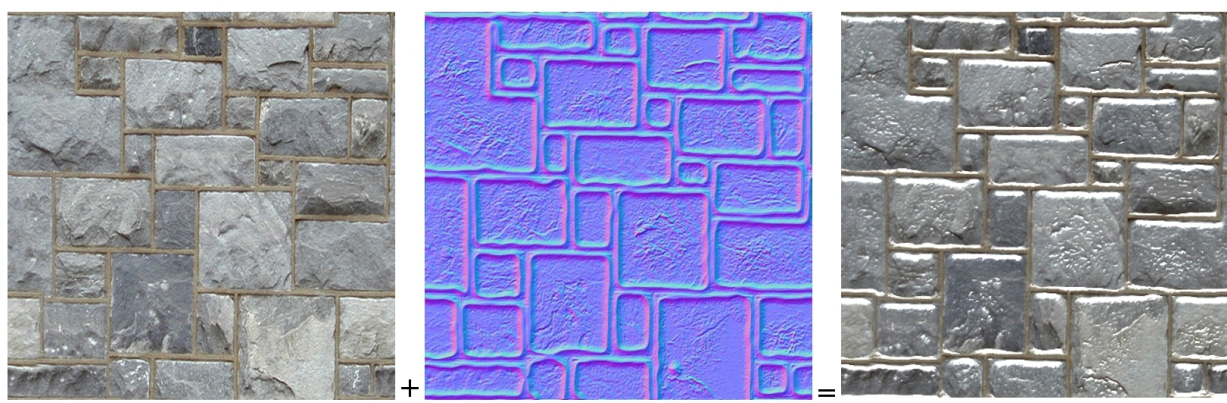


Figure 4.16 depicts the original texture on the left, and in the middle the normal map, and finally on the right the result is shown.

It is possible to build a bump map with some image manipulation software like Gimp or Photoshop, for a determinate texture. The bump mapping, is also known as normal mapping. The texture maps which contains vector data, the values stored are expressed in the coordinate system of the texture

map itself, so that the geometric details are decoupled of any particular geometric, being possible applies a geometric texture map to any triangle mesh, without having to account for the object-space coordinate system used by its vertices.

The x and y axis in a coordinate system of a texture map are aligned to the horizontal and vertical directions of the 2D image, the z axis points upward out of the image plane, see [figure 4.17](#) below.

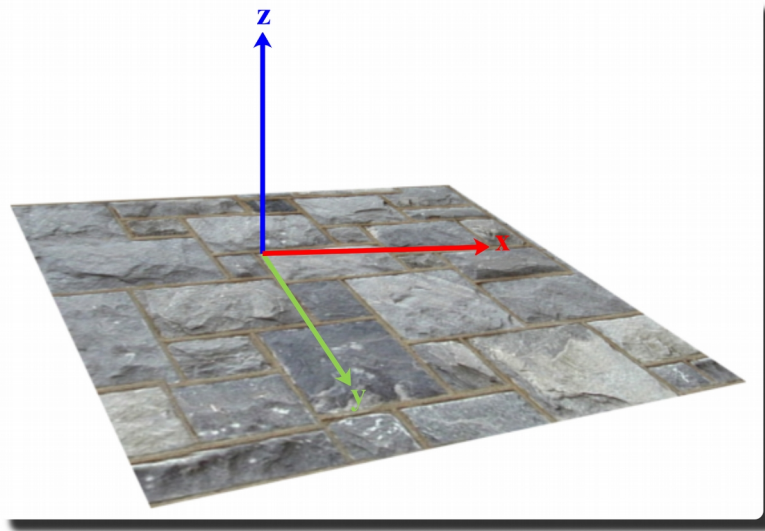


Figure 4.17 shows the x and y axes are aligned to the texel image, and z axis point out the image plane.

In order to perform the bump mapping calculations within the shaders using the geometric information stored in a texture map, is needed a way to transform between the coordinate system of the texture map and model space. To accomplish this, a tangent space is needed. Each vertex in a triangle mesh has a normal vector \mathbf{N} , hence, is needed to find a vector tangent \mathbf{T} , that is perpendicular to vector \mathbf{N} , this is done by identifying the directions in model space that correspond to the coordinate axes of the texture map, the directions within the model space are not constant and vary in each triangle model belongs to, hence, for each triangle vertex, the x and y axis of the texture map are aligned with the texture coordinates UV assigned to triangle vertex, and the z axis of the texture map, are aligned to the vertex normal of the triangle, because the z axis in the texture map points directly out of the plane, hence, the x and y axis of the texture map are tangent to the surface in object space, since its directions goes to the same direction but they do not get intersect, so that, is needed calculate average unit-length tangent vector \mathbf{T} for each vertex in a triangle mesh, thus is created a smooth tangent field on the surface of a model.

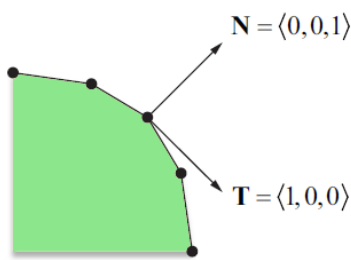


Figure 4.18 depicts each vector at a vertex the tangent space is aligned to the tangent plane and normal vector. This image has been extracted from [\[math3DCGBumpMap\]](#).

As described above, the x and y axis of the texture map are tangent to the surface in object space, so the two tangent directions are perpendicular to each other all the time, but some times this not be strictly true, hence, a second tangent direction is calculated, and is called the bi-tangent \mathbf{B} . All this build a tangent space, where the three vectors \mathbf{T} , \mathbf{B} and \mathbf{N} form the basis of the tangent frame at each vertex, and the coordinate space in which the x , y , and z axes are aligned to these directions is called tangent space.

The following explains the implementation that has been carried out within the engine of what has been explained so far, from a mathematical point of view. To carry out the calculations, it is necessary to use linear algebra again, it is needed to find a 3×3 matrix at each vertex that transforms vectors from object space in to tangent space, we know that the z axis of tangent space always is mapped to a normal vector of a vertex, then the tangent space is aligned such that the x axis corresponds to the s direction in the bump map, and the y axis corresponds to the t direction in the bump map.

Then if,

\mathbf{Q} represents a point inside the triangle.

\mathbf{T} and \mathbf{B} are tangent vectors aligned to the texture map.

P_0 is the position of one of the vertices of the triangle.

$\langle s_0, t_0 \rangle$ are the texture coordinates at the vertex.

and, assuming that we have a triangle whose vertex positions are given by the points P_0 , P_1 and P_2 and whose corresponding the following texture coordinates are given by $\langle s_0, t_0 \rangle$, $\langle s_1, t_1 \rangle$, $\langle s_2, t_2 \rangle$, it is possible to do the following calculations.

$$\begin{aligned} \mathbf{Q}_1 &= P_1 - P_0, \langle s_1, t_1 \rangle = \langle s_1 - s_0, t_1 - t_0 \rangle \\ \mathbf{Q}_2 &= P_2 - P_0, \langle s_2, t_2 \rangle = \langle s_2 - s_0, t_2 - t_0 \rangle \end{aligned} \quad (4.28)$$

then is needed to solve the following equations for \mathbf{T} and \mathbf{B}

$$\begin{aligned} \mathbf{Q}_1 &= s_1 \mathbf{T} + t_1 \mathbf{B} \\ \mathbf{Q}_2 &= s_2 \mathbf{T} + t_2 \mathbf{B} \end{aligned} \quad (4.29)$$

The definition xx.x can be seen as a linear system with six unknowns, this is possible to write in matrix form as follows.

$$\begin{bmatrix} (\mathbf{Q}_1)_x & (\mathbf{Q}_1)_y & (\mathbf{Q}_1)_z \\ (\mathbf{Q}_2)_x & (\mathbf{Q}_2)_y & (\mathbf{Q}_2)_z \end{bmatrix} = \begin{bmatrix} s_1 & t_1 \\ s_2 & t_2 \end{bmatrix} \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix} \quad (4.30)$$

To get the unnormalized \mathbf{T} and \mathbf{B} tangent vectors for the triangle whose vertices are P_0 , P_1 and P_2 , is needed multiply both sides by the inverse of $\langle s, t \rangle$ matrix.

$$\begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix} = \frac{1}{s_1 t_2 - s_2 t_1} \begin{bmatrix} t_2 & -t_1 \\ -s_2 & s_1 \end{bmatrix} \begin{bmatrix} (\mathbf{Q}_1)_x & (\mathbf{Q}_1)_y & (\mathbf{Q}_1)_z \\ (\mathbf{Q}_2)_x & (\mathbf{Q}_2)_y & (\mathbf{Q}_2)_z \end{bmatrix} \quad (4.31)$$

Results obtained with the definition 4.31, it allows to build the following matrix TBN

$$\begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix} \quad (4.32)$$

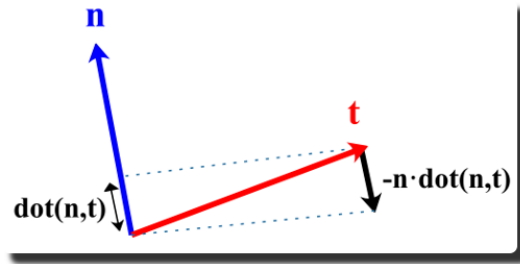
with this matrix is possible to go from tangent space to model space, so is possible transform normals, extracted from the texture map into model space, however, it is usually done the calculations in the other way around, so that, the transformations are done from model space to tangent space within the shaders, keeping the extracted normal as-is, so all computations are done in tangent space. To obtain this, the transposed matrix of definition 4.32 is necessary, such as shows in the following definition.

$$\begin{bmatrix} T'_x & T'_y & T'_z \\ B'_x & B'_y & B'_z \\ N'_x & N'_y & N'_z \end{bmatrix} \quad (4.33)$$

The transposed matrix is applied within the shaders, but it only works if the space that the matrix represents defined in 4.32, is orthogonal, since the results obtained with the definition 4.31, are usually not exactly perpendicular, this perpendicularity is build in function the bump mapping, then the perpendicularity, it depends the texture mapping skewing. To solve the perpendicularity, we can apply the Gram-Schmidt orthonormalization, so the tangent \mathbf{T} is perpendicular to the normal \mathbf{N} , then will be possible to apply the TBN transposed matrix within the shaders, hence, the space represented by the matrix defined in 4.32 must be orthogonal.

The orthogonalization is done with the Gram-Schmidt orthonormalization, such as follows.

$$\begin{aligned} T' &= T - (N \cdot T)N \\ B' &= B - (N \cdot B)N - (T' \cdot B)T' \end{aligned} \quad (4.34)$$



\mathbf{N} and \mathbf{T} are almost perpendicular, hence, to fix it, \mathbf{T} is pushed down in the direction of $-\mathbf{N}$ by a factor of $\text{dot}(\mathbf{n}, \mathbf{t})$

Finally it is necessary to calculate, the handedness, in some cases in symmetric models the coordinate UVs are oriented in the wrong way and \mathbf{T} has the wrong orientation, hence, to check whether it must be inverted or not, is needed to do the cross product between \mathbf{N} and \mathbf{T} and must have the same orientation than \mathbf{B} , so we need to check if $\text{dot}(\text{cross}(\mathbf{n}, \mathbf{t}), \mathbf{b}) > 0$. If it's false, just needed invert \mathbf{T} .

$$B' = T'_w(N \times T') \quad (4.35)$$

Then within the shaders, applying the definition matrix 4.33 the lighting calculations are applied to simulate the roughness of the materials.

The exposed of this section and the implementation within the engine, has been based on the following resources[[math3DCGBumpMap](#)][[FGDevelopRendering](#)], the implementation has been adapted to the engine and to the mathematics library developed for this thesis.

4.1.3 Platform sub-module

The platform sub-module is defined as a name space called **Flwre::Core::Platform**, this submodule is the lowest of library since harbour a collection of routines that are used frequently enough that their interfaces must be exposed through a single header file called "*ResourcesFlwre.h*", this is useful in order to save the programmer time by not constantly having add the various include files repeatedly, since this file exposes the inclusions for most of the standard C/C++ libraries and the Standard Template Library (STL) containers used in the engine. However, as explained below to use the STL library is not the better solution in a real-time environments.

The Standard Template Library (STL) has been used mainly for obvious reasons of time, although for better efficiency it would have been better develop an own library custom made for the engine, since real-time applications sometimes have performance penalties because of the overhead of STL, both in time and memory since the memory allocation patterns of STL, which are not conducive to high-performance programming and tend to lead to memory fragmentation causing cache misses, then this scenario makes unusable in real-time applications.

Any real-time application needs a lot knowledge about memory usage patterns, for this reason is not suitable use a generic set of template containers due to they not have this knowledge, at the same time it depends the hardware where they run for might not have STL support, for example on a video game console, with limited or no virtual memory facilities unlike a computer and with an exorbitant cache miss costs, is probably better off writing custom data structures that have predictable and/or limited memory allocation patterns, although in reality the better solution is writing custom structures anywhere, see[[GEngArchj](#)].

The Standard Template Library (STL) is very popular but exists other third-party libraries which provide these kinds of services too as shows below:

- **STLPort** is a portable and optimized implementation of STL, see[[GEngArchaj](#)].
- **Boost** is a powerful data structures and algorithms library, STL style, see [[GEngArchbj](#)].
- **Loki** is a powerful generic programming template library, in research and proof-of-concepts way, see[[GEngArchcj](#)].

This submodule defines a basic data-types as well as follows:

Type	Description
Vector2i Vector3i	These objects specifies two-dimensional vectors and three-dimensional vectors with integer values.
Vector2{f d} Vector3{f d} Vector4{f d}	These objects specifies a two-dimensional vectors and three-dimensional vectors with single or double precision floating point numbers respectively.
Matrix4 {f d}	Specifies a 4×4 matrix single or double precision floating point numbers, these matrices are all

	organized in column-major fashion.
String	String object contains single string encoded with the UTF-8 universal character set
OfStream, IfStream, Ios, StringStream	The objects are used to operate with files
FlTime	Specifies a double precision time value milliseconds

Table 1.1 shown data types defined in "ResourcesFlwre.h" file in Flow Render Engine.

Sample code snippet from **table 1.1**, implemented in the *ResourcesFlwre.h* file.

```
namespace Flwre
{
    namespace Core
    {
        namespace Platform
        {
            (...)
            typedef Flwre::Core::Math::Vector2<int> Vector2i;
            typedef Flwre::Core::Math::Vector3<int> Vector3i;

            typedef Flwre::Core::Math::Vector2<float> Vector2f;
            typedef Flwre::Core::Math::Vector3<float> Vector3f;
            typedef Flwre::Core::Math::Vector4<float> Vector4f;
            typedef Flwre::Core::Math::Matrix4<float> Matrix4f;

            typedef Flwre::Core::Math::Vector2<double> Vector2d;
            typedef Flwre::Core::Math::Vector3<double> Vector3d;
            typedef Flwre::Core::Math::Vector4<double> Vector4d;
            typedef Flwre::Core::Math::Matrix4<double> Matrix4d;
        }
    }
}
```

In this sub-module is also defined the control symbol visibility, when designing a software API, it is necessary to keep in mind the need to handle the symbols for their correct visibility from other modules or applications that linking with them, these symbols are useful for the linker to decide if different modules (object files, shared dynamic libraries, executables) will share the same data or code. Each platform makes its own management of this visibility, so it is necessary to define the attributes for each platform, so that they are applied at compile time depending on the platform where the engine is compiled, this is defined in this name space within the VisibilityMacros header file with a macro for its proper management on each platforms, see [\[Visibility\]](#).

4.1.4 Utility sub-module

The utility sub-module is defined within the name space **Flwre::Core::Utility**, and contains often needed classes like a dynamic loader libraries, file systems management access and log management, finally a timer is implemented as well, all this is used inside the engine with different targets, due to some this classes are accessed from anywhere of the engine, some of them are instantiate as a singleton as been explained in **section 3.4.3, 4.1.1.2** singleton template class, and see **figure 4.1**. In the following sections each module is explained more widely.

4.1.4.1 Dynamically loaded C++ Objects

The loading libraries at runtime are used within the engine to manage certain modules as a plug-in. In this case, the module that is treated as an add-on, loading it at runtime when the engine is initialized is the **GraphicsOGL3** module. Therefore with this technique it would be possible to change the graphic API with which the engine renders at runtime between different implementations such as Direct3D or Vulkan, if these modules were implemented, however in this work this module is only implemented with OpenGL3.3 but the loading mechanism be leaves ready for future works.

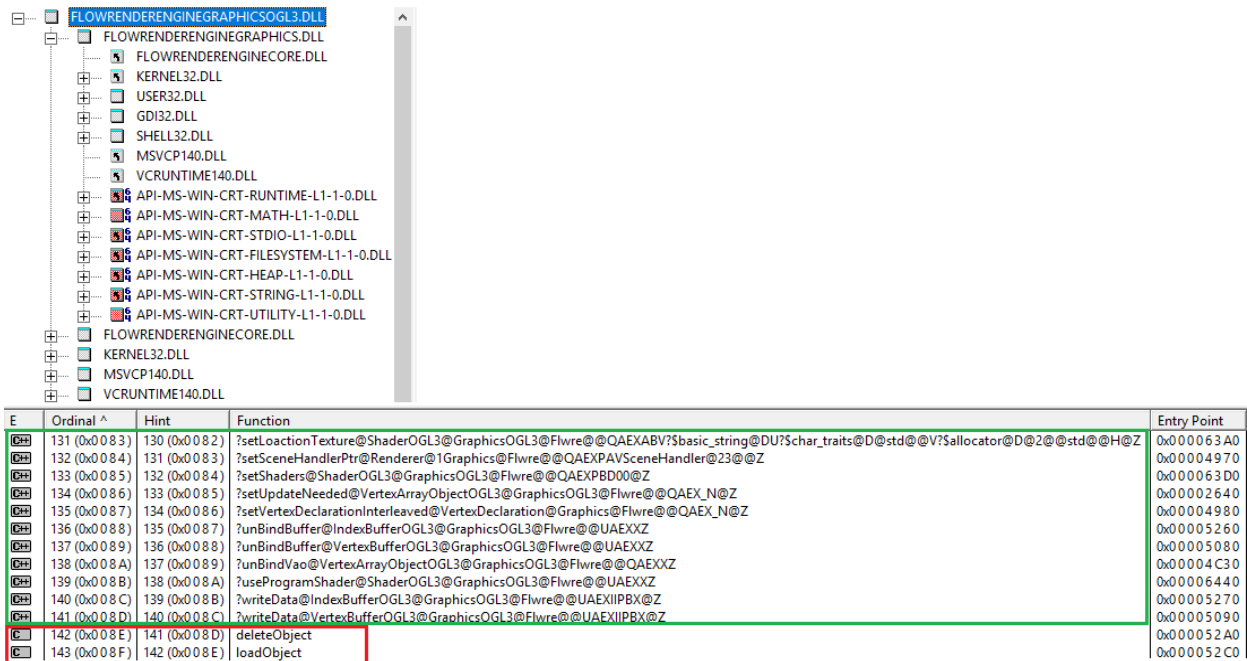
Obviously to change the rendering interface implementation between different graphical APIs implementations at runtime, it is necessary stopping the rendering first and starting again with the new layer is loaded.

In the C++ language does not exist support for dynamic loading libraries, there are, of course, ways to get around the limitation, since C++ programs can call C functions, but this are non optimal from a performance standpoint and error prone. To improve this limitation an object-oriented C++ wrapper has been made based on [\[DynLib\]](#), This has been adapted to the scenario of this project and has been improved, since the rendering engine has a cross-platform approach and must work on both systems Windows and Linux. However, the mechanism loading libraries at runtime does not work the same manner between Windows and Linux, and because of this the container has been adapted and improved to cover this scenario in this project.

4.1.4.1.1 How does it work?

The dynamic loading library dynamically load symbols from an object file. When a source file is compiled, it is placed in an object file, which contains the functions, structures, and any other entities that are a part of the compiled source. To allow other object files to link to a given object file and call its functions and other things, each of the elements in an object file are associated with a symbol, which acts as an id for looking up a determinate element.

In a compiled C program, the symbols in an object file directly correlate to the ASCII function name for the function they represent. This makes dynamically linking to a shared library that was compiled with C very easy. For example, if you want to call a function named **loadObject()**, the symbol name would be **loadObject**. When the source being compiled is C++ instead, the principle is the same, but the symbols are not as clear, the [figure 4.19](#) shown this.



E	Ordinal ^	Hint	Function	Entry Point
131 (0x0083)	130 (0x0082)		?setLocationTexture@ShaderOGL3@GraphicsOGL3@Flwre@@@QAEABV?\$basic_string@DU?\$char_traits@D@std@@@V?\$allocator@D@2@@@std@@@H@Z	0x000063A0
132 (0x0084)	131 (0x0083)		?setSceneHandlerPtr@Renderer@1Graphics@Flwre@@@QAEPAVSceneHandler@23@@@Z	0x00004970
133 (0x0085)	132 (0x0084)		?setShaders@ShaderOGL3@GraphicsOGL3@Flwre@@@QAEPAVSceneHandler@23@@@Z	0x000063D0
134 (0x0086)	133 (0x0085)		?setUpdateNeeded@VertexArrayObjectOGL3@GraphicsOGL3@Flwre@@@QAE_N@Z	0x00002640
135 (0x0087)	134 (0x0086)		?setVertexDeclarationInterleaved@VertexDeclaration@Graphics@Flwre@@@QAE_N@Z	0x00004980
136 (0x0088)	135 (0x0087)		?unBindBuffer@IndexBufferOGL3@GraphicsOGL3@Flwre@@@QAE_X@Z	0x00005260
137 (0x0089)	136 (0x0088)		?unBindBuffer@VertexBufferOGL3@GraphicsOGL3@Flwre@@@QAE_X@Z	0x00005080
138 (0x008A)	137 (0x0089)		?unBindVao@VertexArrayObjectOGL3@GraphicsOGL3@Flwre@@@QAE_X@Z	0x00004C30
139 (0x008B)	138 (0x008A)		?useProgramShader@ShaderOGL3@GraphicsOGL3@Flwre@@@QAE_X@Z	0x00006440
140 (0x008C)	139 (0x008B)		?writeData@IndexBufferOGL3@GraphicsOGL3@Flwre@@@QAEIIPBX@Z	0x00005270
141 (0x008D)	140 (0x008C)		?writeData@VertexBufferOGL3@GraphicsOGL3@Flwre@@@QAEIIPBX@Z	0x00005090
142 (0x008E)	141 (0x008D)		deleteObject	0x000052A0
143 (0x008F)	142 (0x008E)		loadObject	0x000052C0

Figure 4.19 depicts symbol name functions compiled with c framed in red and functions compiled with C++ framed in green in the flowrenderengineOGL3 library.

Clearly can be seen how the symbol names in C has the same name that the functions and otherwise C++ symbol names not. This is due to symbols for a C program are simple because C has no namespaces. Since names are all global, it makes sense to just use the function name for the symbol. With C++, there are a number of different constructs that place scope on names, such as namespaces and classes. This means that when a C++ compiler creates the symbol for a C++ function, it uses some sort of mangled form of the function name and namespace identifiers. To compound the issue, there is no standard for what that mangled form will be. The result is that there is no reliable way for C++ programs to load a symbol through the dynamic loading library using only the symbol name.

To solve this issue, it is possible call functions compiled with C-style symbols from C++ functions. C++ provides a mechanism for indicating that specific sections of C++ source code should be compiled with C symbols, this is possible to achieve with the keyword "extern C", so the sections contained within an "extern C" block are loadable dynamically just as if they were in a library compiled by a C compiler, even if they are mixed in with sections of code compiled with normal mangled C++ symbols. Furthermore, an "extern C" section is otherwise compiled as C++. This means that the code inside "extern C" functions have full access to C++-linked elements. In this way, a C++ function can dynamically load a function with a C-linked symbol name, which can in turn instantiate a C++ object and return a pointer to it. See the following code snippet, where there is an implementation example of the explained above, is just an example is not the implementation made within the engine thought the concept is the same.

```
// This is the main example program
#include <dlfcn.h>
int main(int argc, char** argv)
{
    // Open the libFlowRenderEngineOGL3 shared library
    void* library = dlopen("libFlowRenderEngineOGL3.so", RTLD_NOW);
```

```
//Get the loadObject function, for loading objects
void* loadObject = dlsym(library, "loadObject");
//Get a new renderer object
void* renderer_obj = (*loadObject)();
Renderer* renderer = reinterpret_cast<Renderer*>(renderer_obj);
//processes draw
renderer->draw();
//Get the deleteRenderer function and delete the filter
void* deleteRenderer = dlsym(library, "deleteObject");
(*deleteRenderer)(renderer_obj);
}
//renderer.cpp
// The superclass, known to the program that is doing the dynamic loading.
class Renderer
{
public:
    virtual void draw() = 0;
};
//The RenderEngineOGL3 class can be compiled into a shared object
//library, along with library.cpp, and then loaded by the program.
class RenderEngineOGL3 : public Renderer
{
public:
    virtual void draw()
    {
        /* Process render passes */
    }
};
//library.cpp
extern "C"
{
    // loadObject function creates new RenderEngineOGL3 object and returns it.
    void* loadObject(void)
    {
        return reinterpret_cast<void*>(new RenderEngineOGL3());
    }
    // The deleteObject function deletes the RenderEngineOGL3 that is passed
    // to it. This isn't a very safe function, since there's no
    // way to ensure that the object provided is indeed a RenderEngineOGL3.
    void deleteObject(void* obj)
    {
        delete reinterpret_cast<RenderEngineOGL3*>(obj);
    }
}
```

In Unix based systems like Linux or even though Android systems, to load a symbol from a shared library file, applications first open the library, using the **dlopen()** function, then retrieve the desired symbol with **dlsym()**. The **dlopen()** function takes a filename, which can be either a fully qualified path name or a relative path in which case **dlopen()** searches for it in the normal places where one looks for a library. A successful **dlopen()** call returns a void pointer handle, which can then be used by **dlsym()** to load the symbol, with a "symbol" parameter containing the name of the desired symbol. When **dlsym()** finds a symbol, it returns a function pointer that can be used to call the loaded function.

In windows, run-time dynamic linking is carried out in different manner, since not POSIX-compatible way is used and other different routines are used to operate with. Whereas that the definition in Unix based systems is found in "dlfcn.h" file located in system headers directory that

by default is "/usr/include", in windows this is found in windows SDK directory and its location it depends the windows version used. See [dlfcn][dlopen][windlfcn] for more information in Linux and Windows respectively.

Due to this rendering engine has a cross-compile approach, it becomes necessary an implementation of dlfcn for Windows. To get this target, an implementation of wrapper around the Windows functions has been integrated in project, to make programs written for POSIX that use dlfcn work in Windows without any modifications, see [dlfcn-win32] for more details.

However, the use of **dlopen()** and **dlsym()** functions alone is not enough to achieve a safe and robust approach, therefore it has been carried out the integration of a library that let deal with loading and unloading dynamic objects in object-oriented fashion, combined with dlfcn wrapper implementation for windows to get cross-platform approach.

The library seeks the following goals:

- Cross-platform approach via dlfcn wrapper for windows.
- Don not call any C function directly, everything is object-oriented.
- Loading given only the name in ASCII.
- Outside the loading library anything must deal with a void pointer or reinterpret_cast.
- Delete an object given only doing reference to that object.

4.1.4.1.2 Implementation within the engine

When the engine is used from an application, application context is initialized see **section 4.1.1.1** application context and **section 4.2.3** windowed application, in this process is loaded the **GraphicsOGL3** module at runtime as a plug-in, this module contains the interface implementation declared in **Graphics** module with OpenGL3.3 to render objects, to achieve to load the **GraphicsOGL3** module, is used the mechanism explained in the **section 4.1.4.1** dynamically loaded C++ objects.

Three classes form the C++ wrapper container that let deal with loading and unloading dynamic objects in object-oriented fashion.

Class name	Description
DynamicObject	Is a common base class for any derive class which wants to be loaded at runtime, in this case the renderer class derives from it, defines a deleteObject() function to destroy loaded class, so with this method DynamicObject is deleted itself.
DynamicLibrary	This a shared library that encapsulates the objects loading, through newObject() function, hiding all of the low-level details to the rest of the engine. This is created through the static DynamicLoader
DynamicLoader	Load an object by name and creates a new DynamicLibrary to hold it, to achieve this a DynamicLibrary is created through the static loadObjectFile() function in DynamicLoader class.

A library loaded by DynamicLibrary has defined a **loadObject()** function within the keyword "extern C", for the reasons explained in above section. When the program calls **newObject()** on an instance of DynamicLibrary, **newObject()** in turn calls **loadObject()** on the dynamically loaded library, and passes it the name of the class to instantiate. The **loadObject()** function acts as a

simple factory for dynamic objects, and is responsible for determining which class to load based on the class name it is given, at the same time, a DynamicLibrary has defined a **deleteObject()** to delete it in the same conditions as well. See the code file in render engine implementation "EngineGraphicsDlIOGL3.cpp", and the following **figure 4.20** that shown a simplified UML diagram the implementation explained in the next page.

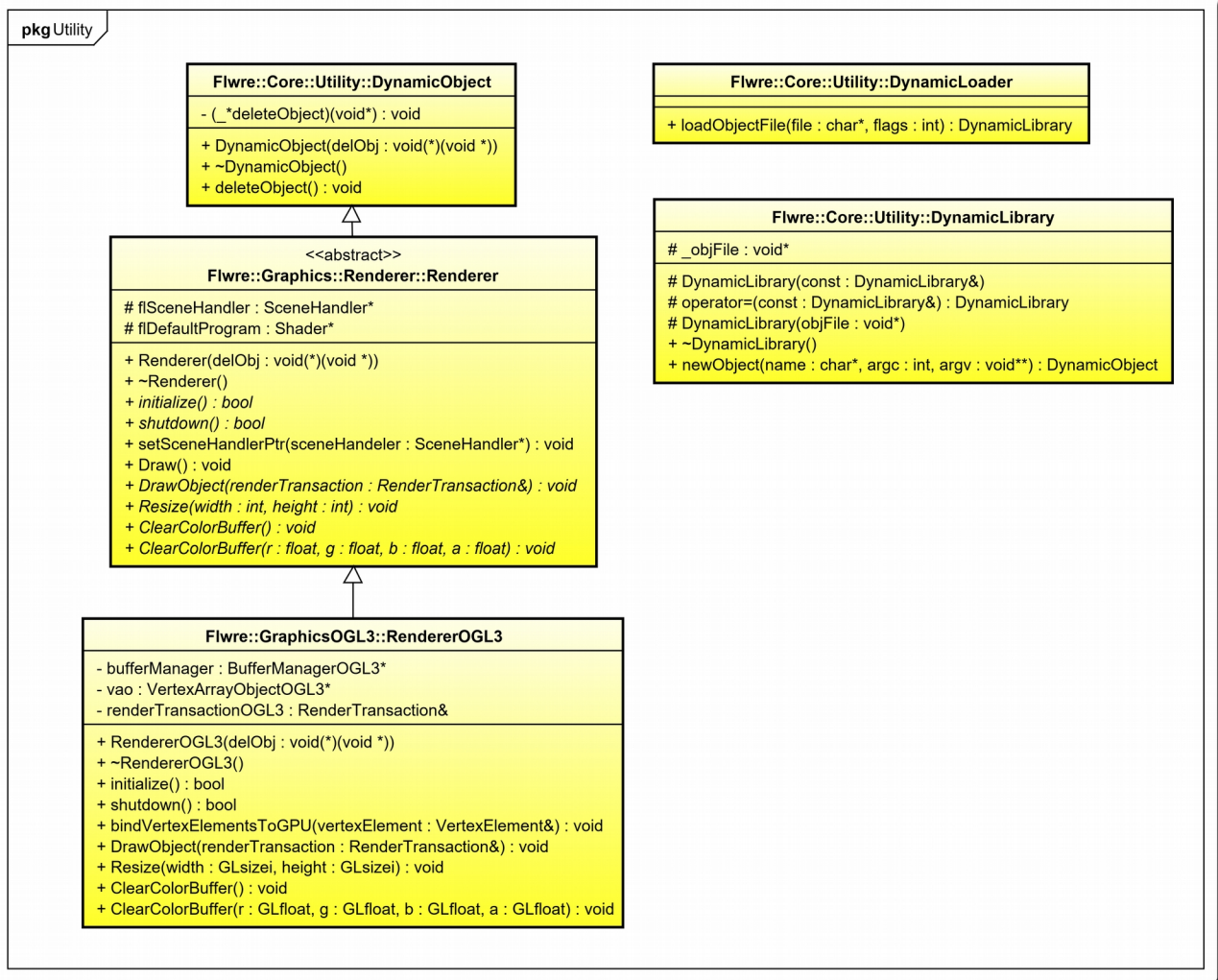


Figure 4.20 shown the dynamic loader implementation to load the RendererOGL3 module at runtime.

