FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Game Engine for Location-Based Services

Tiago Pinto Fernandes

Master in Informatics and Computing Engineering

Supervisor: António Coelho (Auxiliar Professor)

June 2010

Game Engine for Location Based Services

Tiago Fernandes

Master in Informatics And Computing Engineering

Aprovado em provas públicas pelo Júri:

Presidente: Nome do Presidente (Título)

Vogal Externo: Nome do Arguente (Título)

Orientador: Nome do Orientador (Título)

30 de Junho de 2010

Resumo

O objectivo do presente projecto consiste em introduzir uma nova dimensão ao conceito de motor de jogo, através da implementação do conceito de *Location-Based Services* numa Framework de criação de jogos. Hoje em dia, existe um grande número de motores de jogo, comerciais ou gratuitos, disponíveis no mercado. Porém, poucos contemplam a possibilidade de tirar partido das capacidades e conceitos relativos aos serviços de localização geográfica existentes. Os que procuram utilizar estes conceitos, normalmente estão muito longe do que habitualmente é considerado um motor de jogo, devido ao facto de não conseguirem fornecer uma base de programação estável ou uma plataforma de desenvolvimento flexível para a criação de jogos, tendo usualmente por base plataformas Web nas quais os utilizadores devem actualizar o seu estado sempre que jogam um destes jogos.

Assim, o motor de jogo *Punkman*, descrito nesta dissertação, procura apresentar uma nova e inovadora forma de abordar esta problemática e fornecer uma plataforma estável e expansível para o desenvolvimento de jogos que possam utilizar as capacidades dos serviços de baseados em localização. O principal objectivo deste projecto consiste então em apresentar uma arquitectura que seja capaz de suportar um motor de jogo que possa ser utilizado em diferentes plataformas de hardware e software, que seja expansível e capaz de utilizar a informação geográfica e de posicionamento existente, tais como mapas ou pontos de interesse, na criação de jogos.

Porém, alguma parte da implementação é ainda descrita, assim como alguns algoritmos e comparações com outras implementações. Todos os pontos de expansão do motor foram testados e implementados pelo menos uma vez, de modo a garantir que o motor é expansível e simples de utilizar. O presente documento apresenta os detalhes destas implementações.

30th, June 2010

Abstract

The objective of the present project is to give a new dimension to the concept of game engine, by introducing location-awareness to a game creation framework. Nowadays, there are a great number of game engines available in the market, both commercial and non-commercial. However, few are those that contemplate and take advantage of the concept of location-based services. Those who use it can barely be considered real game engines, due to the fact that they do not usually provide a stable programming or developing platform for the creation of games. Rather, they consist more of web-based platforms in which the users must update their status and not necessarily based on specific hardware platforms.

The *PunkMan* engine, explained in this thesis, attempts to present a new and innovative approach to this theme and provide a stable and expansible platform for the development of games that may use the capabilities of location-based services. The main goal of this thesis is to present a suitable architecture for the game engine that can be used in different hardware and software platforms, is extensible and is able of using the geographic location, and underlying geographical information present, such as maps, points of interest, etc. to create games. However, the actual implementation of some concepts is also described, as well as the algorithms used and some benchmarking comparisons. All of the engine expansion points were tested and at least one concrete implementation was created in order to verify that the engine is expansible and easy to use. The current document presents the details on these implementations.

Acknowledgments

In the first place, I would like to present my thanks to prof. António Coelho for the support during the present project and for the help given when the work and time to develop this thesis was in serious risk. I owe my very special thanks to prof. A. Augusto de Sousa and prof. Fernando Nunes Ferreira for guiding me all of these years and for all the support and for introducing me to this amazing world of Computer Graphics and Imagery. Also, I want to thank to my parents who have always given me everything I needed (and more) to reach this point and have always been present when I needed. Finally, but not least important, I want to present my very special thanks to Mariana Teixeira, to whom this work is dedicated, for basically everything, but especially for being by my side in every occasion and helping me to face all of the problems that appeared along this long way.

Tiago Fernandes

Contents

List of Figures

List of Tables

Glossary

CPU - Central Processing Unit

GPU - Graphical Processing Unit

API - Application Programming Interface

PCI - Peripheral Component Interconnect

AGP - Accelerated Graphics Port

HLSL - High Level Shader Language

GLSL - OpenGL Shading Language

HDR - High Dynamic Range

FPS - First Person Shooter

MMORPG - Massive Multiplayer Online Role Playing Game

RTS - Real Time Strategy

GPS - Global Positioning System

GIS - Geographic Information Services

LBS - Location Based Systems

PNA - Personal Navigation Assistant

1. Introduction

The objective of this thesis falls upon the development of a multi-purpose game engine that provides programming facilities to create games in mobile devices that have the capacity of using location based services. A typical game engine is divided in two main components: the first one is the graphical engine, responsible for creating the virtual environment in either two or three dimensions and for providing the necessary tools to interact with this environment; the second component is the logic engine, whose responsibility is to control the behavior of NPC and all of the "business-logic" associated with the game concepts. In addition to these two layers, one more is to be added in the context of this project, which will be able to provide geographic location information to the engine, therefore introducing a whole new set of possibilities to the digital games that may be developed.

An engine of this type can be used for several applications, besides entertainment. One possible, desirable use is to make it so flexible that could be used to develop the whole any kind of location-based applications. For example, in the case of a GPS device, the engine could be used to replace the whole navigation software application. Nowadays, as the game industry evolves, being already one of the most profitable industries in the world, the technology associated with it also evolves in both hardware and software. Today, some mobile devices present characteristics and specifications that would seem impossible some years ago, even for large desktop computers. One example of such a device is the well-known iPhone, which is able to present high-end graphics and multimedia applications. Also, devices that feature Microsoft Windows Mobile are already starting to figure the mobile version of DirectX in order to provide a better graphical API for the development of visual-rich applications. Associated with all this, these devices are experiencing an amazing increase in terms of processing capabilities and storage which allows programmers to create more complex and heavy applications on these devices.

However, it should be still clear that, besides these advancements in technology, the development of such an engine for mobile devices is still a hard task to do, mostly due to the limitations presented by these platforms. The great majority of the existing API's is still very limited, consisting in a small and heavily adapted subset of the full featured API's that were originally built for desktop or laptop computers. So, the architecture of such an engine must be carefully thought in order to consider all these problems and provide a suitable performance for the application. Moreover, there is the necessity of adding a layer of geographical location system and intertwine this layer seamlessly with the remaining layers, which introduces a new level of complexity into the architectural design of the platform that is to be developed.

1.1 Motivation

Nowadays, the use of mobile devices that feature geo-localization possibilities is widespread across the world, namely through the use of mobile phones and GPS devices amongst others. These devices are usually used to provide information about the user's current position, nearby points of interest and a possible route from that position to some other point. However, this type of data can be used for purposes that are very different from navigation, such as gaming. Also, due to the fact that people are becoming increasingly attracted to outside sports, the creation of a framework that could provide the tools for creating a game and associating them with the capabilities of the location-based services could bring several benefits.

Also, although the number of mobile game players is increasing and the market is clearly becoming oriented to this type of gaming, there are still not many engines available for creating mobile applications and games. From the existing engines, even fewer take advantage of the existing geo-localization capabilities or are expansible enough to create new features and support new platforms.

Therefore, it seems clear that the creation of an engine that is able to take advantage from the existing location-based services, that is able of working in multiple platforms and that provides the possibility of being expanded for deployment in new platforms or to create new functionalities, is an innovative work that could even have a great impact on the market, especially on what concerns to mobile gaming.

1.2 Problem statement

The problem behind this thesis resides on the creation of an architectural solution and a methodology that is able of joining a game engine and the capabilities that are provided by the location-based services. Consequently, the engine must be able to work in multiple platforms, including mobile devices which may be able of using geo-localization systems. This is clearly one of the greatest problems that this concept brings, since a game engine groups many components that may be dependent of the actual hardware and software environment.

The architecture must contemplate the fact that new features may be required in the engine. So, the engine must be flexible enough to allow new capabilities to be easily added in the future, by extending existing components or simply creating new ones from scratch. Also, it is also important to consider that some devices may have features that others do not possess. For example, the engine must still work if some device does not have support for the geo-localization system. All of this must be accounted for when designing the architecture, in order to create a framework that can be used freely across platforms.

The engine that is to be developed must then fill this empty space in terms of game engines for location-based services, since the existing alternatives either do not contemplate the existence of geo-localization capabilities or, when these functionalities are included, they do not present the necessary expansibility or features that are required for the creation of games.

1.3 Objectives

The idea behind this project is to develop a flexible game engine that allows the development of games for mobile devices that can use location based services. Although the concept of location based gaming is not new, the creation of a game engine that can use this type of functionality is, by itself, innovative. Most of the existing location based games are very simple and played through web pages, in which the player must register its input of what happened while exploring.

Also, this project requires a very well planned architecture in order to comply with the expectancies of the final users, as the performance must match some mandatory requirements for usage. Besides, as this project is oriented towards mobile platforms, the existing resources in these devices are very limited when compared with the usual, full-featured computers that are available nowadays. But, and perhaps the most important, the engine must be sufficiently flexible to allow programmers to develop any kind of games, either these games use location based services or not.

Finally, a greater objective is to create such architecture that the engine can be easily expanded through plug-ins, maintaining its performance. The idea behind this concept is to allow that a simple, basic engine can be developed and used as the core application. Then, the rest of the features can be developed externally, while maintaining the old capabilities, thereby granting not only the expansibility, but also, retro-compatibility. Therefore, the engine could even be used as the basic infrastructure for other kinds of mobile applications.

1.4 Expected results

At the end of this project, it is expected that the resulting engine is capable of complying with all the requirements that were proposed. The engine shall be able to provide a higher-level interface over the mobile devices graphical capabilities, which must be able to aid programmers and developers to build complex and solid applications on top of it. While developing applications with the engine, the programmer or developer must feel safe to use it. In order to make such a thing possible, the engine must be built taking into consideration most of the possible error conditions that an user can fall upon, and try to avoid them, giving him the possibility to correct the error himself. The flexibility of the engine is also another expected feature that must be present in the resulting product. The engine must provide expansibility features that giver a developer the opportunity of adding new capabilities quickly, and with minimum knowledge of the remaining platform, if possible. Also, it is expected to be easy to maintain, and to do so, it must be well-documented, carefully organized and also possible to configure through external files, to avoid full system compilation.

1.5 Document structure

The present document begins with the current introductory chapter, in which an overview of the current work is presented. In these pages, the concept of the engine is introduced and the problem is described in detail, becoming an essential starting point for the rest of the document.

The second chapter presents a description of the state-of-the-art of the industry. It starts by presenting a brief history of the video-games and follows on by describing the evolution of the graphical hardware. Then it analyses what are the factors that make a game addicting in terms of human psychology related to gaming. Finally, an overview of the current game engine and location-based services technologies is presented, along with the current state-of-the-art for mobile gaming in general and for location-based gaming in particular.

The third chapter presents the suggested solution for the problem described above. The "*Punkman*" engine is introduced in more detail, by describing the architecture and the methodology used in this project. Each layer of the engine is described in greater detail in the following chapters.

Finally, the ninth and tenth chapters present the conclusions obtained from the project and the future work that can be developed in this area, respectively. This document ends by presenting the references that were used to create this work and produce the present document.

2 State of the art

2.1 History of video-games

In 1961 two MIT students, Martin Graetz and Alan Kotok, along with a MIT employee named Steve Russell, created the first video game ever developed. This represented the beginning of a long lasting evolution that reaches our very days. The game, named Spacewar, took around two hundred hours to create and was developed in a Programmed Data Processor-1 (PDP-1) [Way05]. The game mechanics consisted in two ships that had to destroy each other while trying to avoid being destroyed by a star, which occupies the center of the screen, and whose gravitational force attracts both ships.

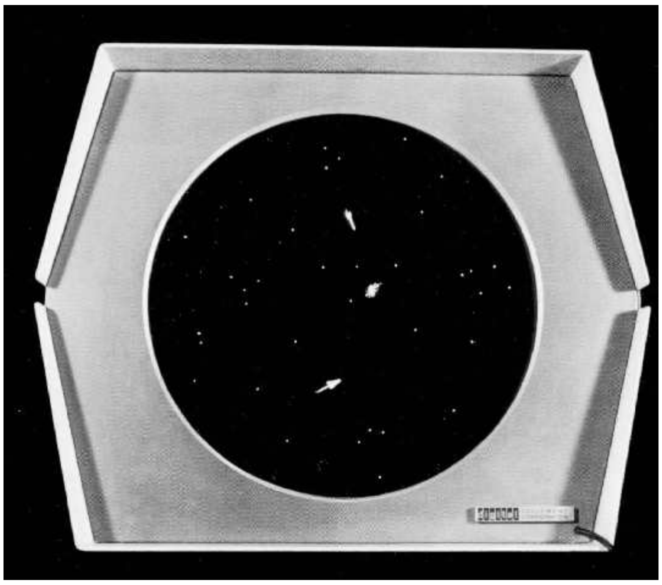


Figure 1 - Spacewar: The first video game

Spacewar represented the first stepping stone in a long way of computer game development. Eleven years later, the concept of video gaming was already a lot different than when it first appeared along with Spacewar. In 1972, Allan Alcorn developed the first profitable video game, distributed by Atari, called Pong. This game was a simple two-dimensional tennis game in which each player controlled one of the vertical pads and had to avoid that the "ball" (actually represented by a square) passed through to the screen border, exiting the field.

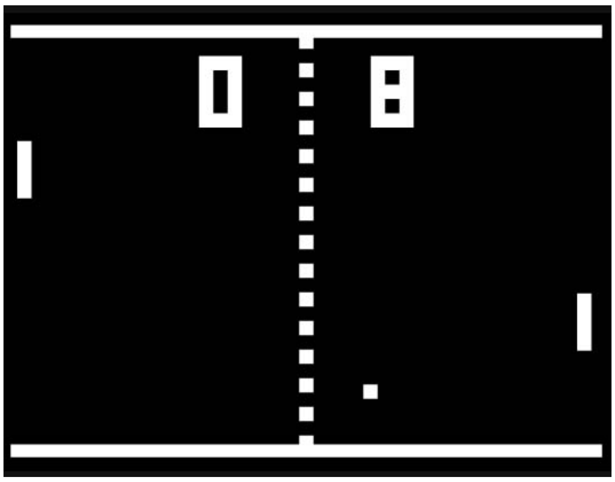


Figure 2 - Pong, the first commercially profitable video game

The outstanding advances in hardware and software technology that happened after this phenomenon uncovered a vast world of possibilities for game developers. Since the first steps given by Spacewar and Pong, a lot more happened during the years in the video game industry. In the beginning of this industry, Pong opened the way for other games in the market by being the first profitable video game ever. Amongst those other games, some titles have become true classics and remain as legends up until this day, being still quoted as some of the best games ever, such as Pacman, Tetris, Qix, Asteroids, Q-bert and so many others. When the graphical capabilities increased, but still considering the 2D era, some different styles of games started to appear. The industry faced a major differentiation in the genre of games being produced, because not only did the level of realism and detail increased, but also new types of gameplay have arisen. This was also due to the introduction of some brand new peripherics such as the joystick or the mouse, which contributed to increase the number of ways that a user could interact with a game. All of these factors contributed for the appearance of some truly remarkable games such as: “Beneath a Steel Sky”, which was one of the first point-and-click adventure games ever; “Sonic the Hedgehog”, which featured a blue supersonic hedgehog which had to save the animals that inhabited his planet and that were made captive by the evil Dr.Robotnik; “Street Fighter”, a beat-em-up game in which the player had to fight his way through a hierarchy of fighters ending up with the well-known boss, Bison; “Excite Bike”, a motocross racing game; and “Super Football”, a football simulation game.

Some time later, around 1981, the first 3D video games made their appearance and granted a very special place that lasts up until today. The most famous are undoubtedly Doom, Wolfenstein and Quake, which were brought to the market in the early 90's. These games were able to set new standards for the whole industry, creating not only a universe of game developers that would use these bases to build today's standards, but also an endless legion of fans that are still addicted to these games even in the present days. However, the games have evolved in a way that tries to achieve technological perfection with better, cutting-edge graphics and effects in order to keep up with the latest graphic cards and technological enhancements in either hardware and software. This can be seen in the most recent games, such as Borderlands, Crysis or FarCry 2, considering the shooting genre. If another genre, such as simulation, is considered, the very same basic gameplay is being kept over the years through games like “The Sims”, “Pro Evolution Soccer” or “Flight Simulator”. Whatever the genre is, the most recent commercial games are always the most technologically advanced, but not the ones that are funnier to play. And this is the very heart of this industry, that seems to be forgotten: “What makes a game fun and addictive?”.

Figure 3 – Borderlands

Figure 4 - Crysis

Figure 5 - Far Cry 2

2.2 Evolution of hardware

With the rising development of the video games back in the 70's, the computers where these games were run also had to take a step forward. The first computers lacked a graphical processing unit that was separate from the central processing unit. Consequently, all of the graphical processing was based on software sent to the CPU. As the processor was necessary for other important tasks, the room left for graphical processing, which is actually quite heavy, was very small. Bearing this problem in mind, IBM released the first video card in 1981 [Kum02].

With the uprising of the video cards and graphical processing units in hardware, two major API for programming computer graphics appeared. The first one, developed as an attempt to create some standards in this area of programming was OpenGL, originally named IRIS GL and developed by Silicon Graphics Inc. Later on, in 1994, Microsoft launched the Windows 95 operating system which brought with it the Microsoft's response to OpenGL, which was called DirectX.

Both of these API's were built with the objective of providing a simple, standard way of programming on top of the graphical hardware. However, the first graphical hardware did not allow much flexibility to program. To understand this, it is important to know what is a graphic pipeline.

A graphics pipeline consists in the sequence of steps that are necessary to transform a three-dimensional input scene into a rendered two-dimensional image. The most common steps of the graphics pipeline are:

- Modelling transformation
- Per-vertex lighting
- Viewing transformation
- Primitives generation
- Projection transformation
- Clipping
- Rasterization
- Texturing
- Shading
- Display

However, the graphics pipeline did not always figured all of these steps. In the first generation of 3D cards, which appeared in 1996 with the 3DFX Voodoo, the card did not do vertex transformations at all. These had to be done by the CPU. Nevertheless, it was already able to perform texture mapping and Z-Buffering. The second generation attempted to remove this overweight from the CPU and place all the transformation and lighting calculations in the GPU. Along with these changes, this second generation also provided facilities for multi-texturing which opened way for a whole set of new effects such as bump mapping, light mapping and many others. In other hand, the hardware interface was also upgraded to the AGP bus, which was up to eight times faster than the regular PCI slots and had a larger bandwidth [Gar08].

Up until this moment, the graphic cards provided some basic algorithms for lighting, clipping and transformation calculations. This architecture had several limitations, such as the fact of being impossible to create more than eight lights in a scene, or applying more than eight textures in an object. For an inexperienced reader, this may look perfectly sufficient. However, if one thinks about a driving game where each car has two head lamps. If the scene had only four cars, there could not be breaking lights, street lamps or even a sun that was not pre-rendered. The same goes for the textures with a very simple example. Consider a simple shooter where each player must defeat the others. If each gun blast produces a decal, representing a hole, when it hits a wall, then after seven shots there wouldn't be room for another texture, assuming there is already a texture on the wall. This would be perfectly normal if some other factors are to be considered like, for example, blood decals, stains, pre-lighting maps, bump maps, etc.

So, at this point it became quite clear that the fixed function pipeline architecture was not flexible enough to support the development of all the ideas programmers had back at those times. To overcome these problems, the third generation of graphic cards brought a very useful feature, called the programmable pipeline. Although it didn't allow huge modifications, it provided some basic ways to alter the standard, fixed algorithms that were built in the graphics card. A programmer could now create small programs written in an assembly-like language, called vertex shaders, which were run for each vertex as it was processed in the pipeline and produced the desired effects in lighting, transformations and texturing. Finally, to achieve greater control and precision, the room for shader programming was greatly increased in the fourth generation. Not only did the vertex shaders started to offer even more flexibility with the birth of higher level shading languages, like DirectX's HLSL and OpenGL's GLSL, but now the programmers could create this kind of shaders for every pixel and not only for every vertex in the scene. By introducing this pixel shading feature, a world of new effects can be created from simple phong lighting to complex rendering effects such as post-processing techniques like bloom or HDR.