4.1.4.2 File system

The engine requires a file handling due to needs to open or create files for data is either read, written, or is appended. This class encapsulate all operations related file handling in a singleton approach so can be accessed from anywhere of the engine see **section 4.1.1.2** singleton template class.

This class hides the particularities of file management implementation on different platforms, Initially the class only implement a create file method and a system to find out the application path where from is executed.

4.1.4.3 Log system

This class implements a basic system log for writing debug/log data to files, is a singleton approach as well since is possible write info to log file from anywhere to the engine, see [section 4.1.1.2](#). The system will define a three level type messages: normal messages, warning messages and error messages.

Initially only the normal messages has been implemented, this messages are written in both standard output and in a plane file text, at the same time, the messages printed for standard output are represented in a console window, but in the future the messages will be rendered in window context and the output text could be improved to a rich text as html or similar.

4.1.4.4 Timer

The real-time applications need to keep track of time, whether for sequencing purposes or for simulation. Operating systems and CPUs have support for a clock, but initially direct access to this clock cannot be done in a platform-independent manner, so the details must be encapsulated to hide the engine layer dependencies, this class implements a methods to manage the time in a cross-platform manner, to get this the chrono C++ library C++11 is used.

The chrono library was designed to be able to deal with the fact that clock access might be different on different platforms and in turn offer an easy way to improve the time precision, since due to the rapid evolution experienced by hardware, every time a higher resolution of time is needed to handle processing times, this evolution in hardware is what caused years ago the introduction of new types of time in the POSIX libraries.

Timer class implements several methods to manage the time, the mainly methods as described as follows, firstly defines three methods to retrieve elapsed time, from the engine initialization or from that the counter is reset in different time resolutions, these are **getElapsedTimeInSeconds**, **milliseconds** and **microseconds**, this methods manage the elapsed time, which includes for example input/output (I/O) operations, with these methods are the basis to calculate the engine frame rate and the actual time when the engine is executed, the method **updateFramesStates()** is called in each frame to calculate the frames per second that the engine is rendering, this method calculates at the same time a frames average per second taking different samples, and store the best and worst frames per second reached as well. Another important value stored in this class is delta time, delta time is the elapsed time between the current frame and the last frame such as can be retrieved with the method **getDeltaTime()**, delta time is used for variably updating scenery based on the elapsed time since the last updated, then the movement will take the same amount of real world time to move across the screen regardless of the frame rate update, whether the delay be caused by lack of processing power or for a momentary workload.

Finally a **getTime()** method is implemented as well, to retrieve the actual time in **hh:mm:ss.s** format and an internal **reset()** method to reset counters when the engine is started. The Timer class has a singleton approach as well, so it is instantiated only one during engine execution and can be accessed via singleton pointer like others engine classes explained above see [section 4.1.1.2](#) and see [figure 4.1](#).

4.2 Graphics module

This module is defined inside namespace **Flwre::Graphics**, like the **Core** module contains several sub-modules defined in another namespaces inside **Flwre::Graphics** main namespace, so maintains an organization. Is built on top the **Core** module, mainly defines all needed to manage

and organize the graphics data to send it to render, so offers an essential organization and management for the graphic data. All this, firstly is carried out by the CPU while the graphic data remain in the computer main memory, then this data is send to the graphic card memory to render it.

At the same time, it propose mechanisms to be able to make more complex managements of the graphic data in the future outside this project, see the managers or basic scene graph components. However, for the moment it makes an essential management of the data as argued in the following sections.

The essential concept must be understood as the initial pillars of the graphic data management for later to be rendered, since advanced management techniques, such as the scene graph or geometric discard techniques in function if an object is visible or not, will not be implemented, leaving them for the future to simplify the scenario, nevertheless this will penalize the efficiency.

All of functionalities implemented in this layer are decoupled of the specific graphic API used in the rendering process, hence this module or layer is platform-independent, to carried out this, an interface is defined in this module and that is implemented in the underlying layer with a specific graphic API, see see [figure 3.1](#) engine architecture diagram.

Along the project, within this layer will be implemented different functionalities to get the target of this project explained above. The following sub-modules depicted in the following table, shows the main engine architecture defined in this module, these have the following definitions.

<i>Submodule</i>		<i>Short description</i>
<i>Resources</i>		It defines a common resources that is mainly buffers: vertex and index buffers and the vertex structure known inside the engine like vertex declaration.
<i>Utility</i>	<i>Managers</i>	It defines managers to handle the creation and destruction of resources, this acts like a factory resources.
	<i>Importers</i>	Initially holds an importer for 3D models with wavefront OBJ file format, that will be supported with triangulate faces only.
<i>Window</i>		A classes set to defines the windowed application and input events handling in the graphic level, implemented with a determinate multi-platform abstraction library at the same time defines the common mechanism needed to manage rendering objects within a window context.
<i>SceneGraph</i>		Initially defines a linear method to access the scene objects and implements the basic components to build a complete scene graph implementation in the future see section 4.2.4 , at the same time it defines the rendering handling of objects to feed renderer interface.
<i>Renderer</i>		It defines the rendering interface to draw.
<i>Effects</i>	<i>Shaders</i>	It provides the definition of the necessary classes to apply effects to the renderable objects.
	<i>LocalEffects</i>	It defines different local effect types supported by the engine.

DataTypes

Encapsulate different data types in a class definition like the color and transformations initially.

4.2.1 Resources

This sub-module is defined inside the name space **Flwre::Graphics::Resources**, and defines a data structures necessary to render objects and apply textures on them, to achieve this can be find the classes to store the vertices definition and the buffers definition. The buffers are memory allocated space that represents a fully typed data collection grouped into elements, that can be used to store a wide variety of data that can be read by the graphics card. This data collection is typed and organized in different ways.

Exists different buffer objects types, these are encapsulated in the engine with the buffer interface definition and initially the buffer types supported by the engine are vertex and index buffers since these are basic to render objects in a modern graphics scenario. The following sections show how it has been implemented.

At the same time this sub-module maintains the interface definition to support texture mapping, the engine initially only support 2D texture mapping.

4.2.1.1 Vertex Declaration

To rendering objects is necessary setting up several data related with the renderable objects, mainly a renderable object is compound by geometrical data this data is represented by many vertices, each vertex contains several data attributes to represent different information that is used when the object is processed in the render pipeline, during the rendering process is necessary that these data are allocated in a determinate memory space with a determinate structure inside the graphic card, this memory spaces are defined as buffers.

The vertex declaration class defines the vertex structure for each mesh created in the engine, the vertex structure holds the vertex attributes necessary to a certain mesh, this inside the engine is known as vertex format.

To create and destroy vertex formats, in conjunction with vertex and index buffers is defined an access by singleton pointer to the buffer managers base. The buffer manager base is part of the interface defined in **Graphics** module. See [figure 4.23](#) below and [section 4.2.2.1.1](#) buffer managers.

On the other hand, holds a binding with the buffers created for a determinate object, the engine maintain a container set with all the buffers created any time during engine runtime, see [section 4.2.2.1.1](#) Buffer managers, however is necessary maintain a relationship of which buffers belong to a certain object.

Also holds the vertex attributes structure, see vertex format and vertex element in the next sections, jointly how the data is represented inside the buffers, since the data inside the buffers can be stored in different manners, for example in interleaved manner or without interleaving, see [section 4.2.1.4](#) buffers. Finally defines the primitive type, used for how the vertex stream should be interpreted by the graphic card, and maintains the number of vertices that a given mesh has, together with the number of indexes defined in case the mesh is indexed.

The [figure 4.22](#) below shows the vertex declaration class in a conceptual way, in conjunction with the explained in the next sections, the [figure 4.23](#) shows an UML diagram of the [figure 4.22](#).

4.2.1.2 Vertex Format

A vertex format contains a vertex elements list, this group of elements describes a determinate vertex format, so vertex format defines the data structure for a vertex, at the same time defines which type of data will be inside the vertex buffers, hence represents how many attributes defined has a vertex.

A simple example of this list, could be the following vertex format:

Vertex Format							
Vertex Element			Vertex Element			Vertex Element	
X	Y	Z	R	G	B	S	T

Figure 4.21 shown a simple vertex format example

In this case the vertex format example, in the [figure 4.21](#) defines a format with three elements, the first element is position data, the second element is color data and the third are the texture coordinates. Each of these elements are defined in a vertex element as explained in the next section.

4.2.1.3 Vertex Element

Inside the engine the vertex attribute concept is represented with a vertex element, the meshes has many vertices each of these vertices are formed with one or several vertex attributes combination. Attributes defines different information related with vertex like position, color, texture coordinate, normal data, tangent data, bi-tangent data and so on. see [figure 4.21](#) for a simple vertex format example with three vertex elements.

However, vertex element stores others related parameters on how the graphic card must read the data, along the elements explained above when this data are stored inside a vertex buffer.

First defines the vertex semantics which indicates the attribute type of element, The [figure 4.23](#) shows the initially defined vertex element semantics in the following enum definition `Flwre::Graphics::Resources::VertexElementSemantic`, along the project other semantics will be defined as needed to cover all engine development.

Jointly to the vertex semantics, the vertex element defines the vertex element type as the number of components per each vertex element in the [figure 4.21](#) example two elements has three components **X**, **Y**, **Z** and **R**, **G**, **B** and the last element has two components **S**, **T**. then its semantics are **POSITION**, **COLOR** and **TEXTURE COORDINATE**.

The [figure 4.23](#) shows the initially defined vertex element type, in the following enum definition `Flwre::Graphics::Resources::VertexElementType`.

The vertex element type, is used to calculate the strides inside the vertex buffers as will be seen later, to achieve this inside each vertex element stores the offset value for each element. The offset variable can contains different values depending if the type of vertex elements will be stored in the vertex buffers in interleaved manner or not. if the vertex elements are stored in not interleaved way, the offset value contains zero all the time, otherwise the offset value stores the position in bytes

where the element is positioned in the elements intercalation. see **section 4.2.1.4.1** vertex buffers for more deeply explanation.

The following **figure 4.22** depicts the explained above in conceptual manner.

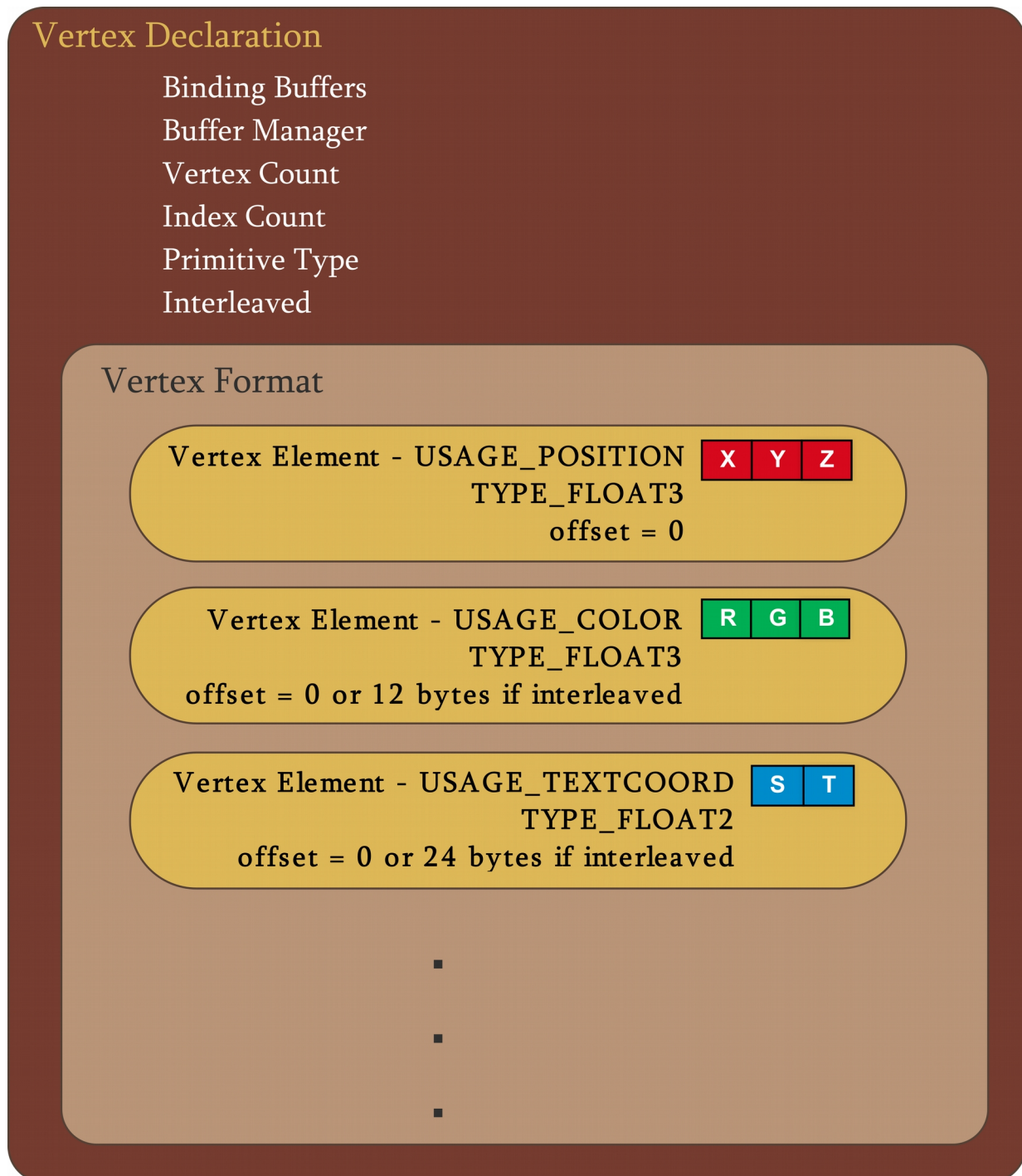


Figure 4.22 depicts vertex declaration class in conceptual manner

The following **figure 4.23** depicts the explained above with an UML diagram.

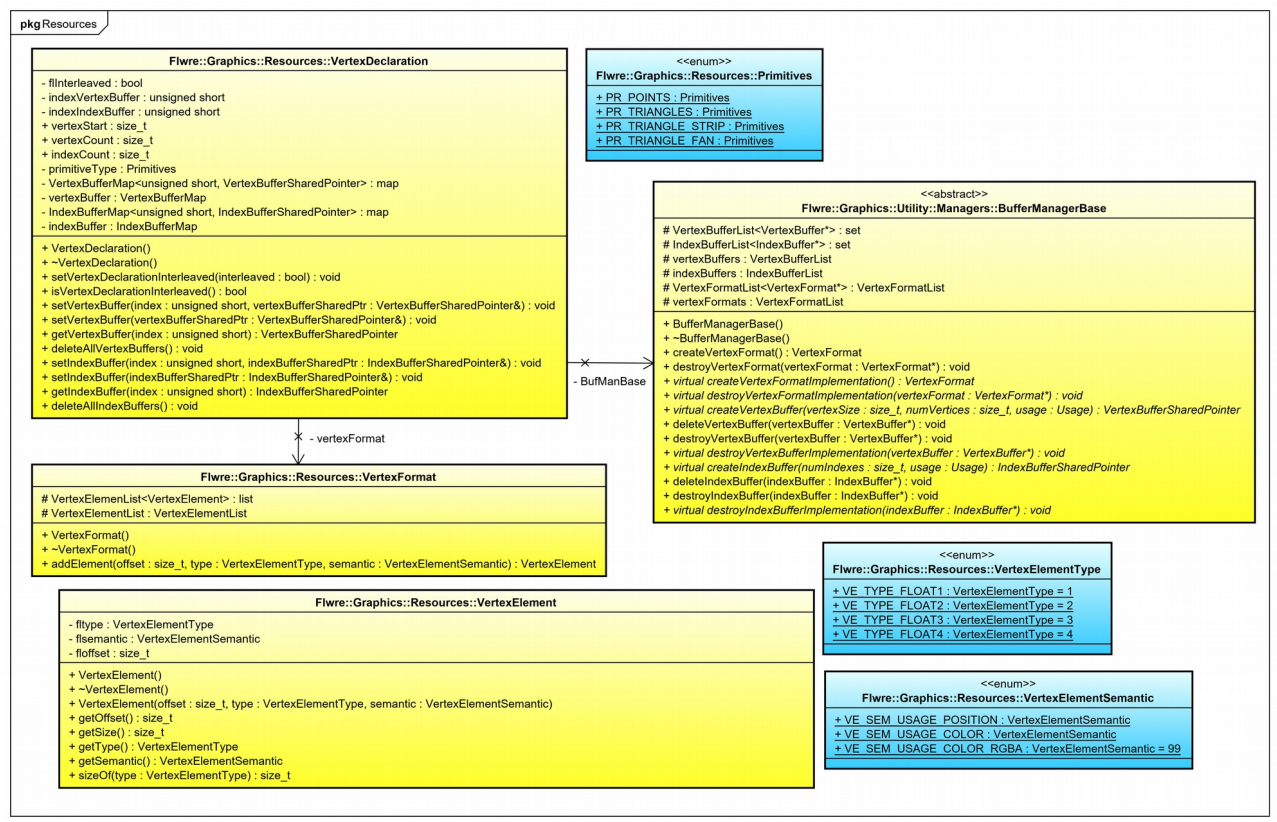


Figure 4.23 shown the vertex declaration class and involved classes in an UML diagram

4.2.1.4 Buffers

The buffers defines a region of physical memory storage used to store data in temporarily manner before it treated. The engine defines an interface called buffer as part of interface definition of this **Graphics** module, this interface encapsulate the buffer types supported by the engine, see **figure 4.24** below.

In this context, the buffers defines a memory space allocated in the graphic card to store initially two type of data streams, vertex data such as defined in above sections and index data explained below section, in the **figure 4.24** can be seen how the buffer interface is specialized in a vertex buffer and an index buffer to store this types of data streams. In the next sections will be explain this buffer types supported by the engine.

Initially the interface defines a virtual method to write data to the graphic card buffers, this data is passed as an array pointer from the computer main memory along with the array size in bytes, at the same time, defines two virtual methods for bind and unbind the buffers inside the graphic card, when a buffer or several buffers are binded the contained data stream is processed by the render pipeline to render the objects, the rendering process can be made by one or several render passes, being unbinded at the end of the render passes, see **section 4.2.6.1.2** render pass. The way to implement the data writing to the buffers or how to bind and unbind it, depends on the graphic API used, so the vertex buffer class and index buffer class are specialized in **GraphicsOGL3** layer, where this behaviours are implemented in this case with OpenGL 3.3, nevertheless this is not shown in the **figure 4.24**.

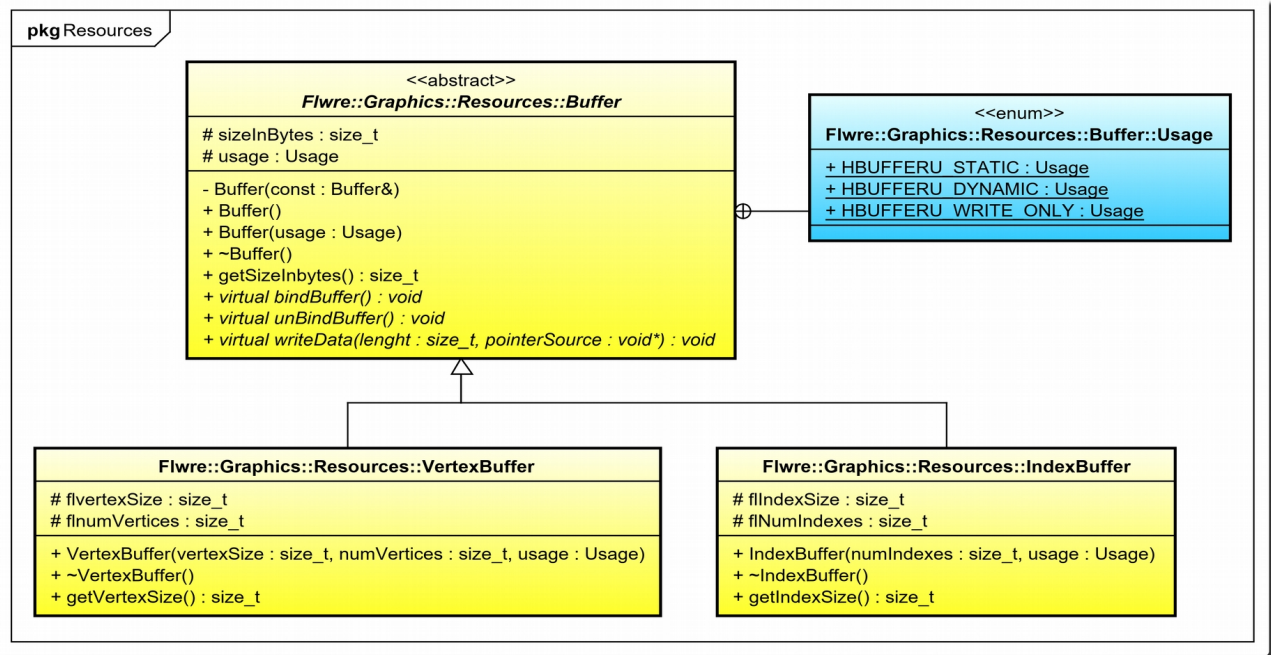


Figure 4.24 depicts the buffer interface definition with the corresponding buffers specialization in graphics layer

The enum **Flwre::Graphics::Resources::Buffer::Usage** definition, enumerate different manners to use a determinate buffer, this is indicated at the buffer creation moment:

HBUFFERU_STATIC or **HBUFFERU_WRITE_ONLY** defines a buffer where the data is static all the time, so no modifications in the contents buffer are made after the first writing, hence this buffer will read many times during the rendering process.

HBUFFERU_DYNAMIC defines a buffer where the data is dynamic, so the contents will be modified repeatedly and used many times in the rendering process.

4.2.1.4.1 Vertex Buffers

A vertex buffer contains vertex data used to define the objects geometrical data or its meshes, a determinate mesh can be build by a lot of vertex of the order of thousands or more if the object is a little complex, vertex buffers can be seen like a sequential vertices collection where each of them can be includes position coordinates, color data, texture coordinate data, normal data and so on, see **section 4.2.1.2** vertex format.

The vertex buffers can store the vertex elements in different manners, normally a renderable object can have one or several buffers assigned, depending if the data structure is defined in interleaved way or is not defined in interleaved way. This concept has been explained above and is now more widely exposed.

The interleaved manner is more efficient, because all vertex elements collection are stored in a unique buffer, by contrast the not interleaved manner, uses a buffer for each collection of vertex elements, so, if a vertex format has three vertex elements **POSITION**, **COLOR** and **TEXTURE_CORDS** for instance, then with a not interleaved buffer, would be created three buffers.

Some not interleaved vertex buffer examples are for instance, one that only contains position data. It can be visualized like the following illustration.

Vertex buffer not interleaved

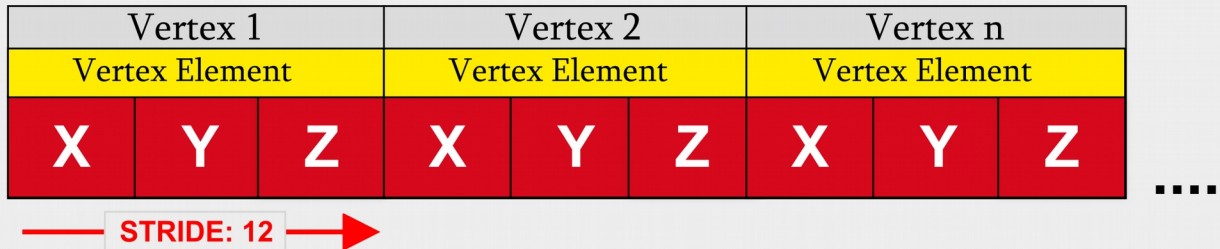


Figure 4.25 shown not interleaved buffer with position data, the stride is 12 bytes

Other examples of this, can be not interleaved buffers with color data or texture coordinates data for example. It can be visualized like the following illustration:

Vertex buffers not interleaved

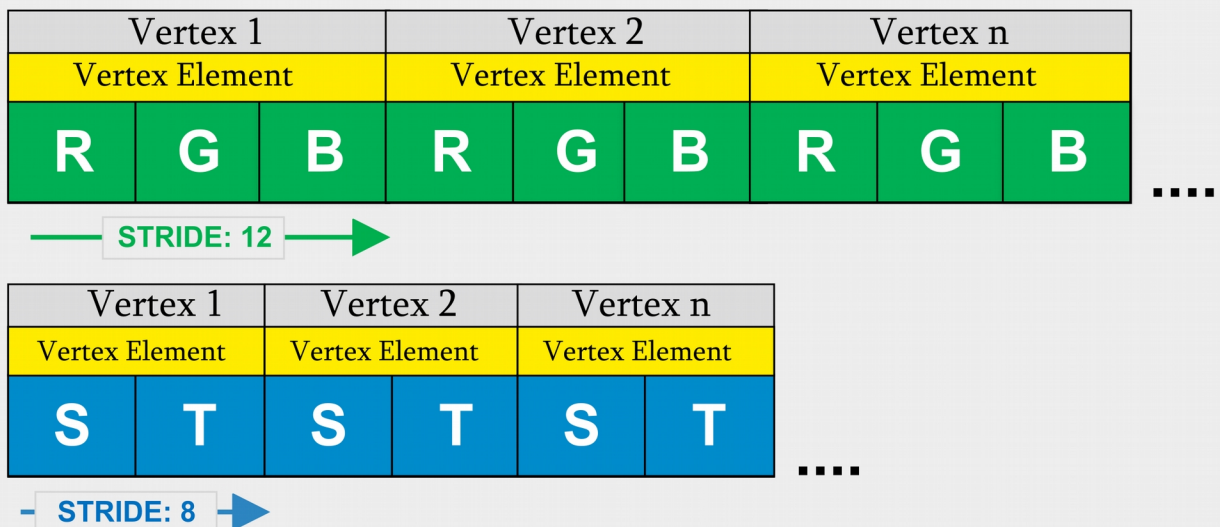


Figure 4.26 shown not interleaved buffers with color data and another with texture coordinates data, the stride is 12 and 8 bytes respectively

To read a vertex buffer the graphic card needs to know, how the data is distributed in the buffer or buffers so the additional parameters are needed, some this parameters as the offset has been defined in [section 4.2.1.3](#) vertex element and [figure 4.22](#). These parameters are explained below more deeply.

The stride means the space in bytes to next vertex and the offset is the displacement in bytes to the next element in a vertex when buffer is interleaved, otherwise offset is equal to zero all the

time. In an interleaved buffer the stride contains the sum of all offsets. See example below and [figure 4.27](#) and [figures 4.22, 4.25, 4.26](#) as well.

POSITION has 3 floats elements * 4 bytes = 12 bytes
COLOR has 3 floats elements * 4 bytes = 12 bytes
TEXTURE_COORDS has 2 floats * 4 bytes = 8 bytes
 32 bytes total.

If a vertex declaration has three elements as **POSITION**, **COLOR** and **TEXTURE_COORDS**, then in an interleaved buffer the stride would be 32 bytes, since each component in a vertex element is a float value and each float has 4 bytes size.

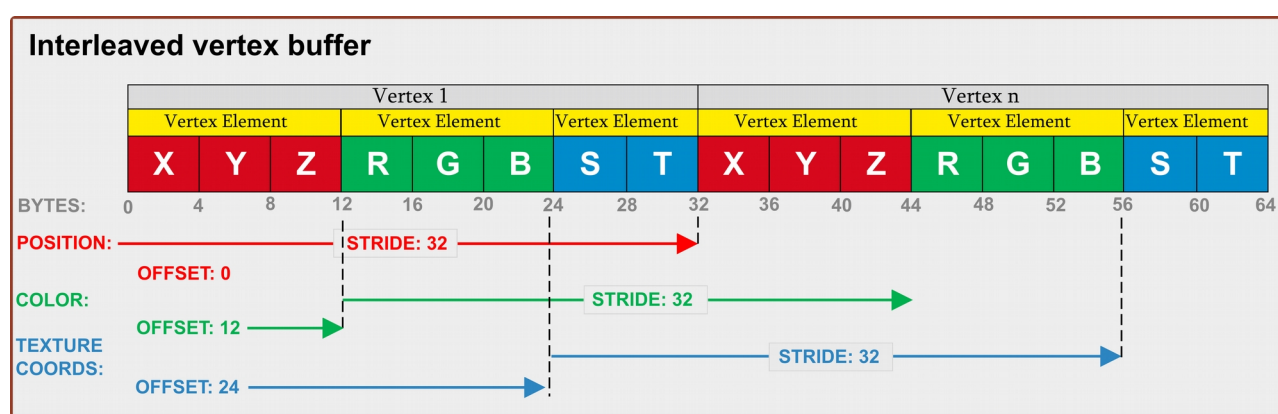


Figure 4.27 depicts an interleaved buffer example, the stride here is the displacement to next vertex element.

All data definition explained above is used in the **GraphicsOGL3** layer to create vertex buffers with a determinate distribution data in the graphic card with OpenGL 3.3, thus the card knows how must be interpret the data stored in the vertex buffers when the objects are rendered.

Nevertheless, all this defined in this layer could be take and implemented with another graphic API, defining another underlying layer as **GraphicsVK** for instance. see [figure 3.1](#).

To see how this is implemented in **Graphics** layer see the **create()** method in the Mesh class.

4.2.1.4.2 Index Buffers

Index buffers contains integer offsets into vertex buffers and are used to render primitives more efficiently, to get this, are reordered the vertex data and reuse existing data for multiple vertices to avoiding data repetition. In other words, provides a mechanism to reference a vertex multiple times in a single draw call, hence instead of just iterating through a list of vertices from beginning to end in the vertex buffers, this mechanism can choose which vertex to draw using a series of index values.

Mainly the indices stored in the index buffers, reduce the number of vertices required to form a mesh geometry, but this mechanism also serves another potential purposes, in a scenario where is wanted changed the color of vertex indexed mechanism allows to change a single color value, otherwise to change the color value in a non-indexed array buffer is should be changed a lot color elements of different vertices.

The indices helps to improve the performance, since in a modern graphics processing approach, normally when the data is processed in the render pipeline, exist a vertex cache before the vertex

shader stage, then to storage the vertex elements comes from a vertex buffer and other cache space before to store the results of the vertex processing, this scenario allows improving the performance, since if a mesh geometry uses indices rather than duplicates each vertex is therefore more likely to be in either the pre or post vertex cache, and so will be processed faster.

In conclusion, indices help to improve the performance globally because is a mechanism that space saver and time saver, allowing rendering faster. Index buffers are memory allocated in the graphic card and store a list of unsigned shorts integers that tell to the graphic card, which vertex in the vertex buffer is currently binded and must be draw.

4.2.1.5 Textures 2D

To improve visual realism the engine support texture mapping to apply images to the meshes, initially the engine has a basic system to load textures, hence only supports 2D decompressed textures. Although in this case practically all the work is done by the graphic API, is needed configure some parameters to achieve the desired graphical results. To prepare the textures to be send to the graphic card, has defined an abstract interface to load and set up the textures to finally will be created on the graphic card. The texture class is abstract, due to the interface definition is implemented within underlying layer **GraphicsOGL3** with OpenGL 3.3 in this case. The [figure 4.28](#) shows the interface defined within the engine.

A texture is an image that firstly is to be needed load it from any source to the main computer memory, before send it to the graphics card memory.

It is important to highlight, that the modern graphics hardware support native texture compression, allowing textures to take up far less memory and at the same time allows to manage any dimension sizes textures, but for the moment this scenario is not implemented within the engine.

So textures are stored as bitmaps so the compressed file formats like PNG, JPEG among others must be decompressed before being send to the graphics card memory, in this scenario is important that the textures has dimensions that being powers of 2, for example 512 by 256 pixels is a good choice. Obviously this must be improved in the future to support compressed textures natively, jointly with the support to other textures types.

To load the images, the engine uses a third party library called **std_image**[\[stdImage\]](#). Once a texture is loaded into graphics memory, the texture coordinates comes into play, since through them is possible to map a texture to a determinate mesh, since is needed to tell for each vertex of the mesh which part of the texture belongs to, the textures coordinates are vector attributes, like the position or color, see [section 4.2.1.3](#) vertex element. In case the textures 2D, the texture coordinates are represented with a two-component vector. The texture coordinates are normalized and no matter how large the texture is, the texture coordinates always are between 0.0 to 1.0 range.

The following UML diagram, shows the interface defined to manage the textures, the interface has been defined with a fluent interface approach, see [figure 4.28](#) below.

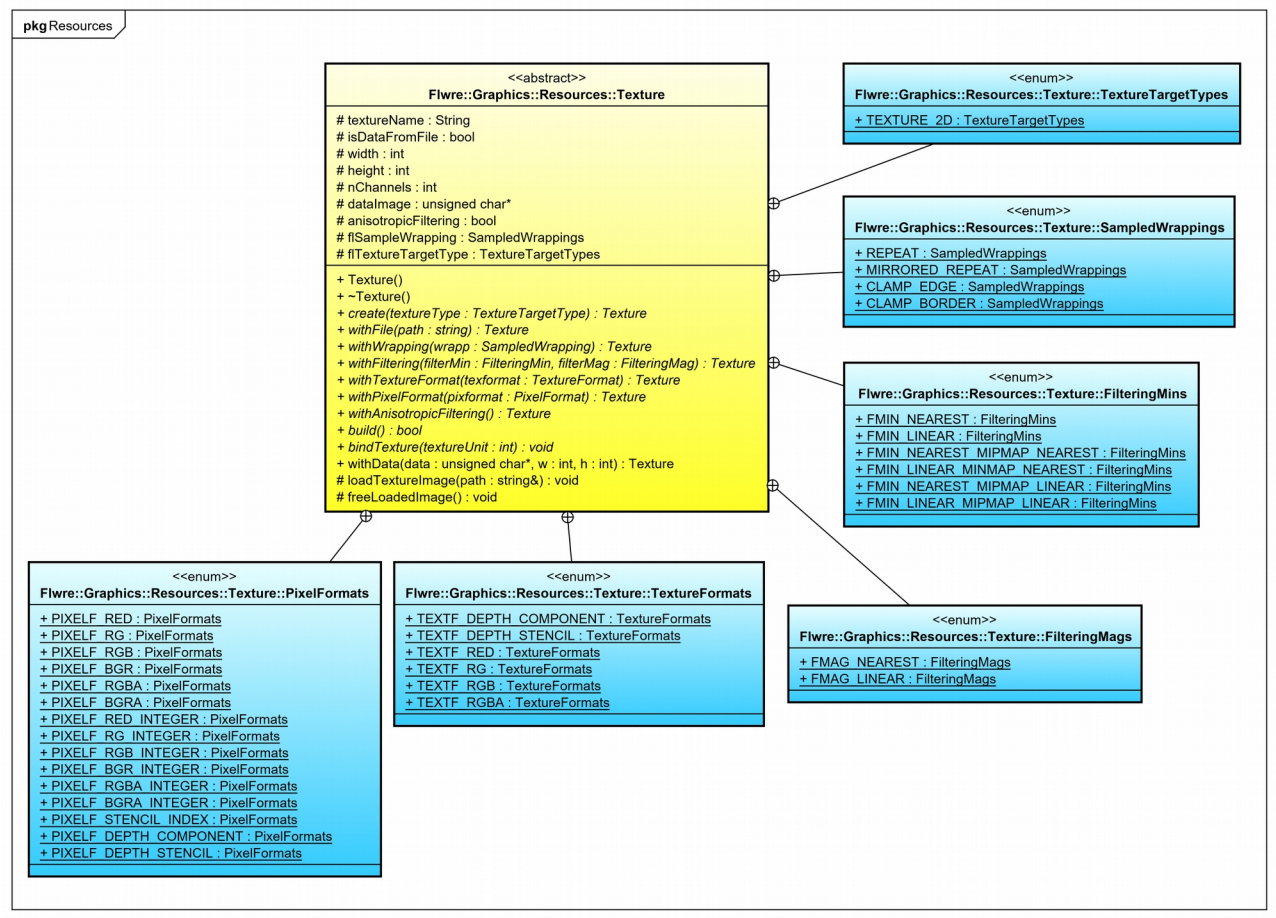


Figure 4.28 shows the interface defined to manage textures with the engine with an UML diagram.

The texture class interface defines different enumerations to manage different parameters related with the textures management. First defines the texture types supported, then defines how the textures are wrapping to the meshes, in conjunction with different filtering types, texture formats and pixel formats.

The methods defined in the texture class, allows applies this parameters when a texture is loaded and created, the create concept must seen as that the texture is created with different parameters to finally to be send to the graphics card memory with the build method.

The interface defined in the **figure 4.28** offers several methods to set up different parameters in a texture, before to be build, its own name defines its utility, these are depicts in the following table.

method	Description
create	For create a determinate texture type, only 2D textures initially.
withFile	For indicate the texture path and filename.
withData	For create a texture in procedural manner.
withWrapping	For assign a wrapping type.
withFiltering	For assign a filtering type.
withTextureFormat	For assign a format texture.
withPixelFormat	For assign pixel format type.
withAnisotropicFiltering	If this parameter is indicates anisotropic filtering is applied to the texture.

build

Send the data to the graphic card memory.

The table shows the methods defined to configure the textures with fluent interface approach. Finally the **bindTexture** method is used during the rendering process to activate the texture unit first, before binding the texture passing the texture unit number as a parameter.

The following code snippet, shows how a texture can be applied to a triangle, here the triangle is defined manually rather than to use the **ColoredTriangle** method defined in the **PrimitiveMeshShapes** class, see [figure 4.4](#).

```
1. Flwre::Graphics::Utility::Managers::TextureSharedPointer texture;
2.
3. (...)
4.
5. void HelloTexturedTriangle::CreateScene()
6. {
7.     texture = Flwre::Graphics::Utility::Managers::ResourceManager::getSingleton().createTexture();
8.     texture->create(Flwre::Graphics::Resources::Texture::TEXTURE_2D)
9.         .withFile("../Textures/fabric.jpg")
10.        .withWrapping(Flwre::Graphics::Resources::Texture::SampledWrapping::REPEAT)
11.        .withFiltering(Flwre::Graphics::Resources::Texture::FilteringMin::FMIN_LINEAR,
12.                      Flwre::Graphics::Resources::Texture::FilteringMag::FMAG_LINEAR)
13.        .build();
14.
15.     triangleMesh = new Flwre::Graphics::SceneGraph::PrimitiveMeshShapes();
16.     /*triangleMesh->ColoredTriangle(0.8f).create();*/
17.
18.     // *** Defining a textured triangle manually ***
19.     triangleMesh->setPrimitiveType(Flwre::Graphics::Resources::Primitives::PR_TRIANGLES);
20.     triangleMesh->setVertexElementSemantic(Flwre::Graphics::Resources::VertexElementSemantic::VE_SEM_USAGE_POSITION);
21.     triangleMesh->setVertexElementSemantic(Flwre::Graphics::Resources::VertexElementSemantic::VE_SEM_USAGE_TEXCOORD0);
22.
23.     triangleMesh->setInterleaved(false);
24.     triangleMesh->setNumVertices(3);
25.     triangleMesh->setVertexAttributes(2);
26.
27.     triangleMesh->setVertice(-0.7f, -0.7f, 0.0f);
28.     triangleMesh->setVertice(0.7f, -0.7f, 0.0f);
29.     triangleMesh->setVertice(0.0f, 0.7f, 0.0f);
30.     triangleMesh->setArrayDataPtr(Flwre::Graphics::Resources::VertexElementSemantic::VE_SEM_USAGE_POSITION);
31.
32.     triangleMesh->setTextureCoordsIndices(3);
33.     triangleMesh->setTextureCoord(0.0f, 0.0f);
34.     triangleMesh->setTextureCoord(1.0f, 0.0f);
35.     triangleMesh->setTextureCoord(0.7f, 1.0f);
36.     triangleMesh->setArrayDataPtr(Flwre::Graphics::Resources::VertexElementSemantic::VE_SEM_USAGE_TEXCOORD0);
37.
38.     triangleMesh->create();
39.
40.     texture2DEffect = new Flwre::Graphics::Effects::LocalEffects::Texture2DEffect();
41.     texture2DEffect->getRenderTechnique(0)->getRenderPass(0)->getDepthStateTest()->setDepthTesting(false);
42.     triangleMesh->setRenderableEffect(texture2DEffect->CreateRenderableEffect(texture));
43.     getSceneHandler()->getRenderableObjectSet()->insertRenderableObject(triangleMesh);
44.
45. }
```

Code snippet 1.1 shows how a texture is created and applied, lines 7 to 13 and lines 40 to 42 respectively.

The **figure 4.29** shows the apply result a fabric texture to the triangle.

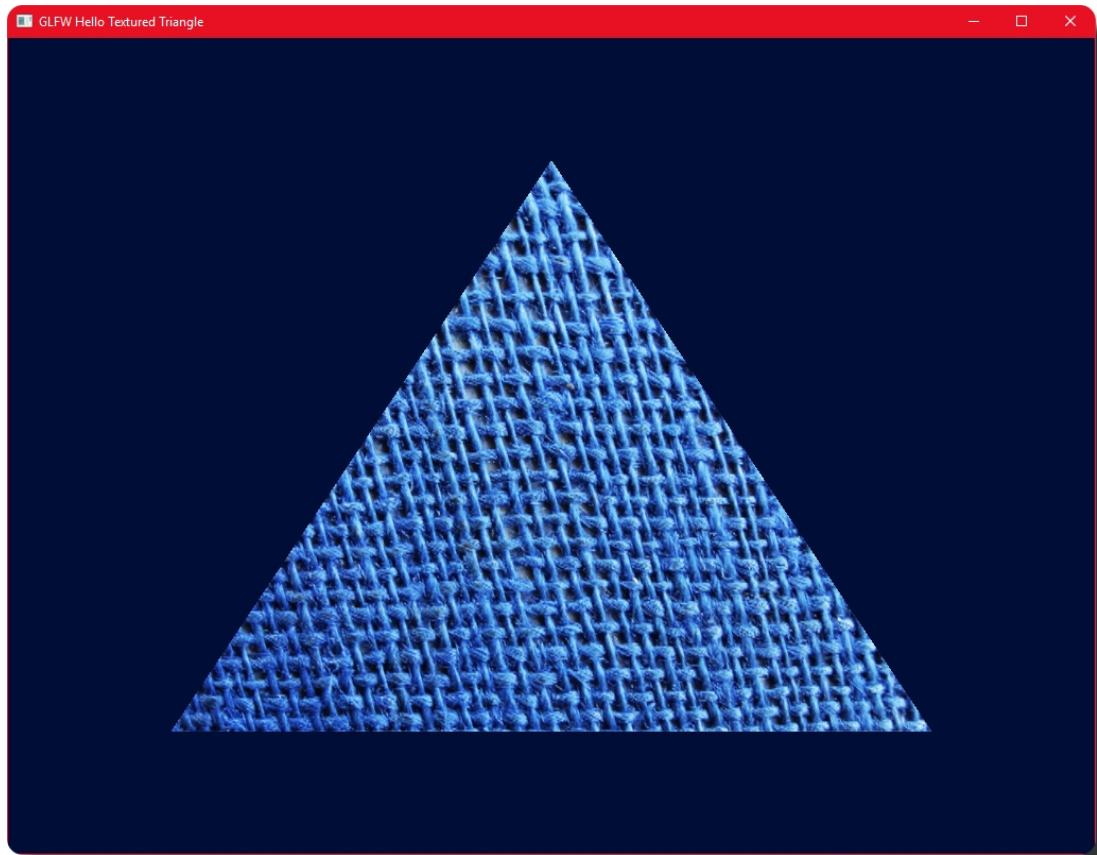


Figure 4.29 depicts a textured triangle with fabric.

4.2.2 Utility

The name space **Flwre::Graphics::Utility** holds different submodules, on the one hand contains the managers explained in the next section, and at the same time, holds the importers, this give the ability to the engine to import 3D models from different geometry definition file formats. Initially a OBJ 3D file format parser has been implemented only, this is collected in the **section 4.2.2.1** OBJ importer.

4.2.2.1 Managers

Within the name space **Flwre::Graphics::Utility::Managers** has been implemented different kinds of managers. Mainly, the managers defines a rendering interface part to create and manage different resources in the engine, with a factory implementation fashion. All managers within the engine are instantiated in singleton manner, because in all rendering process a unique manager instance is needed for each resource or resource groups, see **sections 3.4.3** singleton.

The managers are initialized when the render system is initialized within the engine when starts, and are destroyed when the render system is finished.

In general, the managers defines an interface that offers the suited mechanism to manage different engine resources, mainly maintains a control of the resources created and destroyed during the rendering process, and they offers the needed tools to create and destroy different resources,

defining an interface for it. At the same time, the managers offers the mechanism to encapsulate the destruction of all resources managed when the engine ends, hence the programmer that use the API that offered by the engine, must not be worry about destroying them manually.

4.2.2.1.1 Buffer Managers

Buffer managers holds the mechanism to manage the buffers explained in the **section 4.2.1.4** buffers, more concretely allows create and destroy vertex formats, vertex buffers and index buffers, at the same time maintains a container with all buffers and formats created during the engine execution, this scenario should allow eliminate a determinate object of the graphic card memory but keep it in the computer main memory, being possible to send again to the graphic card, if would be needed at runtime, obviously also expose the mechanisms to eliminate completely a determinate object if this would be no longer necessary.

This is the mainly utility that managers harbor, nevertheless initially only are used to create and destroy resources at the beginning and at the end engine execution.

4.2.2.1.2 Texture Manager

The texture manager is created within the resources manager, basically defines an interface to create textures, from the applications that are implemented over the engine, see line 7 code snippet 1.1 above, to achieve this an abstract **createTexture** method is defined in the managers base class within **Graphics** module, and this is implemented in the underlying layer **GraphicsOGL3** module. At the same time, maintains a STL set container to keep pointers to textures created, for later they can be destroyed when the rendering process finishes, with the methods offered for this function, though when a texture is created, this is wrapped within a smart pointer, so when the engine ends all textures are destroyed automatically without the need to delete a concrete pointers.

The manager offers a couple of methods called **deleteTexture** and **destroyTexture** passing a texture pointer as a parameter. The first method delete it only from the container and the second delete it completely from the container and the graphic card.

At the same time, the manager holds methods to find and retrieve textures by name, this is useful once the ".mtl" files are read within the engine by the OBJ parser, for later find and retrieve a determinate texture for create it.

4.2.2.1.3 Shader Manager

The shader manager is exactly the same concept that the texture manager, applied to the shaders an its parameters, except the shaders can not be retrieved by name. The manager only is used to create and destroy shaders and to maintain a container to store the created shaders.

4.2.2.1.4 Render Effect Manager

Unlike the other managers, the render effect manager is entirely implemented in the **Graphics** layer, so is completely platform independent, at the same time, the objective of this manager is a little bit different than others explained above, since it does not create any resource. Basically this managers has the responsibility to guarantee an orderly control of the storage of render effects and its instances when are attached to the objects once created, for later, during the rendering process, to manage the attached effects to the objects, due to can be changed at runtime. Finally with the

manager, is possible to get an orderly destruction of this resources when the rendering process ends.

The destruction of all stored effect resources are centralized in this case in the same manager when the own manager is destroyed when engine is shut down, however the manager offers methods to delete a determinate effect or instance of it, passing the pointer of a determinate instance effect as a parameter, being possible delete ones at runtime.

4.2.2.1.5 Material Manager

The material manager is implemented within the resource manager jointly the texture manager, unlike the texture manager, the material manager not defines an interface due to is implemented entirely inside the **Graphics** layer, since the materials are totally graphic API decoupled. Initially the main idea is to offer a mechanism to maintain a materials inside an unordered map, for later find and retrieve a determinate material by name, within the map container. To achieve this, the manager offers methods to insert and retrieve the materials by name, and the corresponding methods to remove a particular material.

The reason to use an unordered map, is basically in both cases in the texture and the material managers, the need to keep count of the materials without that a sort is required, where the access to a determinate element being sufficient.

4.2.2.1.6 Mesh Manager

The mesh manager has the same approach to the material manager, and is graphic API decoupled as well.

4.2.2.2 Model importers

The engine needs to export complex geometric models made with other 3D design software like Blender, Maya, 3D Studio Max, SoftImage among others, this software store the 3D models designed with different 3D model formats

The engine needs to import complex geometric models made with other 3D design applications, like Blender, Maya, 3D Studio Max, SoftImage among others, these programs store the 3D models with different 3D model storage formats, being possible move the designed models to another software or render engines, to render it. Exist some third-party libraries that can be integrated within another applications to import these 3D model file in different formats, see[[assimp](#)], however in this project initially an own OBJ importer has been developed and integrated within the engine. The following section explains it.

4.2.2.2.1 Wavefront OBJ importer

The OBJ files stores geometry definition and other properties, this format initially was developed by Wavefront Technologies in the early 80s. The format is open and along the years has been adopted by other 3D graphics application vendors.

The engine implements an own OBJ importer, to load 3D models designed with any 3D modelling software as been explained above, the OBJ parser is based on this other loader, see [[OBJ-Loader](#)], but has been rewritten entirely, adapting it to the engine and improving it as well. However initially the parser only able to parse triangulated meshes, so, the mesh stored in the OBJ file, must be triangulated with any 3D modelling software like Blender, if the model does not come

triangulated in the OBJ file previously, at the same time, all geometry is loaded from the OBJ loader to the graphic card without to be indexed.

In the following lines, a briefly OBJ file format explanation are made, for more information see [\[OBJPaulBourke\]](#) [\[OBJWikipedia\]](#). A simple example has been taken to use for explain the main OBJ file structure, obviously a real file with a complex model stored will be more dense and provably with more parameters, that are not explained here, since only an essential explanation are made, such as it has been said above.

An OBJ file looks more or less like this.

```
1. # Blender v2.79 (sub 0) OBJ File: 'Cube.blend'
2. # www.blender.org
3. mtllib Cube.mtl
4. o Cube
5. v 1.008434 -0.123181 -0.994194
6. v 1.008434 -0.123181 1.005806
7. v -0.991566 -0.123181 1.005806
8. v -0.991566 -0.123181 -0.994194
9. v 1.008434 1.876819 -0.994194
10. v 1.008434 1.876819 1.005806
11. v -0.991567 1.876819 1.005806
12. v -0.991566 1.876819 -0.994194
13. vt -0.002556 0.000000
14. vt 0.997444 1.000000
15. vt -0.002556 1.000000
16. vt 0.997444 0.000000
17. vt -0.002556 1.000000
18. vt -0.002556 0.000000
19. vt 0.997444 0.000000
20. vt -0.002556 1.000000
21. vt 0.997444 0.000000
22. vt -0.002556 1.000000
23. vt -0.002556 0.000000
24. vt -0.002556 0.000000
25. vt 0.997444 1.000000
26. vt 0.997444 0.000000
27. vt -0.002556 1.000000
28. vt 0.997444 0.000000
29. vt 0.997444 1.000000
30. vt 0.997444 1.000000
31. vt 0.997444 0.000000
32. vt 0.997444 1.000000
33. vn 0.0000 -1.0000 -0.0000
34. vn 0.0000 1.0000 0.0000
35. vn 1.0000 -0.0000 0.0000
36. vn -0.0000 -0.0000 1.0000
37. vn -1.0000 -0.0000 -0.0000
38. vn 0.0000 0.0000 -1.0000
39. g Cube_Cube_Material
40. usemtl Material
41. s off
42. f 1/1/1 3/2/1 4/3/1
43. f 8/4/2 6/5/2 5/6/2
44. f 5/7/3 2/8/3 1/1/3
45. f 6/9/4 3/10/4 2/11/4
46. f 3/12/5 8/13/5 4/3/5
47. f 1/14/6 8/15/6 5/6/6
48. f 1/1/1 2/16/1 3/2/1
49. f 8/4/2 7/17/2 6/5/2
50. f 5/7/3 6/18/3 2/8/3
51. f 6/9/4 7/17/4 3/10/4
52. f 3/12/5 7/19/5 8/13/5
53. f 1/14/6 4/20/6 8/15/6
```

This OBJ file contains the cube definition, a cube has been chosen, due to that its simple geometry is suitable to explain the OBJ file structure, An OBJ file includes different type of data, the data type is represented at the beginning of each line. The main data types are related with the vertex data, and elements in this case.

From line 5 to 12 defines the geometric vertices (v), from line 13 to 32 defines the texture vertices (vt) and from line 33 to 38 are defined the vertex normals (vn), all data are related with the vertex data.

The lines between line 42 to 53 defines the object faces (f), the faces are object elements in a polygonal geometry. The faces use reference numbers to identify the vertex data belongs to. This reference numbers starts by 1, so this means that the first geometric vertex in the files is 1, the second is 2, and so on, then in the example, the first geometric vertex is hosted in the line 5. This can be applied with the all different vertex data such as the (vt) and (vn) as well, so the texture coordinate (vt) starts by 1, and the vertex normal (vn) starts by 1 as well, these are hosted in the lines 13 and 33 respectively.

The faces (f) may have a triplet of numbers that reference vertex data. These numbers are the reference numbers for a geometric vertex (v), a texture vertex (vt), and a vertex normal (vn). So each triplet of numbers references a group of vertices separated by slashes such as follows v/vt/vn. The line 42 in the example above shows a three-sided face element with its vertex data for each vertex. So the example can be read as follows.

1/1/1 refers to the first geometric vertex, line 5, to the first texture coordinate, line 13, and to the first vertex normal, line 33, this triplet that refers to one vertex in a face, then the triplet 3/2/1 refers to another vertex in the same face and so on.

It's worth pointing out, that the example above has texture coordinates and vertex normal defined, but is possible that a OBJ file format no has this vertex data defined, or has defined part of them only, so, other combinations will be possible in function of the vertex data defined in the OBJ file. Then will be possible to find the following scenario "f 1//1 3//1 4//1", where the texture coordinates are not informed, or the other possible scenario would be "f 1 3 4" where the texture coordinates nor the vertex normal are not informed. Finally this scenario "f 1/1/1 3/2/1 4//1" will be illegal.

The lines that starts with "#" are comments, see lines 1 and 2. the line 4 indicates the object name, a OBJ file can contains different objects in a unique file. The line 39 refers the group name for the elements that follow it, it is possible to have multiple groups on one line, then the data that follows belongs to all groups, however this does not supported by the OBJ importer implemented here. The line 41 refers to group number but this is another syntax that noy is supported by the OBJ importer.

The OBJ files has much more options, but not are supported with the OBJ importer implemented here, since in this project has been implemented a basic OBJ importer.

The following table shows the parameters supported by the OBJ parser implemented.

Keyboard	Description
#	Comment.
v	Geometric vertices.
vt	Texture Vertices.
vn	Vertex normals.
o	Object name.
g	Group name, create a new mesh.
f	Face, create a vertex.
mtllib	Material library, specifies the material library file for the material definitions.

usemtl

Material name, specifies the material name for the element following it.

The following code snippet shows how to load a OBJ file with the engine.

```
(...)
objImporter = new Flwre::Graphics::OBJImporter();
objLoader->loadOBJfile("./Models/obj/cube.obj").build();
object = Flwre::Graphics::Utility::Managers::MeshManager::getSingleton().getMeshPtrByName("cube");
object->getLocalTransform()->setTranslate(Flwre::Core::Platform::Vector3f(2.0f, 4.0f, 3.0f));
(...)
```

For the moment, the OBJ importer must be instantiated and destroyed from the application, however, the importer could be hosted within the engine as a singleton.

The line 40 in the OBJ file example above, specifies the material name for the element following it, in this case the material name is "Material". The line 3, specifies the material library file used, though the OBJ definition can support different material library files, here one library is supported only. In the following lines are explained the material library files "mtl" related with the OBJ files.

The OBJ format supports materials in a separate file with the ".mtl" extension, that means (Material Template Library), this format also was defined by Wavefront Technologies. These files defines the geometry surface shading properties, see materials [section 4.2.4.6](#), the materials can be defined in one or more files informed within the OBJ file, this use the syntax (mtllib), see line 3 in the example OBJ file. The material library files, contain one or more material definitions, each of which includes the color, textures, and reflection map of individual materials, these are applied to the geometry surfaces in ASCII format equal OBJ files. For more information about the MTL material format, see [\[MTLPaulBourke\]](#).

Although the standard is supported among different 3D modeling software, making it a useful format for interchange of materials and is widely used, is outdated, due to the standard does not support later computer graphics technologies, such as specular maps and parallax maps, though the opened standard nature, allows added new features with a custom MTL file generator. Related with the MTL files, the OBJ parser implemented within the engine, supports the essentials related with this material libraries.

The following table shows the parameters supported by the OBJ parser implemented related with the materials.

Keyboard	Description
Ka	Specifies the ambient reflectivity using RGB values.
Kd	Specifies the diffuse reflectivity using RGB values.
Ks	Statement specifies the specular reflectivity using RGB values.
Ke	Statement specifies the emmissive coeficient using RGB values, belongs to the extensions to MTL to support new techniques for realistic rendering. See [1] .
d or Tr	Specifies the dissolve factor for the current material.
Tf	Statement specifies the transmission filter using RGB values.
Ns	Specifies the specular exponent for the current material.
Ni	Specifies the optical density for the surface. This is also known as index of refraction.
illum	The "illum" statement specifies the illumination model to use in the

	<p>material. Illumination models are mathematical equations that represent various material lighting and shading effects.</p> <p>"illum_#" can be a number from 0 to 10. The illumination models are summarized below.</p> <p>Illumination Properties that are turned on in the</p> <table> <thead> <tr> <th>model</th><th>Property Editor</th></tr> </thead> <tbody> <tr> <td>0</td><td>Color on and Ambient off</td></tr> <tr> <td>1</td><td>Color on and Ambient on</td></tr> <tr> <td>2</td><td>Highlight on</td></tr> <tr> <td>3</td><td>Reflection on and Ray trace on</td></tr> <tr> <td>4</td><td>Transparency: Glass on</td></tr> <tr> <td></td><td>Reflection: Ray trace on</td></tr> <tr> <td>5</td><td>Reflection: Fresnel on and Ray trace on</td></tr> <tr> <td>6</td><td>Transparency: Refraction on</td></tr> <tr> <td></td><td>Reflection: Fresnel off and Ray trace on</td></tr> <tr> <td>7</td><td>Transparency: Refraction on</td></tr> <tr> <td></td><td>Reflection: Fresnel on and Ray trace on</td></tr> <tr> <td>8</td><td>Reflection on and Ray trace off</td></tr> <tr> <td>9</td><td>Transparency: Glass on</td></tr> <tr> <td></td><td>Reflection: Ray trace off</td></tr> <tr> <td>10</td><td>Casts shadows onto invisible surfaces</td></tr> </tbody> </table> <p>* The engine only store the value, but for the moment, it does nothing with the value.</p>	model	Property Editor	0	Color on and Ambient off	1	Color on and Ambient on	2	Highlight on	3	Reflection on and Ray trace on	4	Transparency: Glass on		Reflection: Ray trace on	5	Reflection: Fresnel on and Ray trace on	6	Transparency: Refraction on		Reflection: Fresnel off and Ray trace on	7	Transparency: Refraction on		Reflection: Fresnel on and Ray trace on	8	Reflection on and Ray trace off	9	Transparency: Glass on		Reflection: Ray trace off	10	Casts shadows onto invisible surfaces
model	Property Editor																																
0	Color on and Ambient off																																
1	Color on and Ambient on																																
2	Highlight on																																
3	Reflection on and Ray trace on																																
4	Transparency: Glass on																																
	Reflection: Ray trace on																																
5	Reflection: Fresnel on and Ray trace on																																
6	Transparency: Refraction on																																
	Reflection: Fresnel off and Ray trace on																																
7	Transparency: Refraction on																																
	Reflection: Fresnel on and Ray trace on																																
8	Reflection on and Ray trace off																																
9	Transparency: Glass on																																
	Reflection: Ray trace off																																
10	Casts shadows onto invisible surfaces																																
map_Ka	Specifies that a color texture file or a color procedural texture file is applied to the ambient reflectivity of the material.																																
map_Kd	Specifies that a color texture file or color procedural texture file is linked to the diffuse reflectivity of the material.																																
map_Ks	Specifies that a color texture file or color procedural texture file is linked to the specular reflectivity of the material.																																
map_Kn	Specifies a normal texture file																																
map_Ns	Specifies specular texture.																																
map_Displacement or disp	Specifies displacement map.																																
map_d	Specifies that a scalar texture file or scalar procedural texture file is linked to the dissolve of the material. During rendering, the map_d value is multiplied by the d value.																																
map_Bump or bump	Specifies bump map.																																

When the meshes are parsed, internally the meshes are created instantiating the meshes and are stored within a container for later to be retrieved by name and to be send to the graphics card, the materials related with OBJ files defined within the MTL files, are parsed and are treated in the same manner, all of this management are carried out by the managers explained in **section 4.2.2.1.5** material managers and **section 4.2.2.1.6** mesh managers.

The following screenshots, shows some OBJ models imported and rendered by the engine.