These advancements in the computer graphics technologies ended up splitting the features of DirectX and OpenGL. The first noticeable difference comes mainly from the purpose that both of these API's serve in the present. OpenGL is more directed to learning and researching, due to the fact of being an open-source tool and being supported by a huge community. DirectX became the industry leader technology for both three-dimensional movies and game development, due to its commercially advanced features and side API's. Another huge difference can be seen while comparing both graphic pipelines. As it can be seen in figure 6 below, the OpenGL pipeline is divided in four main steps, corresponding to the vertices, primitives, fragments and pixel stages. The first stage deals with the vertex positioning and coloring while the second deals with the camera positioning and view clipping. The third stage is responsible for the introduction of fragments that can act over the rasterization, such as vertex and pixel shaders in DirectX. Finally, the last stage deals with the actual transference of the computed image to the frame buffer that is to be presented. [Ben97]

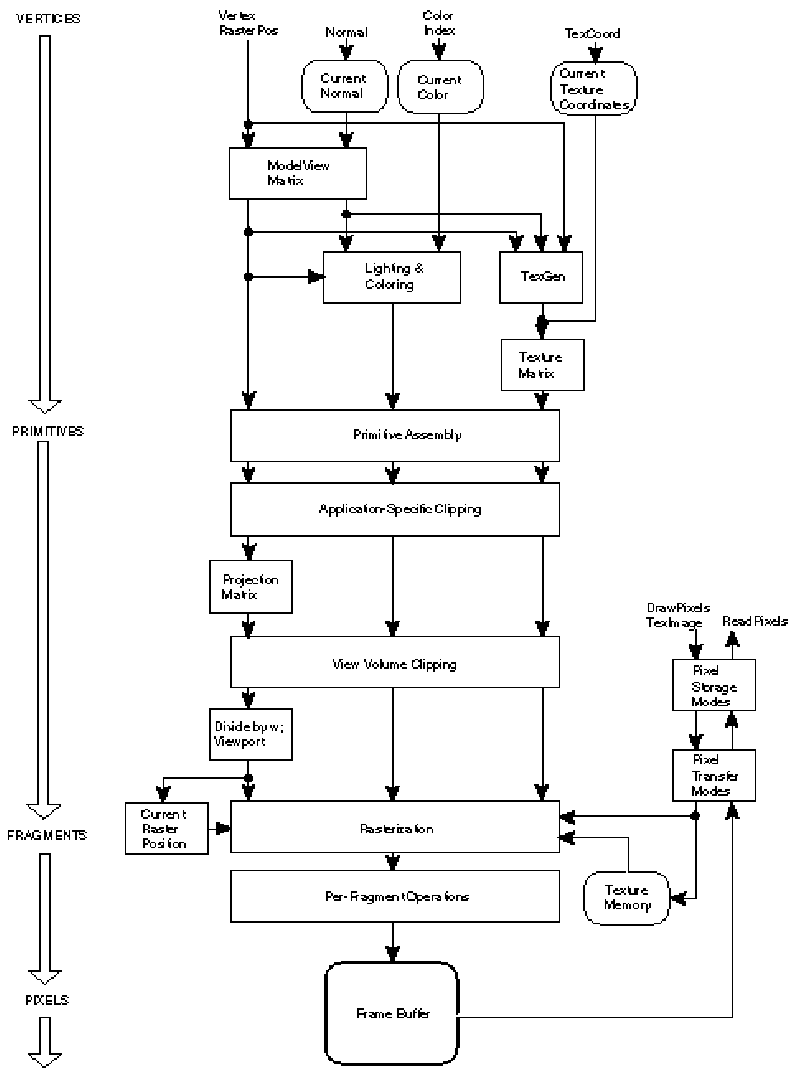
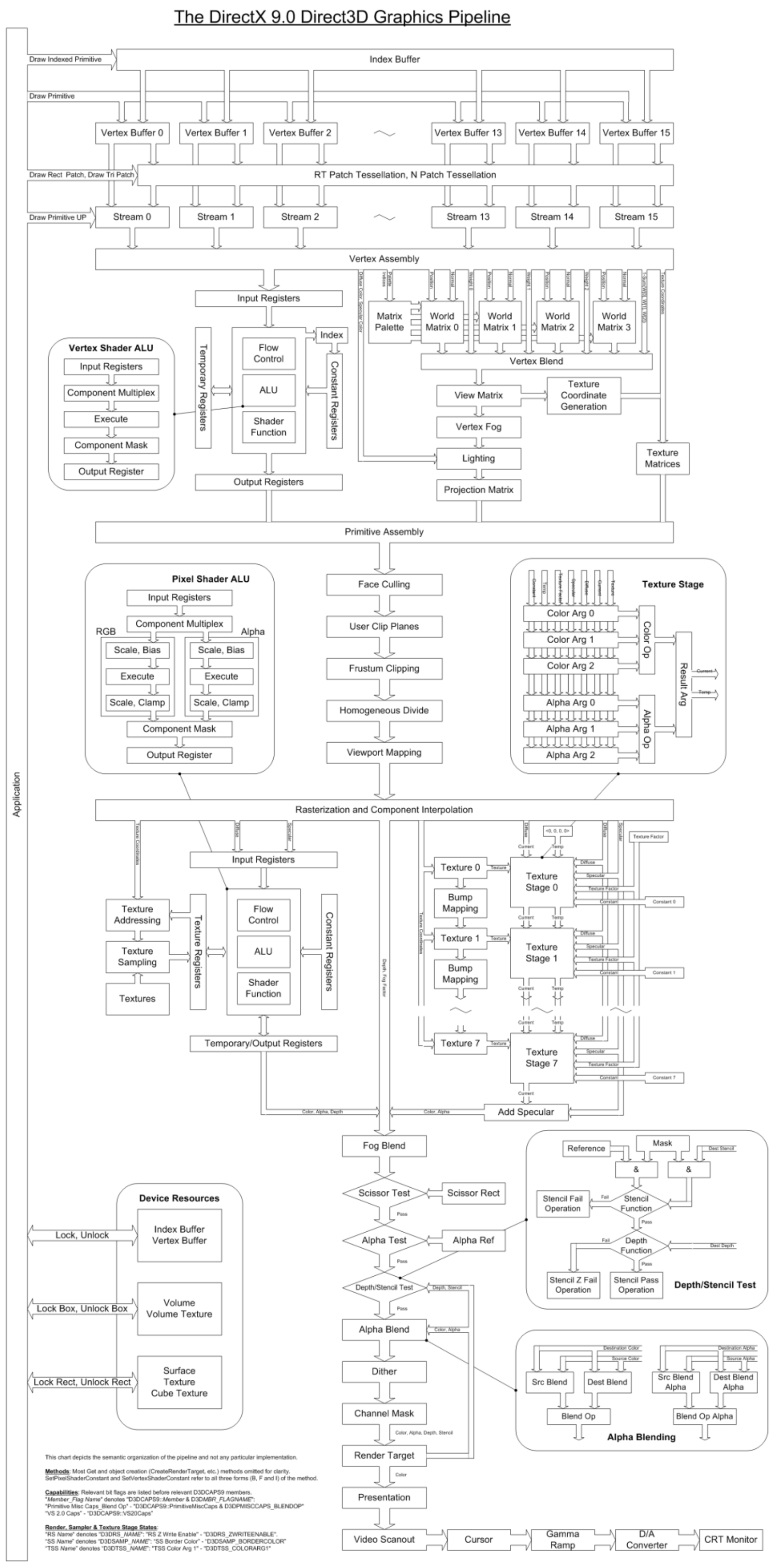


Figure 6 - The OpenGL graphics pipeline

Contrariwise, the DirectX pipeline, presented in Figure 7, features a greater focus on the shading pipeline. Also, it is visible that although the rendering stages are essentially the same as in OpenGL, all of the stages may be directly manipulated by the programmer, by introducing data at any time. The vertex stage is also quite different, as it is divided in vertex and index staging, which results in more optimized objects. After this stage, the vertices are automatically passed through the vertex shader ALU where they are processed directly in hardware, if possible. The data is then passed to the primitive assembly where the pixel shaders are applied in a first pass. Finally, the data goes to the renderization state where the textures are applied through the samplers and the final result is computed by applying fog, alpha, depth tests, dithering and is finally copied to the frame buffer. [Ric06]



This chart depicts the semantic organization of the pipeline and not any particular implementation.

Methods: Most Get and object creation (CreateRenderTarget, etc.) methods omitted for clarity.
SetPixelShaderConstant and SetVertexShaderConstant refer to all three forms (B, F and I) of the method.

Capabilities: Relevant bit flags are listed before relevant D3DCAPS9 members.
"*Member_Flag Name*" denotes "D3DCAPS9::*Member* & D3D*MBR_FLAGNAME*":
"Primitive Misc Caps_Blend Op" - "D3DCAPS9::PrimitiveMiscCaps & D3DPMISCCAPS_BLENDOP"
"VS 2.0 Caps" - "D3DCAPS9::VS20Caps"

Render, Sampler & Texture Stage States:
"RS *Name*" denotes "D3DRS_*NAME*": "RS Z Write Enable" - "D3DRS_ZWRITEENABLE".
"SS *Name*" denotes "D3DSAMP_*NAME*": "SS Border Color" - "D3DSAMP_BORDERCOLOR"
"TSS *Name*" denotes "D3DTSS_*NAME*": "TSS Color Arg 1" - "D3DTSS_COLORARG1"

Figure 7 - The DirectX graphical pipeline

Some differences are immediately visible when comparing the pictures above. As an example, while in DirectX the vertex shaders are computed and applied during the vertex processing phase, in OpenGL the vertex shaders are only applied after everything else was already calculated. This can make it slower, as some calculations will have to be redone during this process. The same happens for the pixel shaders.

But what technology was actually developer after all of the evolution described above? After passing from the fixed function pipeline to the programmable pipeline, the programmers had the capacity to develop new effects and visually stunning appliances that could not exist before. Nowadays, virtually every game uses a pixel or a vertex shader, even if it is a 2D casual game. The actual state of the art in what concerns these technologies includes effects such as Screen Space Ambient Occlusion, which consists in a shading technique that takes into account the attenuation of light due to occlusion, approximating the way that light radiates in real life; Parallax Mapping, which is an improvement of the normal and bump mapping techniques, that simulates the creation of geometry from flat textures, giving surfaces some apparent depth where in reality there is not; HDR which consists in the process of adding new shades and degrees of luminance to a scene, providing more contrast between the lighter and darker areas of a scene; and global illumination, which is an alternative to the default scanline rendering process, in which every ray of light is traced throughout the scene to achieve greater levels of realism at the cost of computational cost.

2.3 Psychology of gaming

Why do people like playing games and why are some games more addicting than others? Throughout the history, many games have drawn people's attention and became notorious milestones in the world of gaming. Some examples of these phenomena are Tetris, Diablo, Mario Kart, Starcraft, and much more recently, World of Warcraft. But why are these games more appealing to players than others? What can a game developer do to make his game become a world class blockbuster?

The reason for this may reside in some basic human psychology. In the first place, the human being is inherently competitive and feels the need to constantly compare himself with the others, and feel that he is or can be better than the rest. This behavior arises even in cooperative actions such as working in a team. For instance, in these cases, there is a huge tendency to fight for the leadership of a group, even if there is already an elected leader or if there is the need to find a natural emerging leader. In any of these cases, there will be competition for the position, either it is an official rank or not. This behavior is a part of the human most basic instincts, and is present in nearly every action that a person makes in its quotidian life. Related to this, comes the need to feel recognized by its actions, to earn the respect of the others that surround him. Once again, this is easily noticeable when working in a team. A good team leader knows that recognizing a teammate's work will make that collaborator become more motivated and increase its integration within the team. To understand why this is so, it should be enough to see that self-esteem, confidence, achievement and respect belong to one of the highest ranks in the Maslow's Pyramid of the Hierarchy of human necessities presented below, and all of these characteristics can be achieved through a simple gesture of acknowledgment for one's actions. [Env09]

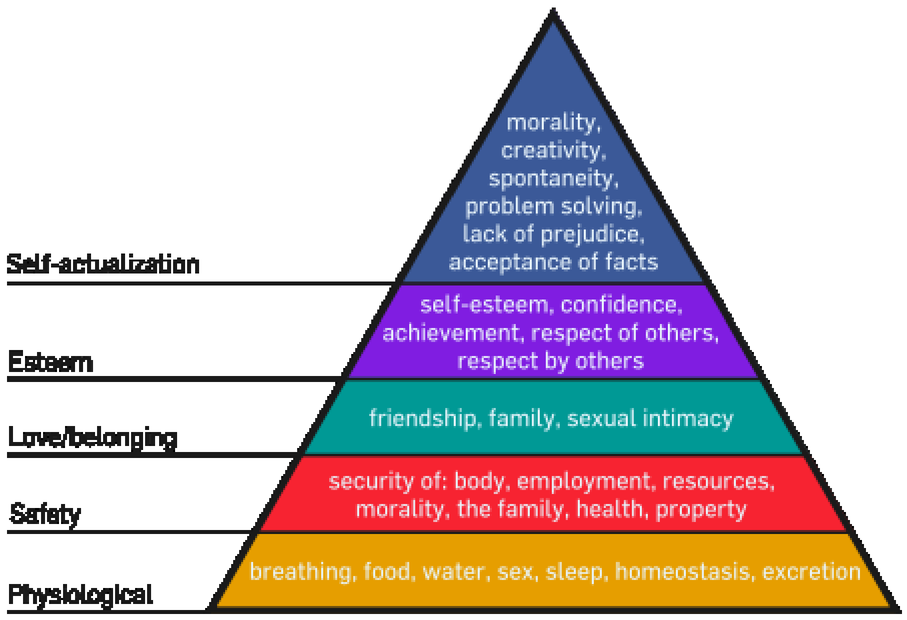

Self-actualization
morality, creativity, spontaneity, problem solving, lack of prejudice, acceptance of facts
Esteem
self-esteem, confidence, achievement, respect of others, respect by others
Love/belonging
friendship, family, sexual intimacy
Safety
security of: body, employment, resources, morality, the family, health, property
Physiological
breathing, food, water, sex, sleep, homeostasis, excretion

Figure 8 - Maslow's Hierarcy of human needs

But there is also another important fact that is closely connected to the points mentioned before. Every person feels the need to have power over something, may it be spiritual, political, social or economic either over individuals or communities. Power is associated with responsibility, but also with self-esteem and an increase in power usually means an achievement in some given area. Also, more power usually also means more competition, and that acts as a challenge to achieve more power, therefore entering a vicious cycle that can lead to a form of addiction.

And how does this relates to games? The answer is quite simple, as nearly every existing game tries to fulfill at least one of the topics mentioned above. Competition comes usually with multiplayer games, or with games that are single player in its essence, but that are able to keep the track of the player's score. In this way, a player can compete directly with others through a network connection, physical multiplayer methods (such as split screen) or by trying to be the best in the ranking. Examples of this type of gaming are the online FPS such as the well-known Unreal Tournament. Individually or in a team, the player must accomplish some objectives that are dependent on the game mode chosen. Also, through this form of competition, the player is automatically rewarded and recognized for his efforts, by its ranking or even by its team position. However, some other games, such as MMORPG give the player some other type of rewards. In this area of gaming, the player is usually rewarded with experience points. The experience points are used to level up the player's character and to unlock some extras, new equipment, new characteristics and features, new areas, etc. By this method, the user actually gets some "tangible" prize for playing and developing its aptitudes. Finally, through all these achievements, a player is able to exert power over the other players, either by being the head-master of some organization such as a guild or by being the team leader in a FPS or an RTS.

In the case of this type of gaming, in which an online component is involved, it is important to consider the factor of socialization. Although some games can be addicting to the point of making the player cut its relationship with society from being so absorbed, this community component can be a counter argument to this fact. It is a widely known fact that some persons have died while playing computer games to due to extreme addiction, and some others live only to play, giving games an extraordinary and overwhelming importance, which ends up isolating these players from society. This point can be argued by analyzing people's profiles, as most of these players probably suffered already from some sort of pathology that made them more probable to be addicted to something or some trauma in the area of social relationships and integration that led them to isolate from the rest of the people. There are records of players that used gaming as an escape for several other serious health problems such as depressions. However, when the idea of online game is considered there is invariably a component of social gaming involved. And in most cases, this factor just makes the game more appealing to the large majority of the players, due not only to the possibility of playing with other players, but also due to the increased competition.

Joining all of the factors mentioned above, computer games have the capacity of being real challenges for the human brain and senses. Besides the sensorial stimulus provoked by the sounds and visual effects shown on screen, most games also defy the humans very own way of thinking and reacting through challenges that are either mental, physical or both combined. In the first case, puzzle and adventure games take the lead quite easily. Although these genres are almost extinct nowadays in the commercial triple-A gaming market, they lead the casual gaming market by a great distance. These games combine a well-defined and strict set of rules that are usually easy to learn but hard to master and give users a series of challenges that must be faced to overcome a certain level. This induces a growing level of difficulty that, if inserted correctly in the learning curve, adds an addicting challenge to the game and makes the player want more. It seems clear though that to achieve such an effect, the difficulty must be adequate and the problems to solve must be solved by using a logical and coherent approach. Non-logical puzzles or an incoherent gameplay leads to frustration, that makes players quit very quickly. This type of gaming, usually related to casual gaming as mentioned above, is usually played by older people with less time to spend on a computer and that just want to quickly play something during a short break. On the other hand, there are the less mental, more physical games, based on reaction and instinct. These games compose the biggest slice of market in commercial and online gaming. Usually composed by few or no puzzles at all, the core of the action resides in being faster, stronger or simply have the correct timing. Although not so hard as the games mentioned previously, these games present a type of challenge that does not attract older players, but rather younger ones. The gameplay is composed by tasks that are usually so simple and repetitive that end up being mechanized by the person playing it in such a way that no mental process is involved. The only requirement is instinct. In this games, the surprises that the game is able to come up with, the game life (i.e., the amount of time that a game can be played until it is over), and the overall feeling of fun defines how good the game is. Usually accompanied by cutting edge graphics, it is commonly the challenge and competition that makes these games popular. For instance, Counter-Strike is still one of the world's most played and favorite game by FPS players and its graphics are not impressive in anyway according to today's standards. Also, the game theme exerts a relative importance on the game success before some players. While some prefer to play more realistic games such as America's Army or

Battlefield, other players prefer games that are less realistic and more related to fantasy, mythology or sci-fi, such as Unreal Tournament 3.

But considering a player's perspective, therefore more personal and less scientific, what does actually makes a game addicting and popular? The answer to this question seems to draw some controversy between the communities, but amongst all the different opinions, the gameplay seems to be the most important factor to the players. And it is also the factor that remains less altered throughout the years. Fortunately, some great examples of unique games have appeared in some very distinct moments in the time. The latest of these examples is "Braid". By being a 2D game that appeared in a market which is actually totally dominated by 3D games, some very low expectations were laid upon Braid's economical revenue. However, it quickly became a stunning surprise due to its exquisite gameplay. The base mechanics behind the gameplay of this title is as simples as the ability to go back in time. This might sound a bit cliché when thinking about some other titles such as the "Prince of Persia" trilogy. The big difference comes when the player must use this power to solve the puzzles throughout the game, especially because the same formula won't work twice and because the puzzles are extremely well designed. Although the art design is superb, this game stands out mainly due to its gameplay that really makes a difference. Another example comes from "De Blob". This game, developed initially as an academic project features an interesting concept of restoring the color to a city by painting it all over. The player controls a small ball of ink that must avoid the black ink police blobs and paint the city from upside down. The game is complemented with some objectives such as painting some special buildings, which are actually three dimensional representations of the city of Ultrecht where the game was developed. For its innovative concept of gameplay, "De Blob" has already deserved the compliments from the Tetris creator, Alexey Pajitnov, for being one of the most original games of all time. [Fil07]

Figure 9 - Braid

Figure 10 - De Blob

2.4 An overview of game engines

The first computer games were not responsible for the creation of the first game engines. Far from that, the first game engines have only appeared around 1995, when the video games started to gain real notoriety. Before this moment, the games were mostly hard-coded into huge, monolithic pieces of code, that were built specifically for one game and that could not possibly be used again for anything else. So, almost everything had to be built from scratch every time a new game was to be produced. The advent of some popular games, mostly first person shooters such as Quake, with multiplayer capabilities made the developers rethink their production processes. Due to the rising commercial needs, videogames evolved from a rather simple work to a complex set of industrial processes, grouped together as two main modules: the game design, which defines the set of rules and goals to achieve, the gameplay, the artwork, the mechanics and the story, and the game engine, which consists in the software that supports the game's implementation.

2.4.1 Concept of game engine

What is then a game engine? Put simply, a game engine is basically a set of reusable components that are game-independent and that serve as the basis for the production of videogames. To provide a more complete answer, a game engine usually consists of an API or a set of API's, complemented by visual designer tools that help developers in the creation process of a game. The most basic task of a game engine is to provide a way to manage a scene composed by zero or more objects. This is done in its main loop, where it must coordinate the rendering of every object, set the scene properties, obtain and process input, apply shaders and post-processing and perform internal management tasks. More advanced game engines provide several features that are extremely necessary, especially according to today standards. Probably the most important is the hardware and renderer abstraction layer. The hardware abstraction layer is usually provided by the renderer software (mainly OpenGL or DirectX) and should not be a concern of the programmers, unless it is their goal to develop their own rendering drivers. The renderer abstraction layer provides a way to draw and manipulate the objects in the scene without actually knowing what rendering driver lies underneath. Although this feature is not currently very used to implement both OpenGL and DirectX drivers simultaneously in a game, mostly because it is not profitable to develop commercial OpenGL-based games, it is still used to build a software layer over the DirectX drivers. If the DirectX changes somehow, all of the top layers remain unaltered, while only some minor changes may be needed at the renderer abstraction layer. Usually, game engines provide some type of data structure to maintain hierarchical and structural order in the scene. The most widely used technique is rendering with scene graphs. A scene graph is basically a graph, composed by at least one root node, which holds scene information in a hierarchical manner and renders it in the correct order, applying the transformations, materials, textures and every other property accordingly through the structure.