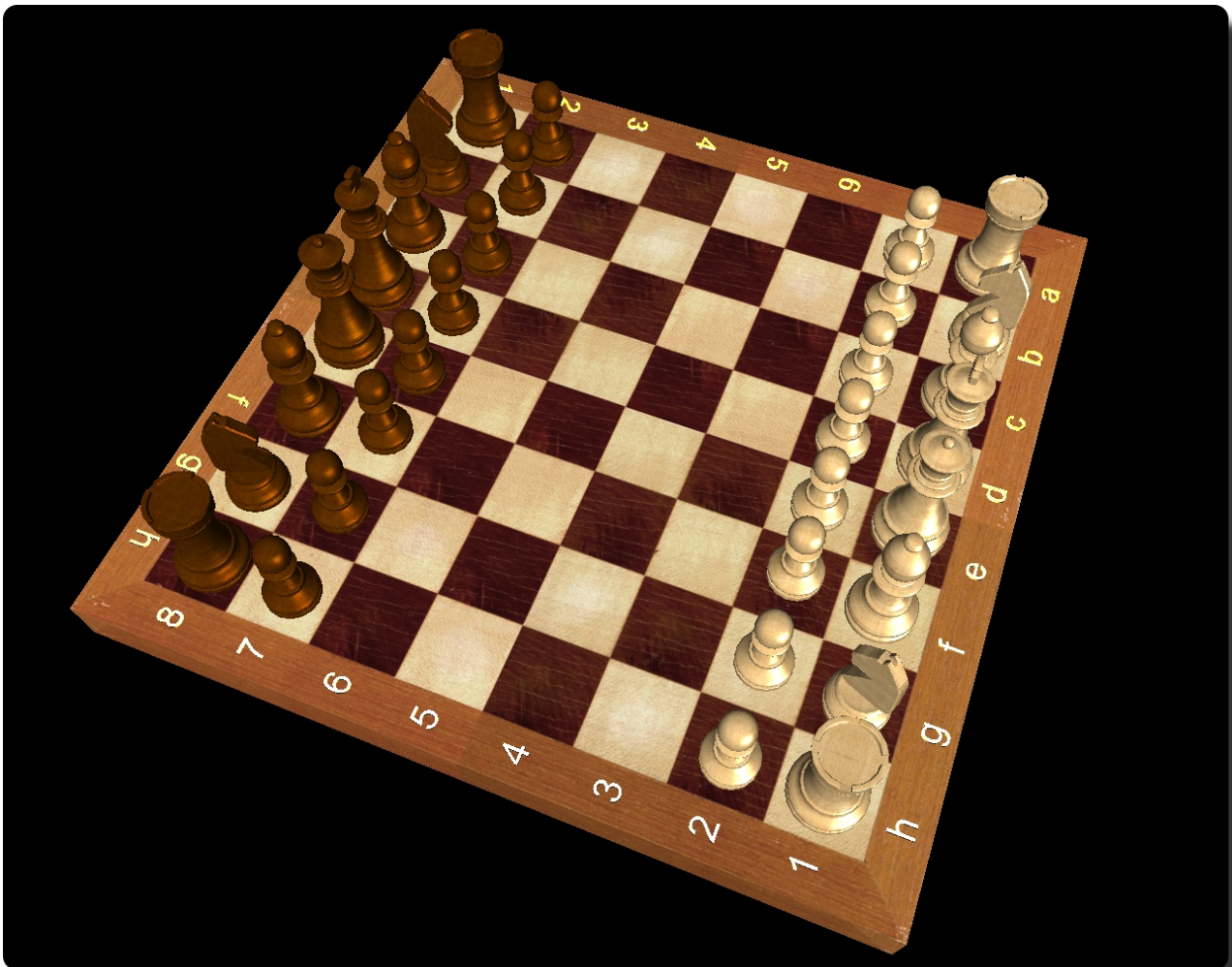


Figure 4.29a. Classical chess, with Phong reflection and one directional light, rendered by Flow render engine.

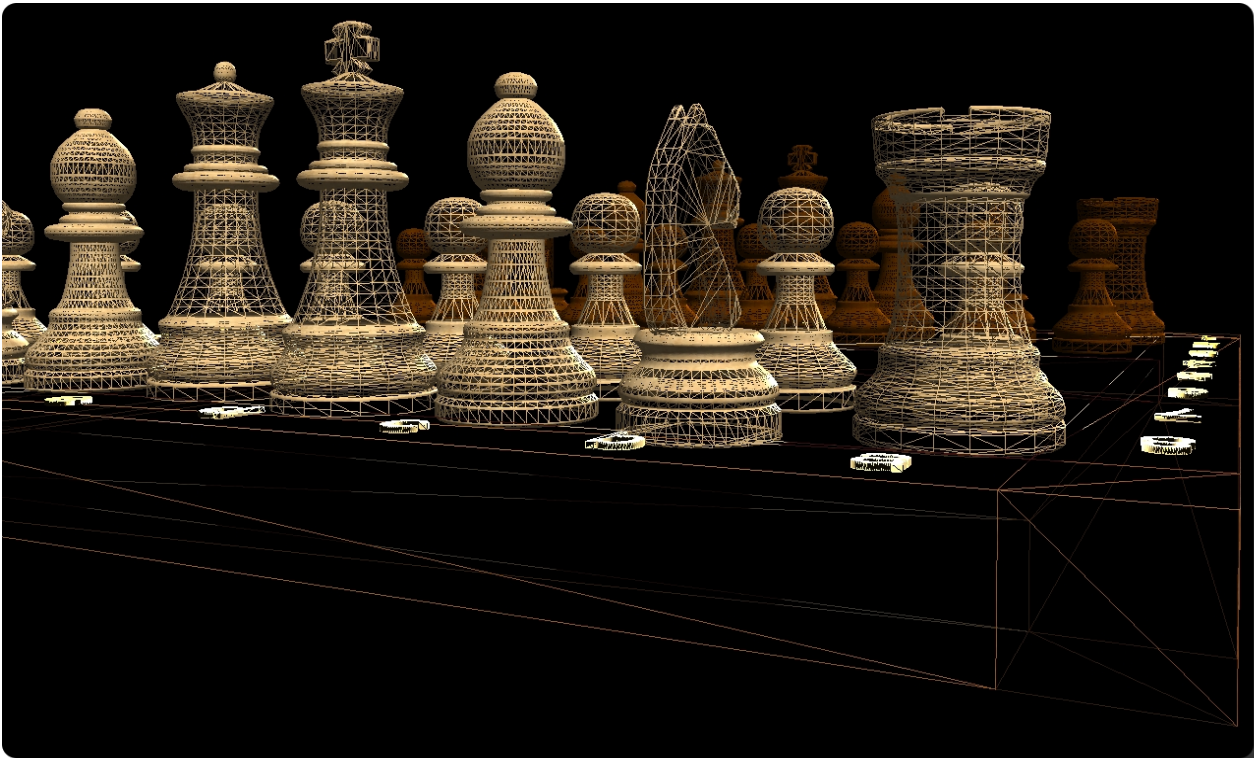


Figure 4.29b. Classical chess wireframe, rendered by Flow render engine.



Figure 4.29c. Suzanne model low, medium and high poly, with diffuse shading, rendered by Flow render engine.

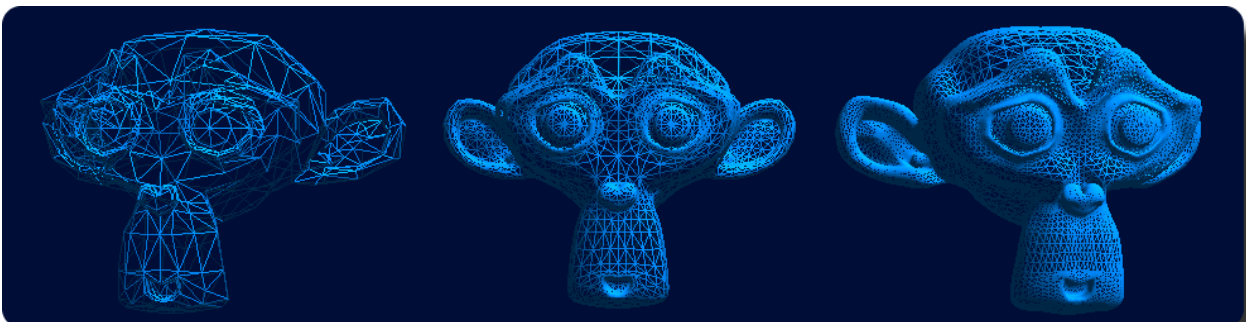


Figure 4.29d. Suzanne model low, medium and high poly, wireframe, rendered by Flow render engine.

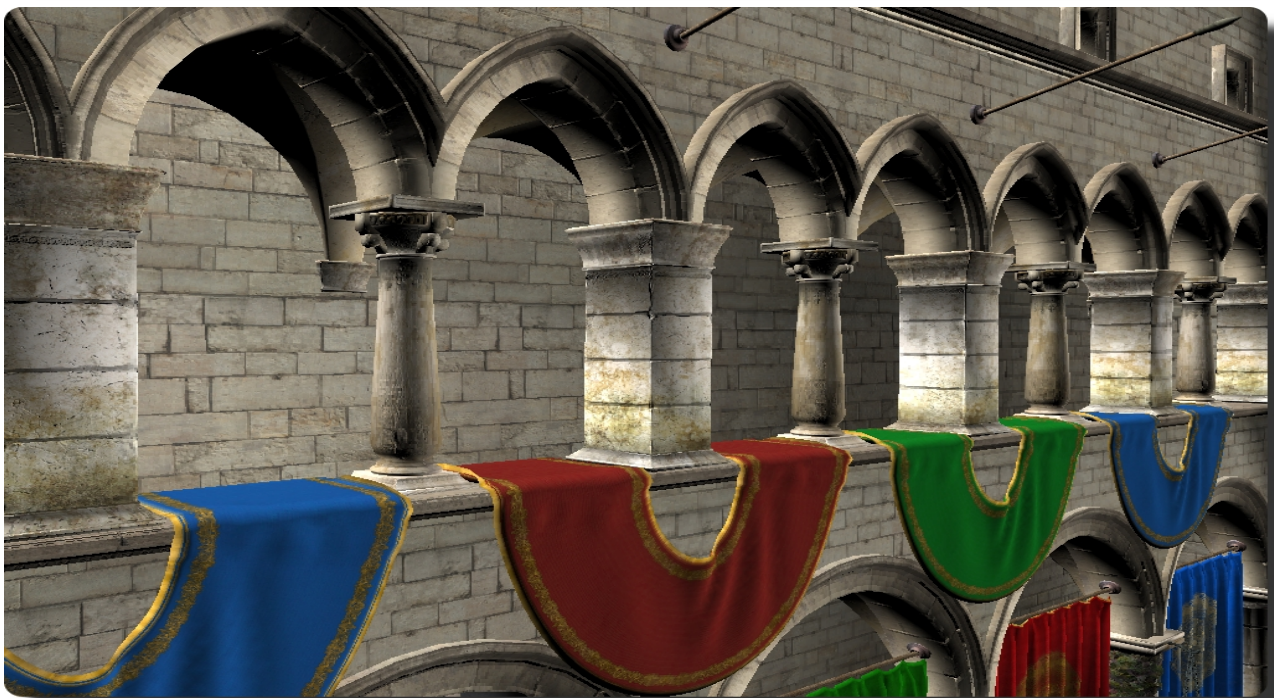
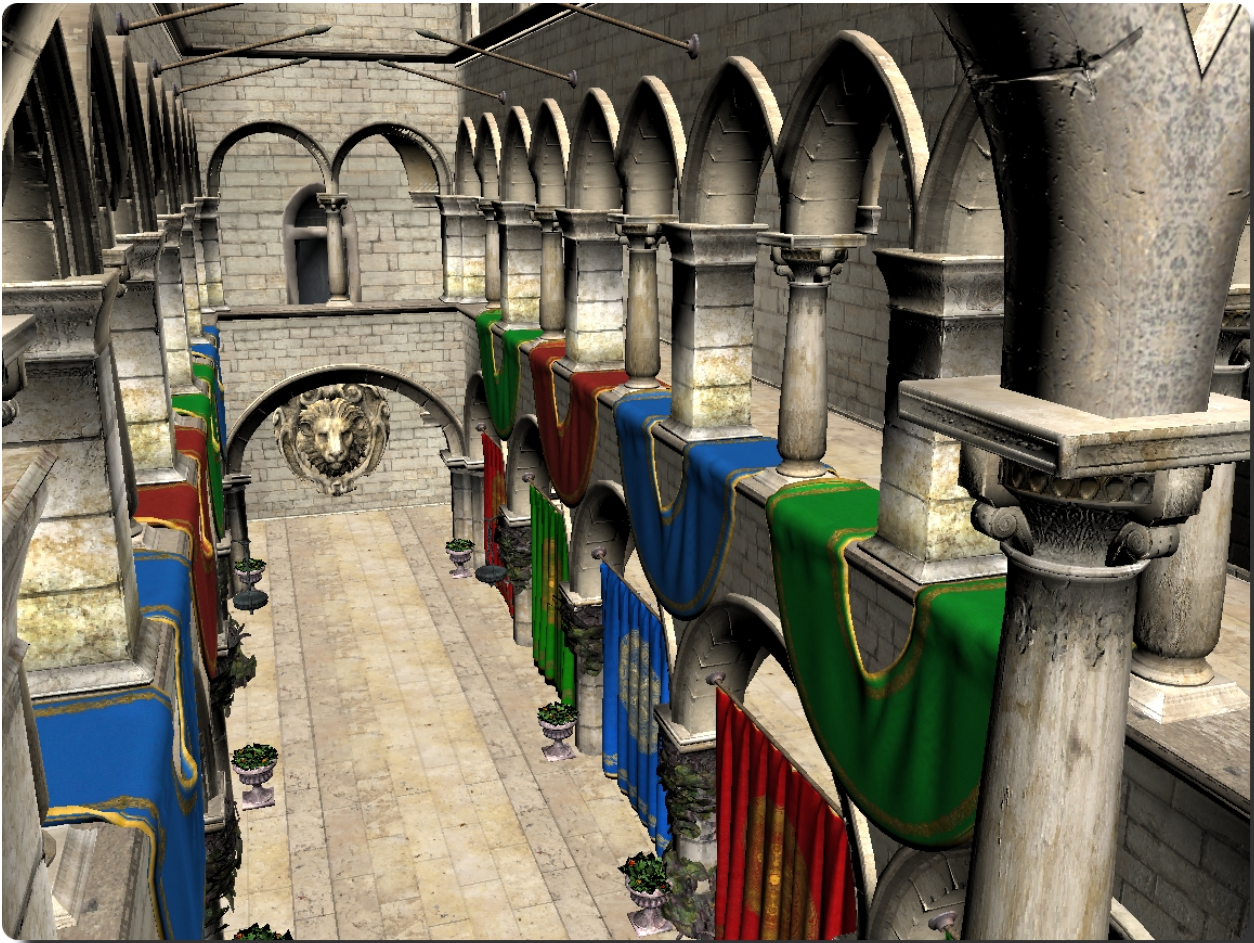


Figure 4.29e, f. Sponza Crytek model, with diffuse shading, normal maps and two directional lights, rendered by Flow render engine.

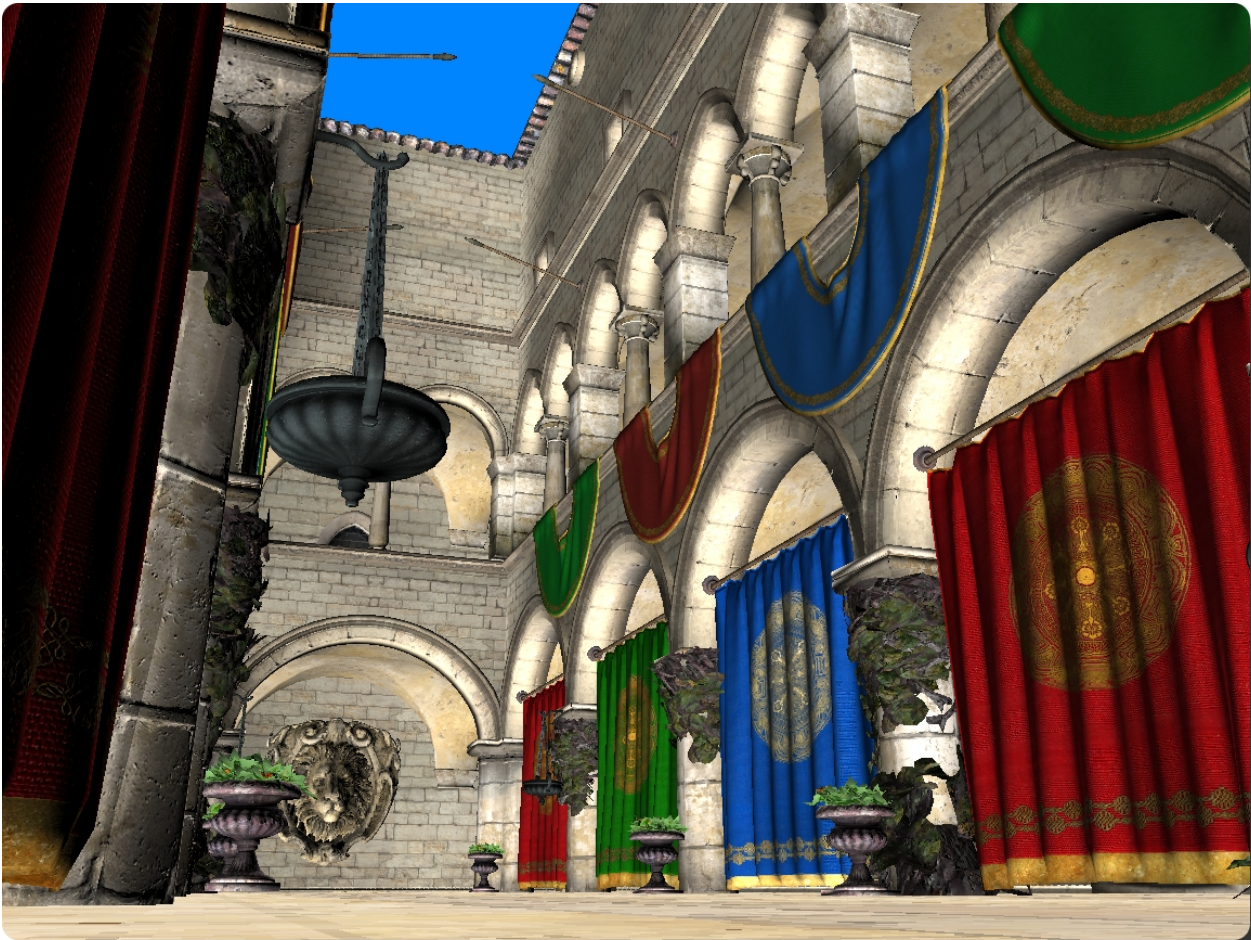


Figure 4.29g, h. Sponza Crytek model, with diffuse shading, normal maps and two directional lights, rendered by Flow render engine.

4.2.3 Windowed application

The window sub-module is defined inside **Flwre::Graphics::Window** name space, inside this name space is implemented the application context specialization explained in **section 4.1.1.1** application context, the specialization defines the windowed type application, at the same time, implements the features to manage the keyboard and mouse handle events, and holds a pointer to the camera implementation, the camera is created when the window application context is initialized, then this pointer is passed to the renderer, and here is used to access the camera methods to move it with the keys and the mouse, during the rendering process, see **section 4.2.4.4** camera.

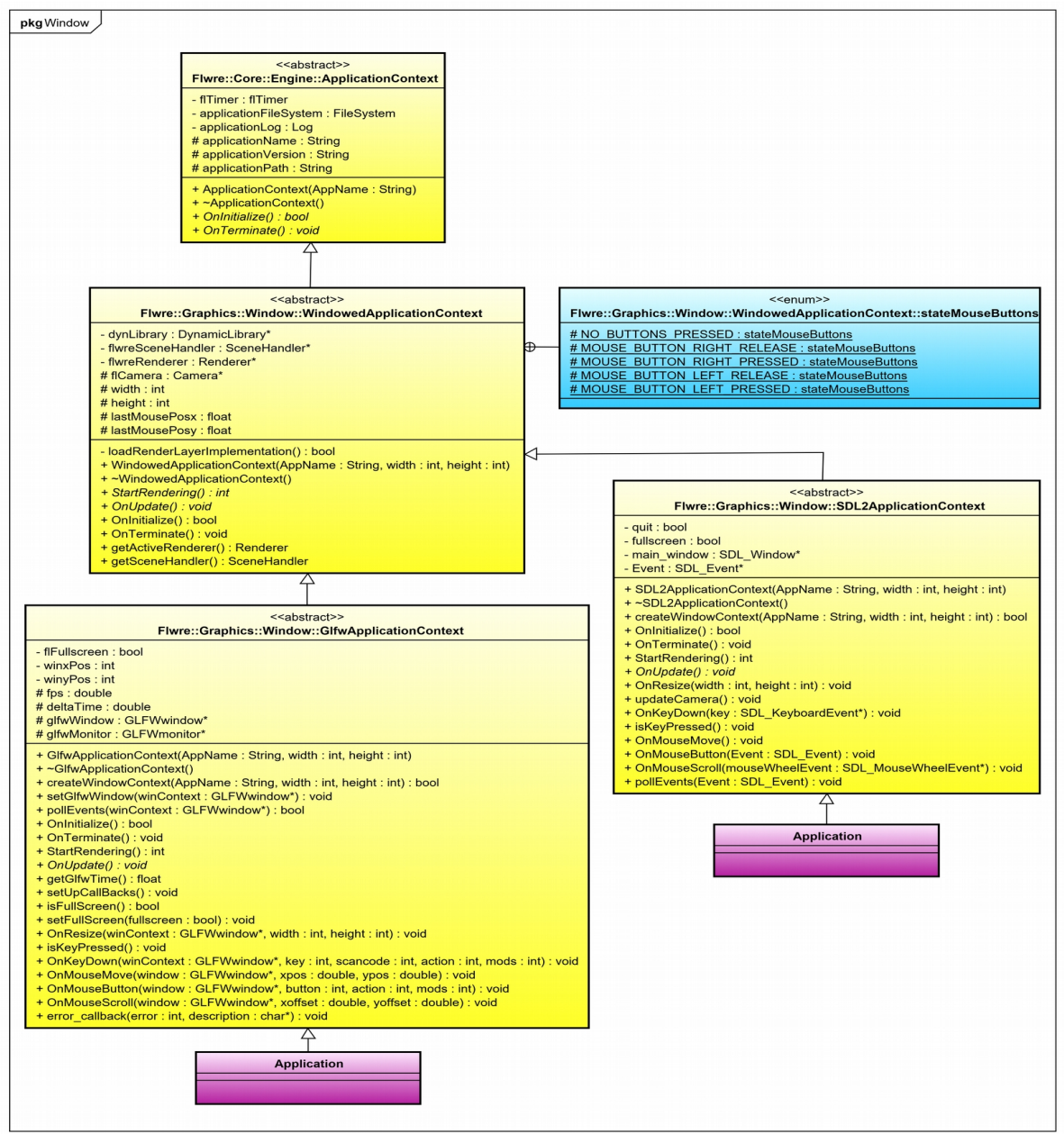


Figure 4.30 depicts windowed application classes hierarchy with two different tool kits.

This design architecture must allow, for example, to create applications with a console approach simply derived from the base application context class and implemented the corresponding behaviour. But for the moment, only implements a mechanism for working with windowed applications.

The **WindowedApplicationContext** and **GlfwApplicationContext** classes implement a windowed system.

Both classes implement the abstract class application context defined in **Core** layer. The reason to implement the abstract interface application context between two classes, is to separate the common window parameters together with the entry points to the renderer system, and the lineal render queue implemented see **section 4.2.5** renderer, of the concrete implementation of a windowed system and the handling events system, with a multi-platform abstraction layer like **GLFW** see **GlfwApplicationContext** class.

The reason to use a library like **GLFW**, is that OpenGL by itself does not provide any mechanisms for creating the necessary rendering context or managing windows, manage user input events even manage timing, although in this project timing is treated separately see **section 4.1.4.4** Timer.

At the same time, the window creation is platform dependent; this means that is handled by a platform-specific APIs and each operating system has the concrete APIs to get this, then this scenario needs a common abstraction level, to simplify the window creation with the engine, since the engine has a cross-platform approach, otherwise different implementations to create windows and event handling for each platform would be needed, this is where **GLFW** comes into play again.

To hide this complexity, different libraries that act as a multi-platform abstraction layer, to hide the windows creation complexity in different platforms, one of them has been seen above.

But nevertheless, the cross-platform engine design in conjunction with the multi-graphic API layer implementation approach, get necessary a design structure allows the integration of different libraries to be able to create windows, rendering context, and events handling, since maybe in a determinate platform does not exist a determinate abstraction library to create windows or simply maybe would be want implemented with a concrete operating system API directly, this approach can view as a tool kit that support multi context windowed applications in compilation time.

For this reason in **figure 4.30** shows the **SDL2ApplicationContext** class, since the explained above allows a window system creation changeable at compilation time between different tool kits to create windows, rendering contexts and manage events from the engine. The de-facto standard toolkit in the engine is the **GLFW** library implemented in **GlfwApplicationContext** class, the **figure 4.30** shows how could be implement another tool kit with **SDL2** library, and following this approach would be possible implements others tool kits, such as freeglut library for instance or even will be implemented directly with a platform-specific API as WGL context in windows or GLX context in Linux and so on.

The tool kit mechanism, offer a major portability to the engine, between different platforms and different implementations of graphic APIs, since in future scenarios, an example of this, would be that **GLFW** library supports OpenGL, OpenGL ES and Vulkan rendering context only, then if wanted to implement a rendering layer with Direct3D, **GLFW** does not support it. Other libraries like **SDL2** supports Direct3D, so this mechanism offer a major flexibility to integrated a determinate library to manage the windows creation, rendering contexts and manage events in a given scenario.

Finally, the decision which library to compile at compilation time is defined in **SysWindowApplicationContext** header via compilation directives.

More in detail, the **WindowedApplicationContext** implements the following:

A private method to load the render layer interface implementation when the engine is initialized, thus maintain a pointer to render system loaded and has a method to retrieve this pointer, see **loadRenderLayerImplementation()** method.

Stores a pointer and method to retrieve it, of the lineal render queue see **SceneHandler**, this class stores another pointer to **RenderableObjectSet** class, which handles renderable objects array see **section 4.2.4.3** renderable object set, and scene handler, this is the minimal approach implemented to manage the objects to be rendered in a scene in lineal manner.

At the same time defines two pure virtual methods **StartRendering()** that is implemented in each window tool kit, and **OnUpdate()** method which is declared in each application and implements the behaviour that is executed in each frame from the applications, see **section 5** applications over the engine.

4.2.3.1 Handling keyboard and mouse events

In the same way that the window creation system, the keyboard and mouse handling is platform dependent where the engine runs, so each operating system handles it differently and consequently is programmed different manner as well over each platform, due to the engine has a cross-platform approach, is necessary a common abstraction level to manage the keyboard and the mouse in a common manner, then the responsibility of this work inside the engine is targeted to the toolkit, that in this case, is the standard toolkit of the engine **GLFW**.

In common manner approach, the responsibility to manage the keyboard and the mouse will be targeted to the tool kits implemented inside the engine in this case here is **GLFW** as been said, due to that is de-facto standard toolkit of the engine, but they could be others like **SDL2**, **freelut** and so on, if would be implemented within the engine, with the offered mechanism to allow change the tool kits at compilation time such as explained in **section 4.2.3** windowed application.

GLFW API offers an abstraction to manage input events, here it is not explained how this API works to manage events, for information about how **GLFW** works see [\[GLFW\]](#) and the engine code implementation, concretely the **GlfwApplicationContext** files.

The **GlfwApplicationContext** class is expanded with the following methods to manage the input events see **figure 4.30**, the **OnKeyDown** method is involved with the keys press management outside the rendering loop in callback way. The callbacks are defined in the method **setUpCallbacks** and are set up when the window context is created when the engine starts.

The methods **OnMouseMove**, **OnMouseButton** and **OnMouseScroll** such as indicate its method names, they manage all related with mouse events.

Related with the key events management, has been defined a method called **isKeyPressed** as well, this method does not work in callback way and is called per each frame inside the render loop instead, to manage the camera behaviours, walk, strafe and lift pressing the keys up, left, right and down arrows keys for walk and strafe, plus Q and Z keys for lift see **section 4.2.4.4** camera, at the same time, that are updated the camera translations in each frame, the reason to implement it in

this way, is to get soft camera movements, when the keys are pressed, since via callbacks does not get camera soft movements.

Finally defines a couple of methods, called **isFullScreen** and **setFullScreen** to set up the engine in full screen mode pressing F11 key, to change between full screen mode and windowed mode. The variables **winxPos** and **winyPos** maintain the window position coordinates on the display screen.

Related with the applications programmed over the engine, the method **OnKeyDown** is overridden on the applications if is necessary, to manage the desired keys on the application. For the moment, this in the applications, must will be programmed directly with the mechanism that uses the API used in the toolkit, however, obviously this should be encapsulated within the engine and offering a common mechanism to the programmer that use the engine API, to program the desired keys with a common interface definition, independent the toolkit that is being used, but for the moment this is not implemented.

4.2.4 Scene Graph

Section 2.3 has been explained the utility of the scene graph, this section explains its utility with a basic example to illustrate it, in a more practical way.

The real-time concept in this context, must be understood like that images are generated online and the rate of generation is fast enough for the image sequence to be looked like animation that simulates something, a real-time rendering engine must be able to render scenes with different levels of complexity, without affecting its performance as the scenes becomes more complex, so, how can a real-time rendering engine be scalable?

To answer this, some mechanism is necessary to keep in track of objects make up the scene for to get high level performance and efficiency when rendering complex scenes, where exists a lot of objects in movement, since a real-time rendering engine must be able to render scenes with different levels of complexity, without affecting its performance as the scenes becomes more complex, logically this will have a limit that will be determined by the amount of objects to render, jointly with other factors such as the power of the underlying hardware used.

The mechanism to achieve this objective is known as scene graph, however, a scene graph is combined with another techniques, like culling while keeping track of objects to increase the rendering efficiency, since a scene graph manage nodes that encapsulate different information types involved in the rendering process.

Getting further into matter, scene graphs consist of a number of scene nodes organized in a hierarchical way via tree-like structure with acyclic graphs usually, each node has a parent node, and a number of child nodes. Everything in a scene can be part of one large scene graph, so it's common for a real-time rendering engine with scene graph implemented to have just a single root scene node, containing everything in the current level as children.

A scene graph nodes can contain graphical information of objects including their transformations, however, doesn't necessarily have to contain graphical information such as a mesh or transformations, since they may be purely transitional, that is, nodes that group together and translate a number of children, but they don't render anything themselves, or perhaps the node will contain only state information, such as a group of sub-nodes that are rendered using a specific shader, or some other specific rendering option set.

Related with culling mentioned above, while traversing the scene graph for rendering, bounding volume of the objects contained in the visited nodes, will be tested against the viewing frustum culling of the camera, and if the object that is contained inside the node is outside the camera frustum, then the node and all its children will be culled and not traversed until are inside the frustum again. This algorithm known as hierarchical view-frustum culling[ViSurAlgo], the overall rendering performance can be increased. Nevertheless, the scene graph provides only a nodes logical organization so spatial data structures are needed could provide better culling results.

To depict the explained above, in conjunction with the transformations in relation with the scene graph, an example to illustrate this can be a car modelled in 3D, a car in 3D has separate meshes for the chassis, wheels and so on, in this scenario, might think in how can we keep track of where the wheels are in relation to the chassis?, this is a simple scenario, but as the scenes get more complicated and starts adding in lots of objects, keeping track of the relative positions of everything starts becoming difficult. Here, is where the scene graph is useful too, due to tree-like structure where each node has a parent node, and a number of child nodes, then the car example could be represented by the following simple scene graph figure.

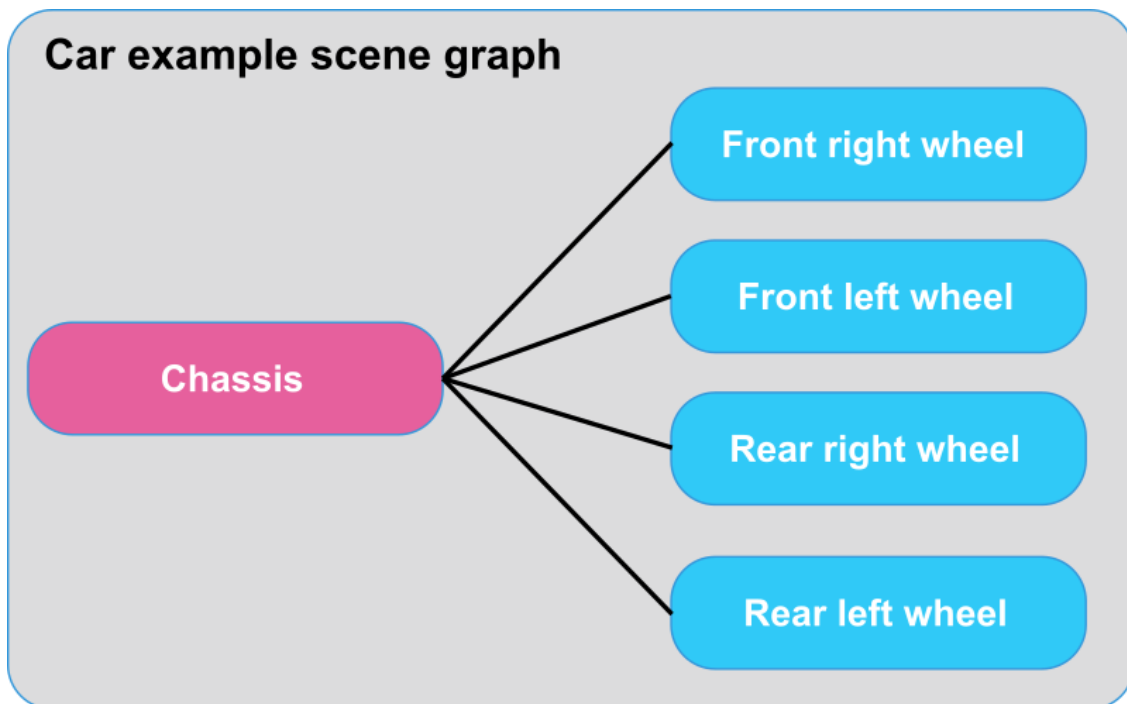


Figure 4.31 depicts the car example in a simple scene graph

Each node in the scene graph, contains information relating to its graphical representation, hence the chassis scene node could maintain a pointer to a car body mesh, while the four wheel nodes contain pointers to a single wheel mesh model, since there is no need to load the same mesh multiple times.

At the same time, the nodes contains pointers to local transformation matrix transforms and the local transformations holds information related with the positions, orientation including scaling of the objects in the scene, the father-child relationship approach between nodes, in the scene graph, brings the possibility of manage all transformation information in cascade way, hence is possible to keep track of where the wheels are in relation to the chassis, to do an entirely car reorientation or being able to size up the car becoming to a big car, if set the chassis scale to 10.0, for example,

then will automatically increase the size of its wheels due to the parent/child relationship of their transformations.

A real-time rendering engine, is compound with many components and all of them are interrelated, some of these engine components, are meshes, transformations, lights, materials, effects and so on. This components at the end, must be well tuned to feed the rendering interface, achieving to render objects with a determinate achievement like the illumination, effects or whatever.

A scene graph combined with spatial data structures, culling techniques, bounding volumes objects, transformations, and the shaders for the effects as has been seen, is an important piece in a real-time rendering engine, to get a good managing and good rendering performance in scalable manner, however, this project is focused on build the essential pillars of a rendering engine, this pillars are different necessary components interrelated of them that compound the real-time rendering engine base and they must be well tuned, before implementing more complex techniques, for this reason, has been preferred first focus on the basic components explained in this project, before implementing spatial data structures, bounding volumes objects and the scene graph management among others more complex techniques.

For this reason, the implementation of these techniques are outside of this project, temporarily has been implemented a linear method with a vector to storage renderable objects instead, this should look like a render queue that in the future will be replaced with a complete scene graph implementation, although in a complex scenes this method becomes less useful for obvious reasons explained above.

Although the fact of implementing it, in a linear way, provide the opportunity to experience how it behaves and to compare it in the future with scene graph behaviour.

In the next sections, is explained how the simple linear method to storage renderable objects is implemented, in conjunction with the mechanism to feed the rendering interface in each frame, at the same time is discussed the classes that compound the renderable objects.

4.2.4.1 Renderable objects and meshes

The geometric data that represents each object is encapsulated in three classes, see UML diagram in the [figure 4.32](#). The main classes for definition a model or object are the renderable object class and Mesh class, the primitives mesh shapes class is a mesh specialization and hides the construction details for some both 2D and 3D geometric primitives objects. See [section 4.1.2.7](#) 2D and 3D geometric primitives.

All data related with model geometrical definition, is encapsulated by the class mesh, although a model can be split in different sub-meshes, initially to simplify the scenario a monolithic mesh definition has been made, in a unique class provisionally, but usually, the complex 3D models can has different sub meshes so in the future this schema would be define a sub-mesh class too, for increase the definition granularity to be closer to reality.

The data contained in mesh class is stored in the computer main memory, and holds entirely information that defines all data related with geometrical objects, then this data is send to the graphic card memory via the create method defined in the mesh class, with the fluent interface approach is possible to do, the following command achieving more readable code.

```
circle->Circle(float posx, float posy, float radiusxy, ColorMeshShape &color).create();
```

The renderable object base class, holds initially a boolean variable `flsVisible` that is true all the time since provisionally is assumed that all objects are visible always, since are not implemented culling techniques for the moment and therefore the objects are rendered in both cases although it is not very efficient, jointly declares a virtual method to set up a render transaction when the object is selected to be rendered, see [section 4.2.4.2](#) render transactions and [section 4.2.5](#) renderer.

The renderable object base class idea, is to hold pointers to information related with geometrical data, see [section 4.2.1.1](#) Vertex Declaration, jointly with the data related with effects and transformations data mainly, thus is being possible share these last data with the same mesh. Due to this, is possible to render a determinate mesh model instantiated repeatedly in a determinate scene, with different positions and different effects applied in each object instance, but all instances point to the same geometrical data stored in the graphic card, being referenced with the vertex declaration, explained in the [section 4.2.1.1](#).

The renderable object class, maintains a pointer to a determinate renderable effect instance, that will be applied to the object during the rendering process, see [section 4.2.6.2](#) local effects, and maintains a pointer to the transform related with this object see [section 4.2.7.1](#) transform class, finally declares a boolean variable to indicate if object is cloned or not, to maintain an internal control, if it is necessary to represent different objects, which maintain the same geometry with different effects applied and different positions within a scene, then the object must be cloned from the original created instance of the object, this is achieved by the method `clone`. At the same time, another two methods are used, `setRenderableEffect` to attach an effect when is created, see [section 4.2.6.2.2](#) renderable effect, and in the same way, this base class implements a method called `getLocalTransform`, for the object transform access, to set up the geometrical transformations related to each object.

This data, later is possible to retrieve through the render transaction mechanism, during the rendering process, to apply the effects to the objects and compute the geometrical transforms per each object, see the following [section 4.2.4.2](#) render transactions.

4.2.4.2 Render Transaction

The render transaction concept, has to be seen as a way to encapsulate all the necessary information that will be applied to the mesh during the rendering process. The container class encapsulates a pointer to a vertex declaration, a local geometric transform and a renderable effect for each object, all these data are passed from the renderable object class, when is selected to be rendered, so, during the rendering process the renderer knows what geometry should choose from all those that exist stored within the graphic card memory, through the data stored within vertex declaration pointer, in turn, also it knows what effect it should be applied to the object, in conjunction with what local geometric transformation should be applied as well, to calculate the model transform defined in [figure 4.10](#) to position the object correctly into world space, see [figure 4.32](#) below.

4.2.4.3 Renderable object set and scene handler

The following UML diagram shows the implemented structure.

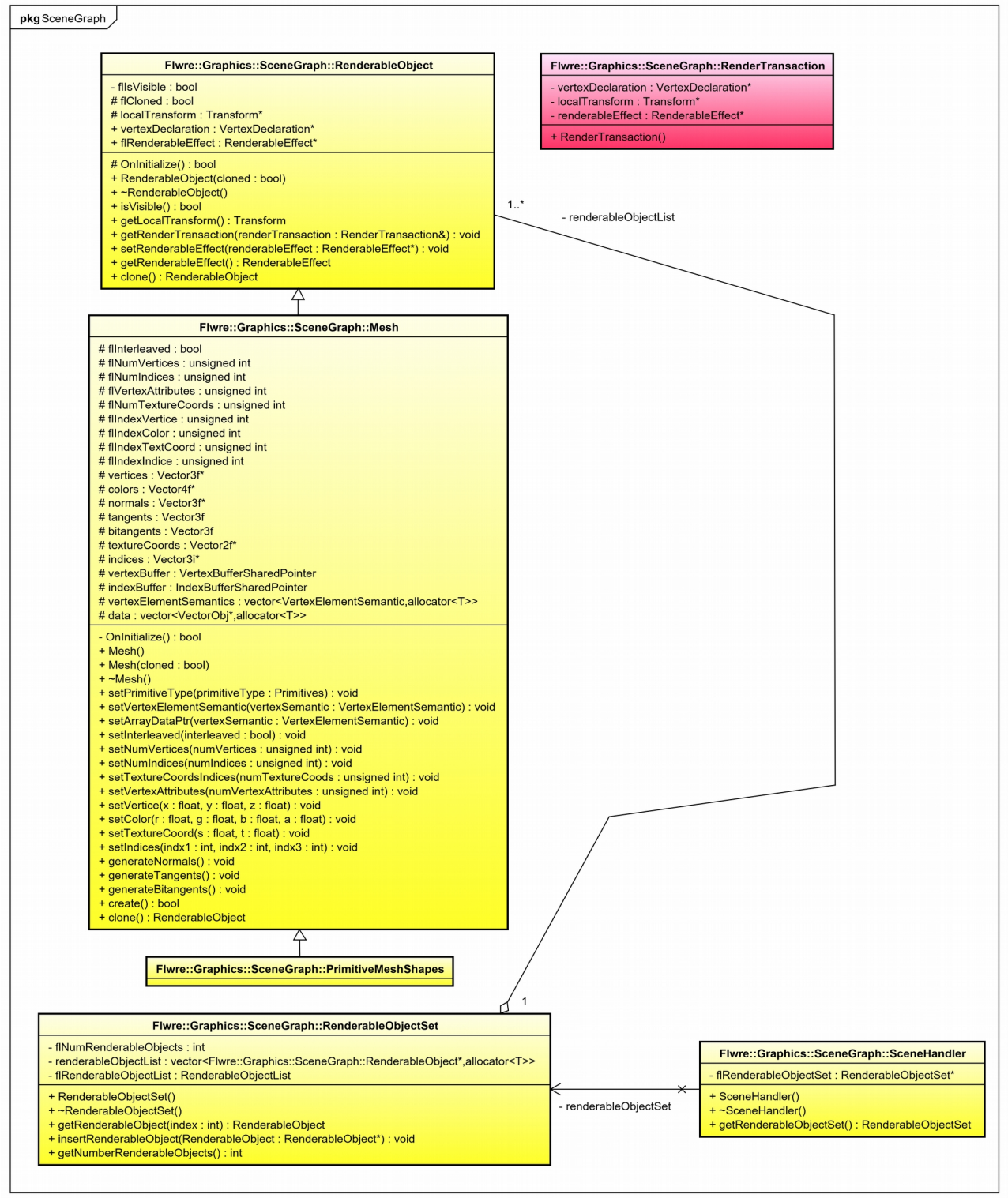


Figure 4.32 shown all the explained above in sections 4.2.4.1, 4.2.4.2, 4.2.4.3 with a UML diagram.

The renderable object set and scene handler is the trivial solution to substitute scene graph mechanism provisionally, since the scene graph not will be implemented in this project, see **section 4.2.4** scene graph, nevertheless, all described above in this section could be defined as the beginning of the path to implement a scene graph.

Basically the renderable object set class, is a renderable objects pointers vector that defines a linear method shaped like array to storage renderable objects, this should look like a render queue, that in the future will be replaced with a complete scene graph implementation.

Finally, scene handler is class container with a pointer to renderable object set instance, with a method to retrieve this pointer, but this could really be eliminated.

4.2.4.4 Camera

In really the camera is an illusion, build with different mathematical concepts, the camera projects a portion of the world space at any one time during the rendering process, this portion of the world is projected onto the view plane, the view plane is a rectangular region that contains the projected data and is known as viewport, the viewport is what is drawn on the rectangular display screen.

There are different camera models that are given determined by the projection type, the most common projection types are the perspective projection and orthographic projection, the engine by default uses perspective projection, though is possible to change the camera model pressing the F2 key to activate the orthographic projection, and F3 key to activate the perspective projection, see projection matrix **section 4.1.2.8.2.2** and its sub-sections.

The projection forms a view volume, all objects outside this view volume are not visible and therefore not drawn, the process that determines which objects are not visible is called culling, the objects that intersect the boundaries of the view volume are only partially visible, the visible portion of an object is determined by intersecting it, with the view volume with a process called clipping.

The view volume is defined by 6-planes, see **figure 4.11** and **figure 4.12** that depicts the orthographic view volume and perspective projection view volume or perspective-view frustum respectively. The engine initially does not implement any type of culling like frustum culling to determine if an object is inside or outside the view volume.

The camera is implemented in the name space `Flwre::Graphics::SceneGraph`, in the **Graphics** module, the camera is totally decoupled from the graphic API, so the graphic API not has any camera, and only receives the projection matrix and the view matrix, in the vertex shader, see **figure 4.10** and **section 4.2.6.1.1** shader parameter data, and **section 4.1.2.8** geometric transforms and its sub-sections.

The camera model is defined as an eye point, and has a coordinate system associated it, such as been defined in **section 4.1.2.8.2.3** view matrix lookAt.

The coordinate system has the eye position e , that indicates the camera position within the scene, then also has the camera direction vector w and the view up-vector called v . The camera direction vector is perpendicular to the view plane or clip space and the view up-vector is parallel to opposing edges of the view port, finally exist another vector called right-vector called u that is perpendicular to the camera direction vector and the view up-vector, since can be chosen such as been defined in 4.22 definition in the section **section 4.1.2.8.2.3** view matrix lookAt. So, the set of vectors $\{u, v, w\}$ defines the camera coordinate systems. See **cameraUpdate** method in camera implementation.

The camera class, defines several attributes and methods needed to build a camera within the engine, in the table below are briefly detailed.

Attribute name	Description
flYaw, flPitch, flRoll	Indicates the camera rotation in Euler angles with degrees, the pitch rotate about x -axis, yaw rotate about y -axis and roll rotate about z -axis.
flFov	Indicates the field of view.
flAspectRatio	Indicates the aspect ratio.
flzNear, flzFar	Indicates where the view volume by the additional planes are truncated, this variables defines the near and far planes position in the view volume projection.
flSpeed	Indicates the speed camera displacement.
flAngle	Indicates the angle of field of view used in camera zoom.
flUp, flRight, flLook	Indicates the vectors explained above $\{u, v, w\}$, where flUp is v , flRight is u and flLook is w , this vectors are normalized.
flPosition	A three-vector component that indicates the camera position obtained from the translation vector flTranslation defined below.
flViewMatrix	Maintains the view matrix, see section 4.1.2.8.2.3 .
flProjectionMatrix	Maintains the projection matrix, see section 4.1.2.8.2.2 .
flRotationMatrix	Maintains the rotation matrix, see section 4.1.2.8.2.4 .
flTranslation	A three-component vector to maintains the camera translation vector.
flInitSceneYaw, flInitScenePitch, flInitSceneRoll, flInitScenePosition	This variables maintain the initial camera configuration to be able to return this initial position pressing F12 key during the rendering process.

The camera supports perspective and orthographic projection as been explained, to set up this projections internally, due to is possible to change the projection at runtime, the camera defines an enumeration with the projection types supported. Pressing the F12 key is possible to initialize the camera to initial position during the rendering process when is moved across the scene.

The camera in all scenes can be moved by arrow keys and rotate by mouse pressing left button mouse, with the Q and Z keys is possible to lift the camera, and with the mouse scroll wheel, is possible to do zoom.

Finally the method names defines their functionality, see the following UML diagram that shows the completely camera definition.

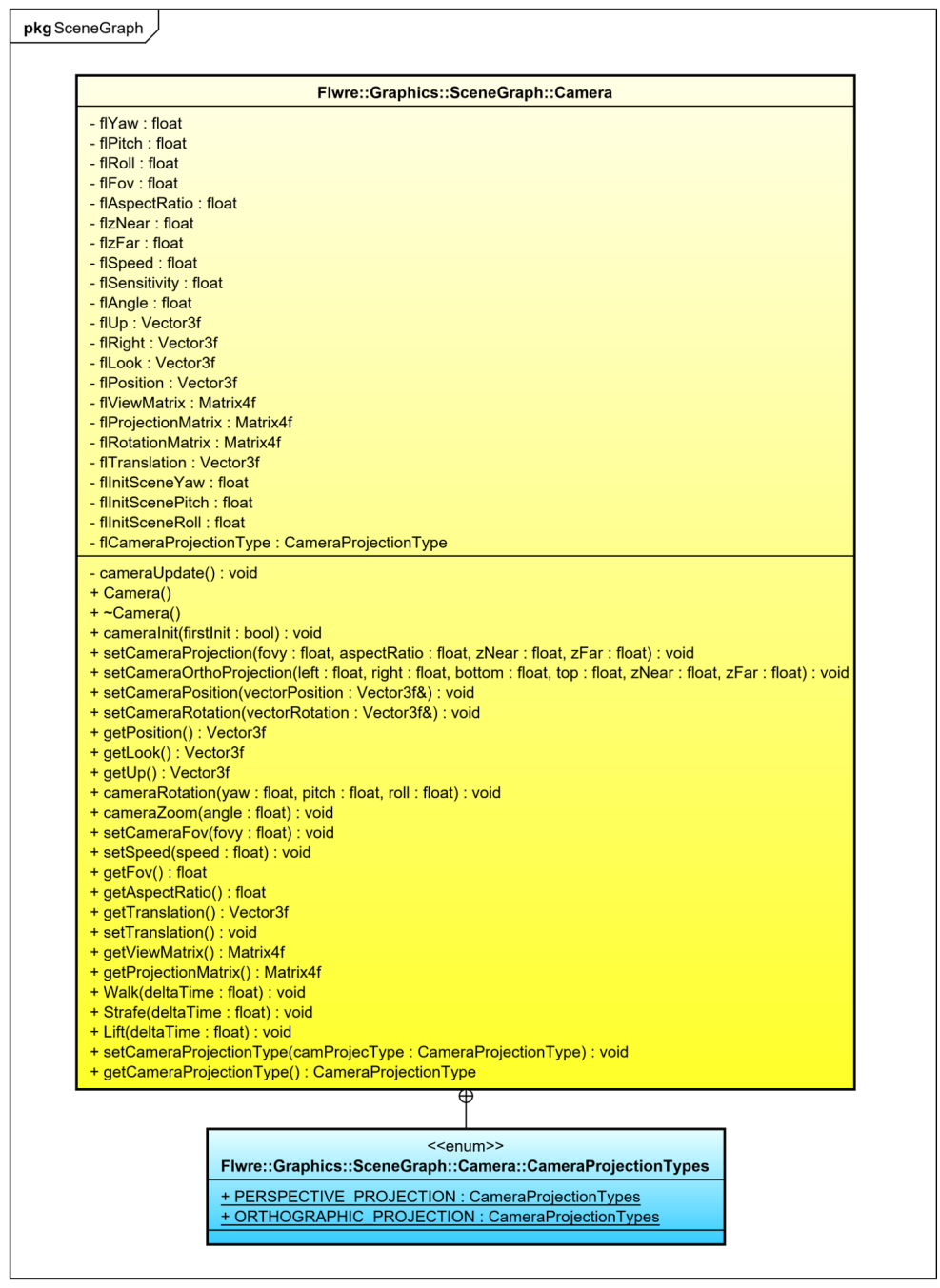


Figure 4.33 depicts the complete camera definition within the engine.

For more information about the camera models and its implementations is possible to review various materials, here are mentioned the following [\[3DGEArchCam\]](#)[\[fundaComGraCam\]](#) but obviously exists others.

4.2.4.5 Lighting

In a graphic system like a real-time rendering engine, the concept of lighting is important, due to if the objects are drawn with textures and color solely, the rendering results lack realism, so, scenes rendered will be incomplete without the presence of lights, for the same reason, that much of the richness our own visual systems provides in the real world they are due to lighting.

Then a real-time rendering engine must support lighting and materials that the lights affect, materials will be discussed in **section 4.2.4.6**. The standard graphics APIs support illumination models that are simple approximations to true lighting, the model concept here is used due to in fact is not possible to simulate exactly what nature does with the light, so, in the field of 3D graphics the lighting is "modelled", though in the concept of real-time the lighting models are designed to the lighting calculations can be performed quickly. More realistic lighting can be found in the graphic systems are not real-time, however, it is worth pointing out, that in the last years the distance of lighting realism between the graphics systems, that are not real-time and those that are, has been shortened due to the power increase of the graphic cards. Everything and so, the systems that are not real-time continue to maintain more realism lighting.

Due to lighting in the real world is extremely complicated and depends on way too many factors, nowadays is not possible to calculate it, with the "limited" processing graphic cards power, therefore, as has been said, the graphic APIs are based in a simplified physic models of the light, one of these models is known as Phong reflection model, being the most popular reflectance model and simpler to implement, this model was created by the Vietnamese pioneer computer graphics researcher Bui Tuong Phong(1942-1975)[[tuongPhong](#)]. This model are applied via shaders in the modern graphics approaches, unlike old approaches where the model illumination was implementing within the fixed function pipeline. With the using the shaders is possible to implement a couple of illumination methods based on Phong reflection model, since the illumination model can be implemented within the vertex shaders or within the fragment shaders, in case to implement it per-vertex, the illumination is applied directly to the geometric vertices before they are converted in fragments in the rasterization stage within the rendering pipeline, in this case the method is called **Gouraud shading** and this model was the implemented within the fixed function pipeline in the old graphics approaches, in case to implement it, within the fragment shader the illumination is applied directly to the fragments, and in this case the method is called **Phong shading**.

All illumination effects implemented within the engine uses the Phong shading only, within the fragment shaders to improve the accuracy of results, since when the shading light equation is evaluated within the vertex shader, causes a results not very realistic, due to normally the resulting color value within the vertex shader is the resulting lighting color applied to a particular vertex only, and the color values of the surrounding fragments are then the result of interpolated lighting colors, and precisely due to this interpolation the lighting looks a bit off, than if is implemented with the fragment shaders where it gives much smoother lighting results.

The following **figure 4.34** shows the visual difference between **Gouraud** and **Phong shading**, the scene on the left is rendered with **Gouraud** (per-vertex) shading, and on the right is the same scene rendered using **Phong** (per-fragment) shading. Underneath the teapot is a partial plane, drawn with a single quad. Note the difference in the specular highlight on the teapot, as well as the variation in the color of the plane beneath the teapot, see[[OGL4gls](#)] for more information.

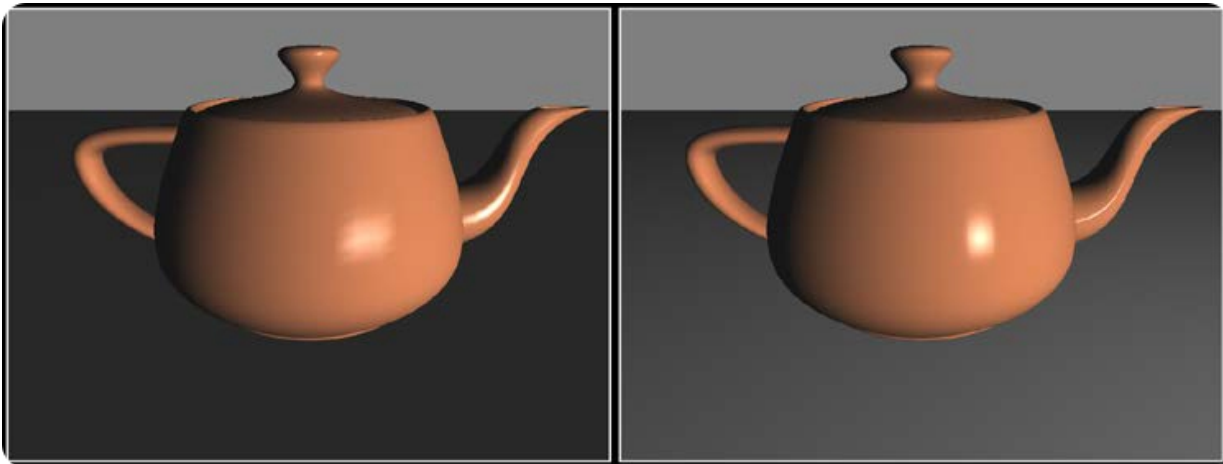


Figure 4.34 shows the difference between Gouraud and Phong shading, this image is not rendered by Flow Render Engine, the image has been extracted from [OGL4gls/]

The Phong equation has three components called ambient, diffuse and specular, these components in fact are physical attributes of the lights that define intensities like color values and in fact, can be seen each one of them as the final color contribution in a determinate surface.

Then the three components are computed as a sum, and the result is the Phong reflection, hence $(Ambient + Diffuse + Specular) = PhongReflection$. But also, the light sources have other attributes as well like intensity and attenuation that they indicate decrease in energy as the light travels over some distance, these are multiplied to each Phong reflection component.

Jointly with the explained above, the materials have an important role with all of this, such will be explained in the corresponding **section 4.2.4.6**.

The lights *Ambient*, *Diffuse* and *Specular* have the following behaviour, explained below.

The ambient light does not have origin nor direction and due to it comes from light that has been scattered by the environment, the materials discussed in **section 4.2.4.6** reflect some of this light due to the quantity ambient indicated in the material that indicates the fraction of ambient light that is reflected.

The diffuse lighting strikes in a surface, the most important property of diffuse light is its direction, since in each point on the object surface the light arrives in some direction and then is scattered equally in all directions at that point, a material has the color component diffuse that indicates how much diffuse light is reflected.

The specular lighting generates specular highlights, this light strikes a surface as well, but its reflection has a preferred direction generating the specular highlights. The specular material color component specifies the fractional amount of reflectance. It is important to highlight, that specular lighting is more a property of the object, rather than the light itself.

The behaviours explained above, has been represented in the following [figure 4.35](#).

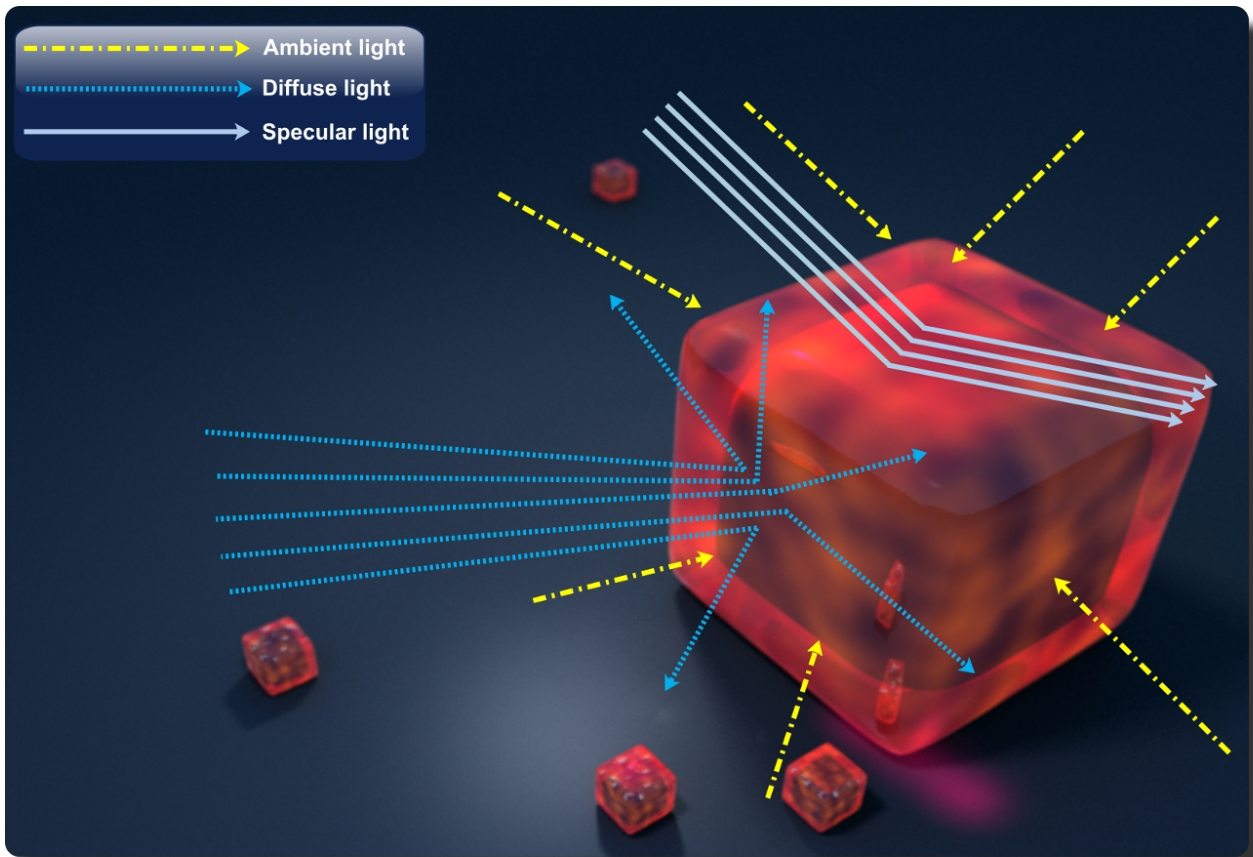


Figure 4.35 the different kind of lights behaviour an object front facing, [cubelImage].

In the 3D scenes created with a graphic system like a real-time rendering engine, usually does not create ambient, diffuse or specular lights directly, light sources are used instead, this light sources has been implemented within the engine distributing them with a hierarchical structure, because the light sources comes in various types, although initially the idea was implement a unique class called light that represents the lights, however, with this approach, not all data members made sense for each light type. The following sections explains the properties per each light type and shows how has been implemented within the engine, to introduce the complete lights hierarchy structure which will be discussed in the following sections is shown the following simplified UML diagram.

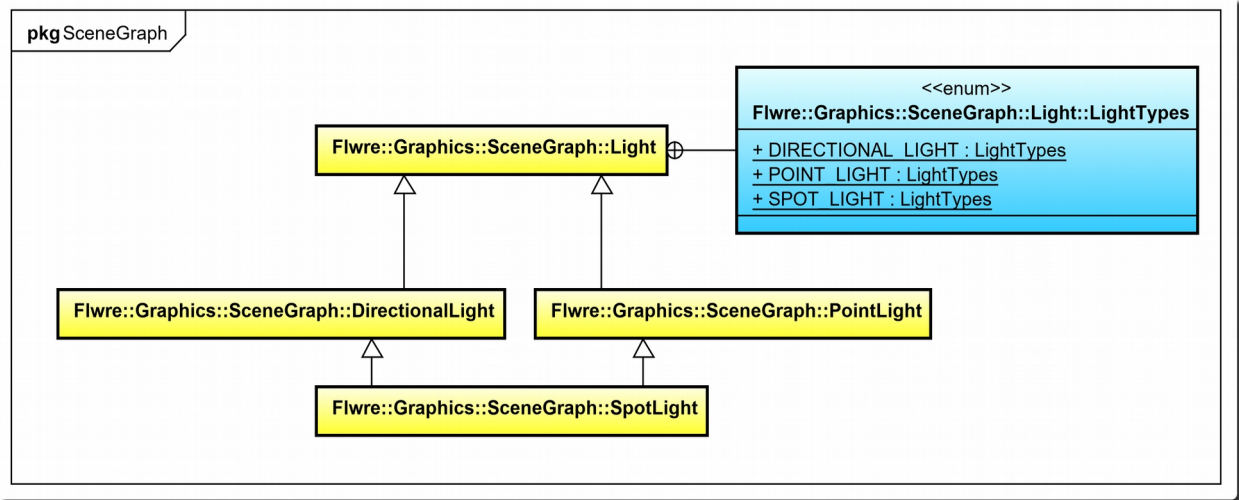


Figure 4.36 depicts the complete structure lights hierarchy within the engine.

As can be seen, the lighting with the different light types has been implemented within the namespace called **Flwre::Graphics::SceneGraph** within the **Graphics** module, both the lighting data stored and the materials data is totally platform independent, so no matters which the graphic API used, with the manner how to storage this data.

The class called light maintains the common values for all light types, this includes the light name jointly the components ambient diffuse and specular, since regardless of the type of light all lights includes this components with its intensities and strength, all this values can be established and retrieved through the corresponding setters and getters. Finally the light class, has an enumeration where is defined the lights types supported by the engine, this is used internally when the lights are instantiated, and could be used to do different light types checks, outside the lighting classes.

It is worth pointing out, that the lights classes structure defined in the **figure 4.36** within the engine, only maintains the definition values per each light defined in a determinate scene, but the light equations are implemented within the fragment shaders for each effect where the lights are involved, the engine during the rendering process, communicate to the fragment shaders and pass these values to calculate the light equations per each pixel in each frame, and rendered the light effects over the scene, see **section 4.2.6.1.1** shader parameters data and concretely the **UpdateUniformsConstants** method. To achieve the desired light effects over the surfaces, the meshes must have the normals calculated, such as explained in the **section 4.1.2.9** normal calculation, otherwise the lighting will not affects to the meshes surface.