This feature is very important to maintain an order and a way to manage the scene easily without having to account for a great amount of lost objects throughout the program. The game engine must also grant space for every other game related entity, such as objects, cameras, materials, textures, artificial intelligence, game logics, etc.

2.4.2 Game engine architecture

With this said, what is the usual architecture of a game engine? The starting point involves the core systems of the engine. This module will support every other directly or indirectly through a set of useful functions that comprise the most essential tasks of the application and should therefore be split in several smaller modules. The first of these modules is the low-level system that is responsible for providing an abstraction layer over the hardware, the programming language and possibly even the existing software platform as good as possible. This layer is composed by the implementation of the most widely used and common data structures such as arrays, hash tables, lists, stacks, queues and strings. By using data structures developed specifically for this purpose rather than the language original more generic libraries, a higher performance can usually be achieved due to the removal of unnecessary code from the compilation and from the runtime application. Still in the low-level system, some concerns must be directed to the system specific specs such as the byte endianness, which may vary between different platforms. Also, if intended, platform-specific concepts can be encapsulated in this layer, providing support for multiple operating systems if desired. Some other basic but important functions can be included in this layer such as a system time management module, file handling and a system for custom memory management. In parallel with the low-level system comes the mathematics system. This will be a crucial part of the engine, as most of the computer graphic subjects end up falling in the realm of mathematics sooner or later. This layer must take care not only of the basic mathematic functions such as sine, cosine and square roots but also provide a set of structures that can be used later on, such as points, vectors, quaternions and matrices. Finally, still regarding the core systems, a memory management layer may be implemented to provide fail safe capabilities to the engine as well as fast resource allocation and deallocation, smart pointers and a basic object API system. Above this layer comes the renderer layer. This consists in a set of functions that are actually responsible for presenting the images on the screen. The most basic chore of this layer is to provide a way to group objects in a scene. As mentioned above, the most commonly used structure is a scene graph that interrelates each scene node with some others to build a logical data structure for representing the scene. It is convenient to notice that a scene node can be basically anything that is somehow related to the scene, for example, an object, a camera, lights, textures, etc.. However, the geometry must support transformations and so, it is necessary to keep the geometric state for each transformable object in this layer, so support must be provided for this feature. In order to support collision detection, bounding volumes must also be computed at this stage for each object and considering every transformation for the current object. Bounding volumes are usually boxes or spheres created to overlap the limits of an object to provide a way to determine if there was a collision between two objects by comparing geometry that is much simpler than the original, to avoid unnecessary calculations. This layer of abstraction is also responsible for setting the rendering states that determine the overall parameters and properties of a scene, such as the global state, lighting and texturing modes and much more. Finally, is responsible to set up the camera systems for the current scene and set its basic parameters in order to render a scene according to the developer needs. [Dav05]

The later characteristics sum up the basics of what a game engine should provide. On top of the layers described above, much more could appear and are usually necessary for a more serious engine. Nowadays, game engines usually implement advanced scene graph features like dynamic detailing. For example, a small object that is rendered far away from the view point does not need to have so much detail as an object that is very near the camera. Consequently, the distant objects can have less detail than the ones that are closer to the camera. Therefore, the object's detail level must be adaptable in a dynamic way to provide full scene optimization. Also in what concerns optimization, further features can be implemented in a middle layer to improve the engine performance. For example, Binary Space Partitioning Trees (BSP Trees) are data structures that divide the scene space in nodes to improve rendering performance. Through this method, the space is divided into convex sets by hyperplanes that provide an implicit ordering to the scene, thereby improving greatly rendering times by rendering only what is strictly necessary. Another useful technique is the Portal Rendering. Through this technique, interior scene rendering performance can be increased by making each room as an independent model and grouping the interior as a graph where two rooms are represented by adjacent nodes if the rooms are physically connected. The result is that a culling system can be built around this model to draw only what is strictly visible from one room to the other, cleaning useless geometry that is not visible through the walls of one room to another, decreasing the number of calculations to perform and increasing the performance. Besides all of this optimization techniques, and using the basic engine presented before, new features such as terrain rendering or animation controllers can be easily implemented by recurring to the core systems implemented. [Ped01]

Finally, on top of everything described up until this point, some top-level features can be plugged into the engine. The most commonly implemented features are the custom visual effects, improved collision detection, artificial intelligence and a physics engine. The first feature is usually quite hard to implement because it requires fetching data from nearly every layer built into the engine. Special visual effects are nowadays commonly built with pixel and vertex shaders combined. However, when a vertex or a pixel is about to be treated by the respective shader, it does not know its transformed position in the screen, the position of the lights in the scene, or even anything about the neighbor vertexes or pixels. This information must be passed through a special data pipeline that is responsible to establish the data communication link between the main application and the shader. The problem resides in the fact that the world matrix is usually implicit in the engine, as well as the view and projection matrix, raw texture information, light positions and most of all, vertex and pixel information. Besides all this, every piece of require data resides in different layers, which obliges the programmer to be very careful when developing a new game in order to be able to maintain an architecture that is flexible enough to respond to this demands. In the other hand, collision detection does not traverse every single layer in the engine, but is nevertheless hard to implement. It would be an easy problem to solve if every individual triangle of an object could be compared with every other triangle in the scene for each frame to check if a collision happened and react to it. Clearly, this method induces a great overload of comparisons even for a rather small scene and is not usable for relatively large scenes containing some millions of triangles. So, the basic approach is to use bounding volumes for each object and check if, given the distance between the bounding volumes of each object, a collision could have happened. In the positive case, a deeper analysis could be undertaken if necessary. A more accurate test, given that the bounding volumes test indicated that a collision could have happened, would probably be a intersection based method. These methods focus in testing the collisions between linear components, either these are edges or vertexes, and the other object's triangles. All of this analysis must be supported by object to object collision detection methods and tested against spatial and temporal coherence verifications. In other perspective, physics and artificial intelligence engines are usually implemented in a relatively external way in what concerns the engine. The physics engine is usually plugged around each object in the scene, by saying: “This object here will be a rigid body and shall obey to these laws of physics”. Usually, the physics engine is an external module and only wraps its functionality around the already built engine, acting as an outer layer that can easily be attached or detached from the main engine, performing its calculations side-by-side with the main application. Finally, the artificial intelligence module is even more external, as it is implemented as scripts, created in small scripting languages such as Python or Lua and used directly by the entities inside the engine. Nowadays, the most used technique for building artificial intelligence within games is to build Behavior Decision Trees (BDT) in Lua Scripts and plugging them to the engine. BDTs consist in a set of nodes in which each node identify a state in the world and possible actions to respond to it. The main disadvantage of this method is that a large amount of states must be covered. However, the main advantages are clear, as this is a simple, non-expensive, efficient, clear, non-linear and complete way of achieving a rather competent intelligence in a game. [Dav05]

Currently, the game engine market is enriched by a series of competent projects that are thriving for their very own place in this area. They can be split upon two main logical categories, as they can either be commercial or non-commercial engines. Starting by the later, the world has known some great independent initiatives, from which the most well-known is probably the Ogre game engine. Although it is widely used in several projects, it is quite hard to work with, mainly due to its documentation that is hard to understand and lacks some updated steps on configuration and implementation details. Another well-known project is the Panda 3D, developed by Disney in partnership with Carnegie Mellon University. This engine has already been used for several Disney related projects and investigation, but as an open-source initiative is also widely used for the development of independent games. Featuring a great set of capabilities such as shader generation and python integration, utilities like performance measurement, and a coherent step-by-step documentation and examples, it consists in a easy-to-use but rather complex and complete game engine. However, the most noticeable project that has been growing through the years in the realm of freeware open-source game engines is Irrlicht. This game engine has drawn the attention of developers due to its extremely easy form of being used. It includes a huge set of libraries that abstract almost every detail of a game implementation. The core of Irrlicht is the scene manager which is responsible for loading almost everything, except for some things upon which the user has to have control, such as cameras. And that is one important characteristic of the Irrlicht engine, because it gives the programmer the chance to explore the engine capabilities to a point that is as far as he wants. Irrlicht can manage almost everything on its own, or give the control of everything to the user, if it desires to do so. Added to this comes a set of visual development tools and programming capabilities that transform this engine into a serious candidate for game development, even capable of matching against commercial engines. [Nik09]

In terms of commercial engines, the market is clearly more competitive but dominated by some well-known names. One of the biggest names in the industry is the Source Game Engine, which is the base for Half-Life 2 and one of the most complete game engines available. Used by many developers and companies to develop its own games, such as “Dark Messiah of Might and Magic”, it has proven to be a flexible platform for game development. However, it is already a bit old for the current standards, and is starting to show some signs of its advanced age. Crytek has given the world a capable adversary for the Source Engine, with some features that are much more advanced than the later, namely the CryEngine. Currently in its second version, this engine is the base of the Crysis game, produced by the same company. The main focus of this engine is the interactivity, the possibility of destroying pretty much everything that the player can see, and the capability of rendering huge terrain areas. The competence of this engine is recognized by the market, as the first version of this game engine was heavily modified by Ubisoft developers to produce the Dunia engine, which is the core of the FarCry 2 game. But the commercial span of this industry is still dominated by a game engine that exists for a long time and that is still responsible for setting the cutting edge in terms of computer graphics and game engine development. This engine is called Unreal Engine 3, developed by Epic Games. The visual game editor allows the rapid creation of not only new maps and the importation of new models and art, but is also supported by the Unreal Scripting Language which traverses the whole engine and allows the creation or modification of game rules and modes. Through this, not only the developers can quickly develop new games using this engine, but also the enthusiasts can create new game modes and modifications that can be shared and played by some other users. The power of this game engine is told by the amount of developers and games that are currently using it, for example: Gears of War 1 and 2, Alpha Protocol, America's Army, BioShock 1 and 2, Mass Effect 1 and 2 and, of course, Unreal Tournament 3. [Dev10]

2.5 Location Based Services

As the name implies, Location Based Services (LBS) are “information services accessible with mobile devices through a mobile network and utilizing the ability to make use of the location of the mobile device”. [Vir01] In other hand, a Location Based System can also be described as “A wireless-IP service that uses geographic information to serve a mobile user. Any application service that exploits the position of a mobile terminal” [OGC05]. Location Based Services can be considered as the result of the fusion of three different types of technologies, namely the Internet, New information and Telecommunication Technologies and Geographic Information Systems (GIS) [Bri02].

A LBS is composed by several distinct components. The first basic component is the mobile device that supports the LBS and acts as an interface between the user and the system. Through this mobile device, the user sends its request to the communication network. The network is the second necessary component to be able to use a Location Based Service as it supports the transfer of messages between the mobile device and the service provider. But logically there is a core component that must be present. As the main function behind a LBS is to provide a set of services based on the user location, it is clearly necessary to have a way to know the geographical position where the user currently is. To do that, the mobile device needs to be equiped or connected to a positioning component that is able to determine what the current location of the user is. This can be done by using a Global Positioning System (GPS) or the mobile communication network. However, due to the possible use of proxies, the later method is much more propense to failures or innacuracies. Finally, two more components are required. A service provider that adds new sets of functionality to the LBS, such as routing, position calculation, yellow pages services, etc. and a data content provider that is responsible for providing maps, location of points of interest and other position related information.

Location Based Systems can be divided in two main types. Perhaps the most common are the services that are directly required by the user, such as the GPS. In this kind of systems, the user asks directly for the service that is expecting to receive. For instance, when a user turns on the GPS device, it is because he wants to know his current position or how to reach some other location. Therefore, the request is sent from the user to the network on demand and gets back to him after processing the required information. This type of system is called a pull-based service as the user “pulls” the information to himself. In the other hand, some services are not requested by the user but attempt to reach him nevertheless. This is called a push-based service as the information is pushed to the users although it is not requested by them. Mobile advertising is a very popular service that uses this technique. Based on the user location some relevant advertising information is sent to the user like, for example, a message about discounts in a local store nearby. One other common type of usage for this kind of systems is the roaming information when one crosses the border between countries. Usually, when this happens, the mobile communication network detects that a foreign mobile card entered its authority region, detects its original country and sends a message with relevant information to the user, such as emergency numbers, prices, etc. [Ste06]

LBS can be used for several different purposes. The most widely known is navigation. By using a LBS through a Personal Navigation Assistant (PNA) such as a GPS device, and combined with routing and maps for a certain area, a user can quickly obtain directions, routing, avoid traffic and, of course, know its position. Associated to this comes the purpose of obtaining information. Usually, these services provide information about nearby points of interest such as restaurants, shoppings, tourism spots, hotels or even nearby people. Also, some devices are also equipped with the capability of planning tourism routes, travels or even shopping tours. Still in a leisure-oriented perspective, LBS can also be used for playing games like geocaching or mobile games that use positioning information of some sort and to find friends and start instant messaging conversations with other people. But LBS can also be used for more professional purposes. One widely know usage of this systems is tracking. Nowadays, the majority of the logistics companies already use LBS in their vehicles, connected to a central security station, for safety purposes. For example, if a truck is stolen, the information provided by the LBS device installed on the vehicle can be used to track down the truck and its cargo in a matter of instances. This kind of tracking capabilities is also useful in the case of emergencies. When a distress signal is emitted by someone that needs help, its position can also be immediately sent to the competent authorities that will be responding to the distress call and therefore provide a more efficient response. Still in a professional point of view, LBS can also be used for billing directly the users of a highway, by knowing its position and recent path, making the fares more just and accurate, based on the fact that with this information, the user would only pay for the exact amount of way that he actually travelled in that road. This method is also useful to reduce costs with employees and infrastructures throughout the toll stations in a freeway. Also, as mentioned before, advertising can also benefit from LBS. By knowing the location of a user, position relevant advertising messages can be sent to him in an attempt to provoke interest in the products, events, promotions or opportunities happening nearby its location. Finally, in a different perspective, knowing a user location is very important in the context of a virtual augmented reality environment system. This kind of systems provides an overlay of computer generated graphics over real life images to provide extra information about it. When this system is integrated in a live video feed installed in a helmet (or glasses) carried by the user, it is clearly necessary to know where it is located and in what direction it is looking and moving to provide an as-accurate-as-possible information about the scene that is being visualized. [FID09]

2.6 Mobile Gaming

The concept of mobile gaming made its public appearance with the Nokia cell phones in the year of 1997. In their 6110 model, Nokia introduced the first pre-installed mobile game, which is still played and installed in a great number of mobile phones nowadays, and goes by the name of Snake. The game's objective was to feed a small snake with small pieces of fruit that appeared randomly on the screen. For each fruit caught, the snake would grow a bit. If the snake bit its own body, then the game was over. In some versions, there was the option to turn on some walls to make the game more difficult, making the player lose if the snake hit any wall. The concept of the game was originally created in 1970 for the arcades and has drawn many attentions to it. However, it reached mass popularity only when it started being bundled with Nokia phones back in 1997.

But what is then considered mobile gaming? Mobile gaming consists in any game developed for any handheld device, including those that were not originally developed for playing games. This includes devices such as mobile phones, GPS, PDAs, smartphones or portable media players and gaming-specific mobile platforms such as GameBoy, Playstation Portable, etc.. It is then clear that mobile games face some problems that a normal computer game wouldn't have to consider. In the first place, the hardware architecture of a mobile device is much more limited than a normal computer's. Added to this, the mobile devices are usually not equipped with as many input possibilities as a computer, either in terms of keyboard limitations (although some devices already have a full keyboard layout available) or in terms of existing peripherals. Due to this, the software development for these platforms is restricted to a limited set of libraries when compared to what a programmer is used to for developing software on a full featured computer platform. Although some other platforms are available, the most commonly used for the development of applications for mobile devices are based upon Windows Mobile, Palm OS, Symbian, Adobe's Flash Lite, Java ME, DoCoMo's DoJa and Qualcomm's BREW. The limitations presented above are responsible for an important characteristic of the games developed for mobile platforms. To be able to impose in this market, which has gained a huge notoriety since its commercial appearance with Snake, the games focus in one of two main aspects: a differentiating gameplay and not necessarily great graphics, that makes it very addictive and fun to play, thereby standing tall amongst the rest of the games; or being based in a very strong and popular license that has already thrived its place in the market, such as Assassin's Creed, Tomb Raider or Batman. Usually, the later are based on recent movies to promote themselves just by relating to the name of the movie.

Looking at gaming through an economic perspective, this industry is proving to be one of the most profitable sectors even in with the difficulties presented by the actual economic context. This is due to the fact that, given the actual standards, people turn to gaming in a way to improve their quality of life but also as an escape of the real life difficulties. Besides this, mobile games are usual fun, addictive and quick to play, being therefore considered casual games. As people start to have less free time, they are interested in playing something quickly, that can be fit into a small span of time, but that is at the same time both fun and challenging. Also, people are not always interested in physically going to a game store to buy a copy of a game, and probably they don't even have time to do so. Consequently, these gamers choice fall upon the casual games, which are easy to learn, hard to master, are usually very fun and addicting and can be downloaded from the internet or from some mobile service provider portal. So, a successful mobile game will probably be either easy to play, interruptible, based on subscriptions, have some sort of social interaction capability or able to take advantage of mobile innovation. To fully understand the mobile gaming industry, it is interesting to take a look on how this business value chain actually works. The first entities on this market are the developers, which are responsible for creating the applications and make them available to the industry. However, most of the times, these developers are not capable of handling the mass distribution of their products and are therefore recruited by or work for a publisher. The publisher works as the entity that organizes the products from several developers and injects them into the market. Publishers put a margin over the original prices, which will be their revenue, but will also certify the game's quality and correctness. After this process is completed the publisher's name will be put upon the product, giving it a stronger market impact. One good example of this, are the Gameloft games which are recognized for their quality and for being very addicting and fun. Usually, the publisher sell the games to wireless carriers, which are the core of mobile gaming and application distribution market, as they control not only the network, but also the information that reaches the end-users. In the end of the value chain are the customers, which represent the end-users of these applications. As the market is characterized by using a push-based model, the customers usually search for an existing product in the publisher or wireless carrier website and download it to their mobile devices, paying the subscription or the price of the application, which includes the margins of every entity that is present on the value chain. In parallel to these entities comes the mobile device manufacturers, which create the actual handheld devices and are responsible for introducing innovation and new capabilities on mobile platforms. By their relative position on the value chain, they represent an independent force between the developers, publishers and carriers, and the customers. [Mic03]

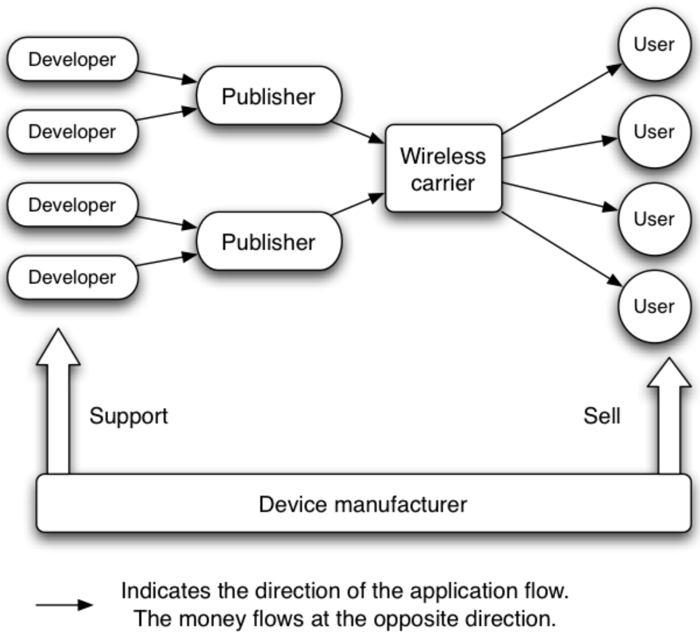


Figure 11 - Mobile gaming industry value chain

New trends are starting to appear in the world of mobile gaming. Nowadays, these devices are growing in complexity and some handhelds with advanced processing, graphics and networking capabilities are starting to appear in the market. This has opened several new possibilities in the area of mobile gaming. Maybe the most notorious is the capability of creating multiplayer games for mobile devices by using wireless technologies such as infrared, bluetooth, GPRS, 3G or even Wireless LANs. From simple games, such as chess, checkers or TicTacToe to more complex games such as golf, racing games and even MMORPG. In a matter of fact, MMORPG are experiencing an impressive growing for the mobile market, as the companies start to create more and more sophisticated games for this platform, or providing the capability of managing computer MMORPG characters, quests and inventory through a mobile device. For example, some applications already give the user the capacity of managing and automating some World of Warcraft tasks through a mobile application. In another perspective, devices such as the iPhone present very advanced graphical possibilities combined with different types of interaction capabilities such as accelerometers and geospatial location. The concept of playing by using the player movements resembles the original Wii concept and results in a completely different dimension in playing, opening room for a huge set of possibilities in gaming. The geospatial location capabilities are also drawing the attentions of both players and developers, as some phenomena such as Geocaching begin to win their space in the market. Location-based games are still relatively recent, but are already becoming very popular amongst players. Although most of these games are still very related to a specific place (for example, "The target" is only playable in Barcelona and "PacManhattan" is only available on Manhattan") and very close in concept (usually they only require to find a "treasure" that is hidden somewhere by unveiling some quest or jigsaw), a great ammount of new projects and initiatives are starting to appear, some of them powered by great companies such as Google and Sony. It is also important to mention that these games are usually closely intertwined with a strong component of social networking, therefore becoming massive multiplayer games with a planetary radius, which as it was mentioned above, becomes an important factor for the success of this type of games.

Besides the development platforms described above, two lower level alternatives are available, based on the two main renderers mentioned before, namely OpenGL for Embedded Systems, or OpenGL ES for short, and DirectX Mobile. OpenGL ES consists in a well defined subset of the OpenGL main system and is primarily oriented for the development of graphical applications for embedded devices such as handheld platforms, vehicles or consoles. There are two versions main branches available concerning this software platform. OpenGL ES 1.x is oriented to platforms where the hardware is not programmable, thereby enabling hardware graphical acceleration and focusing on image quality, being optimized for a great performance on low-specs hardware. In the other hand, OpenGL ES 2.x was created mainly for use with programmable hardware, in which the graphics pipeline can be programmed through special shaders. One of the greatest advantages of this technology asset is that it can work upon multiple hardware and software platforms, being therefore very easy to port between different systems. Also, it can be used in devices with very modest specifications such as devices with a 50 MHz processor and 1 MB of RAM. In other perspective, Microsoft has also released a mobile version of the DirectX software development kit. Like OpenGL ES, this platform is also created from a subset of the original software environment for normal computers, in this case consisting of the entire functionality of DirectX 9, but without vertex and pixel shading capabilities and the possibility of performing multiple locks upon vertex or index buffers simultaneously. However, DirectX mobile is targeted only at Microsoft Windows Mobile 5.0 or greater, requiring the installation of the .NET Compact Framework on the device. As some mobile devices do not support floating point arithmetics, DirectX mobile contemplates these cases by introducing some special classes which abstract these cases from the programmer, although it is strongly recommended to always use fixed point arithmetics to improve performance in these platforms. In conclusion of everything mentioned above, and comparing these two software development kits, OpenGL ES seems to be a better approach when developing multi-purpose, multi-platform game engines for mobile devices, mostly due to its ambit, platform maturity and feature set.

2.7 Location Based Gaming

According to ABI Research [TFL06], in the year 2011 there will be 315 million Global Positioning System (GPS) users and Location Based Services (LBS) in the world. It seems then clear that GPS enabled devices are becoming very important in people´s daily lives. Although these systems are intended for telling the user where it is located in the world through its coordinates and map routing between two or more places, the characteristics of these devices soon became relevant in order to make it a very rich multimedia platform. The touch screen, speakers, memory card slots, PDA-like architecture and powerful rendering capabilities for two and three-dimensional graphics have placed drawn to these platforms a great amount of attention to mobile gaming on LBS devices. Like the multimedia facilities now available on these platforms, also the concept of mobile GPS gaming is rather new. However, some initiatives have already become very popular amongst PNA users, like, for example, the game Geocaching. The objective of this game is quite simple, having its roots on a treasure hunt. One player must hide a small container, called geocache, in some outdoor place and must give instructions to the other players in the form of a quiz or treasure map. When one player finds the geocache, he must keep its contents and replace them with some other objects. [GSI09]

But some other kind of games or applications using LBS can be easily found. One of the most original projects is the Project PacManhattan. [Amo09] This consists in a game in which one player represents the well-known character Pacman, which must flee from the ghosts that chase him. The remaining players represent the ghosts. The objective of Pacman is to go through every street in a set of pre-defined pathways to win. This objective is similar to the original Pacman objective in which the yellow character had to eat all the small yellow balls to win a match. On the other hand, the ghosts must touch the Pacman to win in order to make him lose a life and must avoid him to go through every path. The result is a very fast and fun game, played in the heart of a big city (Manhattan in this case) which is able to produce hilarious moments. This game is particularly fun when played by parkour runners using small headset devices for communication and a micro GPS platform for positioning.

Besides these independent projects, the multimedia industry is already seeing this area as a possible future in terms of business. For instance, the Locomatrix company has developed an infrastructure that allows users to create new games, start gaming servers and/or join existing games that are currently running [LML08]. This is done by using two distinct applications. The Locomatrix website, which acts as the company product front page and presents the basic rules, and a mobile application that is installed after request and interacts with a GPS device and the Locomatrix main servers in order to play the game and find nearby players. The mobile application must be requested to Locomatrix by sending a text message to their services and then downloading it from an URL given in response. Except for the message and the optional GPS unit sold by Locomatrix, the whole game is free. Also, GPSGames.org provides a set of games that can also be played for free. There are actually seven games that, although very similar in what concerns the gameplay concept, are still free of charge as the company lives from donations that are made by the players. The concept behind all of these seven games is equivalent to the geocaching and doesn't quite provide “error-checking”, i.e. a player can easily cheat while playing. For example, one of these games is the GeoPoker [GG09], in which the players must find a geocache to be awarded a new random card. The objective, as in poker, is to get the best possible hand. However, as stated in the site, the game “uses the honor system to prevent cheating”.

It is important to notice that, except for the case of Geocaching, most of the games mentioned before present some sort of limitation related with the number of simultaneous players in one game. There is, however, some games that try to include the greatest number of possible players in the same game. One example of this is TurfWars [MFP09]. The concept is truly innovative for a LBS game. The basic objective is the creation of a turf controlled by the player. The turf can be expanded by competing with other players, which represent rival mobsters. Each player has its own set of stats that can be evolved throughout the game in a RPG-like manner. These stats can be improved by accomplishing missions, buying new weapons, joining a stronger mafia or even building its own family and becoming its capo. New missions appear according to both the player current rank and its location. When two players are near, they can engage in combat. The result of the fight is automatically calculated according to the current statistics of each player, in a process common to many online RPG.

Although this genre is not really inserted into the realm of gaming, it is relevant to notice that the concept of LBS gaming was also expanded to location based social networking. The most well-known example is probably the Google Latitude project. This software allows the users to find nearby friends as they move throughout the city or when they log into a google account. The concept of social networking based on the geographic information of an user has been used into several other applications that even offer some rewards, in the form of achievements to the most active and exploratory players. Good examples of this are Foursquare, which is constantly updated with information relative to the players and its achievements, which are rewarded to players when they interact with each other, accomplish objectives or find new places; GoWalla [GWI09] that has a direct connection with Twitter and Facebook and provides the user with a virtual passport that is stamped with a distinct mark for each new visited place; and finally BrightKite [BKI10] that, although it is much more simple then the previous, also has the capability of presenting events happening near each user and share photos between them.

Finally, there is one innovative concept that remains to be mentioned, which consists in the ability of drawing with the GPS. Once again, this concept is not directly linked to the usual concept of gaming, but is still considered relevant to the subject in hand. The main activities in this area are led by GeoDrawing.com [Sha08] which holds a gallery of GPS drawn pictures. The concept is very simple. By using only a path recording enabled PNA the user must use the streets of a city as the guidelines for a drawing. As the user travels through the city, lines are drawn in the GPS representing its drawing. So, the basic challenge is to walk through the streets in order to produce an artistic drawing. The result is a huge gallery of drawings made by GPS that reflect the imagination of a great amount of users that support the GPS Drawing community.

3. A game engine for LBS

This section presents the solution for the problem mentioned above, describing the chosen architecture and methodology used in the development of a game engine that attempts to respond to the necessities referred in Chapter 2. This project resulted in the creation of the *PunkMan* Game Engine, which consists in a multi-platform game engine that is able to take advantage of the location-based services capabilities. The engine architecture was built in order to create a flexible working software base that is capable of working on any platform with only minor adaptations. It was created with the concern that almost everything can be expanded to create new features or to adapt the engine to a new device. Nevertheless, the engine features a whole set of capabilities that already greatly complement the underlying infrastructure, giving wide rendering and optimization possibilities. The engine's architecture, philosophy and methodology will be described below, in the following sub-chapters.

3.1 Engine requirements

The development of the current project was guided by a set of requirements that represent the facilities that the final product must provide. So, the first step towards the resolution of the problem mentioned above started with the creation of a list of functionalities that the final engine should be able to provide to its users. The production of this list was based on the capabilities that some existing game engines, described in Chapter 2, are able to provide; in game development frameworks, such as XNA; in the capabilities of the current personal navigation assistant devices, such as GPS; and in the experience granted by some years of game creation and game engines development. Therefore, the *Punkman* engine must provide the following list of requirements:

- Creation of a base programming framework that abstract the underlying hardware and software environment
    - Abstraction over the data types
    - Abstraction over data structures
    - Creation of a mathematical library mostly oriented to computer graphics
    - Implementation of a fixed-point library

- Serialization system
    - Creation of a base serializable type
    - Creation of objects from the serialized representation
    - Objects may be transferred between entities in the serialized format
- Pseudo-reflection system
    - Each entity must know its type and the types of its parents
    - Each entity knows its variables, functions and their parameters
    - Each entity may expose some or all of the variables and functions for remote execution
    - Each entity may be controlled remotely by an external controller in order to obey to an external script or artificial intelligence routine, for example
- Portable rendering system
    - Development of a structure that can be used for easily implementing different rendering APIs
- Rendering capabilities
    - Creation of a canvas for drawing
    - Rendering initialization and configuration
    - Support for rendering 2D objects
        - Sprites
        - Text
    - Support for rendering 3D objects
        - Triangles
        - Meshes
    - Lighting support
        - Implementation of several light types, where available
            - Point
            - Omni-directional
            - Spot light
        - Materials and mapping support
            - Creation, configuration and application of materials
            - Support for different map types, such as textures
    - Camera support
        - Target cameras

        - Free cameras
    - Transformation
        - Flexible transformation system
        - Implementation of transformation queues
            - Add transformations to an object in runtime
            - Remove transformations from an object in runtime
            - Changing transformation parameters
            - Swapping the order of transformations in runtime
        - Pivot-point transformations
        - Possibility of adding new types of transformations when necessary
- Animation
    - Flexible animation system
    - Inclusion of keyframed animations
    - Creation of a linear animation system
- Scene management and optimization
    - Creation of a scene graph structure
        - New objects may be added, removed or changed in runtime
        - Must provide a way for easily loading a scene graph from an external file
    - Extensible and configurable scene optimization system
        - Creation of a scene management system
        - Scene managers may be interchangeable
        - A scene may be optimized by any scene manager available
- Location-based services
    - The engine must provide a way of knowing the user's current position
        - Street
        - Address
        - City
        - Country
    - The navigation system must be sufficiently expansible to work with the several location-based services available
        - GPS systems
        - Geographical web services

        - Logging systems
    - The engine must include the capability of calculating the route between two points
    - The navigation system must include a POI system
        - POI's can be loaded from external files
        - Each POI must have, at least:
            - Name
            - Location (in coordinates)
            - Type
        - The POI types must be dynamic
            - May be added in runtime
            - May be loaded from a file
            - A POI is always associated with a type
    - The navigation module must consider the existence of different coordinate systems
        - Representation of coordinates in:
            - Degrees, minutes and seconds
            - Decimal format
            - UTM format
    - The navigation system must be able of seamlessly connecting to the graphical and logical system.

3.2 Methodology and engine philosophy

Bearing in mind these problems, the development of the present engine was guided by a strict, but rather useful philosophy. The first step was to write a list of the basic features that are mandatory for an engine for Location-Based Services, as presented above. After listing these required features, it was necessary to define which capabilities depend on the platform on which the engine will be deployed. The philosophy used for these cases was to create an abstract interface for each platform-specific component. Each of these interfaces must be implemented by a class for a specific platform. So, an abstraction level over these platform-specific concepts is provided by these interfaces, but also by a factory class that is built on top of these classes. Each factory class follows both the Factory Method pattern [Gam95], depicted in Figure 15, which is able of choosing the correct implementation for a given interface according to the application settings; and the Singleton pattern, which provides a global instance across the whole application. For example, to have a window manager, a *PMWindow* interface class was created. After that, in order to have a system that can run on a 32-bit Windows® system, an implementation of this interface for this specific platform was created, which was named *PMWindowW32*. On top of these two classes, a *PMWindowFactory* class was developed. So, to instantiate a window, it is simply necessary to use the following code snippet:

PMWindow window = PMWindowFactory::createWindow();*

The calls to the factory methods are parameterized according to the interface constructor. In the case presented above, the creation of a window can be customized with a title, width, height and an option that states if the window must be expanded to fullscreen mode.

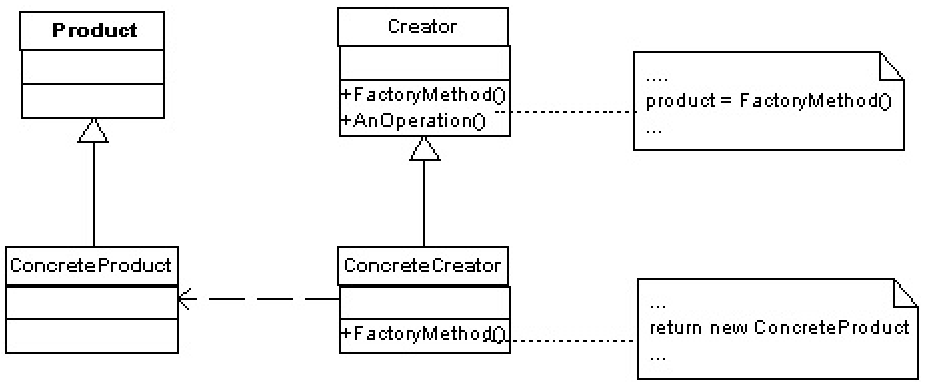


Figure 12 - Factory Method pattern

Due to the possibly limited resources in which the engine may operate, memory management and performance were issues that were always present during the development. Although impossible in some cases, recursion was avoided as much as possible by using dynamic programming. Also, the math layer, which is the base for nearly everything else, was created by using some of the fastest and most efficient algorithms available, such as the Square Root algorithm from John Carmack [Ebe02].

Also, the behavior of the engine can be fully customized by changing the definitions at PMDefinitions.h. By modifying these settings, the engine is able to readapt to the operative system, to the renderer driver type, to the arithmetic library, toggle the use of STL, etc. For example, to change the renderer driver from OpenGL to DirectX, it would be necessary to change the respective setting in the file and compile the engine for DirectX. The engine would automatically link with the correct libraries and maintain all of the remaining functionality.

The last important point to mention about the game engine design philosophy is the focus on the layer encapsulation. A lower level layer never depends on a top level one. By achieving this, the lower layers can be reused for a different purpose or, for instance, as the base for an upgraded version of the engine. Besides that, due to the extensive use of interface classes to abstract the top layers from the platform-specific components, these low-level elements can be easily changed or updated without compromising the entire engine. New components can also be easily added, if necessary, by deriving the interface class and adding a new entry to the factory class. So, the interface classes can also act as stated by the Façade pattern [Gam95], providing a "translation" interface between what a module provides and what the system actually expects.

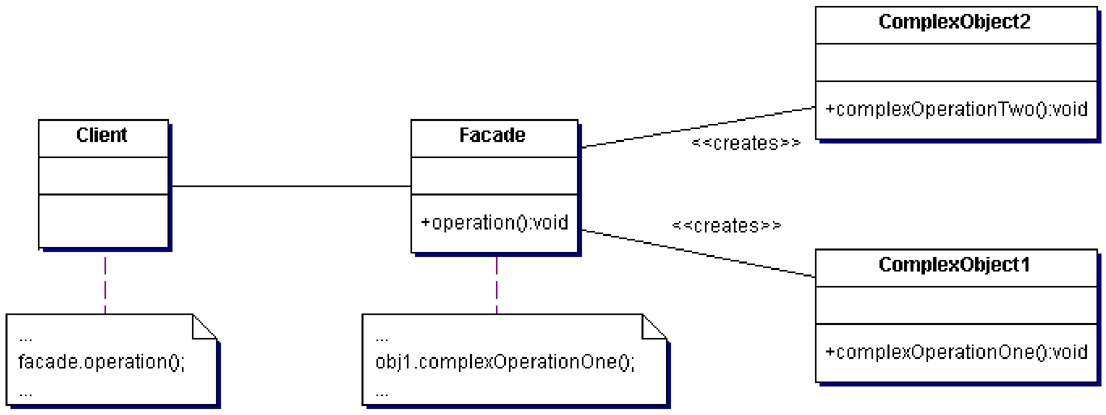


Figure 13 - Facade pattern

3.3 Architecture

The main problem behind the creation of the present engine is the development of an architecture which can work in several different platforms, provide a flexible and extensible programming interface and that is capable of delivering a useful set of features to the developer. Given that, it seems clear that the architecture must be carefully planned before implementing any engine features.

Starting by the beginning, the first big problem is the engine portability between several platforms, including mobile devices. This could be implemented by modifying the engine's code base for every platform. However, this is not a desirable solution mostly due to two facts: in the first place, no one can know in beforehand how many changes will the engine need, in order to compile and run in another platform; but most importantly, no one can predict what will be the impact of those changes for the rest of the engine classes that are directly or indirectly dependant of these base classes.

Due to these facts, the first architectural decision was the sub-division of the engine in four main layers. This layout is common amongst game engines, with the difference that the scene management and the application layers are usually unavailable to the programmers. The first layer, called the core components layer, consists in a low-level system management whose main goal is to provide a solid and stable abstraction layer over the underlying development system. Immediately above this layer lies the renderer layer. This layer is responsible for handling everything that is to be drawn and to provide a set of operations over the window, if there is one, upon which the graphics will be drawn. To manage these facilities, a scene management layer was created. The idea behind this layer is to provide a set of operations that can be used to create and modify a scene easily, such as scene graphs. But, and perhaps more important than managing a scene is to provide some means to optimize its presentation. On top of these layers, there is a fourth layer, namely the application layer, which is actually composed of some features that are useful, but not essential. A good example of such a feature is the insertion of a possible physical engine. This feature would fit on top of the other layers, but it would not be an essential feature of the generic engine that is to be developed. In that case, it would become an external feature which could be implemented on top of everything else.

Designing the architecture for the core components layer was one of the greatest challenges presented by this project. This layer is responsible for providing the abstraction over the underlying system and it must consist in a coherent stepping stone for the rest of the engine built upon it. It is subdivided in five main parts:

- Low-Level System
- Data Structures
- Event System
- Networking
- Thread Management

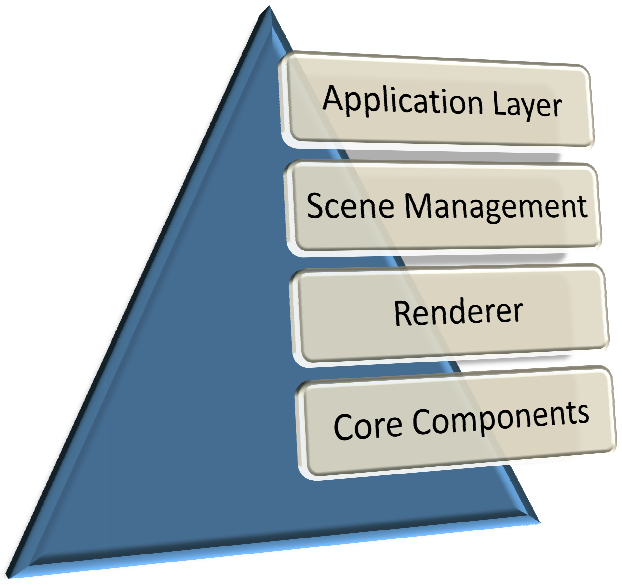


Figure 14 - Engine layer organization

Starting by the beginning, the low-level system is responsible for file management across platforms. This part of the core components is composed by an interface to the file management entity and a factory that chooses the correct implementation for this interface according to the project settings. The data structures sub-layer consists in a set of classes that provide a way of storing data throughout the engine and a way of performing operations on the data saved in those structures. This sub-layer is composed by a set of generic data structures, such as *arrays*, *strings*, *hash maps* and several others.

Besides these data structures, some other classes were created in order to store data in an organized form, but are logically separated due to its nature. One of these cases is the graph library which provides support for graphs across the application. These structures will be used not only by the scene management layer above the core components, but also by the XML parser and are also useful for game logic. The graph library consists in three generic classes, which represent a graph, graph node and graph edge that can hold any type of data. Finally, there is also one generic class called *PMGraphLibrary* which is responsible for applying the algorithms upon graphs. These algorithms can be as diverse as the Ford-Fulkerson algorithm for maximum flow, Kruskal and Prim algorithms for the minimum spanning tree, loop detection, D* Lite for path finding, among others. [Wei99]

The other case is the math library which was implemented in this sub-layer. As the engine is also composed by a graphical engine, there is the need of having a specialized library to handle all of the mathematical part of the processing. This library is composed of data structures such as matrices, vectors and quaternions. The main idea behind these mathematical abstractions is to give independency over the renderer driver that is to be used. All of the geometrical transformations are carried out by the matrices themselves instead of letting the renderer do the job. This way, DirectX, OpenGL or any other rendering system can be used, because the matrices are pre-calculated by the engine and submitted to the renderer to be applied. Also, some rendering systems do not have direct support for quaternions and by applying the transformations before submitting the matrices to the renderer, this problem is avoided. This is also important because the implementation of quaternion rotations can avoid problems such as the Gimble Lock, caused by using rotation matrices calculated through Euclidean Angles. In parallel with the later files, a Singleton class *PMMath* was created to support some of the most common mathematical operations, such as square root and trigonometrical functions.