In the old graphics approaches, the number of lights on a scene was limited, in modern graphics approaches, as the implemented in this engine, the number of lights are not limited by a determinate number of lights if not by the graphic card power.

4.2.4.5.1 Directional light

A directional light could be seen like the sun in the real world, a directional light has a direction but no specific origin, so that all light rays are parallel to each other, then its direction within the engine is specified by a three component vector, that is used to set up the light direction in the scene where the light is set up. Although in the real world, the sun has a determinate position, due to the long distance where is located simply is disregard its position and take only the direction into account. Finally, the directional light brightness remains the same regardless of the distance from the lit object.

The **DirectionalLight** class is derived from the **Light** class and maintains a direction vector jointly the methods to set up and retrieve it.

The following **figure 4.37** shows the directional light in conceptual manner.

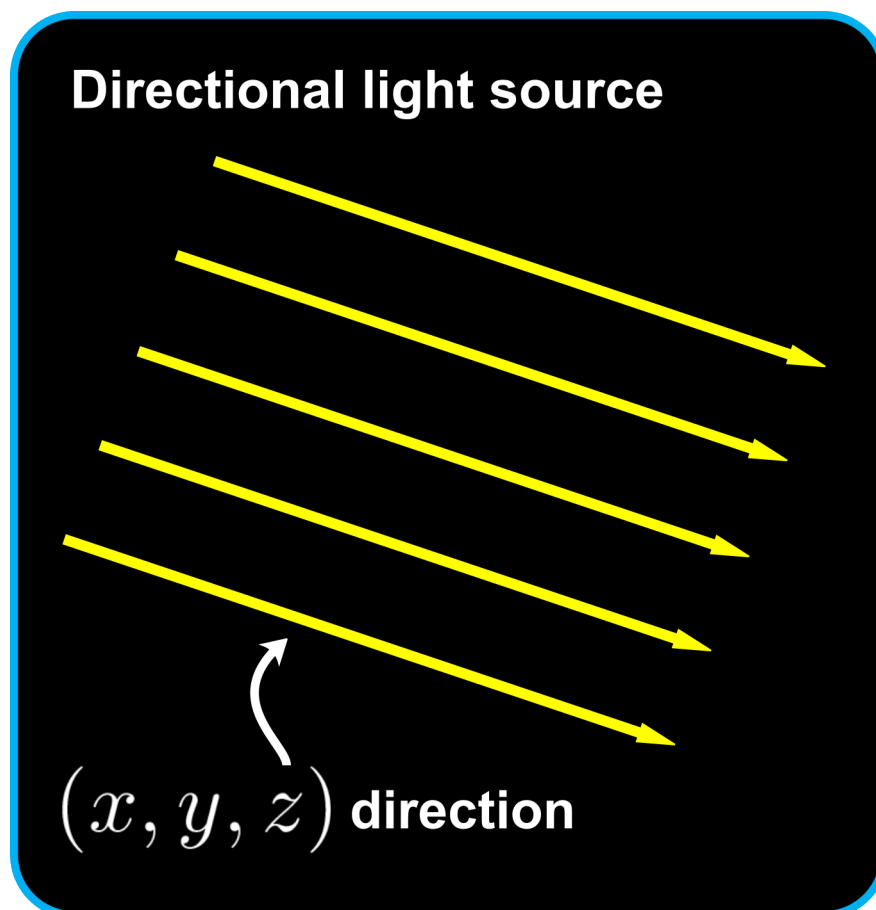


Figure 4.37 depicts the directional light in conceptual manner.

4.2.4.5.2 Point light

A point light source, unlike a directional light, has both an origin as well as a fading effect, which grows stronger as objects move away from it, hence, a classic example for a point light is a light bulb. In a point light the direction of light is constant across the scene, then the directional light becomes dynamic due to in a point light, the light emanates in all directions equally, this causes that the direction must be calculated per each object by taking vector from the object towards the point light origin, for this reason this light type specifies an origin position rather than a direction like the directional light.

The **PointLight** class is derived from **Light** class and maintains a three component vector position, jointly the methods to set up and retrieve it, at the same time, maintains different values to set up the lighting attenuation, such as constant attenuation, linear attenuation and the exponential attenuation, jointly with the methods to set up and retrieve it as well.

The following **figure 4.38** shows the point light in conceptual manner.

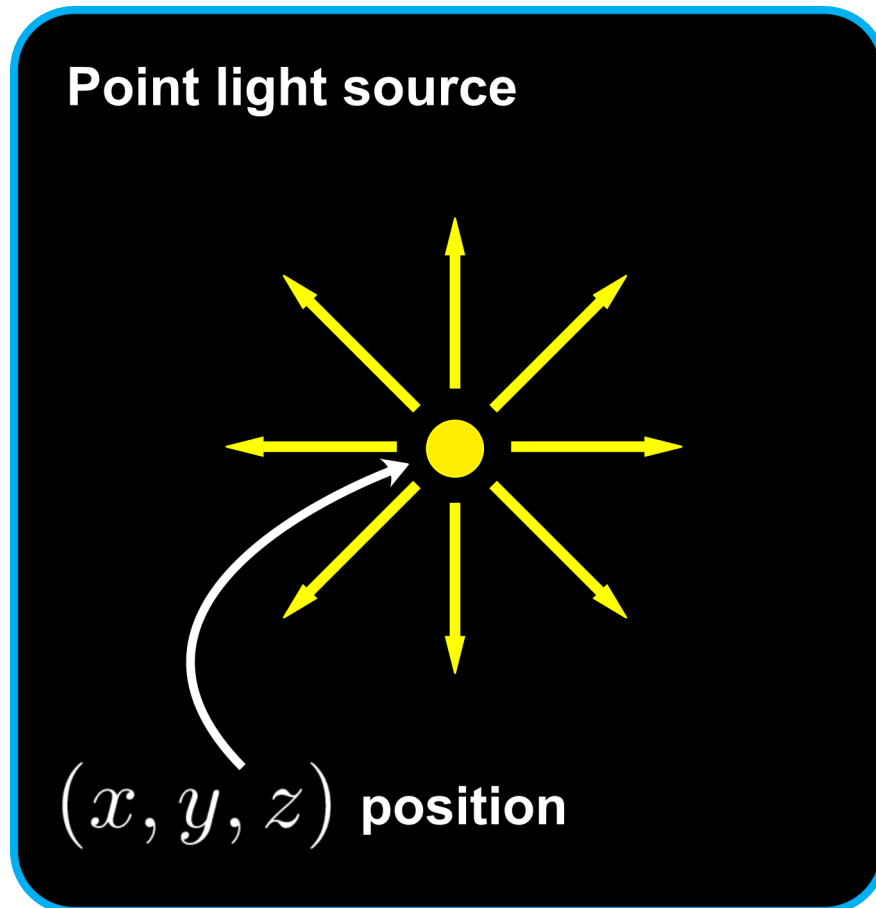


Figure 4.38 depicts the point light in conceptual manner.

4.2.4.5.3 Spot light

A good spot light example is a lantern, this light type is more complex than the other lights explained above, essentially borrows components from both, for this reason the spot light class is derived from directional light and point light class, see [figure 4.36](#). This light has an origin position, and is under the effect of attenuation as distance from target grows as the point light, and its light is pointed at a specific direction as the directional light. The spot light adds the unique attribute of shedding light only within a limited cone area that grows wider as light moves further away from its origin, outside the cone, the spot light emits no light.

The **SpotLight** class maintains different methods to define the cone area diameter, jointly with a method to on and off the spot light.

The following [figure 4.39](#) shows the spot light in conceptual manner.

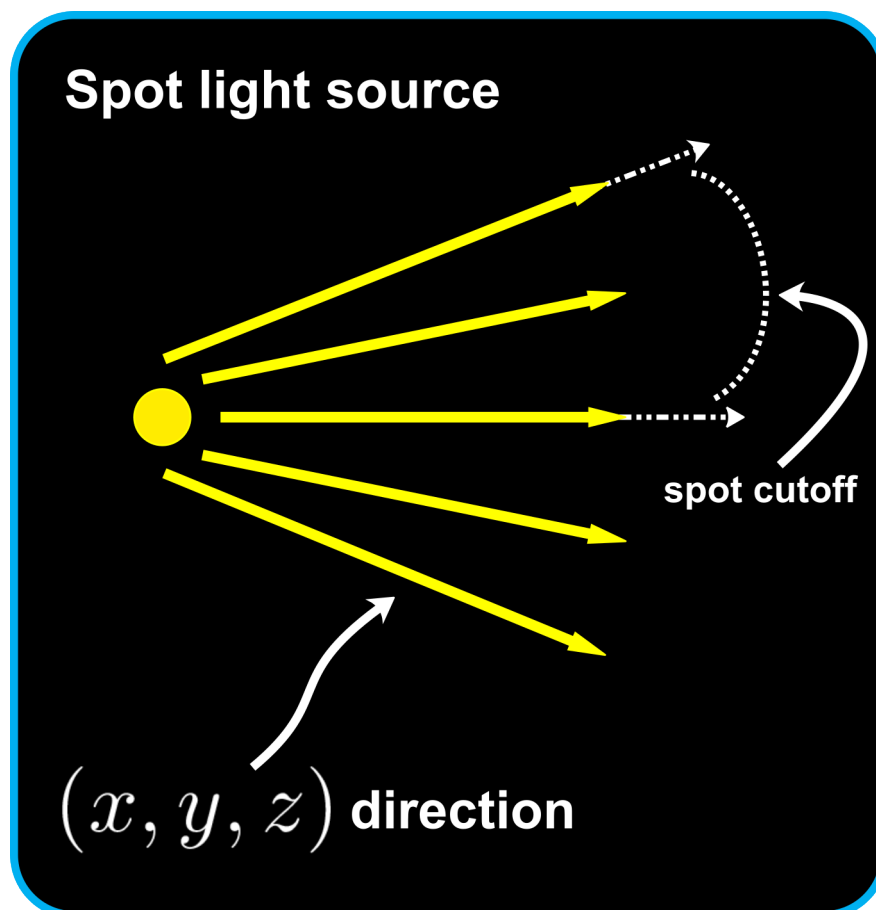


Figure 4.39 shows the spot light in conceptual manner.

Finally, the **figure 4.40** shows the complete lighting structure with an UML diagram for greater understanding to explained above.

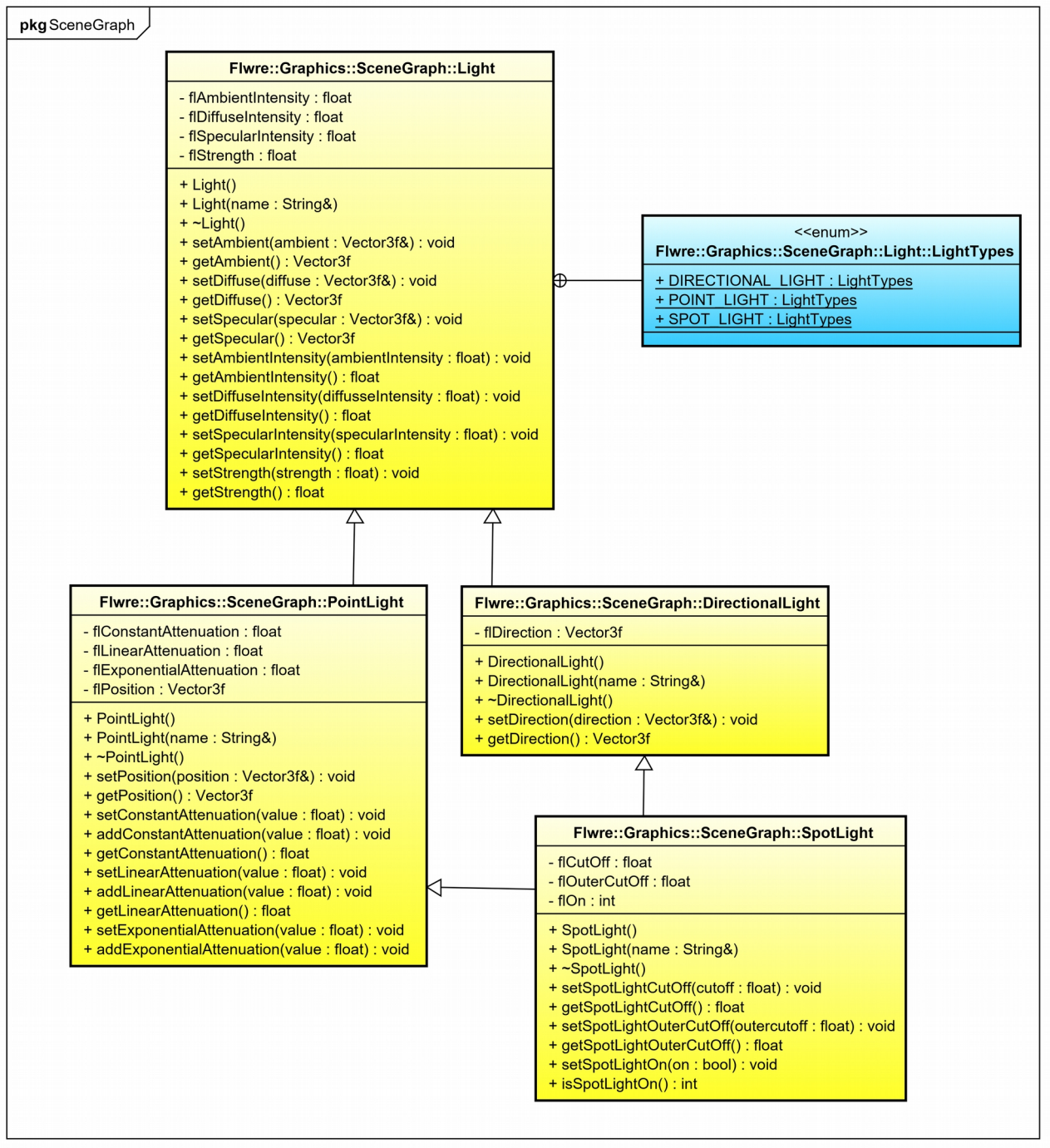


Figure 4.40 shows a complete lighting structure implementation with an UML diagram.

In the **section 4.1.2.9** normal calculation, has been explained the normals calculation, and how are applied to the meshes in order to be able to simulate, how the rays of lights bouncing off over the meshes surfaces in the scenes simulating a reality. if a mesh does not have its normals calculated and applied on it, the illumination effects will not works.

4.2.4.6 Materials

Materials are used to enhance the realism in the rendered scenes jointly with the lighting and texture mapping. The materials allows for objects simulate different types of materials of the real life and are strongly related with the lighting such as will be seen below.

In the real world, each object reacts differently in front of the light depending on the material it is made, hence, steel objects are often shinier than a clay vase for example or a wooden table does not react the same to light as a steel table, at the same time, each object also responds differently to specular highlights since some objects reflect more than others. To simulate this within a graphic system is necessary to define material properties for each object.

The materials solely applied to an object do not take effect, for the material to take effect, it is necessary to involve the lights, at least one, see [section 4.2.4.5](#) lighting, and obviously are needed the vertex normals calculated explained in [section 4.1.2.9](#) normal calculation.

The materials are composed of various components called ambient, diffuse and specular, these components will determine the ambient, diffuse and specular reflection of the objects in front of the light, in fact, are color components that indicate how much of each component light is reflected, hence, can be seen as attenuation factors per each RGB component for each light type. The computation of this can be seen as follows.

$$ambient_{light\ model} \cdot ambient_{material}$$
$$diffuse_{light\ model} \cdot diffuse_{material}$$
$$specular_{light\ model} \cdot specular_{material}$$

Finally, the shininess component is related with the specular component, and defines which specular highlight scattering or radius the material has, and the emissive color component simulates the light that the material itself generates, which is usually nothing, and this light does not reflect with the other objects. With this scenario it is possible to simulate real-world materials in a rendered scene.

The engine offers a basic system to store the material properties explained above to apply them manually to the objects. The materials also can be stored as part of a 3D model format.

The engine implements an essential OBJ parser, where the materials in the OBJ 3D file format are stored in a separate material file, with the extension ".mtl", these files are associated with the wavefront ".obj" files. Hence, the OBJ parser internally will use this system to store the material properties read from ".mtl" files for later to set up them to the objects to get the final rendered results.

The materials are implemented within the namespace `Flwre::Graphics::SceneGraph` in the **Graphics** module, this data is totally platform independent like the lighting, so no matter the graphic API used, with the manner how to store this data. To maintain the different parameters involved in a material a unique class called material is defined such as depicted in the UML diagram in [figure 4.41](#).

This class maintains different data members to define the components explained above, such as emissive, ambient, diffuse, specular and shininess with a three-dimensional vector, at the same time defines the setters and getters methods to manage this data. Although the components initially are implemented with three-dimensional vector, it would have been better implemented with four-dimensional vector, to take into account the alpha component. However, this is implemented in color data type see [section 4.2.7.2](#) color class, since the idea was to implement the components

of the materials with the color data type instead three-dimensional vector, but for now it is left that way.

The following UML class diagram, shows the material class with its attributes and methods, in a simplified way, since the class has a lot attributes and methods defined, but here to simplify the diagram, shown the basic components involved with the materials.

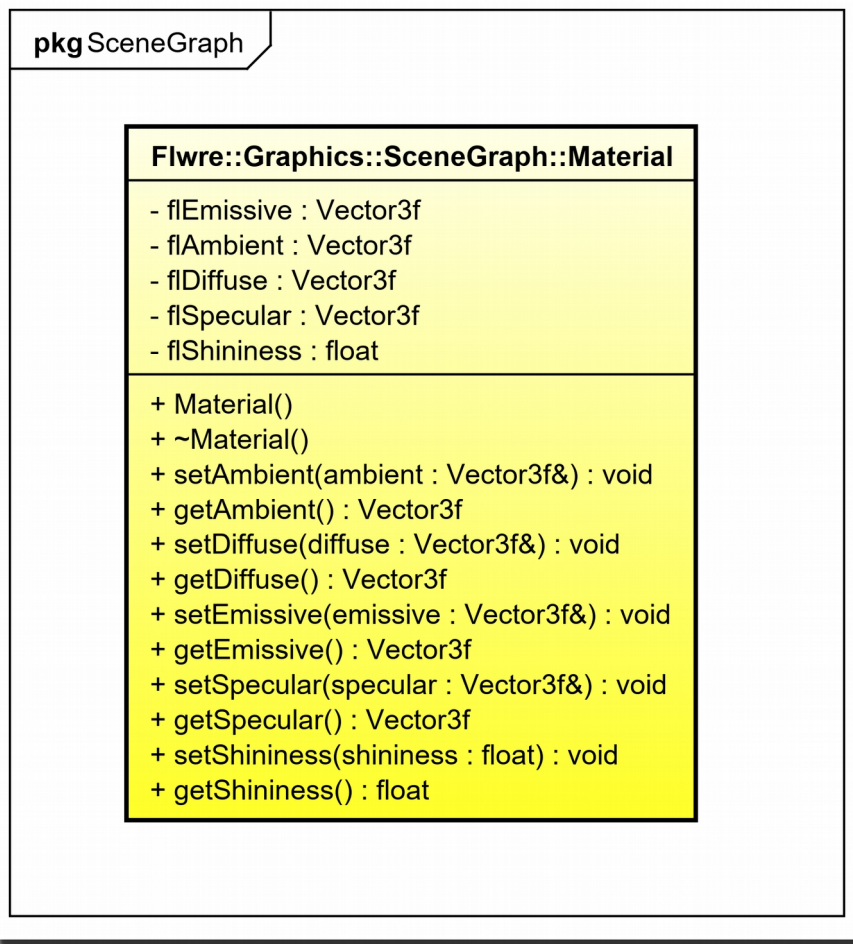


Figure 4.41 shows the Material class with a UML diagram, in a simplified way.

4.2.5 Renderer

This sub-module is defined in **Flwre::Graphics::Renderer** name space and holds the interface definition for all common rendering operations, more concretely is the top-level entry point into the engine drawing system. The interface is implemented in the underlying layer **GraphicOGL3** with OpenGL3.3, which can be loaded at runtime and could be switched between other modules implemented with different graphics API at runtime as well, if would be implemented, see **section 4.1.4.1** dynamically loaded C++ objects. The interface approach makes the **Graphics** module completely independent from the underlying hardware and any graphic API.

Inside **Flwre::Graphics::Renderer** name space, can be find the render class, this is an abstract class with the following definition.

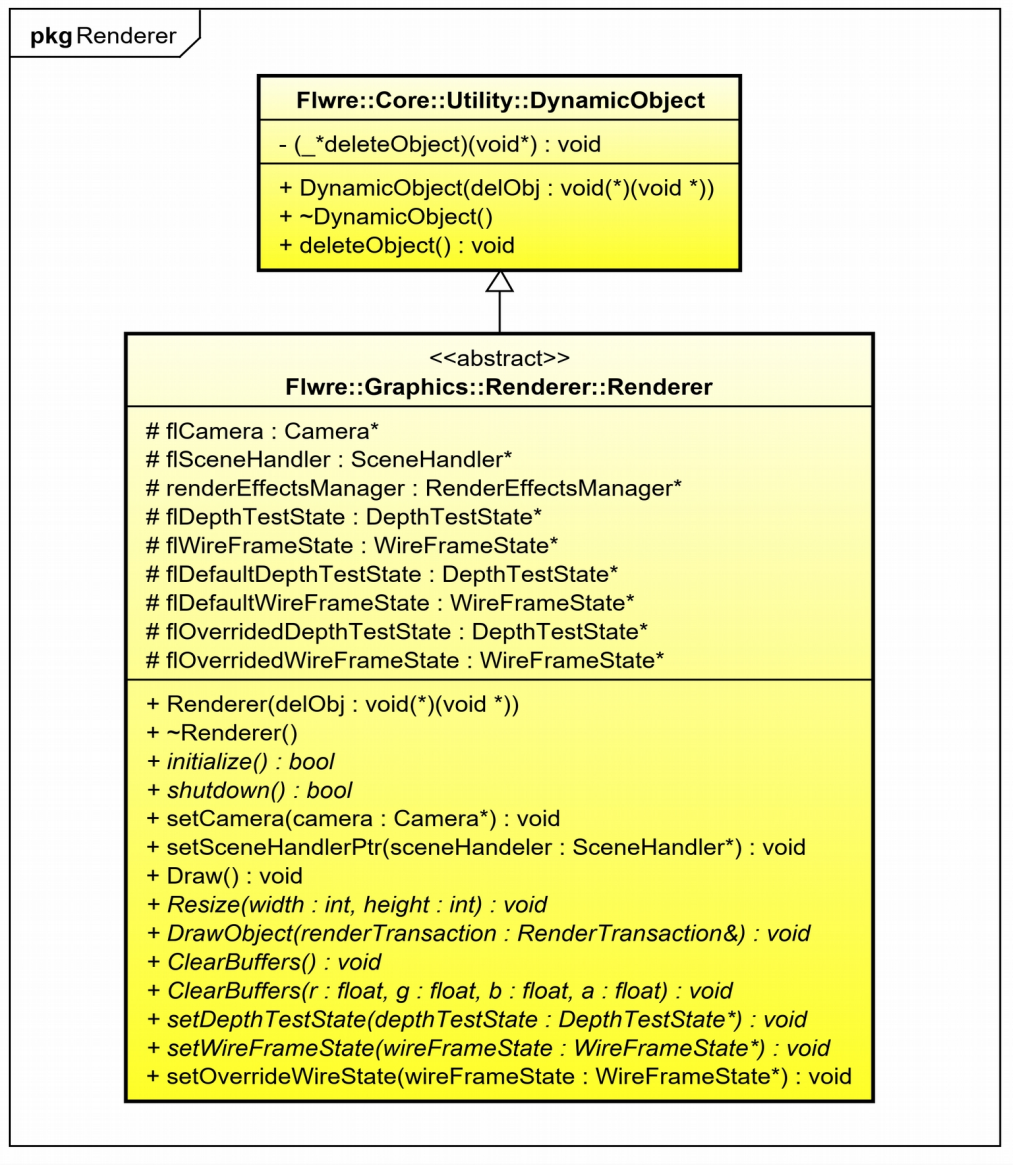


Figure 4.42 depicts the abstract renderer class derived from Dynamic Object class

Notice that the renderer class, is derived from DynamicObject class, this is the mechanism that allows to load and to switch the modules at runtime explained, see **section 4.1.4.1** dynamically loaded C++ objects.

The class holds a pointer **flSceneHandler** jointly a method to set up the pointer to access into the render queue, see **section 4.2.4.3** renderable object set and scene handler. The renderer access to the effects mechanism exposed in **section 4.2.6.2** local effects, to get it, a pointer to the effects mechanism is encapsulated within the render transaction container, defined in the **section 4.2.4.2** render transaction, see **DrawObject** method below, jointly with other data needed during the rendering process for each object. So, the renderer own the mechanism to unpack all information defined in render transaction for each object selected to be rendered, handling the graphic API implemented in the underlying **GraphicsOGL3** layer, to feeding the functions involved in handling buffers, effects and objects drawing.

On the other hand, the renderer has another features, firstly maintains a pointer to camera system implemented, see [section 4.2.4.4](#) camera. And maintains different pointers to different render states explained in [section 4.2.5.1](#) render states, and finally maintains a pointer to render effect manager see [section 4.2.2.4](#) render effect manager.

When the renderer is initialized, the render subsystem is started, it is worth pointing out, that the managers are initialized when render starts, all managers acts to graphic API renderer level, except the render effects manager that acts in **Graphics** module level, for this reason the pointer to render effects manager is declared in this level, nevertheless is instantiated in **GraphicsOGL3** module for a subsequent orderly destruction of resources when render system is shuts down.

The method **Draw** is the top-level entry point into the drawing system, and starts the sequential reading of lineal render queue implemented, see [section 4.2.4.3](#) renderable object set and scene handler, for the moment no culling techniques are implemented, so, all objects stored in this vector are send to be rendered independently if the object is visible or not. At the same time, here would begin a depth-first traversal of the scene hierarchy, in case the scene graph would be implemented, then the lineal access method would be replaced, since this is not better suitable scenario for obvious efficiency reasons, so, in the future will be replaced with a complete scene graph implementation, jointly with culling techniques, see [section 4.2.4](#) scene graph.

According renderable objects are accessed from **Draw** method the **DrawObject** method tells to the renderer to draw its geometry, to achieve this, a render transaction is passed as a reference to the method **DrawObject** see [section 4.2.4.2](#) render transactions and the [figure 4.42](#), for the moment one render pass is supported per object only.

Finally, the **Resize** method is used to resize the viewport in case the window size was modified during the rendering process, see [section 4.2.3](#) windowed application, the **ClearColorBuffers** methods sets the entire window to a given background color.

4.2.5.1 Render states

The render states defines information associated with the geometric data for the purposes of drawing the objects, more concretely they configure some render pipeline stages that are configurable, see [section 2.1](#) the rendering pipeline, [figure 2.1](#). the engine initially defines two basic render states explained in the following sections, this states are unpacked in the rendering process, from the render pass due to each render pass has associated the render states, or in other words each render effect has different render states associated and configured in certain manner, see [section 4.2.6.2.1](#) render effect.

The renderer defines its own global render states configured by default, when the renderer is initialized, at the same time each render effect when this is implemented see lines 53 and 54 of code snippet 1.3 in texture2D effect implementation in [section 4.2.6.2.1](#) render effect. On the other hand, each effect can define their own render states, then when the render effect is applied to a determinate object, the render states defined are applied in local manner during the rendering process, otherwise if the render effect did not has a determinate render state defined, then the global render state prevails.

All render states derived from the base class State, this class implements the common functionalities used for the render states, initially implements the enumeration used for comparison function used in different render states, although for the moment is only used in depth test state, but in the future other render states could be implemented it, for example, Alpha test state and Cull

face state among others. The following sections explains the render states implemented and the [figure 4.43](#) shows with an UML diagram this implementation.

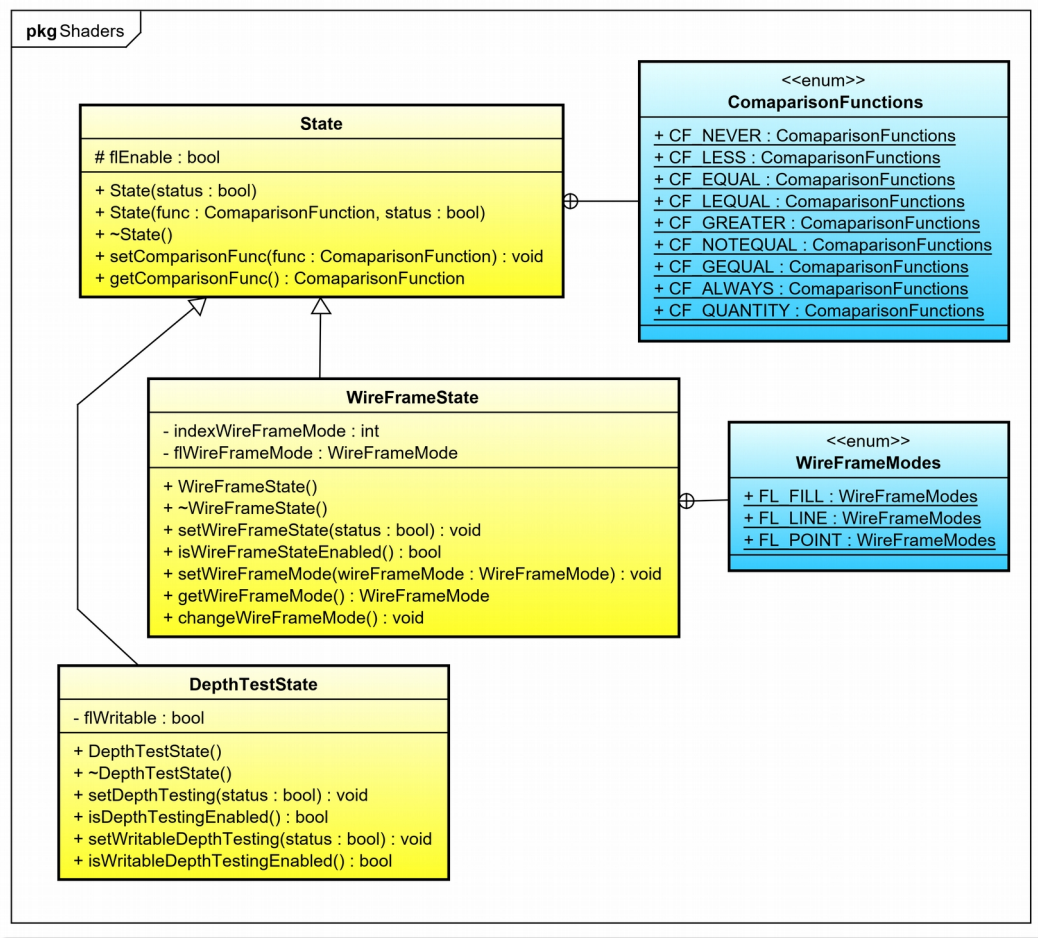


Figure 4.43 depicts the render states implemented with an UML diagram.

4.2.5.1.1 Wire frame state

The **WireframeState** class, is a simple class that offers the methods to manage the wire frame state configuration, and defines the different rasterization modes such as the **FL_FILL** mode to render the interior of the polygons filled, **FL_LINE** mode where the boundary edges of the polygon are drawn as line segments, and finally **FL_POINT** where the polygon vertices that are marked as the start of a boundary edge are drawn as points during the rendering process.

This state has been overridden to apply the different rasterization modes globally to the scene by pressing F1 key during the engine development for debugging reasons. For this reason the possible configurations defined in the render effects will be omitted, see **setOverrideWireState** method in the renderer class in the [figure 4.42](#).

4.2.5.1.2 Depth test state

The **DepthTestState** class, implements the mechanism to enable and disable the depth buffer. The depth buffer is also known as z-buffering. When this buffer is enabled, prevents faces rendering to the front while they are behind other faces using. To achieve this, this buffer stores the pixels depth of the objects are rendered during rendering process, since many points of the 3D space may end up projected to the same pixel on the 2D screen space, then the graphics system needs to keep track of the actual depths of z -axis in the 3D scene to determine which of those points is the visible one.

To determine if a pixel is visible or not, the depth test is enabled by default and tests the depth value of a new pixel against the content of the depth buffer storing others pixels depths, then performs a depth testing, and if this test passes, the depth buffer is updated with the new depth value otherwise if the depth test fails, the pixel is discarded.

It is important to highlight, that this test is realized in screen space after the fragment shader has processed within the render pipeline, see Per-Sample Operations[[pipelineOGL3](#)].

The test is based on a function ($N \text{ func } P$) where N is a determinate depth of pixel, while P is the depth stored in the depth buffer (z-buffering), func can modify the comparison operators it uses for the depth test, this allows to control when should pass or discard pixels and when to update the depth buffer, setting different comparison operators.

The function func accepts several comparison operators that are listed in the table below, at the same time this operators has been implemented within an enumeration called **ComparisonFunctions** shown in the [figure 4.43](#) in the **State** class.

Function	Description
CF_NEVER	The depth test never passes.
CF_LESS	The depth test, passes if the fragment's depth value is less than the stored depth value, this is the option configured by default in the engine.
CF_EQUAL	The depth test, passes if the fragment's depth value is equal to the stored depth value.
CF_LEQUAL	The depth test, passes if the fragment's depth value is less than or equal to the stored depth value.
CF_GREATER	The depth test, passes if the fragment's depth value is greater than the stored depth value.
CF_NOTEQUAL	The depth test, passes if the fragment's depth value is not equal to the stored depth value.
CF_GECUAL	The depth test, passes if the fragment's depth value is greater than or equal to the stored depth value.
CF_ALWAYS	The depth test always passes.

The reason to implement the enumeration in the base **State** class, is due to the comparison operators, can be used for others render states that will be implemented later such as alpha state or stencil state.

4.2.6 Effects

A real-time rendering engine needs implement a solid effects system to avoid major changes in the engine when a new effects are added to the engine. Solve this scenario is not has a trivial solution to achieve a solid mechanism, and obviously there are different ways to solve it, and some better than others.

Aside the possible mechanism that can be proposed within the engine to manage the effects, the advances raised in the last years related with computer graphics field, and concretely together with the increase of the GPUs power, the effects becomes to interesting and exiting field of work to delve in new lines investigation and consequently appears new effect techniques along with the existing techniques.

This drives that an engine must be frequently will be modified to fit a new effects features appeared or simply because of the requirements in a determinate project maybe change, and the engine does not support the new features that project demands, so, an engine needs a solid effects system to front face this scenario with guarantees.

This scenario provides different manners to apply effects to the scenes, since each effect or effect group them need different application techniques, to get the desired effect over the scenes or objects or even combine effects over the scene totality, and at the same time different effects applied to each objects.

Hence, a professional real-time rendering engine needs must supported different manners to apply the effects, mainly the effects can be global effects or local effects, namely the global effects are applied to everything that appears on to the scene, and the local effects are applied to a determinate object or group of them, per each effect will exist a render technique which will own one or several render passes to get the desired technique, see **section 4.2.6.1.3** render technique and **section 4.2.6.1.2** render pass.

This project, as stated previously, intend to defines the essential pillars of a rendering engine, simplifying to the maximum without losing a good scalability, for this reason here initially this scenario has been simplified as well.

The engine offers a mechanism, where initially only the local effects are supported with a unique render pass within a render technique, and a mechanism to apply the effects attaching them to the objects. The following sections explain all this in more detail.

4.2.6.1 Shaders

In a modern computer graphics approach supported by the engine, the shaders are a very important piece, since allows a programmable render pipeline leaving behind the old fixed pipeline exposed in older computer graphics generations.

A basic shader management is implemented for the moment, a unique shader class is implemented with the basic functions to load, compile, links and delete the shaders. The engine supports vertex and fragment shaders, for this reason a good approach could be a hierarchy of classes definition to depicts this structure with a base class called **shader** and couple of derived classes called **VertexShader** and **FrangmentShader**, nevertheless to simplify the scenario temporally a unique shader class is defined and implemented. The shader class is a pure abstract class since is implemented in the **GraphicsOGL3** module with OpenGL 3.3.

More concretely, to support shader programs by the engine, a subsystem that encapsulate the shader programs and its parameters has been implemented, at the same time, this subsystem will be useful in a scene graph management in case of being implemented in the future.

Exists different components that are effects related such as lights, materials, textures and so on, such as explained in the previous sections, thus on the one hand, the shader class maintains stored the number of components for a determinate effect, that is, the number of lights involved in a shader, the number of textures and so on. Together with this, emerges the requirement to maintain a constant communication with the shaders stages defined inside in the graphics pipeline, see [figure 2.1](#) in [section 2.1](#), during the rendering process, once the shader is compiled and linked. So the shader class exposes a mechanism to store uniform and attributes locations values, for later, to be accessed them by name at a later stage, the later stage in this case is the rendering process, since the engine needs to access to the locations of all of the shader's attributes and uniforms respectively, defined within the shader program. This data is stored within the shader class, with a STL library map container, and are mapped by a name, that correspond to a determine location value within program object being possible to be accessed by name.

To achieve this, the shader class is expanded with new methods, see methods **AddAttribute** and **addUniform** together with the overloaded access operators **() []**, and the setters and getters to maintain the number of components stored per each shader, see [figure 4.44](#) depicted below as UML diagram.

The engine only supports shaders written with GLSL 3.3, and the shaders must be written with a determinate structure so that the engine to parser it correctly, otherwise, the engine will not read correctly the shaders and does not work. In like manner that nothing related with OpenGL3.3 not will be explained, nothing related with GLSL 3.3 will be explained neither.

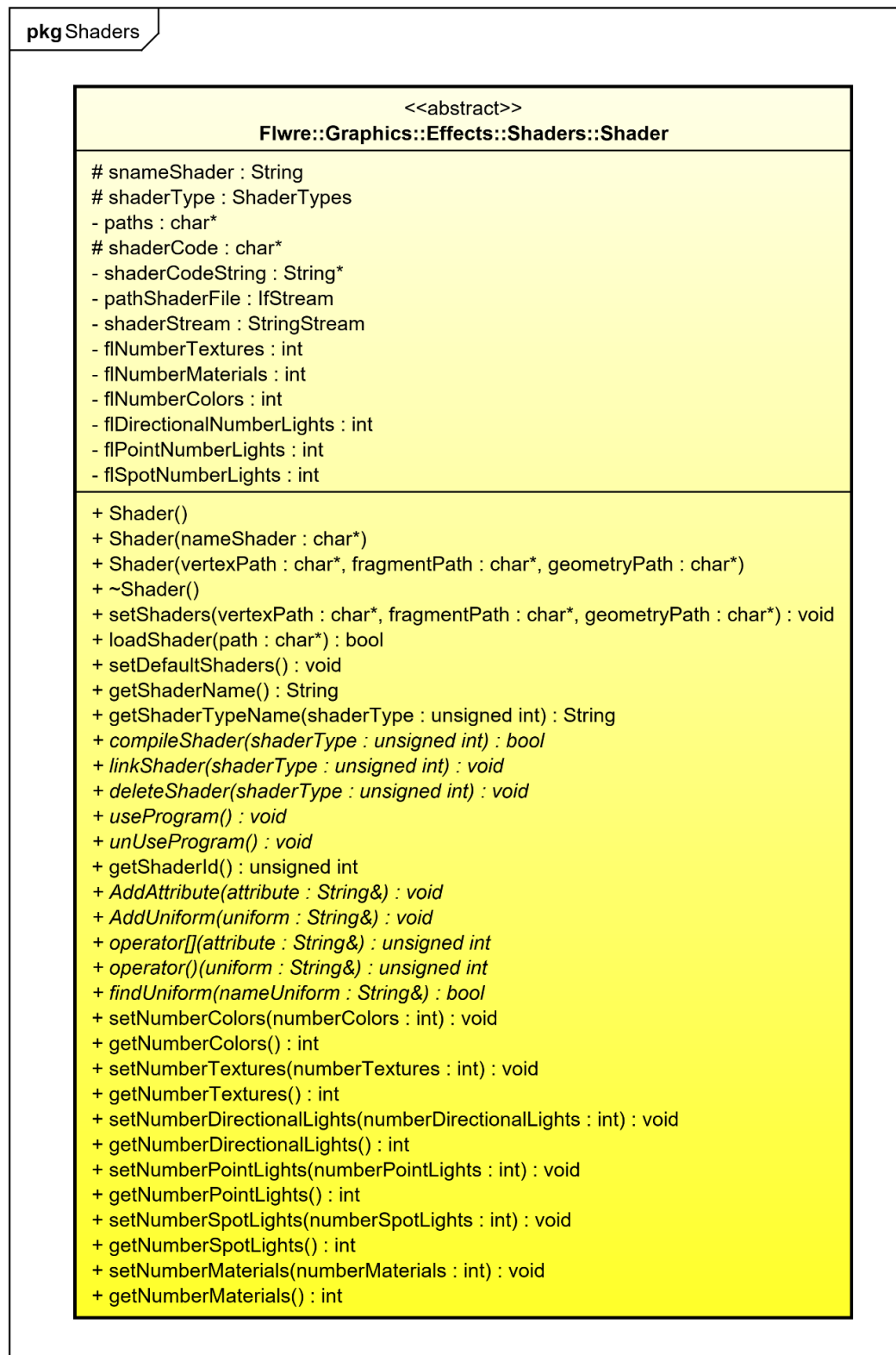


Figure 4.44 shown the abstract class Shader implemented in GraphicsOGL3 module with an UML diagram.

4.2.6.1.1 Shader parameters data

To complete this scenario, a new class has been created called **shaderParametersData**, this class maintains a relation with a determinate shader instantiation, via a pointer to keeping the parameters values by a given shader for a determinate render pass. This relation is created at the time when is created a determinate render effect instantiation just like will see later, see **section 4.2.6.2.2** renderable effect.



Figure 4.45 shown the abstract class ShaderParameterData with an UML diagram.

The **figure 4.45** shows the UML diagram for this class, such as can be seen the class is abstract, due to the complete implementation is performed in **GraphicsOGL3** module, and involve the direct graphic API manipulation. The class defines the setters to store the pointers to lights, materials, textures and colors defined in a given render pass, all this data is defined in the **Graphics** level module, so, are independent of graphic API used. However, is needed pass all this data to the graphic card, so, this class defines a method called **updateUniformsContants**, this method is entirely implemented in **Graphics** level module, although communicates to the underlying layer **GraphicsOGL3** to pass the values of stored components to the shader stages inside the render

pipeline, see [figure 2.1](#) in [section 2.1](#), which it involve the direct graphic API manipulation, to achieve this defines an interface that is implemented in **GraphicsOGL3** module, for a determinate graphic API that in this case is OpenGL3.3. see the abstract methods defined in the [figure 4.45](#) more in detail below.

```
virtual void setIntParameter(const int location, const int value) = 0;  
virtual void setFloatParameter(const int location, const float value) = 0;  
virtual void setVector3Parameter(const int location, const float x, const float y, const float z) = 0;  
virtual void setVector3Parameter(const int location, const Flwre::Core::Platform::Vector3 &vector3) = 0;  
virtual void setVector4Parameter(const int location, const Flwre::Core::Platform::Vector4 &vector4) = 0;  
virtual void setMatrix3Parameter(const int location, const Flwre::Core::Platform::Matrix3 &matrix3) = 0;  
virtual void setMatrix4Parameter(const int location, const Flwre::Core::Platform::Matrix4 &matrix4) = 0;
```

At the end, these methods pass the values of components defined in a determinate shader for a concrete render pass in **Graphics** module, to the graphic card. The components data can be different data types, for instance the direction of a light, that can be defined with a three component vector, so it would be used the **setVector3Parameter** method defined above in code snippet 1.2 to pass the vector data to the shader object within the render pipeline. Exist different methods obviously, since the different components data can be stored with vectors, matrices, integers and so on.

It's worth pointing out, that the method **updateUniformsContants** is executed in each render loop, and updating all data defined above, but also updates, all the matrices involved in a geometric transformations depicted in the [figure 4.10](#), passing them to the vertex shader within rendering pipeline, see [section 4.1.2.8](#) geometric transformations and the [figure 4.10](#).

4.2.6.1.2 Render pass

The render pass class is part of the class group to operate with effects within the engine, basically the render pass class maintain a unique pointer to a given shader see **section 4.2.6.1** shaders, and maintain a unique pointer to different render states implemented within the engine, see **section 4.2.5.1** render states. The render passes are stored within a render technique, see **section 4.2.6.1.3** render technique, because a determinate render technique could has different render passes.

Conceptually, a render pass can be seen as a single "draw call", the term "draw call" means exactly what it says: calling any graphic API function to draw, in OpenGL this functions has the following form **gl*Draw***, these commands cause that vertices to be rendered, and is where the kick off the entire rendering pipeline, see render pass class with an UML diagram below.

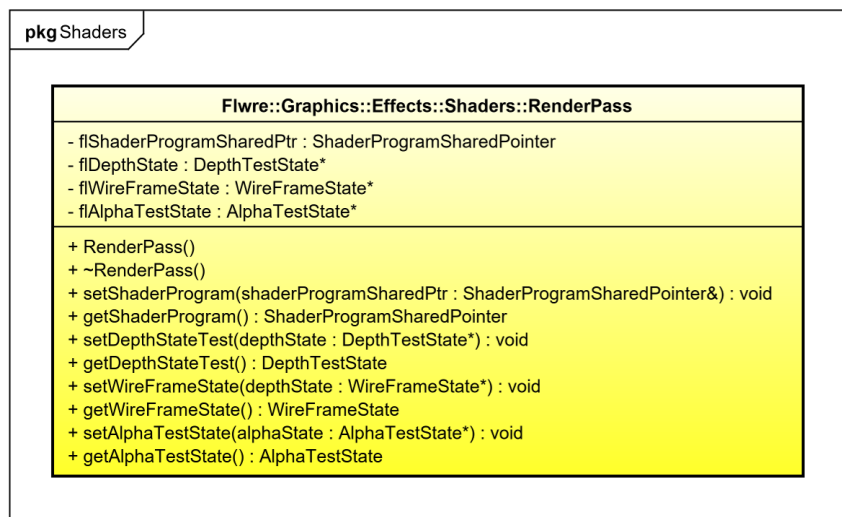


Figure 4.46 depicts the render pass class with an UML diagram.

Thus during the rendering process, the render passes are unpacked and applied to the objects to achieve a determinate effect in an object, since a given render pass is associated to a shader.

4.2.6.1.3 Render technique

There are many rendering techniques some may be more complex than others, is not the objective of this project implements a complex rendering techniques, but nevertheless in this project a mechanism is raised initially simplified that could be applied in more complex techniques in the future if is tuned up. Below is shows the render technique class with an UML diagram.

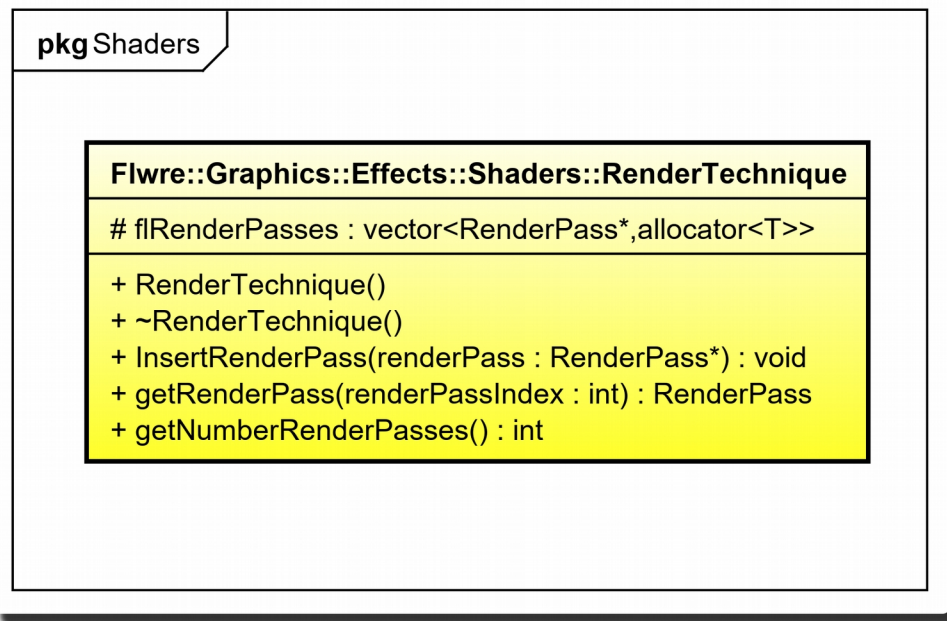


Figure 4.47 shows the render technique class with an UML diagram.

A render technique defines a determinate rendering technique inside the engine, a render technique can has one or different render passes see [section 4.2.6.1.2](#) render pass. To achieve this, holds a vector that maintain pointers to different render passes, and exposes methods to insert and retrieve this render passes with an index, at the same time, offers a method to retrieve the number of render passes stored for a determinate render technique, all of this, is stored when a render effect is defined see [section 4.2.6.2.1](#) render effect, and is unpacked during the rendering process.

Thus a determinate render technique can be defined within a render effect by one or different render passes, and each render pass, can involves a different shader. Thus in multi-pass techniques, different render passes over the same mesh occurs, rendering the same object multiple times, where each rendering does a separate computation that gets accumulated into the final effect over the mesh, thus each object, can be rendered with a particular shader, and then this is called a "pass" or "render pass".

It is worth remember here, that each render pass holds within it a shader pointer that is applied to a determinate mesh in a given render pass, so, depending the technique defined within a render effect this technique could have one or different render passes.

Nevertheless, to simplify the scenario, initially all render effects implemented within the engine has a render technique with a unique render pass. The following sections exposes how the render effects are implemented within the engine using all components defined above.

4.2.6.2 Local Effects

Mainly a local effect is an effect that is applied to a determinate object in a scene, the render effect class introduced in the following section, initially is designed to define a local effect template within the engine, so, with this mechanism is possible to implement different local effects as a templates, for later to do an effect instantiation at runtime.

4.2.6.2.1 Render effect

The render effect class, encapsulates all the relevant information and semantics for producing a desired visual result over a mesh, all components involved in a determinate effect separately way has been exposed in the sections above. This class group all of this as a local effects base class, so any local effect template defined within the engine should specialize this class. This works as an effects factory at runtime within the engine, creating a renderable effects instances, see **section 4.2.6.2.2** renderable effect. The following figure shows the render effect class with an UML diagram.

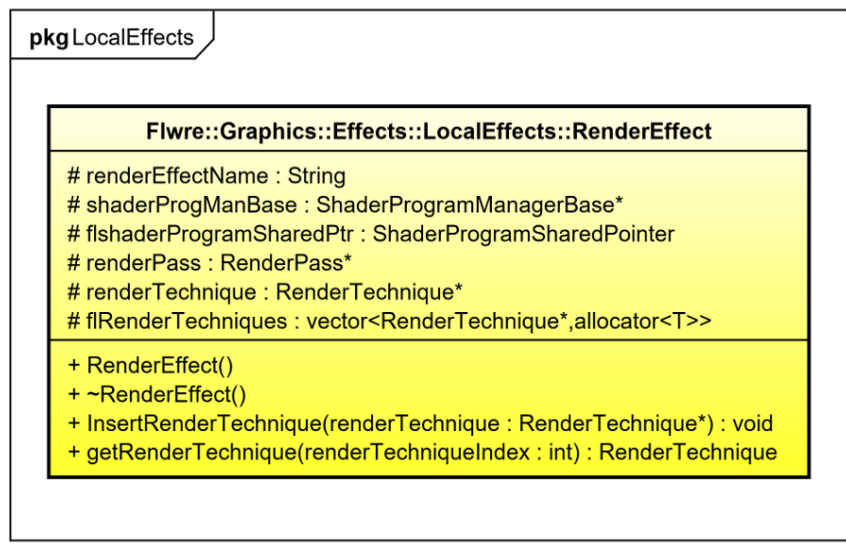


Figure 4.48 depicts the render effect base class with an UML diagram.

The UML diagram depicts the class structure, as can be seen firstly, keeps the name of render effect, maintains a pointer to shader program manager base, to be able create shaders within the render effect, and maintains a pointer to the created shader. At the same time, defines a pointer to the render pass and a pointer to the render technique, since at the end, a render effect has a determinate render technique which encompasses all other defined components inside.

All this structure defined in the base class, allows build a determinate render effect with a this class specialization, defining the shaders and the render passes involved in a render technique that they composes a determinate rendering local effect.

The following UML diagram, shows a simple effect specialization to depict the explained above, at the same time, is shown a simple render effect implementation in a code snippet to gain a better explanation understanding.

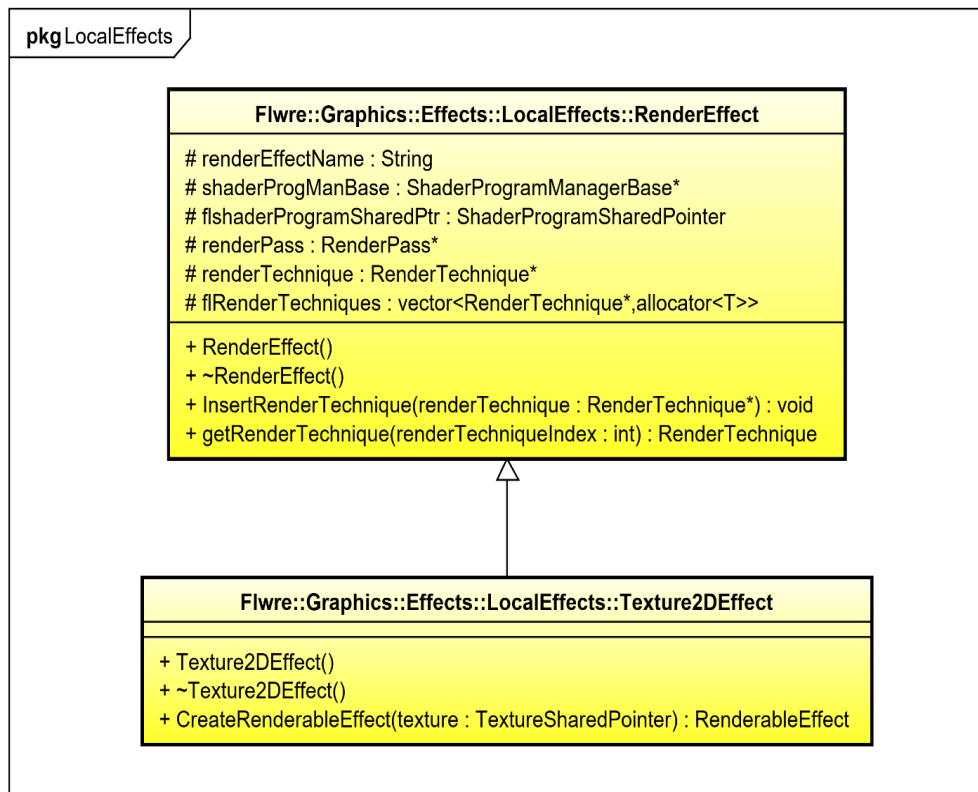


Figure 4.49 depicts the Texture2D effect specialized from render effect base class.

The following code snippet shows the [figure 4.49](#) implementation within the engine.

```

1. class FLOW_API RenderEffect
2. {
3.     private:
4.
5.     protected:
6.
7.         Flwre::Core::Platform::String renderEffectName;
8.         Flwre::Graphics::Utility::Managers::ShaderProgramManagerBase *ShaderProgManBase;
9.         Flwre::Graphics::Utility::Managers::ShaderProgramSharedPointer flshaderProgramSharedPtr;
10.
11.         Flwre::Graphics::Effects::Shaders::RenderPass *renderPass;
12.         Flwre::Graphics::Effects::Shaders::RenderTechnique *renderTechnique;
13.         std::vector<Flwre::Graphics::Effects::Shaders::RenderTechnique*> flRenderTechniques;
14.
15.     public:
16.
17.         RenderEffect();
18.         virtual ~RenderEffect();
19.
20.         void InsertRenderTechnique(Flwre::Graphics::Effects::Shaders::RenderTechnique* renderTechnique);
21.         Flwre::Graphics::Effects::Shaders::RenderTechnique* getRenderTechnique(int renderTechniqueIndex);
22. };
23.
24. class FLOW_API Texture2DEffect : public Flwre::Graphics::Effects::LocalEffects::RenderEffect
25. {
26.     private:
27.     protected:
28.     public:
29.
30.         Texture2DEffect();
31.         ~Texture2DEffect();
32.
33.         Flwre::Graphics::RenderableEffect*
34.         CreateRenderableEffect(Flwre::Graphics::Utility::Managers::TextureSharedPointer texture);
  
```

```

35. };
36.
37. Texture2DEffect::Texture2DEffect()
38. {
39.     renderEffectName = "Textures2DEffect";
40.
41.     flshaderProgramSharedPtr = ShaderProgManBase->createShaderProgram("texture2DEffect");
42.     flshaderProgramSharedPtr->setShaders("Shaders/textureVertexShader.glsl", "Shaders/textureFragmentShader.glsl");
43.
44.     flshaderProgramSharedPtr->setNumberTextures(1);
45.     flshaderProgramSharedPtr->AddUniform("texture1");
46.
47.     flshaderProgramSharedPtr->AddUniform("projection");
48.     flshaderProgramSharedPtr->AddUniform("view");
49.     flshaderProgramSharedPtr->AddUniform("model");
50.
51.     renderPass = new Flwre::Graphics::Effects::Shaders::RenderPass();
52.     renderPass->setShaderProgram(flshaderProgramSharedPtr);
53.     renderPass->setDepthStateTest(new Flwre::Graphics::Effects::Shaders::DepthTestState());
54.     renderPass->setWireFrameState(new Flwre::Graphics::Effects::Shaders::WireFrameState());
55.
56.     renderTechnique = new Flwre::Graphics::Effects::Shaders::RenderTechnique();
57.     renderTechnique->InsertRenderPass(renderPass);
58.     this->InsertRenderTechnique(renderTechnique);
59. }
60.
61. Texture2DEffect::~Texture2DEffect()
62. {
63. }
64. }
65.
66. Flwre::Graphics::RenderableEffect*
    Texture2DEffect::CreateRenderableEffect(Flwre::Graphics::Utility::Managers::TextureSharedPointer texture)
67. {
68.     Flwre::Graphics::RenderableEffect *flRenderableEffect = new Flwre::Graphics::RenderableEffect(this, 0);
69.     flRenderableEffect->setTexture(0, 0, texture);
70.
71.     return flRenderableEffect;
72. }

```

The **CreateRenderableEffect** method, see code line 66 above, acts as a factory creating renderable effects instances, see **section 4.2.6.2.2** and the **getRenderTechnique** method, see code line 21, that retrieve the technique during the rendering process, when the render effect is unpacked from a render transaction by the renderer, see **section 4.2.4.2** render transaction. The result obtained of apply the effect shown in code snippet 1.3 to a triangle mesh, can be seen in the **figure 4.29**, hence, this effect serves to apply 2D textures to the meshes.

Although the render effect base class has defined a vector of render techniques pointers, see code line 13, the engine only supports a unique render technique per render effect, this is set within the code, and initially only a render pass has been implemented inside each local effect proposed in this project.

The following section explains the mechanism to instantiate a render effect defined within the engine, with the mechanism explained above in factory way at runtime, when the effect is attached to a determinate object.

4.2.6.2.2 Renderable Effect

An effect instantiation within the engine is a renderable effect, this section explains how is created an effect instantiation from the effect template implemented in the render effect base class, and its specialized classes, such as seen above, in **section 4.2.6.2.1** render effect.

The effects are created with the factory pattern at runtime, when the effect is attached to a determinate object with the following command.

```
triangle->setRenderableEffect(texture2DEffect->CreateRenderableEffect(texture));
```

The method **CreateRenderableEffect** implemented in each render effect derived class with different signatures depending the effect, acts as an effect instantiation factory, creating an effect, to be attached to the object. The different signatures allows pass the components involved in a determinate effect, in the example has been passed a unique texture, but depending the effect, the components passed will be different, for instance could be a directional light, a point light and so on, or even a combination of them, so depending the effect, the components passed will be different.

A complete definition of a render effect has been shown in [figure 4.49](#) texture2D effect, and the code snippet 1.3, this is an effect template defined within the engine. The engine has other effect templates defined such as will be seen in the following section, hence, with this mechanism is possible add new effects to the engine modifying a small portion of the engine only.

When an effect instantiation is created with **CreatedRenderableEffect** method, this is attached to the object via method **setRenderableEffect**, the method is defined in the renderable object class, see [section 4.2.4.1](#) renderable objects and meshes, if the object is cloned, firstly maintains the same effect instance, unless is replaced by another effect, hence, if there are 100 objects cloned, all objects has the same effect instantiation and hence, everything points to the same effect.

When **CreateRenderableEffect** method is invoked, an effect instantiation is created inside, during the instantiation process, the render effect template is passed as a parameter, then the render technique is unpacked, jointly with the render passes involved, and is created a shader instantiation with the related parameters for each render pass defined in the render effect template, see code snippet 1.3 line 68 and [figure 4.49](#).

4.2.6.2.3 Effects implemented within the engine

This section explains the effects implemented within the engine with the system described in the previous sections. Obviously, it would be possible implement many more to achieve other visualization effect types, nevertheless, depending the effect implemented maybe the shaders system parser of the engine should be adapted, see [section 4.2.6.1](#) shaders and sub-sections, and concretely the [section 4.2.6.1.1](#) shader parameters data.

The following table shows and explains the effects implemented, the effect name is the class name that implements it, and implements a determinate effect is shown in the table illustrate the effect type that is.

Effect name	Description
DefaultEffect	Applies a basic effect with colours no parameters are needed, the colours are packaged within vertex elements. (vertex attributes), and the object is unlit.
UnlitColorEffect	Applies a basic effect with a color, the color is passed as a parameter and the object is unlit.
CheckerEffect	Checker Effect can creates a checker texture created procedurally, if is applied in a quad, it is possible to build a tiled surface, some scenes has this effect applied to created the scene subsoil. It is possible to pass as a parameter a texture as well.
Texture2DEffect	Used to apply a unique texture passed as a parameter.
DirectionalLightTextureDiffuseEffect	This effect applies a directional light with a texture passed as a

	parameters and calculate diffuse shading.
DirectionalLightMatDiffuseEffect	This effect applies a directional light with a material passed as a parameters and calculate diffuse shading.
DirectionalLightMatPhongEffect	This effect applies a directional light with a material passed as a parameters and calculate specular reflection.
PointLightTextureDiffuseEffect	This effect applies a point light with a texture passed as a parameters and calculate diffuse shading.
SpotLightTextureDiffuseEffect	This effect applies a spot light with a texture passed as a parameters and calculate diffuse shading.
SkyBoxEffect	This effect applies a sky box effect.
NormalMapPhongEffect	This effect applies a normal map with a phong reflection.

4.2.7 Data Types

The name space **Flwre::Graphics::DataType**, encapsulates the different data types implementation within the engine, with the different classes definition within the **Graphics** module, initially, defines a couple of data types explained in the following sections.

4.2.7.1 Transform class

The transform class encapsulates the model matrix, to be applied the geometrical transformations to the objects, this matrix has been depicted in the [figure 4.10](#) marked it in red, hence, the class contains methods to encapsulate translations, rotations and uniform and non uniform scaling. So, maintains the model matrix instance involved in each object, see [section 4.1.2.8.1](#) model matrix. Each renderable object keep a pointer to the transform instance built when the renderable object is created, initially the transform class maintain an identity matrix, so, the objects initially if not translated will be remain in the origin coordinates in the world space.