Moving on to the event system, it seems quite clear that without input by the player a game cannot exist. On the other hand, creating an event system that can be supported in several platforms is not an easy task. For instance, the Microsoft Windows® window management system and the event dispatching system are deeply interleaved, which makes the architectural design a hard task. Moreover, an event is not necessarily an external input from the user; it can also be a notification that a file has finished loading, for example. The solution implemented to solve this problem was the Subscriber pattern. To listen to events, a class must inherit from the *PMEventSubscriber* class. After that, it can choose from which listeners it wishes to fetch events from, just like Java. Then, every time an event of the same type of the chosen ones happens, a notification is fired to every subscriber of the listener. Portability is ensured by the fact that the *PMEventListener* class is merely an interface for the event listener system and must be implemented for each desired platform or event type. Using this architecture, one can easily create a new class for capturing and firing event notifications from a joystick and at the same time make a new 3D file object reader that inherits from this class and launches an event when the loading is complete.

One other sub-layer is the thread management module. This component is responsible for handling the threads that are launched by the application. As before, this is also a platform specific issue and therefore must be handled carefully. Once again, the portability problem is solved by adding an interface which can be implemented for each platform and instantiated through a Factory class. There remains, however, the well-known problem with threads and concurrency, imposing the need of creating security measures that can avoid this issue. To solve this security problem, the threads can only be created through a Singleton *PMThreadManager.* The thread manager uses a Factory to create each thread and saves them into an internal list of active threads. Also, it saves a list of *PMCriticalSection* objects, which can be used by the programmer to impose access restrictions upon the existing threads, i.e., a thread can only enter a specific critical section if no other thread is inside section. By using a Singleton thread manager, this control class is shared among every thread and possesses a global view of what is happening. Therefore, it can control what enters or leaves a critical section, by providing it with the clearance to enter a section. If a section is occupied, then the requesting thread is put in a queue in order to enter as soon as possible.

Finally, the networking module is responsible for providing an abstraction over the socket facilities. If there is a possible network connection, then the *PMSocket* interface can be used to implement a communication class that can be used by the rest of the engine, guaranteeing the portability of this subsystem. This networking system can be used for creating both client and server sockets, and is mostly useful when used together with the thread management module, because some sockets can be blocking reading. The network capabilities are used mostly for fetching data from the web, using several protocols, useful for possibly upgrading engine components or download new data.

The following diagram represents the architectural layout of the Core Components layer, represented by its classes and relations between them. Due to its dimensions, the data structures package is not described here in terms of its classes and relationships, but a more detailed explanation is provided below, in Chapter 4 which is dedicated to this package.

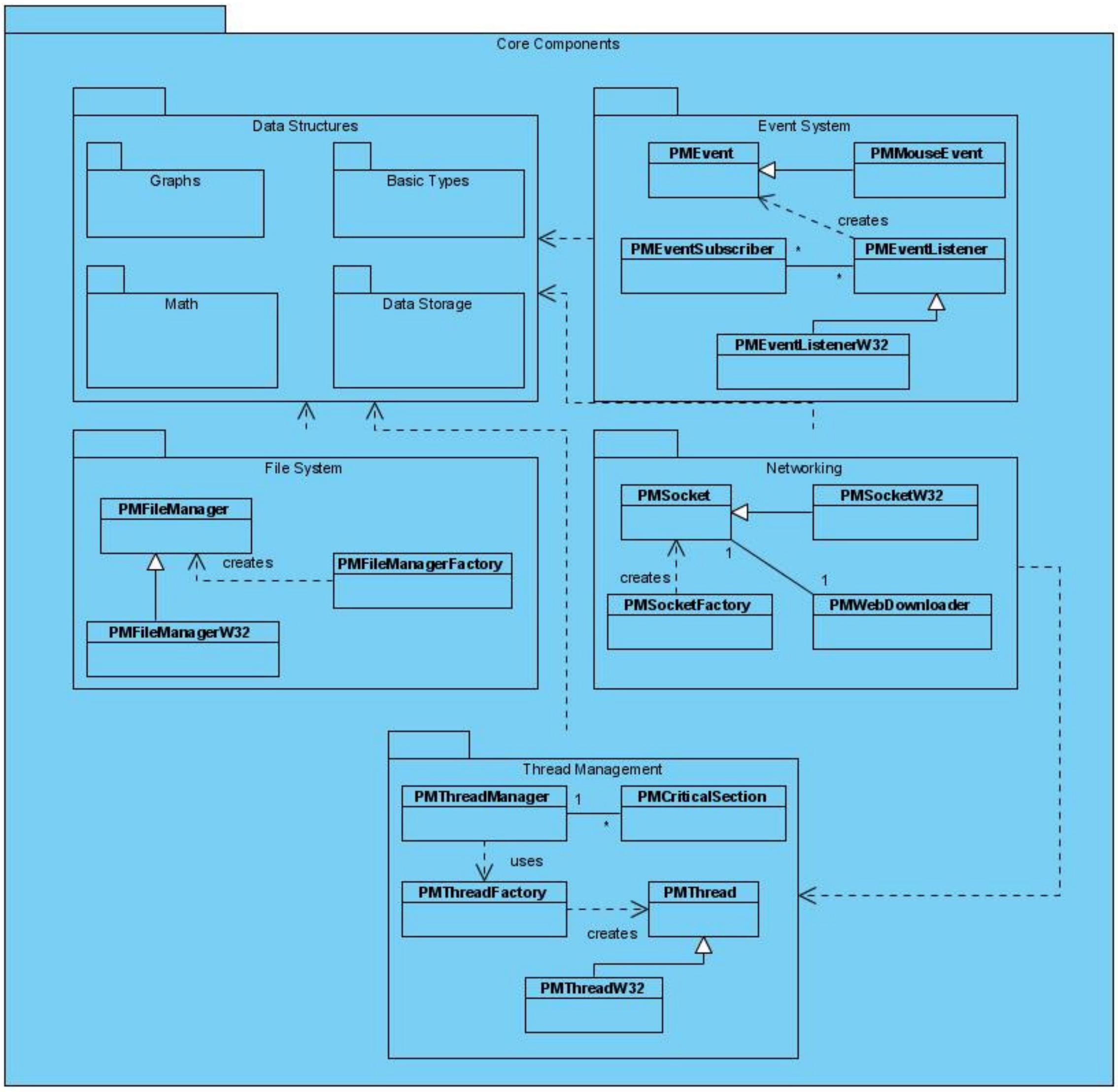


Figure 15 - Core components class diagram

The core components layer, described before can be the base of any application that was not necessarily related to computer graphics and games. Based on this layer, described above, sits the rendering layer, which is responsible for handling every aspect related to the actual graphics that are displayed on the screen. The major problem presented by this layer is the fact that there are at several renderers that can be used. Two of these well-known software platforms are OpenGL® and DirectX®. Although they serve the same purpose, these renderers are not compatible with each other and must work in exclusive mode. But, even if the inter-compatibility was assured, the way they work is completely different. While OpenGL presents a

C-like API composed by a set of functions split across a core library and several satellite libraries, some of which are created by users, DirectX presents a unified and structured object-oriented API which condenses every aspect of graphical programming into a single SDK which can be extended by user created plug-ins. Nevertheless, the main difference between these two platforms is the portability. While OpenGL can work in virtually any device and operative system with graphical support, DirectX works exclusively upon Microsoft Windows compatible platforms.

To be able to create an engine that is independent of the actual renderer being used it is necessary to develop a layer of abstraction that captures the essential common features of these platforms and is capable of transposing in a seamless form to the rest of the application. The development of such a layer started with the production of a list of the essential features mentioned above. After writing down these required functions an abstract class could be defined to represent a common rendering interface that could be used by the rest of the engine. The implementation of this interface works just like the rest of the platform specific concepts that were explained above. According to the application settings, a renderer implementation is instantiated and returned by using a renderer factory class.

To support the rendering facilities, a canvas to draw to must be created. As the canvas are usually contained within window areas, a window management interface was created, that provides the essential methods used to manipulate windows, canvas, viewports or any other type of area where the drawing can occur. Like before, as the windows are platform-specific, the window manager objects are instantiated through a factory class that chooses the correct implementation based on what was defined in *PMDefinitions.h.* Taking advantage of the abstraction level provided by the window and the renderer interfaces, the definition of an architectural scratch for a graphical scene could be established. A typical graphical engine setup is divided in two main components: 2D and 3D objects. These two components can then be used by several other mechanisms that manipulate these entities on the screen.

The 2D entities are mostly sprites that do not possess the concept of depth. However, the implementation of such concept varies drastically between renderers. For instance, while DirectX includes the concept of sprites, which have their own rendering cycle and can be drawn directly to the screen without worrying about any previous setup besides texture loading, OpenGL does not provide native support for 2D entities. To create a 2D object in an OpenGL context, an orthographic camera must be initialized and pointed at a 3D object in order to make it look false. Although 3D objects use an extra coordinate, most of the operations that can be performed on these entities are also performed on 2D objects. So, it seems clear that the architecture for the operations on both entities can be shared up to some point in order to increase stability, efficiency and reduce code replication. The architectural layout of this layer consists in six sub-layers, each of which takes care of a specific aspect. The most important is perhaps the loading. 2D entities must be loaded in a different way from 3D entities since 2D objects are much easier to load then 3D. To support the loading of the files, a geometry sub-layer was created, therefore implementing 2D and 3D specific data structures such as vertices and faces. As an architectural decision, these data structures were implemented in this layer and not in the core components because they implement concepts that are specific to the renderer. Consequently they do not belong in the data core components but rather in the rendering layer, although they are higher level data structures.

An imperative concept present in computer graphics is lighting. This is essential both to 2D and 3D structures, although its usage is usually overlooked when working with 2D elements. However, 3D objects rely heavily on lighting to achieve realism or other types of effects such as cel-shading. Lighting includes not only the lights per-se, but also the object's material properties, color and texture mapping. Some of these concepts are, as some of the data structures mentioned above, renderer specific. So, as before, the solution is to create a common interface and instantiate the correct object by using the Factory pattern. Although this solution works for the present problem, the definition of an interface for lights is not at all trivial due to the fact that, once again, each renderer does things in its own way. This problem is mostly evident when a user tries to create lights with different types. While DirectX supports spotlights, point lights and directional lights, OpenGL doesn't make any type of explicit distinction. To overcome this problem, a list of common light properties was written and placed in the interface. The type of the light, if necessary, is automatically found by the engine according to the parameters used.

Two other key concepts that must be present in the renderer layer are transformations and animations. Both of these concepts depend heavily on the renderer used, which uses matrices to compute the result of some set of transformations. Matrices present two problems. The first big problem is the matrix format that is used. Some renderer API's use row-major transformations while others prefer to use column-major order operations. From a mathematical point of view, this is solved by performing a simple transpose operation, which is a simple operation. From a software engineering point of view, the big problem is to find out when should these operations be used and how can the engine be design to remain transparent to the programmer when the renderer changes. The answer to this problem was to remove the transformations from the renderer and perform them directly through the matrix library. Only after all the transformations matrices are calculated internally in the engine, the matrix is submitted to the renderer. Then the renderer implementation does its job upon the matrix, transposing and modifying it if it eventually needs to. The diagram below presents the general architecture of the renderer layer:

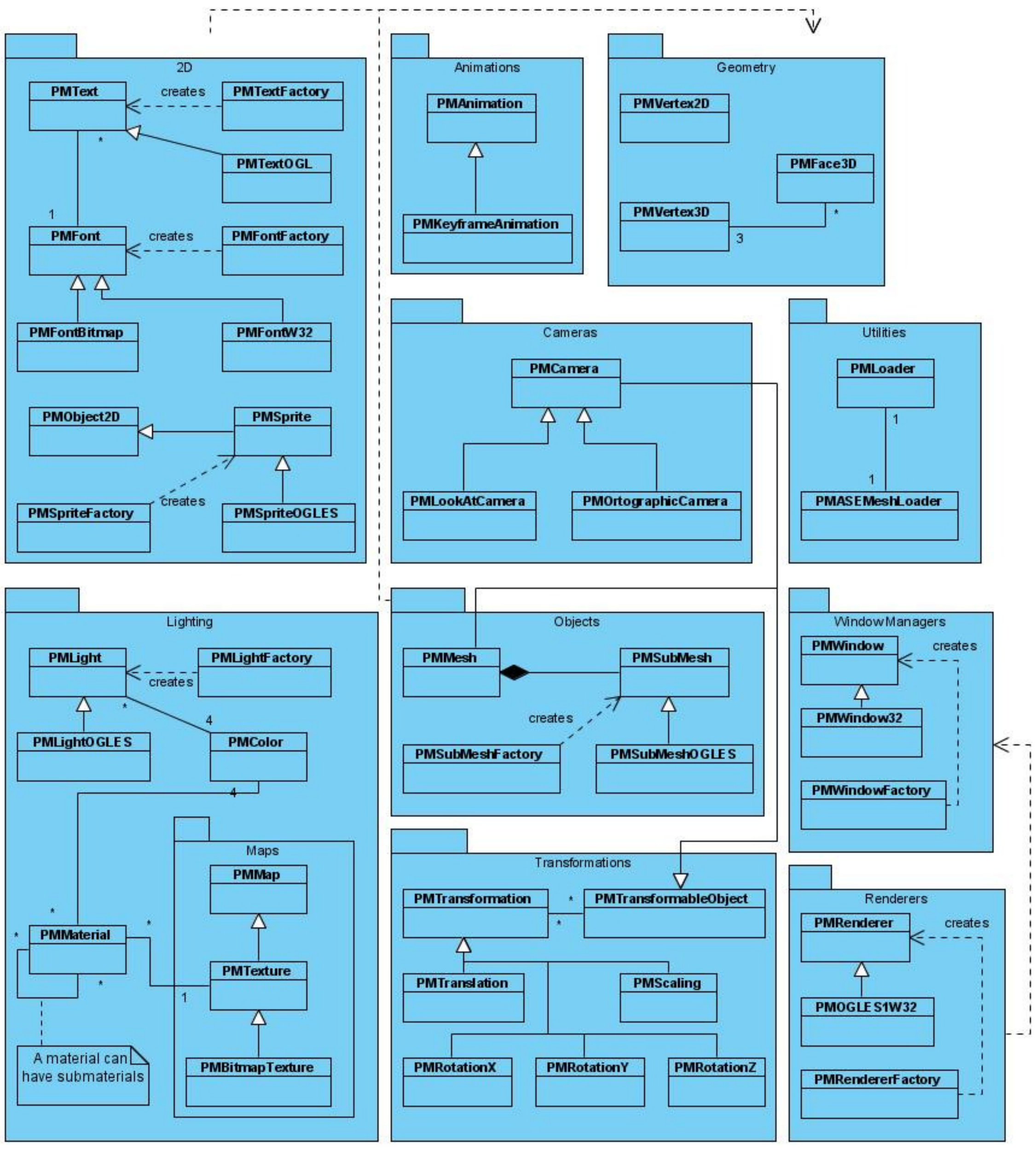


Figure 16 - Renderer layer architecture diagram

Finally, it is important to mention the creation of a 3D mesh support layer. A mesh is a set of sub-structural atoms (triangles, quads, patches, etc.) that are part of an object and usually share some properties. For the sake of portability between renderer drivers, each mesh is composed of at least one submesh. Each submesh may have its own set of properties such as material, map, color, position, etc. Also, this implementation allows for some interesting variations such as composing a mesh from several different external files, or decomposing a mesh in several components. Besides the properties mentioned above, a submesh stores information about the faces, vertices and bounding boxes of an object. This information can be ignored, but it can be useful later when running optimization algorithms such as octrees.

The scene management is a rather small but nevertheless mandatory layer. It provides a simplified way of controlling a scene and helps to keep everything organized. Maintaining the scene organized is not merely a way of easing the burden of finding entities in the current set, but most importantly, to be able to optimize the rendering of a scene. Therefore, the main component of this layer is the scene graph, whose function is to represent the scene in the form of a graph, storing the connections between the components and applying the logical and drawing routines accordingly. As a utility to ease the task of loading scene graphs, a XML reader class was created. By using this class, a programmer can easily specify a scene graph using XML and then loaded seamlessly. The scene graph is automatically loaded and presented on the screen. An octree was also included in this layer to optimize the scene performance when required. The octree is calculated in parallel with the scene graph, but it is only built on demand, due to the fact that it can take some time to compute the node information. However, as this is done on load time there is no real impact in the final product.

4. Core Components Layer

4.1. Data Structures

The data structures layer is a part of the Core Components Layer that provides the necessary abstraction over the existing data types in order to give the engine the possibility of working in multiple platforms. Basic types such as integer and decimal values are redefined in this layer, to provide platform independency, but some new data structures are also added to provide new and expanded functionalities to the engine, such as strings, dynamic arrays and hash maps.

This layer is represented in the diagram below, which presents an architectural overview over the engine data structures layer:

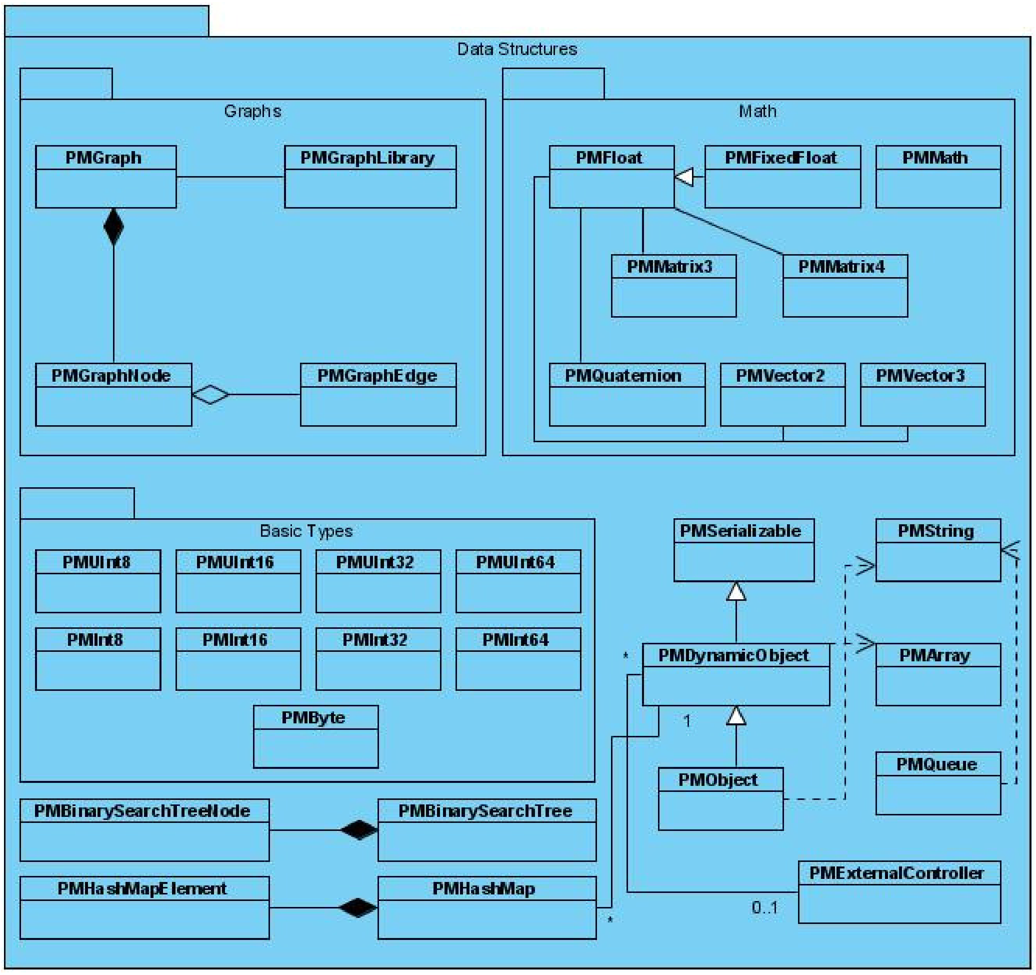


Figure 17 - Data Structures architecture diagram

4.1.1. Integer types

The core components layer consists in a set of features that provide platform independency to the upper layers of the engine. As some devices do not provide support for certain language facilities or data types, the most common data types were encapsulated in classes or redefined by engine-specific types. The most notorious example is the redefinition of the integer types to *PMInt8*, *PMInt16*, *PMInt32*, *PMUInt8*, *PMUInt16*, *PMUInt32* for the signed and unsigned eight, sixteen and thirty two bits integer types, respectively.

4.1.2. Floating vs fixed point arithmetics

However, there is a more important type redefinition that was implemented on the engine. Most mobile platforms do not provide support for floating point arithmetics such as a normal desktop or laptop computer. So, to be able to use decimal values, a fixed point arithmetic library had to be developed in order to be able to perform mathematical operations upon these values. Fixed point numbers are stored in integer data types. Considering that 32 bit integer types are available, these bits must be divided between the integer part and the decimal part of the number to represent in fixed point. The PunkMan engine assumes a default setting of 15 bits for the integer part, 16 bits for the decimal part and 1 signal bit. However, this parameter can be changed by setting the PM_FIXED_FLOAT_FRACTIONAL_BITS to the desired value. Storing a decimal number in an integer format raises one serious problem. If the integer part of the number has only 15 bits, then it is limited to values in the range [0, 32768[, which corresponds to 2^{15}. The same can be said about the decimal part, which is composed of 16 bits, reaching a maximum precision of approximately 1.523×10^{-5}. In some situations, the problem could be minimized by reducing the number of fractional bits. This would solve some issues were operations with big values are essential, therefore sacrificing precision. The same solution could be applied in cases where a bigger precision is required, by increasing the number of fractional bits. Nevertheless, it seems clear that this solution would not be useful for the most common situation, where both of the previous situations appear intertwined in the same problem.