The method **setTranslate**, allows to translate an object passing as a parameter a three-component vector that indicates the world space position where is wanted to go, the method **setRotate** serves to rotate the objects passing as a parameters the angle to rotate in degrees and a three-component vector which indicates about axes wants to rotate.

To scale the objects a couple of methods has been defined, **setUniformScale** to do the uniform scaling passing a scalar as a parameter, and **setScaling** to do a non uniform scaling passing a three-component vector as a parameter, this three component vector indicates the values with different amounts along the x , y and z -axes to scale such as been defined in definition 4.7 in the [section 4.1.2.8.1.2](#) scaling.

The following code snippets shows how to apply this translations to the objects.

Apply a translation to an object.

```
torus->getLocalTransform()->setTranslate(Flwre::Core::Platform::Vector3f(2.0f, 4.0f, -2.0f));
```

Apply a non uniform scaling to an object.

```
torus->getLocalTransform()->setNonUniformScale(Flwre::Core::Platform::Vector3f(6.0f, 4.0f, 2.0f));
```

Apply a uniform scaling to an object.

```
torus->getLocalTransform()->setUniformScale(8.0f);
```

Apply a rotation to an object.

```
torus->getLocalTransform()->setRotate(90.0f, Flwre::Core::Platform::Vector3f(1.0f, 0.0f, 0.0f));
```

It is worth pointing out, that if the methods are applied in cascade for a determinate object, in really is being multiplying the model matrix with different geometrical transformations, to get the final model matrix applied to the object during the rendering process. Nevertheless, but it is pretty common to first scale the object, then rotate it and finally translate it.

As can be seen in the code snippets above, the vector passed as a parameter is a three-dimensional vector, but this vector in really is a four-dimensional vector, since has a homogeneous coordinate w , but as this value is one all the time, then is encapsulated within the mathematical library.

Finally the **getModelMatrix** method, retrieves a pointer to the model matrix to pass it to the vertex shader, see **updateUniformsContants** method in the **section 4.2.6.1.1** shader parameter data.

The following figure shows the transform class with an UML diagram.

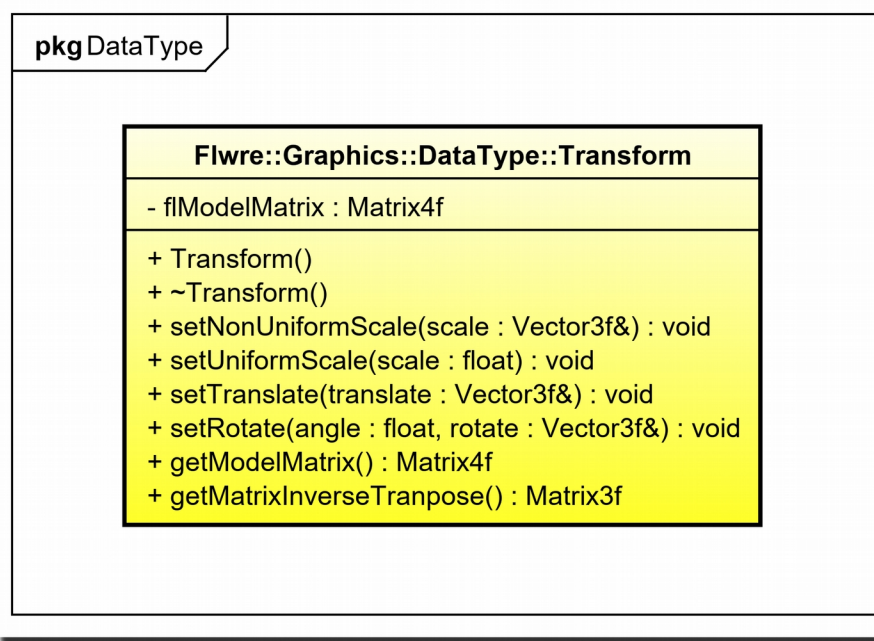


Figure 4.50 shows the transform class with an UML diagram.

4.2.7.2 Color class

The color class encapsulates the color definition as a four-component vector, that represents the RGBA components, at the same time, predefines a several basic colors encapsulated within different methods, see UML diagram. So, initially the main idea is to represent a color entity within the engine, the idea is that this data type should be implemented within any class that needs to define colors, for example, the material class, nevertheless for the moment not it is integrated into any class yet.

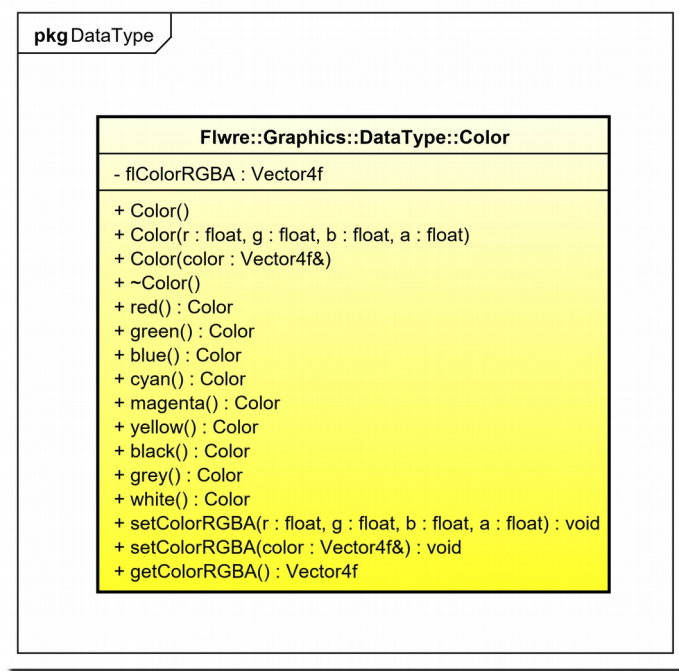


Figure 4.51 shows the color class with an UML diagram.

4.3 GraphicsOGL3 module

This module is defined within a namespace **Flwre::GraphicsOGL3** and holds the interface implementation defined in the **Graphics** module explained above in **section 4.2** graphics module, building the bridge between the rendering interface with a determinate graphic API, so this layer becomes a platform dependent, and in this case, is implemented with OpenGL3.3, but could be implemented with another graphic API as depicted in **figure 3.1**.

The engine has been designed to make this module interchangeable at runtime, see **section 4.1.4.1** dynamically loaded C++ objects and **section 4.2.5** renderer, between others graphic APIs implementations, if they were implemented, for manage to change the rendering behaviour between different graphics APIs at runtime, obviously the rendering must be stopped, swap the layer and initialize the render again with new layer loaded.

When the engine starts, this layer due to is implemented with OpenGL, is responsible to load the pointers to OpenGL functions, core as well as extensions at runtime, to get this, glad is used, see [\[glad\]\[OGLoadingLib\]](#). The reason for using glad, is because OpenGL is not stand-alone library, is a specification created by Silicon Graphics in 1992, so the specification implementation it depends the platform where developing for, since each vendor might has been implemented the specification in different manner in the graphic card driver for a specific hardware with a determinate features supported, thus the location of most of its functions is not known at compile-

time and needs to be queried at run-time. Then it is the task of the developer to recover the location of the functions and store them in function pointers for later use, the recovery of these locations is specific to the operating system, and doing it manually is a cumbersome process, then glad automates this process.

The next **section 4.3.1** defines a reference to the OpenGL functions used in this project, the functions are shown doing reference which section of this document they belongs to, but no more explanation is made, for more deeply compression, see the official OpenGL documentation or related tutorials.

4.3.1 Reference to the OpenGL functions used

```
//Vertex Buffer Section 4.2.1.4.1
glGenBuffers(1, &f1BufferId);
glBindBuffer(GL_ARRAY_BUFFER, f1BufferId);
glDeleteBuffers(1, &f1BufferId);
glBindBuffer(GL_ARRAY_BUFFER, f1BufferId);
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBufferData(GL_ARRAY_BUFFER, sizeInBytes, pointerSource, getGLUsage(f1Usage));

//Index Buffer Section 4.2.1.4.2
glGenBuffers(1, &f1BufferId);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, f1BufferId);
glDeleteBuffers(1, &f1BufferId);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, f1BufferId);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeInBytes, pointerSource, getGLUsage(f1Usage));

//VAOs Section 4.2.1.2 and Section 4.2.5
glGenVertexArrays(1, &VAO);
glDeleteVertexArrays(1, &VAO);
glBindVertexArray(VAO);
glBindVertexArray(0);

//Retrieve strings describing the current GL connection Section 4.2.5
glGetString(GL_VENDOR);
glGetString(GL_RENDERER);
glGetString(GL_VERSION);
glGetString(GL_SHADING_LANGUAGE_VERSION);
glGetIntegerv(GL_NUM_EXTENSIONS, &numExtensions);
glGetStringi(GL_EXTENSIONS, i);

//Create and destroy generic vertex attribute array (define an array of generic vertex
attribute data) Section 4.2.5
glVertexAttribPointer(indexLocation,size,attributteType,GL_FALSE,stride,bufferDataPointer);
glEnableVertexAttribArray(indexLocation);

//Renderer Section 4.2.5
glDrawElements(primitiveType, indexCount, GL_UNSIGNED_INT, 0);
glDrawArrays(primitiveType, 0, vertexCount);
```

```
//View port Section 4.2.5

glGetIntegerv(GL_VIEWPORT, GL_VIEWPORTParameters);
glViewport(GL_VIEWPORTParameters[0], GL_VIEWPORTParameters[1], width, height);

//Clear buffers Section 4.2.5

glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
glClear(GL_COLOR_BUFFER_BIT);
glClearColor(r, g, b, a);
glClear(GL_COLOR_BUFFER_BIT);

//Managing shaders Section 4.2.6.1

glCreateProgram();
glCreateShader(GL_VERTEX_SHADER);
glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(shaders[shaderType], 1, &shaderCode, NULL);
glCompileShader(shaders[shaderType]);
glAttachShader(id_program, shaders[shaderType]);
glLinkProgram(id_program);
glGetShaderiv(shader, GL_COMPILE_STATUS, &success);
glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &maxLength);
glGetShaderInfoLog(shader, maxLength, NULL, &errorLog[0]);
glGetProgramiv(shader, GL_LINK_STATUS, &success);
glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &maxLength);
glGetProgramInfoLog(shader, maxLength, NULL, &errorLog[0]);
glDeleteShader(shaders[shaderType]);
glUseProgram(id_program);
glUniform1i(glGetUniformLocation(id_program, name.c_str()), value);

//Shader Parameters Data Section 4.2.6.1.1

glUniform1i(location, value);
glUniform1f(location, value);
glUniform3f(location, x, y, z);
glUniform3fv(location, 1, vector3.data_ptr());
glUniform4fv(location, 1, vector4.data_ptr());
glUniformMatrix3fv(location, 1, GL_FALSE, matrix3.data_ptr());
glUniformMatrix4fv(location, 1, GL_FALSE, matrix4.data_ptr());

//Textures 2D Section 4.2.1.5

glGenTextures(1, &textureId);
glBindTexture(target, textureId);
glTexParameteri(target, GL_TEXTURE_WRAP_S, sampledWrappingParam);
glTexParameteri(target, GL_TEXTURE_WRAP_T, sampledWrappingParam);
glTexParameterfv(target, GL_TEXTURE_BORDER_COLOR, color);
glGetFloatv(GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT, &largest_supported_anisotropy);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAX_ANISOTROPY_EXT, largest_supported_anisotropy);
glTexParameteri(target, GL_TEXTURE_MIN_FILTER, filteringMinification);
glTexParameteri(target, GL_TEXTURE_MAG_FILTER, filteringMagnification);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_BASE_LEVEL, 0);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, 4);
glTexImage2D(target, mipmapLevel, textureFormatParam, width, height, 0, pixelFormatParam,
pixelTypeParam, dataImage);
glGenerateMipmap(target);
glActiveTexture(GL_TEXTURE0 + textureUnit);
```

```
glBindTexture(target, textureId);
glDeleteTextures(1, &textureId);

//View port Section 4.2.5, Handling keyboard and mouse events Section 4.2.3.1

glGetIntegerv(GL_VIEWPORT, GL_VIEWPORTParameters);
glViewport(GL_VIEWPORTParameters[0], GL_VIEWPORTParameters[1], width, height);

//Shaders Section 4.2.6.1

attributeLocationList[attribute] = glGetAttribLocation(id_program, attribute.c_str());
uniformLocationList[uniform] = glGetUniformLocation(id_program, uniform.c_str());

//Render States Section 4.2.5.1

glEnable(GL_DEPTH_TEST);
glDepthFunc(compareFunc);
glDisable(GL_DEPTH_TEST);
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
glPolygonMode(GL_FRONT_AND_BACK, GL_POINT);
```

5 Applications over the engine

Any application that uses the engine to render objects needs to define a class derived from the tool kit class, that has been exposed in **section 4.2.3** windowed application, see the **figure 4.30**. The class must override the following methods: **OnInitialize()**, **OnTerminate()** and **OnUpdate()**, see code lines 16,17 and 18 in the code snippet above. But nevertheless, obviously the application can defines many more methods if required.

The following code snippet shows the essential main structure that an application built over the engine must have. The application example below, draws a triangle onto screen.

```
1. class HelloTriangle : public Flwre::Graphics::Window::Application
2. {
3.     private:
4.
5.         Flwre::Graphics::SceneGraph::PrimitiveMeshShapes* triangle;
6.
7.     protected:
8.
9.         void CreateScene();
10.
11.     public:
12.
13.         HelloTriangle();
14.         ~HelloTriangle();
15.
16.         bool OnInitialize() override;
17.         void OnTerminate() override;
18.         void OnUpdate() override;
19. };
20.
21. HelloTriangle::HelloTriangle() : Flwre::Graphics::Window::Application("Hello Triangle",
    1024, 768)
```

```
22. {
23.
24. }
25.
26. HelloTriangle::~HelloTriangle()
27. {
28.     OnTerminate();
29. }
30.
31. bool HelloTriangle::OnInitialize()
32. {
33.     if (Flwre::Graphics::Window::Application::OnInitialize())
34.     {
35.         CreateScene();
36.     }
37.     return true;
38. }
39.
40. void HelloTriangle::CreateScene()
41. {
42.     triangle = new Flwre::Graphics::SceneGraph::PrimitiveMeshShapes();
43.     triangle->ColoredsTriangle().create();
44.
45.     this->getSceneHandler()->getRenderableObjectSet()->insertRenderableObject(triangle);
46. }
47.
48. void HelloTriangle::OnTerminate()
49. {
50.     delete triangle;
51. }
52.
53. void HelloTriangle::OnUpdate()
54. {
55.     this->getActiveRenderer()->ClearColorBuffer(0.0f, 0.05f, 0.21f, 0.0f);
56.     this->getActiveRenderer()->Draw();
57. }
58. FLOW_WINDOWED_APPLICATION_MAIN(HelloTriangle)
```

The methods **OnInitialize()** and **OnTerminate()** initializes and shuts down the engine and the application, see code lines 31 to 38 and 48 to 51. for the moment, the triangle deletion is implemented in the code line 50 but really this should be encapsulated within the engine.

In the **createScene()** method, all objects involved in a scene are defined and initialized, see code lines 42 and 43, in this case, a unique coloureds triangle is initialized, then in the code line 45 the triangle mesh (renderable object) is inserted in the linear render queue for later to be rendered.

The method **OnUpdate()** is executed during all rendering process, and is called in each frame, inside the method has been implemented the minimal necessary to render the triangle, see code lines 55 and 56.

The macro **FLOW_WINDOWED_APPLICATION_MAIN** in the line 58, encapsulates the application main entry point, at the same time, holds the mechanism to be able to exchange between different tool kits at compile time in transparently way, the macro must be defined in each tool kit header as **GlfwApplicationContext** or **SDL2ApplicationContext**, in case of being implemented. Finally, the macro must be present at the end in all applications.

5.1 Applications implemented to test the engine

To test the engine development has been implemented different scenes, see the [xfigueraTFM1219_Flwre_rendered_scenes_presentation.pdf](#), to see different screen shoots.

6 Summary

Along this thesis a real-time rendering engine has been exposed, the work has been focused to show a possible manner to design a real-time rendering engine as simplest as possible with solid foundations without losing a good scalability, but nevertheless, to achieve this simpler approach the efficiency has been penalized in certain manner, since has been implemented the essential pillars of the real-time render engine, nevertheless, the design allows to improve the efficiency without major changes in the main engine structure.

Although third-party mathematics libraries implemented for computer graphics exist, the engine implements an own mathematics library, the library is implemented with templates approach, this allows use the library with different data types, the library implements all essential necessary components used in computer graphics, like vectors, matrices, and geometric and projection transformations. The implementation has been done as simplified as possible, being a good starting point to learn how implement a mathematics library for computer graphics, however the library could be improved.

The engine has been implemented in layered fashion way, with this approach is possible to decouple the different level of responsibilities in each layer, a rendering interface via abstract classes has been defined, so that has done a total decoupling between the common responsibilities and the graphic API implementation with OpenGL 3.3, hence, with this approach is possible to implement the rendering interface with other graphic APIs in an easy way, specializing a common interface, and implementing it, with a specific graphic API.

A real-time rendering engine must be able to render scenes with different levels of complexity, without affecting its performance as the scenes becomes more complex, to accomplish this target, data structures are used which arranges the logical and often spatial representation of a graphical scene using a scene graph and spatial data structures, combining them with culling techniques to avoid to render geometry that is not visible.

However, to simplify the scenario, the engine not implements this techniques initially, in place, the engine implements a lineal render queue stored in a vector, without any kind of technique to determine if a geometry is visible or not, however has been explained and located on this thesis the techniques used to achieve it.

Scene graph implementation together with spatial data structures and culling techniques would improve the engine performance considerably, getting much more efficiency, the implementation of this mechanisms within the engine structure proposed, it shouldn't be hard extremely.

At the same time, although the engine supports indexed geometry, the OBJ importer build-in inside the engine, does not support indexed geometry, so that this importer creates non-indexed arrays that loading the vertices in directly mode without indexing it, this scenario works, but penalizes the rendering efficiency. To improve the rendering performance when the models are imported via OBJ importer, the importer should be improved to support the creation of indexed arrays when the models are imported from OBJ files.

Obviously the proposed project is not the only neither the better, simply is a possible manner to front facing this kind of project from scratch, from a simplified and solid point of view, and people with more experience than me, surely they could contribute with good ideas.

The project encompass an essential knowledge about real-time rendering and computer graphics field related, but due to the field extension, is impossible embrace more subject-matter in this pages for obvious reasons of time and dimension. But nevertheless, this project can be a good starting point, for the beginning of deeper challenges.

The rendering interface has been implemented with OpenGL3.3 only, to get a real-time rendering engine with a modern graphic approach, however, the engine design, along the layered approach allows to do an implementation with other graphic APIs like Direct3D or Vulkan in an easy way. At the same time, the layer that implements the rendering interface, it has been implemented as a plug-in, this allows to avoid the recompilation of the engine, in case to change the graphic API for rendering, being possible change it at run-time.

The engine offers an API to handle graphics in easy way, is a software development kit and that can be seen as a middleware, since it hides the complexity that exists in the real-time computer graphics development, and the developers that use the engine, they can focus on the business logic of the application, without diverting attention to rather complicated graphics rendering issues.

Finally, to deepen the concepts exposed in this thesis, is highly recommended to dive in the resources shown in **section 7** without which, this project would not have been possible, although in certain manner and in some cases, they tends to be more complex resources to digest.

6.1 Future Works

Below shows some the possible improvements.

Firstly, the most important improvement is the implementation of a complete scene graph system, along culling techniques, to improve the efficiency, for the time being, the engine render all the geometry, whether it is visible or not with a lineal render queue that is stored in a vector.

Other possible improvements are listed below.

- OBJ importer improvement, geometry indexing support.
- Implementing the rendering interface with another graphic APIs, like Vulkan or Direct3D.
- Quaternions implementation.
- Support text rendering on screen.
- Implementation a GUI, rendered on the viewport or windowed.
- Support for geometric shaders.
- Improvements in shaders parsing, find out another techniques.
- Global effects implementation and support for different render passes.
- Write a complete API documentation with examples.
 - Write mathematics library documentation.
 - Write the rest of the API documentation.
- Works on Linux.
- Among others ...

7 Bibliography and resources

7.1 Section 1

[3DfxVd1] Diamond Monster 3Dfx Voodoo 1

<http://www.vgamuseum.info/index.php/news/item/548-diamond-monster-3d-3dfx-voodoo-1>

[RTGraRenEn] Real-Time Graphics Rendering Engine, Prof. Hujun Bao, Dr. Wei Hua, Zhejiang University, Hangzhou, China, Springer 2011, Pag 8, Fig 2.1, Fig 2.2 and Fig 2.3

[PS4Arch] [https://en.wikipedia.org/wiki/Jaguar_\(microarchitecture\)](https://en.wikipedia.org/wiki/Jaguar_(microarchitecture))

<https://www.anandtech.com/show/6976/amds-jaguar-architecture-the-cpu-powering-xbox-one-playstation-4-kabini-temash/4>

[GEngArchPS4] Game Engine Architecture, third edition, 2015 CRC Press, Jason Gregory, Chapter 4 Parallelism and Concurrent Programming, section 4.3 Figure 4.18 Simplified view of PS4's architecture.

[GProceUnits] - T. Akenine-Möller and J. Ström, "Graphics Processing Units for Handhelds," Proceedings of the IEEE, vol. 96, Issue 5, pp.779-789, 2008.

[iPackMan] - J.Ström and T. Akenine-Möller, "iPACKMAN: High-Quality, Low-Complexity Texture Compression for Mobile Phones," in Proc. Graph. Hardware, pp.63-70, 2005.

[BHEG] - Nvidia Corporation, "Bring High-End Graphics to Handheld Devices," Nvidia white paper, 2011

[ListNvidia] https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units

[ListAMD] https://en.wikipedia.org/wiki/List_of_AMD_graphics_processing_units

[ListIntel] https://en.wikipedia.org/wiki/List_of_Intel_graphics_processing_units

[TiOmap3] Texas Instruments Inc. OMAP3 family of multimedia application processors
<http://www.ti.com/lit/ml/swpt024b/swpt024b.pdf>

[Qualcomm] Qualcomm Inc. <http://www.qualcomm.com/snapdragon>

[khg] <https://www.khronos.org/>
https://en.wikipedia.org/wiki/Khronos_Group

[OpenGLES] <https://www.khronos.org/opengles/> - https://en.wikipedia.org/wiki/OpenGL_ES

[DirectXhis] https://en.wikipedia.org/wiki/DirectX#Version_history

[FixOpenGL] A. Sampson, "Let's Fix OpenGL," 2nd Summit on Advances in Programming Languages (SNAPL 2017), vol. 71, pp. -, 2017
<http://drops.dagstuhl.de/opus/portals/lipics/index.php?semnr=16032>
<http://drops.dagstuhl.de/opus/volltexte/2017/7130/pdf/LIPIcs-SNAPL-2017-14.pdf>

[VulkanBench] VComputeBench: A Vulkan Benchmark Suite for GPGPU on Mobile and Embedded GPUs
2018 IEEE International Symposium on Workload Characterization (IISWC)
Workload Characterization (IISWC), 2018 IEEE International Symposium on: 25-35 Sep, 2018

[GraphicShaders] - Graphics Shaders, Theory and Practice, 2nd edition, Mike Bailey, Steve Cunningham
CRC Press 2012.

[PIC] https://en.wikipedia.org/wiki/Pixar_Image_Computer

[Reyes] https://en.wikipedia.org/wiki/Reyes_rendering

[RenderMan] <https://renderman.pixar.com/product>

[ShadeTrees] - Shade trees. Cook, Robert L. Conference | Proceedings of the 11th Annual Conference: Computer Graphics SIGGRAPH '84, p223-231, 9p. <https://graphics.pixar.com/library/ShadeTrees/paper.pdf>

[sgk] <https://www.ilm.com/vfx/young-sherlock-holmes/>

[slgh] Marc Olano and Anselmo Lastra. "A Shading Language on Graphics Hardware: The PixelFlow Shading Language." In *Proceedings of SIGGRAPH '98, Computer Graphics Proceedings, Annual Conference Series*, edited by Michael Cohen, pp. 159–168. Reading, MA: Addison-Wesley, 1998

[rtps] - Kekoa Proudfoot, William Mark, Svetoslav Tzvetkov, and Pat Hanrahan. "A Real- Time Procedural Shading System for Programmable Graphics Hardware." In *Proceedings of SIGGRAPH 2001, Computer Graphics Proceedings, Annual Conference Series*, edited by E. Fiume, pp. 159–170. Reading, MA: Addison-Wesley, 2001

[SysProgGrH] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. "Cg: A System for Programming Graphics Hardware in a C-like Language." *Proc. SIGGRAPH '03, Transactions on Graphics* 22:3 (2003), 896–907.

[CgTutorial] Randima Fernando and Mark Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Boston: Addison-Wesley Professional, 2003.
http://developer.download.nvidia.com/CgTutorial/cg_tutorial_chapter01.html

[DirextX9intro] Craig Peeper and Jason Mitchell. "Introduction to the DirectX 9 High-Level Shader Language." In *ShaderX2: Shader Programming Tips and Tricks with DirectX 9.0*, edited by Wolfgang Engel, pp. 1–61. Plano, TX: Wordware Publishing, 2003.

[GameEngineList] Game engines list, https://en.wikipedia.org/wiki/List_of_game_engines

[OffLineRenderersList] Offline renderers list, https://en.wikipedia.org/wiki/List_of_3D_rendering_software

[DKBT] <http://aminet.net/package/gfx/3d/DKBTrace>

7.2 Section 2

[GEngArchPipeLine] *Game Engine Architecture*, third edition, 2015 CRC Press, Jason Gregory, Chapter 4 Parallelism and Concurrent Programming, Chapter 11.2 The Rendering Pipeline.

[GEngArchGPUprog] *Game Engine Architecture*, third edition, 2015 CRC Press, Jason Gregory, Chapter 4 Parallelism and Concurrent Programming, Chapter 4.11 introduction to GPGPU programming, pag 349.

7.3 Section 3

[geomTools] <https://www.geometrictools.com/>

[3DGEArch] *3D Game Engine Architecture Engineering Real-Time Applications with Wild Magic*, David H. Eberly Magic Software, Inc. Morgan Kaufmann Publishers, 2005 by ELSEVIER Inc.

[fluentInterface] https://en.wikipedia.org/wiki/Fluent_interface

[dPatterns1] <http://www.blackwasp.co.uk/gofpatterns.aspx>

[dPatterns2] Gamma E., Helm, R., Johnson, R., Vlissides J.: Design Patterns: Elements of Reusable Object Oriented Software, Addison Wesley, 1995.

[dPatterns3] Head First Design Patterns O'Reilly Media, 2004 ISBN-13: 978-0596007126

[stdImage] <https://github.com/nothings/stb>

7.4 Section 4

[Torus] <http://mathworld.wolfram.com/Torus.html>

[Sphere] <http://mathworld.wolfram.com/Sphere.html>

[math3DCG] Mathematics for 3D Game Programming and Computer Graphics, Third Edition, Eric Lengyel, Course Technology, a part of Cengage Learning, 2012 – Chapter 4 - Section 4.4.

[songho] <http://www.songho.ca/math/homogeneous/homogeneous.html>

[fundaComGra] Fundamentals of Computer Graphics third edition, Peter Shirley (NVIDIA Corp.), Steve Marschner (Cornell University), et al. CRC Press 2009 Taylor & Francis Group. Pages 423 – 425 section 17.2.2 Interpolating Rotation.

[MathComGra] Mathematics for Computer Graphics, John Vince, Springer 2010. Page 80 Section 7.5 3D Transforms, subsection 7.5.4 Gimbal Lock.

[MathComGra1] Mathematics for Computer Graphics, John Vince, Springer 2010. Page 99 Section 7.8 Rotation a Point About an Arbitrary Axis, subsection 7.8.2 Quaternions.

[pipelineOGL3] https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

[persDivide] https://www.khronos.org/opengl/wiki/Vertex_Post-Processing#Perspective_divide

[coordSys] <https://docs.microsoft.com/en-us/windows/win32/direct3d9/coordinate-systems/>

[persDivide1] <https://www.learnopengles.com/tag/perspective-divide/>

[math3DCGBumpMap] Mathematics for 3D Game Programming and Computer Graphics, Third Edition, Eric Lengyel, Course Technology, a part of Cengage Learning, 2012 – Chapter 7 - Section 7.8.

[FGDevelopRendering] Foundations of Game Engine Development, Eric Lengyel, Published by Terathon Software LLC, 2019 - <http://foundationsofgameenginedev.com/#fged2>
<http://foundationsofgameenginedev.com/FGED2-sample.pdf>

[GEngArch] Game Engine Architecture, second edition, 2015 CRC Press, Jason Gregory, Chapter 1, section 1.6.4.1

[GEngArcha] STLPort - <http://www.stlport.org/product.html>

[GEngArchb] Boost - <https://www.boost.org/>

[GEngArchc] Loki - <http://loki-lib.sourceforge.net/>

[Visibility] <https://gcc.gnu.org/wiki/Visibility>

[DynLib] Dynamically Loaded C++ Objects William Nagel, 2005
<http://www.drdobbs.com/dynamically-loaded-c-objects/184401900>

[dlfcn] <https://pubs.opengroup.org/onlinepubs/7908799/xsh/dlfcn.h.html>

[dlopen] <https://www.tldp.org/HOWTO/pdf/C++-dlopen.pdf>

[windlfcn] <https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/>

[dlfcn-win32] <https://github.com/dlfcn-win32/dlfcn-win32>

[stdImage] <https://github.com/nothings/stb>

[assimp] <http://www.assimp.org/>

[OBJ-Loader] <https://github.com/Bly7/OBJ-Loader>

[OBJPaulBourke] <http://paulbourke.net/dataformats/obj/>

[OBJWikipedia] https://en.wikipedia.org/wiki/Wavefront_.obj_file

[MTLPaulBourke] <http://paulbourke.net/dataformats/mtl/>

[GLFW] <https://www.glfw.org/documentation.html>

[ViSurAlgo] James H. Clark, Hierarchical Geometric Models for Visible Surface Algorithms, Communications of the ACM, vol. 19, no. 10, 1976, pp.547–554.
<https://dl.acm.org/citation.cfm?id=360354>

[3DGEArchCam] 3D Game Engine Architecture Engineering Real-Time Applications with Wild Magic, David H. Eberly Magic Software, Inc. Morgan Kaufmann Publishers, 2005 by ELSEVIER Inc. Page 259, Section 3.5.1 Camera Models.

[fundaComGraCam] Fundamentals of Computer Graphics third edition, Peter Shirley (NVIDIA Corp.), Steve Marschner (Cornell University), et al. CRC Press 2009 Taylor & Francis Group. Page 141, section 7 Viewing.

[tuongPhong] https://en.wikipedia.org/wiki/Phong_reflection_model

[OGL4gls] OpenGL 4.0 Shading Language Cookbook – Second Edition – David Wolff. 2013 Packt Publishing.

[cubeImage] <https://www.flickr.com/photos/racchio/218529117/>

[pipelineOGL3] https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

[glad] <https://glad.dav1d.de/> - <https://github.com/Dav1dde/glad>

[OGLLoadingLib] https://www.khronos.org/opengl/wiki/OpenGL_Loading_Library

7.5 Other resources

<https://www.ogre3d.org/>

<https://www.cg.tuwien.ac.at/research/publications/2007/bauchinger-2007-mre/>

<https://www.cg.tuwien.ac.at/research/publications/2007/bauchinger-2007-mre/bauchinger-2007-mre-Thesis.pdf>

<https://research.ncl.ac.uk/game/mastersdegree/graphicsforgames/>

<https://learnopengl.com/>

<http://ogldev.atspace.co.uk/>

https://www.khronos.org/registry/OpenGL/index_gl.php