The main arithmetic operations were rewritten to cope with the fixed point number implementation described above. By the inherent properties of the fixed point values, implementing the addition and subtraction operators is a straight-forward task, as it can be performed as usual upon the integer values that represent the fixed point number. However, the multiplication and division operators can't be implemented with such ease, due to the fact that multiplying or dividing two 32-bit integers can easily result in values that can't fit a 32-bit integer, thereby creating unpredictable over/underflow situations that are very hard to detect and solve. To solve these cases, the most common solution is to use 64-bit integers to hold the values and the result of the operation. After the operation is performed, the result is shifted N bits to the right, where N is the number of fractional bits of the current fixed point representation. Another solution would be to perform the multiplication incrementally and shifting each intermediate result to its place. By doing this, it is possible to avoid the usage of 64-bit integers, but much more operations would be required to perform the calculation. An example of such code would be as follows:

```
int ntemp = rhs.number;
int temp, result;
char sign = 0;

if (this->number < 0)
{
    sign = 1;
    this->number = -this->number;
}
if (ntemp < 0)
{
    sign ^= 1;
    ntemp = -ntemp;
}

result = (((this->number & 0x0000FFFF) * (ntemp & 0x0000FFFF)) +
            1048576) >> 21;
result = result + ((((this->number >> 16) * (ntemp & 0x0000FFFF)) +
            16) >> 5);
result = result + ((((ntemp >> 16) * (this->number & 0x0000FFFF)) +
            16) >> 5);
temp = (this->number >> 16) * (ntemp >> 16);
result = result + (temp << 11);
this->number = (result * -sign);
```

4.1.3. Strings

The data structures implemented in this layer provide a set of useful data types that were built to provide an abstraction above the existing facilities of the language. Across the engine, the most used is probably the PMString. This class consists of an abstraction above the C standard strings (*char**) and the *STL strings*. The behavior of this class can be set in the configuration parameters present in *PMDefinitions.h*, choosing if the *PMString* should use *char** or *STL strings*.

4.1.4. Arrays

A dynamic array structure, called *PMArray*, was also implemented. Just like what happens with the *PMString*s, this class can also be configured to work either with C-style arrays or with *STL vectors*. If the C-style configuration is chosen, then an array is allocated with a fixed size. When all of the free positions are filled, then a new array with more space is allocated, the old one is copied and deleted when the process is complete. So, adding new elements is quite fast and easy, having the burden of re-allocation when the base size is exceeded. However, the base size can be configured by changing the *PM_ARRAY_BASE_ALLOCATION_SIZE*. Deletion of elements is performed in two different ways. If only one element is deleted, then the *PMArray* pulls every element back to fill the empty space, pushing it to the last position:

```
for (PMUInt32 i = index; i < this->length() - 1; i++)
        data[i] = data[i + 1];

this->size--;
```

If the array is to be completely erased, then a different approach is followed. Instead of operating in each one of the elements, the whole array is deleted and reallocated as a new one. A lazy deletion algorithm could be used, but this approach is preferred due to a simple fact. Lazy deletion could be implemented in a very easy and fast way, by simply setting the size to zero. So, when an object was to be added to the *PMArray*, it would overwrite the existing values, thereby performing the lazy deletion. However, there is an important problem that can’t be ignored. While in the first case the array is fully reallocated but no values are present in indices greater than the size of the array, in the later case an attempt to access to an index that is greater than the size would return a valid value that should have been deleted, but is still present and waiting to be overwritten by a valid value. An extra flag could be added to each element of the array to check if an element is valid, but by using the first approach this additional memory does not need to be used. The following table presents a comparison between the adding and emptying operations on both types of arrays present on the engine, namely the engine managed arrays and STL vector.

Elements to delete	PMArray	STL vector
100	0	0
500	0	0
1000	0	15
5000	0	16
10000	0	47
50000	0	967
100000	0	3494
500000	0	85924

Table 1 - Benchmarking for the empty operation in PMArray and STL Vector

Elements to add	PMArray Allocation size = 10	PMArray Allocation size = 100	PMArray Dynamic allocation size	STL vector
100	0	0	0	0
500	0	0	0	0
1000	0	0	0	0
5000	15	0	0	0
10000	31	0	0	0
50000	764	78	0	31
100000	3010	296	0	62
500000	114504	11544	31	312

Table 2 - Benchmarking for the add operation in PMArray and STL Vector

All of these tests were performed in the same computer and the results were obtained by calculating the average values of a series of twenty independent tests. The times are measured in milliseconds and were obtained by using the windows multimedia library, namely the *timeGetTime* function. So, it should be clear that the times that are reported as zero simply represent times that are below the precision obtained by this function. By analyzing the first table, it is possible to see that the deletion of all elements of a *PMArray* is much faster than deleting the elements of an STL vector, mostly due to the fact that it only requires deleting the array pointer using the *SAFE_DELETE_ARRAY* macro and reallocating it with the same size. By its turn, the times registered for adding new elements present great variations depending on the allocation size. This value represents the number of memory positions that are reserved when the array reaches its capacity and needs to expand. The results presented above clearly show that for arrays that require a small number of elements, a small allocation size is enough. This avoids wasting to much unnecessary memory by reserving small amounts of memory each time. However, if the number of necessary elements tends to increase, reallocating the array in such small increments may become a very slow process as shown above. Although a bigger allocation size increases the performance for bigger arrays, the same problem will arise sooner or later, as shown in the column that represents a *PMArray* with a value of 100 for the allocation size. The best option seems then to use a dynamically calculated allocation size, such as presented in the third column. The allocation size is computed based on the previous size, adding an amount of new memory positions equal to half of the previous allocation size. The results obtained and presented above seem conclusive about this method, which is even faster than the STL vector approach.

4.1.5. Hash Maps

To increase the performance of specific items in a collection an implementation of the hash map data structure is provided. This structure is a generic collection, accepting any kind of object as key and as value for each stored element. The idea behind the hash map concept is that of a table in which a key corresponds to a value. The values are stored in the table in a position that corresponds to the key that was provided. Therefore, searching a value in the hash map is very easy, because its position is given by the specified key. As the key may be an object of any type, a hash function must be provided. Given a key, this function must return an integer value that corresponds to an index in the table. An ideal hash function would always return unique values for each different key, but such a function may be hard to create in some cases.

There is, however, a problem to consider when implementing hash maps. If two different values have the same key, they will occupy the same position in the table. To solve this problem, there are four possible approaches. The first one is simply overwriting the previous value and store the new value in its place. Very similar to this, another approach is to ignore the new value if another value already occupies the slot for that key in the table. These two options are very easy to implement but always result in loss of data, thereby provoking the loss of reliability in this data structure.

Another option is to rehash the value of the key to another position in the table. For example, by adding a certain offset to the value returned by the hash function and store the value in the position key + offset. However, this alternative might also collide with other value that is already present in the table or, if that is not the case, steal the space for a value that should legitimately occupy that slot. Consequently, the idea of hash mapping would be somehow lost, search efficiency would be compromised and this data structure would become very close to a linked list.

The last option was the alternative implemented for the hash maps in this engine and consists in separate chaining. This technique implies using linked lists for each entry in the table. Each list is initially set to NULL if no value is present. Then, when an element is added to a given position, corresponding to a specified key, the list entry is initialized as a new PMHashMapElement pointer. These objects store not only the value of that entry, but also a pointer to the previous and next PMHashMapElement. When a new element matches the position specified by this key, a new PMHashMapElement is created accordingly, filling the previous and next values depending on the previous list state. If it is the first element, then the previous one and the next one are set to NULL.

4.1.6. Math Library

Since this engine requires very specific and optimized mathematical routines, a math library was implemented in order to perform the necessary operations over the newly created data structures.

- **PMFloat** – Represent a floating point number. If *PM_USING_FIXED_POINT* is active, then PMFloat actually represents an object of type PMFixedFloat. Otherwise, it implements its own functionality using floating point arithmetic.

- **PMFixedFloat** – Represents a fixed point value. If *PM_USING_FIXED_POINT* is defined, using PMFixedFloat or PMFloat is equivalent. Defines the set of possible operations over these arithmetic types.

- **PMMatrix3** – Represents a square matrix composed by three rows and three columns. These matrices can be used to represent 2D transformations in the engine and define the required set of operations that abstract them from the actual renderer driver.

- **PMMatrix4** – Similar to PMMatrix3, this class represents square matrices of size 4. These matrices are mainly used to represent 3D transformations and camera settings in the engine, supporting a wide range of operations.

- **PMQuaternion** – Represents a quaternion, which consists in a 4D value composed by four coordinates: x, y, z and w. These objects are useful to perform rotations in 3D space without using Euclidean angles. By using quaternions, rotation errors such as the Gimbal Lock can be avoided. This error occurs when using Euclidean angles to perform 3D space rotations, causing two of the object axis to collapse and become coincident. This is a very common and hard to solve issue that can be overcome by using quaternions.

- **PMVector2** – Represents a bi-dimensional vector, which can be used for 2D specifications. It can interoperate directly with PMMatrix3 by using the provided functions.

- **PMVector3** – Represents a three-dimensional vector, which is mostly used for 3D entities. These objects also interact directly with PMMatrix4 by using the provided methods, defined either in PMVector3 or in PMMatrix4.

- **PMMath** – Consists in a Singleton class that is able to perform some mathematical operations such as square root, conversions, and trigonometric functions. Also, it declares a set of constants such as PI that can be used throughout the engine.

4.1.7. Pseudo-Reflection System

To make the engine expansible and provide superior possibilities of controlling some of the engine objects from other objects or from external scripts, a pseudo-reflection system was developed. As the engine was fully developed in C++ and there is no native support for a real reflection system in this language (except within .NET framework with CLR support), a similar system was created that mimics some of the essential features of reflection.

Every object that does not represent a basic data structure (such as strings) inherits from a class named PMDynamicObject. A dynamic object is a special object that knows its own type and the types of the classes that it inherits from. It also saves the class member variables and methods in a hash map. The variables are stored under the form of a PMDynamicVariable, which represents an engine variable that is either a class member or a parameter for a function. These objects contain the name of the variable, its type and a pointer to the actual variable inside the class.

The information about the class methods are saved into objects of the PMDynamicFunction structure. Like the PMDynamicVariable mentioned above, these objects can be used to represent a class function and to expose it externally to the exterior. Each function saves its name, return type and required parameters. The parameters for a function are stored in the form of an array of PMDynamicVariables to preserve the coherence between methods and variables and provide a seamless interoperation between components. In the other hand, these dynamic functions also provide a useful feature as they can be remotely invocated. Therefore, by specifying a function name and its parameters, the class finds the correct function and executes it, returning its value, if any, in the form of a PMDynamicVariable.

Both the dynamic variables and functions are stored in hash maps in order to increase search performance. However, as C++ allows function overloading, there can be several functions with the same name, which will be stored in the same slot on the hash map. As the hash maps are implemented using separate chaining, the risk of losing or overwriting information do not exist, but in case of overloaded functions it may be needed to obtain all of the values in the hash map entry that corresponds to a certain key and iterate through its elements to find out the correct function.

Also, a dynamic object contains a pointer to a PMExternalController object, which is initially set to NULL. A PMExternalController represents a state machine or a similar mechanism that, given an object state, provides an action to be executed. An action specifies how the object should behave in a certain moment. Therefore, by specifying an external controller and attaching it to a certain object, that object can be controlled by the actions that are specified in the controller. Usually, these actions are dependent of one or more conditions that are also specified in the external controller. One type of usage for these controllers is the creation of behavior trees [Cha07]. A behavior tree knows its current state and the action that can be taken after a certain condition is met. If no conditions are met, then the tree executes a default or no-operation action. So, by using this, a certain object can easily be controlled either by an external script, behavior tree or any other kind of mechanism capable of translating states and conditions into usable actions.

However, this whole pseudo-reflection system requires that each class explicitly declares which variables and functions are available for external usage. This can improve security, as the external definitions are specifically written in the class. Also, it is very dynamic, because new variables and methods can be added, removed or modified in runtime. But it can also require a considerable amount of extra work, as the parameters for the variable and functions stores must be declared one by one in each class. This issue is reduced by the levels of hierarchy that compose the engine, but it may consume some time specifying every detail and configuring the application accordingly, nevertheless.

4.1.8. Serialization system

To support the interoperability between engine components and to facilitate the communication between external controllers and the pseudo-reflection functions, a serialization system was included in the architectural specification. Serialization consists in the process of converting a data structure into a sequence of bits or into a human-readable format that can be transmitted across an application, network or written to a file. After receiving a serialized object, it can be deserialized to the original object which contains all of the data from the original.

Serialization presents several problems and decisions that are not easy to make. The first problem is whether to perform serialization in a human readable or in a binary format. While the first has the advantage of being read by humans and allow easier debugging, the binary format is not easy to decode and understand. However, the binary format is usually much smaller than the human readable alternative. Another problem is to find out what is the data size of each object when it becomes serialized in either format. If the object is of a fixed size data type, such as a 32-bit integer, this problem is irrelevant as the object can be directly read by its size.

Otherwise, if the object is, for instance a dynamic array, then it becomes quite hard to know its data size, as it may have a variable number of elements. This problem can be overcome by using delimiters in the data representation, such as XML tags if a XML representation is chosen. Nevertheless, this approach has the problem of introducing an additional overhead both in size and in processing time, affecting efficiency.

The object serialization could be done by simply using a C routine such as *memcpy* to copy all of the data beneath a certain pointer address to an array of bytes. This method is very efficient and works quite well in all situations, except those in which an object has pointer attributes. In that case, the address of the pointer is copied to memory, but not its contents. Consequently, this method could work as long as the pointer address is valid, but if the pointer is invalidated or destroyed, the deserialization of the object would point to a possibly invalid or at least unknown zone of memory, which could bring serious consequences.
So, the best approach to these problems seems to be the creation of a simple interface, similar to the ISerializable interface in C#, which every object that wants to be serializable must implement. This interface defines only a pure virtual function *serialize* which returns an array of *PMByte* that represents the object and a second pure virtual function *deserialize* that is responsible for unmarshalling the object. But how does this solve the problems mentioned above? By using this method, both problems can be solved by the same solution. As each object implements its own version of *serialize* and *deserialize* functions, the representation to use, either human readable or not, is chosen by the object itself, as it is responsible for the translation of its own representation. This implies, however, that an empty constructor is supplied for every serializable object, because the object must be initialized before being used. This requirement could be avoided by using static functions in a Factory-like class that initializes every object in the engine, but using such a class from nearly every object in the engine would compromise the whole architecture. Therefore it was considered that providing a constructor without arguments for every serializable object was a better option.

The second problem would also be solved by the same method. As each object knows its own attributes, it can initialize itself accordingly and avoid the problems posed by pointer initialization and resizable data structures such as dynamic arrays or hash maps. Besides that, this method removes the necessity of providing special forms of serializing simple data structures such as integer values, because each class can serialize and unserialize them as it pleases. Also, as this alternative puts all the responsibility of the serialization process upon the object, it can produce a final result that is hybrid, i.e., as each object chooses its own representation, the final serialization product can be a mixture of both human readable and binary formats, using different representations. It seems clear that this fact can open new perspectives of optimization by choosing the best representation for each data type, reducing the final size of the serialized object and improving the debugging, as the result becomes quite modular.

4.1.9. Graph Library

An important part of a game engine is the capacity of organizing a graphical scene or any other relevant concept of development that requires some sort of representation of the relations between data objects. Moreover in an engine where location, navigation and routing are core topics, these functionalities become essential. So, to represent these relations, a graph class, named *PMGraph*, was created. A graph is composed by nodes and the edges between these nodes. To be able to use these graphs in nearly everything, this class uses template objects, to be capable of supporting every object type in its nodes. A node knows its value, which can be of any class type, the nodes to which it is connected through the specified edges, its parent and its unique ID in the application. An edge, which connects two nodes, can be bidirectional or not, knows which nodes are connected, which node is the origin and which node is the destination and its weight (which is useful in the case of a weighted graph). Consequently, based on all these specifications, a graph can store the information about its nodes and relations, be directed or undirected and be weighted or not. This definition of a graph was thought to provide a general and expansible way of using a graph, in such a way that it can be used within most of the algorithms that can be used for operations in graphs, such as routing algorithms like Dijkstra and A*, maximum flow algorithms like Ford-Fulkerson, minimum spanning tree like Prim algorithm or others. [Wei99]

To apply these algorithms upon the graphs, an approach similar to the *PMMath* class was implemented, called *PMGraphLibrary*. This class is basically a Singleton object that can perform all of the required operations in a graph either they are simple operations such as detecting if a graph is connected or if it has loops, or perform more complex operations like routing. The choice of this architecture in what concerns the creation of such a class is based on the fact that it makes the engine more coherent in the way that operations are performed upon the structures, thereby easing the engine utilization.

This class is easily expanded by adding more functions whenever required. Some operations were already implemented, but if in the future a new functionality is needed, it can be added to the graph library in order to provide a new facility to the engine.

4.2. Event System

Like any interactive computer software application, a game engine is heavily based on the input that a player gives to the game. Also, the engine must give to its components and subsystems a way of communicating between them. An example of such a case is sending a notification of the end of a file loading operation to the object that required the loading of the file. For that reason, it seems clear that it is necessary to provide a way of transmitting events throughout the engine.

As mentioned above an event can be of various types, such as keyboard or mouse input, file loading, or any other type of event that the engine or the user requires. So, the event system must be expansible enough to support any kind of predefined or customized type of event that is necessary. This poses a serious problem to the engine architecture and consequent implementation, because the events can arise from nearly anywhere in the application, depending on the requirements. Also, it is once again necessary to consider that for the same type of event, the place where it should be created and from where it should be transmitted may vary between hardware and software platforms. For example, in Microsoft Windows® systems, a mouse event can only be caught inside the WndProc function, which is a callback function that acts as the message receiver for a window. So, the window class must be able to create and throw events in Windows®. However, the keyboard events can be caught either in the WndProc function or by performing a call to the *GetAsyncKeyState* function, which can be done anywhere.

Based on all this, there can be several points in the application where an event can be thrown. Most importantly, these points cannot be previously listed, which increases the complexity of the architectural decisions concerning the creation of an event system. To solve this problem, it is necessary to use an architectural solution that is known to work and expansible enough to provide an answer to all the problems mentioned above. The solution chosen to overcome this issue was to implement the Subscriber software pattern.

The Subscriber pattern provides a way of dealing with events similar to the one used in RSS systems. The basic idea behind this pattern is that there is a class that acts as a listener of some kind of events. If another class is interested in being informed of the occurrence of an event of that type, it subscribes to that listener. The listener stores all the subscribers in some data structure and when an event is raised it informs every subscriber that an event occurred. Then, each subscriber acts accordingly as it desires. The subscription can be held forever or can be subject of a life span that ends after a certain period of time. Also, a subscription can be terminated by the subscriber itself or by the listener under certain circumstances. For example, if a file load completion event was sent to all subscribers and the file was closed, then there is no need of keeping that listener in memory. Consequently, it can be destroyed and all of the subscription requests are deleted.

An event in the engine is a subclass of the main *PMEvent* class, which specifies the name of an event. For instance, the *PMMouseEvent* class extends the functionality of the *PMEvent* class by adding the required parameters for the mouse position, pressed buttons or wheel scroll. So, the events can represent nearly everything that is necessary in the engine or even for a game that is developed on top of it. Therefore, it seems clear that this mechanism for transmitting events is quite expansible for any kind of situation. A class can be a listener or subscriber of any event, or both, and treat the event according to its needs.

4.3. File System Access

In most applications there is the necessity of accessing external configuration or data files, which change the way the software works or simply acts as a database for the program. The same is valid when dealing with both game engines and mobile applications. This is especially critical in game engines due to the necessity of creating and reading save-game files, configuration files, textures, fonts, et al. However, the support for file reading and writing in mobile devices can be severely reduced by the software or hardware platform for which an application is deployed. File systems change drastically between platforms and most of the times, file input and output usually raises security issues that must be handled carefully.

Nevertheless, the operations that can be performed upon files are essentially the same between platforms and include opening a file, reading from it, writing to it and closing the file handle. As before, this platform specific concept follows the engine philosophy and is present in the engine under the form of an interface which provides the generic methods for accessing and controlling a file. Similarly to the rest of the engine platform specific concepts, the file managers are created through a factory, which returns the correct type of implementation given the current definitions used during the compilation.

Besides the generic methods for opening, reading and writing to a file, the interface was taken a little further, as it provides methods for reading specific structures or data types, such as a whole line, a byte, the existing types of integer values or the whole file at once. Also, it specifies virtual functions that are used to check if the file was successfully opened and if the end of file was reached.

4.4. Networking

Nowadays, especially after the popularization of the web 2.0, most applications are somehow connected to the internet. Besides the communities and internet content, connection to a network might be important when an application needs to be updated or configured remotely, for example. However, most game engines do not provide a way of interacting directly with a network. Instead, the ruling tendency is to oblige the programmer to create a new layer upon the whole engine which deals with networking. This is not necessarily wrong, when dealing with high-level protocols and remote data management. But when low-level data structures such as sockets are involved, the engine must be carefully thought in order to adapt to several platforms if required.

So, as most of the aspects mentioned before, also the sockets are platform specific. Nevertheless, all socket implementations are somewhat alike in some aspects such as being either a client or a server socket and providing functions that allow sockets to be created, connect to a remote address, read and write data and close the data stream.
Similarly to the rest of the platform specific components, the socket layer architecture relies upon a socket interface which provides the common functions mentioned above. The socket is then created through a socket factory, according to the specified platform for which the engine is compiled.

It is not unusual to join threads and sockets when dealing with network capabilities. The reason for that is simple. In most platforms, sockets are read-blocking, which means that until a socket receives the specified amount of data that it is expecting, it will block the current thread. One "exception" to this, is the Windows Asynchronous Sockets which do not block even when the data is not received. Instead, this system relies upon the Windows Messaging System in which a message is received for any socket event that occurs. However, when such a feature is not available, it can be simulated by using threads. While one thread reads data from a socket and blocks until receiving the length of data that it expects, some other thread monitors the networking thread and proceeds with its job while the data is being retrieved. In this case, if the thread that is downloading data becomes unresponsive for too long, the control thread can terminate it, thereby freeing resources immediately and, if necessary, order it to try reading the data again or simply report an error.

An example of such case was implemented in the engine in a class named PMWebDownloader. The function of this class is to obtain web pages from the internet given its link, although it can be used for several purposes and expanded or modified to any other finality. It uses the HTTP protocol as the base for retrieving the web page data and is currently being used to, for example, download a XML data file from the Google Maps Geocoding API in order to find out the nearest street or location from the coordinates where the user is located. From this, it should be clear that implementing a protocol using the PunkMan engine is very easy, as the platform independence can be guaranteed through the base interface and the protocol classes can freely use any of these features to communicate across a network.

4.5. Thread Management

An important feature in any game engine is the capability of handling threads. In most game engines keep the thread control out of reach for the users in order to simplify the application programming interface with the programmer. However, although the threads usually fulfill their work successfully, a consequence of this is that the programmer is usually unable to control what happens in each thread. Also, usually the game engines do not provide a way to create and manage existing threads and to avoid deadlocks and thread-safe concurrency.

Moreover, threads are platform specific as they can be implemented in several ways. So, the usual approach was followed, creating an interface for the threads and implementing it as necessary for each platform. But, threads are not created as usual because the thread constructor is not public. Every thread must be created through another class called a thread manager. The thread manager class is a singleton class and is responsible for controlling every thread in the application, thereby guaranteeing that all the threads are centralized in one single component which administrates every existing thread inside the engine. Due to the fact that the class methods and variables are static, they are global to the entire application, regardless of the thread that is currently running. Therefore, by creating the threads through the thread manager, access security to critical sections can be easily guaranteed.

Critical section exclusive access is also ensured by the thread manager class. To specify that a code zone is a critical section it is only necessary to call the static method *enterCriticalSection* and specify a name for the critical section. When the thread manager class receives a request for entering a critical section, it searches its internal database and checks if the critical section name already exists. If there is no section with the specified name, then no other tried to access that specific piece of code and thereby, access can be freely granted. Otherwise, if the critical section is already registered in the thread manager it is possible that a thread is currently inside. To check if another thread is inside the critical section in which the current thread is trying to enter, the thread manager checks who is the owner of the critical section. If the owner is the thread itself, then it can enter. Contrariwise, in the case that the owner is not the current thread, the thread manager will suspend it until a *leaveCriticalSection* command is received from the owner.

Due to the fact that threads usually receive a callback function as the argument and the callback function declaration can drastically change between platforms, the engine defines a PM_THREAD_START_FUNCTION constant which represents the current platform's callback naming convention. Similarly, a thread must provide some way of being manipulated in order to pause, resume and terminate it, for example. Once again, these handles can be completely different in distinct platforms. For instance, while in Linux the thread id is a *pid_t* value, in Windows the system provides a HANDLE value when a thread is successfully created. Therefore, the engine includes also a PM_THREAD_HANDLE_TYPE definition which can be set accordingly to represent the correct data type for the thread handle values.

4.6. Other features

Besides the features mentioned above, it is also important to mention two more classes that were implemented in the core components layer. The first one is the logging system. This class obeys to the Singleton pattern and provides a way of logging the system activity to an external log file. This is particularly useful due to the fact that the engine can't throw exceptions due to the limitations of some platforms which do not support this language feature. So, in that case, instead of throwing an exception, the error is recorded in a file and the engine exits silently whenever possible. However, this is clearly useful for situations where everything went according to the plan and the programmer merely desires to register the engine activity in an external log file. This class relies heavily on the *PMFileManager* class, therefore being a good example of how this feature can be used.

The other important class is called *PMTime*. Although it does not provide a lot of functions it is quite important to maintain the coherence of a game. Time management classes are essential to game engines due to the fact that they are responsible for controlling the animations, input times, frame rate, etc. So, this is the place where the time is controlled inside the engine and can be easily extended to provide date and time information, application milestones, benchmarking, and to tell the total time for which the application is running, for example.

5. The Renderer Layer

The renderer layer is responsible for the creation and initialization of every component that is necessary to be able to draw to the screen and to support further rendering options. Therefore, this layer comprises not only the renderer and a canvas setup, but also a set of objects that can be used to draw to the screen, such as sprites, text and meshes. The great challenge of this layer is, once again, maintaining the engine portability mostly due to the difficulties presented by the differences between rendering API's. The diagram below shows the architectural layout for this layer:

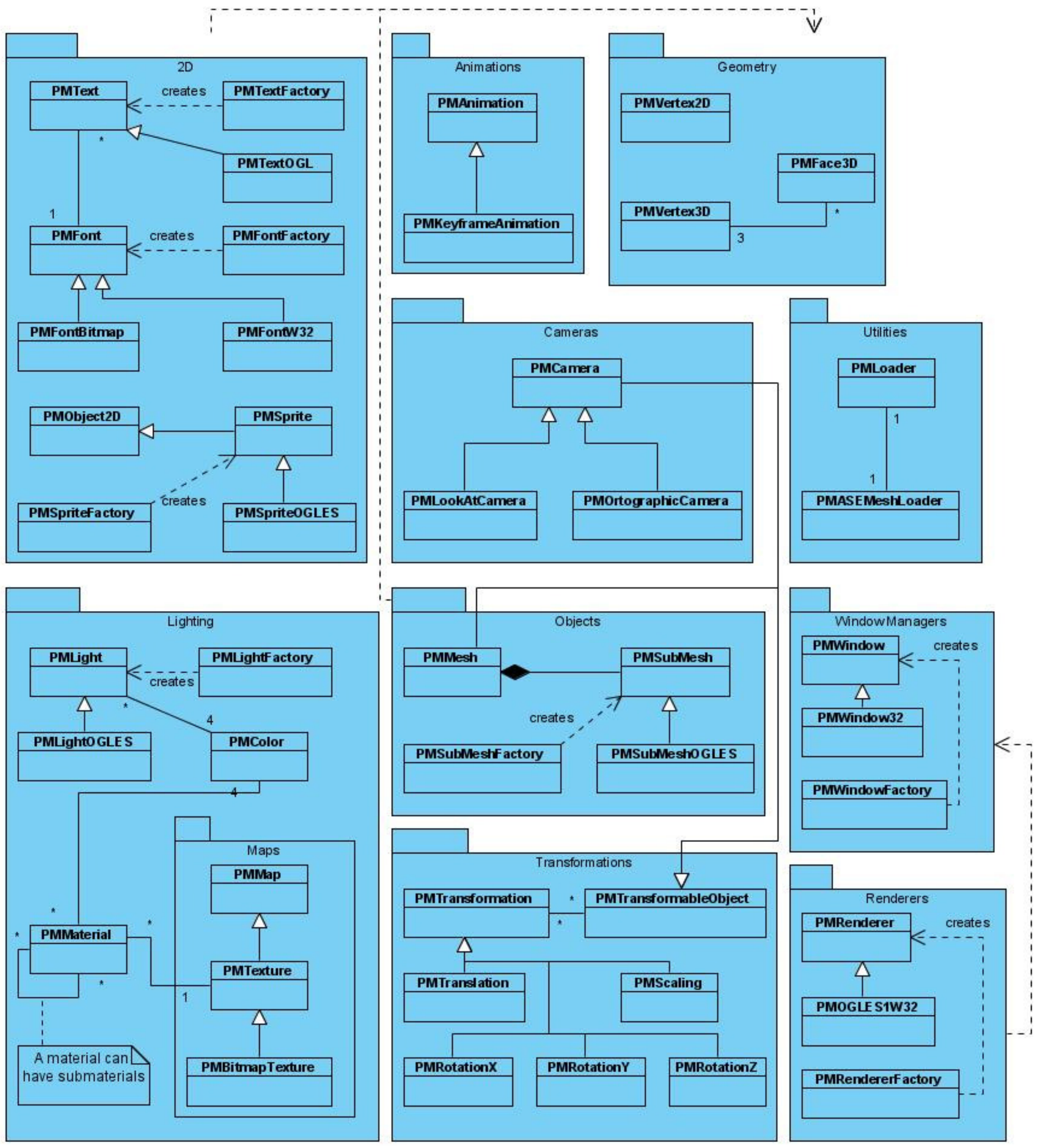


Figure 18 - Renderer layer

5.1. Window Managers

The main objective of a game engine is to provide a simple way of creating them and presenting them on a screen. In order to draw onto a screen, two things are necessary, just as in real life: something or someone that paints and somewhere to draw to. The first component is called a renderer and is responsible for all the drawing activities. The later is called, in the context of this engine, a window. Therefore, it is important to make it clear that a window is simply a canvas-like object to where the renderer can draw. However, this is once again a topic that is platform-specific. For example, while in Windows systems the drawing can occur in the client area of a simple window, in a mobile device this may not be so straightforward. In those cases, it might be necessary to use a specialized structure that is not necessarily a window, but rather a canvas, viewport or panel, in order to be able to draw.
To address this issue, an interface called *PMWindow* was created that specifies the common features of this type of drawing target, such as width, height, title, frame rate and update methods. However, some specific functions, most of which are useful for connecting to a specific renderer, are not specified in the interface. Instead, they must be created on the implementation itself. For instance, a Windows window is connected to a renderer through a special handle, specified by the HWND value. This value exists only in Windows and is definitely necessary in this case. But, in some cases, there is no need to specify where the drawing is taking place, if the platform system takes care of that all alone, such as what happens with XNA. For that reason it is preferred to keep these platform-specific methods in the implementation to avoid engine overhead.

But why is a window management system necessary then, if the renderer itself could handle the possible creation and drawing to a window? There are two reasons for that. By separating these concepts, the engine becomes more logical and modular, promoting expansibility and easing modification, bug corrections and updates whenever it becomes necessary. In the other hand, by using a split system such as the one implemented in *PunkMan*, the rendering can take place in multiple surfaces at the same time, by simply switching the underlying *PMWindow*. This eases the work of drawing to multiple windows or viewports, which can be very useful, for example in the case of a split-screen multiplayer game.

5.2. Renderers

Being one of the core aspects of the engine, the renderer's layer comprises the initialization, configuration and utilization of the underlying rendering system. This is the base of any game engine, as without a renderer nothing could be drawn to the screen. It is also one of the hardest features to implement in a game engine that is intended to be portable, especially in mobile platforms.

There are two widely known main rendering API's, namely OpenGL and DirectX. Besides these two alternatives, there is a great number of other rendering API's available on the market, mostly performing its work in software. However, although these platforms have a common goal in mind, the way they follow to reach it is totally different. For instance, while DirectX is completely object oriented since its origin, OpenGL has been unsuccessfully trying to convert itself to an object oriented design for some years, thereby following up until today a merely procedural architecture. Also, the capabilities of each existing API and the provided methods share some contact points, but differ greatly in some other aspects. Although the very basic rendering functions such as clearing the backbuffer or initializing the essential definitions are very similar between these API's, some other features such as transformations are very different in these alternatives. While OpenGL uses column major notation for matrices and performs transformations by using *glPushMatrix* and *glPopMatrix*, DirectX prefers to use row major notation and uses a different approach. Everything in DirectX is centralized and controlled by a Device object pointer, concept that is inexistent in OpenGL. The transformations are submitted to the device which applies them to the scene. The main difference here is the lack of the concept of a transformation stack as the one figured in OpenGL.

Given all this it should now be clear that implementing a renderer layer that is able to work with such different implementations is not an easy task. However, there is yet one important point to consider, which could help to design an architectural model for the engine. This engine is supposed to work in several platforms and is mainly oriented to mobile devices, which do not support some of the complex features that are available in the full API.

So, some of the conflicting parts of the API could be ignored in favor of the basic features which are supported by all, or at least most of, the platforms. However, the philosophy for this engine requires the effort of making it a truly multiplatform system, without limitations up to where is possible.

To accomplish this goal, and considering the serious differences between API's, one fundamental idea must be present at all times. Every aspect that is related to the renderer must be forcefully contained inside the renderer class. And why is that so important? A renderer is a transversal component of a game engine and is used throughout many layers. Consequently, there is the tendency to spread renderer functions across the whole engine and outside of its own layer in which it should be contained. This is a common practice, that eases the work of enclosing every renderer aspect inside a specialized class and it always leads to chaos in the engine. Most importantly, this is clearly one crucial step to make the engine support multiple rendering API's.

Based on all this, the renderer layer starts, as before, with an interface that reflects the common characteristics of the renderer API's or those which can somehow be converted to support a common method format. The correct renderer for the current configuration is given by a factory that initializes its base parameters and returns the created pointer to a renderer. The functions can then be used through the common interface *PMRenderer*. It is worth to notice some aspects in the implementation, related to the problems described above. The first point is that, in order to guarantee the enclosure of this layer, the geometrical transformations are all done by the *PMMatrix3* or *PMMatrix4* classes internally. Only after all the transformations were calculated is the resulting matrix submitted to the renderer and applied to the scene. The reason for that is simply to avoid using renderer-specific transformation functions such as *glTranslatef*. So, all the calculations are done by the engine automatically and transmitted to the renderer in the end.

In what concerns transformations, there is still another relevant issue. OpenGL, as mentioned above, has the capability of storing and restoring the transformation matrices in each matrix mode by using a matrix stack. Although DirectX does not use this concept, it uses something similar called scene states. So, although the actual implementation is quite different in both API's, the same result can be obtained in the end. Consequently, two functions were specified in the *PMRenderer* interface, called *saveMatrix* and *restoreMatrix* that must be implemented by the concrete renderer class that inherits from this interface. By using this method, each renderer API chooses the best way to save and restore a given matrix state.

A similar problem happens with textures. OpenGL does not provide any way of loading a texture by itself. The data loading has to be done manually and saved in an array of bytes, in order to be stored in OpenGL's memory through *glTexImage2D*, for example. Contrariwise, DirectX is already capable of loading the most common types of images formats directly by using the function *D3DXLoadTextureFromFile*. The result of loading a texture and saving it in the OpenGL system is an integer number that uniquely represents the texture in the application (after calling *glGenTextures*), while in DirectX the result is a pointer to an IDirect3DTexture9 object, which can then be used freely across the whole application. So, in order to provide a way of working with both approaches and any other that might become necessary, the *PMRenderer* specified a *saveTexture* function which receives an array of PMByte representing the texture, its width and height, a name to identify the texture if necessary and finally a parameter that specifies if the texture uses the alpha channel. As a result, by specifying a name, each implementation can store the texture in a hash map, for example and map the name to the respective value, which can then be retrieved and used directly.

In order to complement the previous function, there are two more functions that are also essential, namely *setTexture* and *removeTexture*. Once again, as each renderer does this differently, there is the necessity of letting the implementation decide what to do in each case. The purpose of these functions is quite straightforward. The first one simply activates a texture and applies it to anything that will be drawn while the second function deactivates the application of the texture.

The remaining functions do not raise any problem to understand or implement, as they are mostly related to the setting of some parameters such as clear color, lighting state, material state, culling, ambient color, etc. However, these setter and getter functions that were specified in this interface are merely the most basic ones, not only for the sake of simplicity, but also because these are the functions that are indispensible for the renderer utilization. Nevertheless, introducing new functions and capabilities to the rendering layer can be added very simply, if necessary, in the implementation.

Finally, it is important to refer one last point. In order to be able to perform its drawing work, a renderer must have some canvas-like structure where it can actually draw, regardless of what type of rendering API it is being used. In the engine, this is called a *PMWindow* and has the responsibility of providing the necessary means to communicate with a renderer in order to allow the drawing work of the later. Therefore, when a new renderer is created, a *PMWindow* pointer must be passed in order to let the renderer know where it should draw. This is an essential detail, because by doing this, the renderer is capable of rendering to multiple surfaces or canvas, merely by switching the underlying *PMWindow* that is associated with it.

5.3. 2D system

Two dimensional elements are an essential component in a game engine for several reasons. The first and most common usage in game engines is clearly is to create the "Heads Up Display" and user interfaces. Although some engines are already featuring 3D user interfaces, these features consume a lot of space and memory, and aren't usually as usable as they should. Some engines, even 3D also use 2D entities to use something known as billboarding. Although this technique is disappearing slowly, it was widely used to give the illusion that something that was apparently a three-dimensional object was always facing the viewer. This was mostly used for simplifying complex objects such as trees and, sometimes for interface elements like the life meter above a character that follows it around. 2D objects are also still widely used to create particle effects such as explosions, gun shots and smoke, for example. Even though the volumetric effects are gaining importance nowadays, these 2D effects are cheaper to achieve in most graphic cards, as most of the computers are still unable to easily cope with the industry achievements such as these 3D volumetric effects.

Given the importance of such a topic, a 2D object sub-layer was added to this engine. However, another renderer related problem appears when trying to develop such a feature. While DirectX provides direct support for two-dimensional objects through the *ID3DXSprite* interface and its methods, OpenGL does not support 2D rendering directly. In this later case, it is necessary to find a workaround to make it support the creation and display of 2D entities. Fortunately, the solution is simple and consists in creating an orthographic camera that, due to the parallel projections that characterize it, matches the screen viewing space. Therefore no more viewing transformations are applied, forcing the object to remain in screen space and not in a real world space, which makes the object vertices match the screen coordinates.

The implementation of 2D components in this engine start in a generic abstract class called a *PMObject2D*, which inherits from *PMObject* and stores a pointer to the renderer and an orthographic camera. If this camera is specified to the object, then it will be used to render.

Otherwise, if the camera remains unspecified or is NULL, the object will trust that a camera was previously setup correctly and will try to render itself anyway. The *PMObject2D* specified a pure virtual function called *draw* that every other object that inherits from this class must define. Consequently, in order to draw the object to the screen, a *PMSprite* class was developed that sums up the characteristics of a sprite. A sprite is basically a 2D rectangular element that is placed on the screen, which has a position and a texture. However, although DirectX supports sprites directly, as mentioned above, OpenGL does not. So, it is considered a platform-specific concept and as a result must be abstracted. So, *PMSprite* represents a generic sprite interface that must be implemented for each renderer, such as lights for example. Therefore, a *PMSpriteOGLES* class that represents an OpenGL ES sprite was created and deployed on the engine. It can be created, such as all of the engine platform-specific concepts through a factory that is able of creating the correct type of sprites according to the engine specifications and definitions.

5.4. Transformations

In what concerns Computer Graphics, one of the most important topics is clearly geometrical transformations. They are responsible for placing the objects in the world as the programmer desires them to appear and provide the possibility of changing the position or aspect of the objects while an application is running, in order to create an animation, for example. Three-dimensional geometrical transformations are represented by square matrices with dimension four and the most important and well-known transformations are translation, rotation and scaling. It is worth to mention, however, that these are not the only existing transformations, as there are some others such as shearing, but they aren't as common as the former three.

In addition, when dealing with transformations, several problems may appear. The first one is related to the fact that they are a renderer-specific issue, i.e. renderer API's use transformation matrices differently. For example, while OpenGL uses a row-major notation for the matrices, DirectX prefers a column-major notation. Besides that, if a geometrical transformation is represented by a matrix, the transformation that results from applying two transformations to an object is equal to the multiplication of the two transformation matrices. Consequently, as the matrix multiplication is not commutative, so it happens with transformations. Therefore, the order in which transformations are applied is relevant and must be done carefully.

The solution to these problems was implemented in three different parts. The first problem, related to the engine expansibility in what concerns geometrical transformations is quite easy to solve. Instead of considering that a transformation is merely a matrix, it was considered a separate class instead. Therefore, a generic interface called *PMTransformation* was created to represent a transformation. From this interface it is important to refer that each transformation has an optional name, and three important virtual functions named calculateMatrix, applyTransform and *stopTransform*. Then, some classes were developed that implement this interface and perform some concrete geometrical transformations, namely *PMTranslation*, *PMRotationX*, *PMRotationY*, *PMRotationZ* and *PMScaling*.

The second problem, related to the renderer format, can be solved by taking the responsibility of calculating the transformations from the renderer. As the transformations are merely matrices, the calculation can then be taken care by the matrices themselves. So, the *PMMatrix3* and *PMMatrix4* objects are responsible for doing their own calculations of the transformation matrices. The transformations are computed internally by the engine, multiplied by the other transformations that are to be applied and only after the final transformation matrix has been calculated, it is submitted to the renderer. The renderer must accept a *PMMatrix3* or *PMMatrix4* object, depending if the application is working with 2D or 3D graphics, and then uses the matrix as it pleases, transposing it if necessary.

The answer to the last problem can pose a larger challenge, because it is not necessarily a real problem. If the programmer is sufficiently careful, then no big problem will rise from the fact that he has to guarantee that the correct order is maintained at all times. But in certain cases, it is desirable or even mandatory to remove a transformation from an object after it has been set, for instance, during an animation. There are two possible approaches to solve this problem. The first one is to keep track of the pivot point of the object. By knowing its position, further transformations can be applied seamlessly, by translating the object to the origin, applying the transformation and putting it back where it was previously. This solution is very easy to implement and to use, but does not provide the means to do a real "rollback" on the transformations. Instead, it just eases the work of adding new transformations on top of the existing ones.

The second solution is more expansible and provides the means for an easier manipulation of the transformations applied to an object. It is called a transformation queue and simply consists in the sequence of transformations that are applied to an object. This has several advantages over the previous approaches, but also obliges to implement some data structures such as a queue, the transformation classes mentioned above and a special object called a *PMTransformableObject*. This object stores the transformation queue information and exposes it to the exterior. Thereby, any object that wants to take advantage of this technique must only inherit from this class and it is ready to use the transformation facilities that become available, because there are no pure virtual functions that must be implemented. Transformations in a queue can be removed or added in runtime, and their position in the queue can be changed. Also, the transformation can be obtained by its name, and its parameters can be modified in runtime directly. This makes the job easier for anyone that wants to change or control a specific object behavior in a given time or to create animations. Also, by using this method, the pivot point of the object can also be easily obtained and if the user prefers to use the technique described above, it is free to do so.

5.5. Animations

There are not too many examples of games that do not have animations. An animation consists basically in a transformation or sequence of transformations applied to an object or a part of an object for a certain period of time. Animations are useful to create the illusion that a character is walking, a car is moving or even to simulate physics in an engine for example. However, animations are not restricted to geometrical transformations. A texture or material can also be animated, making it change somehow through time. A good example of such a thing is an animated sky with clouds. Usually, the top layer consists in a blue gradient texture representing a cloudless sky and beneath that layer come other layers with clouds that may be moving to simulate a real environment. In these cases, the geometrical layers do not need to move as it is only necessary to animate the texture coordinates of its vertices to mimic the movement of the clouds.

It is then necessary to find a way of including animations in the game engine. However, considering these types of animations, a very generic interface must be provided to be able to both reflect all the needs of any kind of animation and to be sufficiently expansible that it can be easily extended to be used in a whole new type of animation type that is necessary to create. The first big step in this direction is clearly to detach the animations from anything else in the engine that is to be animated. For example, it would be extremely easy to build an animation layer that received a *PMTransformableObject* and simply applied the transformation directly to it. That solution, however, would be too rigid and hardcoded and consequently would not work for texture, material or even vertex animation. The solution passed by creating a very simple and generic *PMAnimation* interface class, which holds a pointer to a renderer and three important animation member functions, namely *update*, *applyAnimation* and *removeAnimation*. The first must be called in order to advance the animation in time, updating the current transformations that will be applied by the concrete implementation. The second function simply applies the calculated transformation to an object, while the last method stops the animation from being applied.

In game industry nowadays, the most commonly used types of animations are those based on keyframes. Keyframes consist in a sort of milestones that are set on the animation timeline to indicate that something has changed. In the case of a 3D object, a keyframe might hold a difference in the object's position, rotation or scale, for example. The engine provides a way of working with keyframed animations, through a class named *PMKeyframeAnimation* that inherits directly from *PMAnimation*. When a *PMKeyframeAnimation* is created it is empty, meaning that it possesses no keyframes at all. Keyframes may be added afterwards by using the *addKeyframe* function. After the keyframes are added, a function called *computeDuration* is called automatically to calculate how much time does the animation lasts. This information is useful because it is necessary to know the total length of the animation in order to calculate the interpolation factor between keyframes. After all this, the animation is calculated based on the specified keyframes. In the time between keyframes, the transformations of the same type are linearly interpolated to provide a smooth transition in the transformations that happen between these moments. But, if at frame 10 there is a translation to the position (10, 0, 0) , at frame 20 there is only a rotation in X of 90º and finally in frame 30 there is another translation to the position (20, 20, 0), the rotation is interpolated from the keyframe 0 to the keyframe 20 and the translation is interpolated from the keyframe 0 to the keyframe 10 and then, from the keyframe 10 to the keyframe 30, in two different sections of the animation.

Animations are then very easy to use, as most of the job is done automatically. If a new animation type is to be added to the engine, it can simply use the functions defined by the *PMAnimation* class to update and render, and implement any other that is necessary to provide the remaining support for the new animation type. Then, the objects that are subject to animations, such as *PMTransformableObject*, simply call these functions and the animations will be updated and drawn accordingly to what is necessary.

5.6. Geometry

The geometry sub-layer consists in an extension to the math layer that is present on the Core Components Layer. It comprises all of the necessary data structures that are required to draw two or three-dimensional objects. For that reason, it is the base of the object layer and might be expanded accordingly if new needs appear. It was an architectural decision not to join the data structures present in this layer with the ones on the math layer presented above, due to the fact that these are closely tied to the object geometry and can be considered to be one abstraction level above then the ones mentioned above.

This sub-layer comprises three different classes that serve as the basic support for complex object creation, namely *PMVertex2D, PMVertex3D* and *PMFace3D*. The first two classes represent a vertex and are very similar in all aspects, with the understandable difference that the first represents a two-dimensional vertex while the later holds all the necessary data for a three-dimensional vertex. In what concerns this engine at this level of abstraction, a vertex must be composed of four different components. The first, and perhaps most important, are its coordinates that specify where is the vertex located in the world. After these appear the normal coordinates that are only relevant to three-dimensional vertices. These represent a 3D vector that indicates what is the vertex's normal in order to calculate if this vertex is visible or not. Of course, this is only relevant after the vertex is included within a polygon. Then comes the texture coordinates, which are used to calculate the placement of a texture that may be applied to the face or faces to which this specific vertex belongs. Finally, the vertex color is the component that indicates what color the vertex shall have if no materials have been specified and if the renderer is using vertex colors to draw.

However, although all of the basic parameters were covered in the implementation that was described above, these components might still not be enough sometimes. For example, when using multi-texturing techniques, two texture coordinate sets might have to be used, and in that case, this specification is not enough. The same applies to some cases where vertex or pixel shading is applied in order to change the way lighting appears. For instance, to use a normal mapping shading technique, not only two textures must be used but also two sets of normal coordinates. Therefore, it is important to guarantee that these data structures are simple but powerful enough, but also able to be extended to match the needs of the end-application.

It is still significant to mention that each vertex has the capability of being transformed by itself. Generally, this is not what happens when the objects are rendered. Instead, before one object is drawn a matrix is set in the renderer to transform it accordingly. Still, the method of transforming an object by using vertex-by-vertex transformations is very used nowadays and therefore, as it does not bring a big overhead to the engine, it was implemented within the vertex data structures. So, all of the basic operations (translation, rotation in X, Y and Z, and scale) as well as a generic *transform* function that takes a matrix and transforms the vertex accordingly where implemented both in *PMVertex2D* and *PMVertex3D*. To complement these functions, an overloaded operator for multiplying vertices by matrices was also implemented in these classes.

As a final point, the *PMFace3D* specifies a three-dimensional triangle in the engine. It is called a face in order to obey the general nomenclature given in 3D applications and therefore maintain a certain level of coherence between these applications and the engine. A face is thereby composed by three vertices and a face normal. It also holds a material identifier that uniquely identifies the material that is assigned to the current face, in order to be applied later. This class consists in the base atom for building and drawing objects in the engine.

5.7. Lighting

Lighting is one of the most important computer graphics topics that are present in games, because without it nothing could be seen and a good level of realism couldn’t be achieved. Lighting starts with colors, which are usually defined in a RGBA system. Consequently, the first step when implementing a lighting system is to include the concept of color in the engine. That is done by the *PMColor* class which represents a RGBA color value whose individual channel values range from 0.0 to 1.0. The color class also provides a *normalize* function which clamps the values to its range, if they have been exceeded.

After the colors come the materials, which are responsible for representing the way that the light interacts with an object. They are responsible for making an object look shinier or dimmer, its color and give the viewer an idea about what an object is made of. The materials are, in its simplest form, defined by its ambient, diffuse, specular and emissive components, and the shininess. However, it is also important to consider that a material may contain a texture that complements it and therefore it must be saved within this class, due to the fact that materials and textures are interdependent. If this material uses no texture, then the pointer is simply *NULL* and the texture is not applied. To promote expansibility, materials work in a hierarchical way, i.e. a material can have several sub-materials. Most 3D file formats use materials that can have submaterials, such as OBJ or ASE files. This can be very useful for the cases where an object is composed by several different parts that require different materials. After setting these properties, or leaving the default settings, the material can be applied by using the *draw* function and removed afterwards by using the *end* function. It is important to mention that if an object calls the *draw* function of a *PMMaterial* object and does not call *end*, all subsequent objects will inherit the material characteristics, unless other material is specified and activated meanwhile.

Finally, the lights themselves consist in a set of parameters that can be used to specify the exact behavior of lighting in a scene. However, OpenGL does not separate different light types such as point, directional and spot lights. DirectX requires that the programmer specified what light type should be used. Therefore, the engine provides a class named *PMLight* which is can be parameterized with all of the possible attributes of any type of light. Then, according to which parameters are set, the system decides what type of light should be created and submitted to the renderer.

The parameters that can be set for a light are:

- **Position** – The light's position in the world
- **Direction** – Useful for directional lights, represent the direction in which they are pointing
- **Ambient** – Ambient color component for the light
- **Diffuse** – Diffuse color component for the light
- **Specular** – Specular color component for the light
- **Exponent** – Exponent factor of the light
- **Cutoff** – Cutoff value for spotlights

- **Constant Attenuation Factor** – The constant component of attenuation of the light
- **Linear Attenuation Factor** – Linear component of attenuation of the light
- **Quadratic Attenuation Factor** – Quadratic component of attenuation of the light

It is also important to mention that light creation is renderer dependant and consequently can't be directly abstracted by a single class. Therefore, *PMLight* represents a light interface that must be inherited by a concrete light implementation that is created correctly according to the renderer being used. Consequently, like before the lights are created through a specialized factory that creates and returns a light based on the current renderer and application settings.

5.8. Maps

In its most common form, a map can be a texture that is applied to a material and used as a mean of giving a different look to a certain object. However, a map can also be used as a bump, displacement, normal or lighting map, for example. A map is generically something that is placed over a surface to change how the surface looks or how it reacts to light. Consequently, it makes sense that these structures are commonly associated with the materials and lighting altogether.

Given this, it seems logical to design a way to allow new map types to be added easily. To promote expansibility, a *PMMap* class was created that concentrates the common main aspects behind a map, namely its width and height, a pointer to the renderer, a name and two member functions for starting and ending the drawing of the map. Upon the creation of this class, it was inherited to create a common texture class, called *PMTexture*. Besides the members inherited from *PMMap*, this class also includes a specific texture identifier, an array of bytes to store the pixel data, an eight bit integer value to store how many bytes does the texture have per-pixel and one boolean value that indicates if the texture uses the alpha channel or not. It is important to say that it is necessary to store the texture's pixel data because of OpenGL and renderers alike. Due to the fact that OpenGL does not recognize and is unable to load any texture file format, the only way it has of loading textures is to read bytes directly to memory and save them there. In the other hand, DirectX does that alone and does not require the programmer to manually set every byte manually in the texture memory. However, it gives the opportunity of doing so, if it is desired. So, to address this problem it was necessary to obey the "least common multiple" as soon as it is expansible enough and for that reason raises the necessity of saving the array of bytes containing the pixel data.

Finally, after the *PMTexture* class was created, it required some way of using texture data correctly. Consequently, a simple class called *PMBitmapTexture* was created, which is able to load texture data from simple BMP files. This is a really simple class, but is still able of successfully loading and presenting a bitmap image on the screen as a texture. So, the goal of this class is, beyond the fact of serving as a texture loader, acting as an example of how the texture mapping can be implemented to load any kind of texture file easily.

5.9. Objects

Usually called a mesh, an object is, putting it very simple, anything that is visible on a scene. The reason why objects are called meshes is due to the fact that they are composed by an interconnected set of adjacent triangles, which are the only thing that the renderers can actually draw. Therefore, the objects are actually meshes composed by triangles that are then submitted to the renderer to be drawn on the screen. Still, it is important to refer that some renderers such as DirectX already contemplate the concept of mesh and are able of loading and storing meshes directly in memory. In these cases, drawing a mesh is very simple, being sufficient to call a draw method included within these objects. Contrariwise, OpenGL does not contemplate the existence of such concept. So, in order to ease the work and provide a layer of abstraction over the concrete renderer that is being used, a *PMMesh* class was created to store the information related to a mesh in the engine.

Actually a mesh is a very simple class, due to the fact that it is merely a top-level container. It is composed by a set of sub-meshes that are what actually contains geometrical data. Besides this, a mesh is very similar to the *ID3DXMesh* interface present in DirectX, having a *draw* method and a bounding box, which may be used to test it against collisions. Also, it has a member function that gives the programmer the possibility of adding new sub-meshes to the current mesh. Then, when *draw* is called, every sub-mesh is drawn to the screen.

A sub-mesh is an object of type *PMSubMesh* and is responsible for holding the necessary geometrical data that corresponds to a given object. So, a sub-mesh holds the information about the vertices that compose it, which may be later used for augmenting the precision of some optimization algorithms such as octrees. The vertices are stored in an array present in the class and can either be added manually or loaded from a file, if necessary. None of this is mandatory to use, but the engine provides both capabilities in order to ease its utilization. A sub-mesh also stores information about the faces that compose it, in order to ease the task of loading them. Also, as a sub-mesh can have at most one material that can be applied to it, it stores it in the class as well. It is important to refer that, if a mesh is composed by parts that take each one a different material, and then each part must become a sub-mesh of the container mesh, due to the fact that one sub-mesh must have one material only. Finally, just like the main mesh, a sub-mesh also contains a bounding box that is calculated after the sub-mesh is built. This might be useful not only in terms of collisions, but also when an animation is loaded, for example. Think of the case of a humanoid model that figures a walking animation. In that case, it may be desirable to access each arm's bounding box individually in order to find out if it is interacting with something.

5.10. Cameras

Regardless of the renderer used, cameras are one of the hardest topics to implement in any game engine, due to the fact that they are usually represented by two distinct matrices that are created from a great number of parameters that can easily be mistaken. Also, the implementation of the camera matrices varies between renderers, augmenting considerably the difficulty of creating a multiplatform layer for cameras. The solution for this problem was somehow similar to the problem of geometrical transformations, also described in this chapter, and consists in making the *PMMatrix3* and *PMMatrix4* objects do all of the work.

After saying this, the camera implementation in the engine starts with the *PMCamera* class. This is a generic free camera that is unbound to any restrictions and has no specific target to look. Instead, it only knows its position, look, up and right vectors. Besides these positional and orientation vectors, the camera also keeps its view and projection matrix that will be calculated when the camera definitions are set by the user.

In what concerns the projection matrix, two options are available. The camera can figure either an orthographic or a perspective projection. Choosing which alternative is to be used is only dependent on choosing the *setOrthographicParameters* or *setPerspectiveParameters* in the camera. This may be changed during the execution of the program, making the camera re-adapt to the new settings automatically. These alterations are immediately communicated to the projection matrix which automatically calculates the new matrix that represents the new settings. In the other hand, the view matrix is calculated automatically from the position, right, up and look vectors. If any of these values changes or if a transformation is applied to the camera, it automatically updates the view matrix to reflect the changes.

New camera types are also very simple to add. In the case of this engine, a look-at camera was added simply to the engine by specifying how it should behave when a change occurs. It inherits directly from the *PMCamera* class and uses most of its variables, except that it also needs a target vector to look to. However, as this camera type doesn’t need a look vector, it would be rendered useless. Therefore, instead of declaring a new variable for the target vector, the look variable is reused for such a purpose and consequently, the look-at camera simply differs from the camera implementation in the calculations being applied to the view matrix. The remaining calculations are already being performed inside the *PMCamera* class.

By using this method, the cameras can easily be detached from the renderer. Their matrices are calculated externally and when all the calculations have been performed, the renderer receives the matrices ready to upload to the graphical pipeline. As a consequence, creating new types of cameras, changing the behavior of the existing ones or customizing the calculations becomes very simple through this technique.

6. Scene Management

The scene management layer consists in the set of classes that control when, how and what objects appear on the screen. It is responsible for maintaining the organization of a graphical scene of any type and providing straightforward manipulation options over it. More important than that is the fact that the scene management layer must be able of easily loading scenes and provide some way of optimizing its rendering, especially when the engine is targeted at devices that may have low hardware and software capabilities.

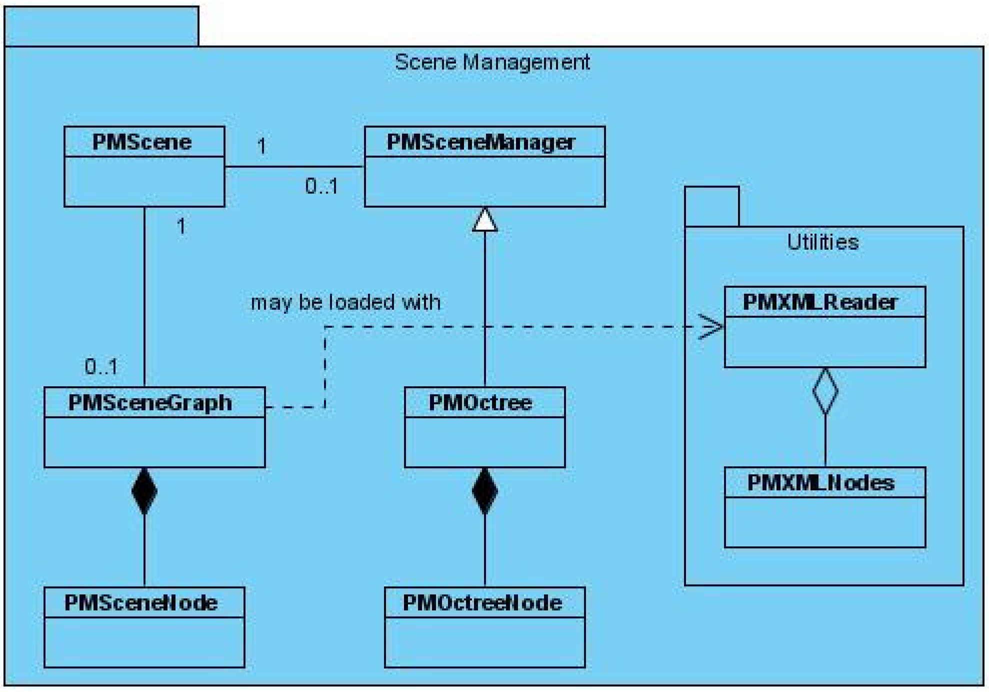


Figure 19 - Scene management layer

6.1. Organization of a scene

The problem of managing a scene is one of the most complex and studied problems in the area of computer graphics. In most cases, performance is essential, mostly when talking about mobile devices with reduced computing capabilities. Even if the target platform has a high performance and integrates the best hardware for rendering extremely complex scenes, a graphical application that is not optimized can run slowly or with some delay. Therefore, it is important to attempt to guarantee that a scene can be rendered in an optimal way, increasing the performance and efficiency of the application.

But before addressing this problem it is relevant to consider how a scene can be implemented in an engine. A scene holds everything that is required for rendering, either it is an object, a light, a camera or any other entity that is necessary for supporting the drawing, such as materials or transformations, for example. A scene is represented in the engine by the *PMScene* class which provides methods for adding basically any kind of object to the graphical scene. Also, to create a graphical environment it is merely necessary to initialize a *PMScene* object and the engine will be ready to be used. For instance, a basic application can be created by using the following code:

```
void main()
{
	PMScene* scene = new PMScene();

	while (scene->getWindow()->isAlive())
		scene->draw();
}
```

This initializes the entire graphical environment to support rendering and displays an empty window that is kept open in the application main loop. The loop is terminated when, by some reason, the window is no longer alive. This might happen when the user presses a certain key or due to some error, the window is forced to close.

After the scene was created, new objects can be added to it dynamically by using the functions specified in the *PMScene* class. Also, the underlying renderer and the window that support the scene rendering can also be obtained and configured by using the methods that are present in the scene. For example, to set some of the renderer definitions such as lighting and material states and the ambient color, it would be sufficient to use the following code:

```
PMScene* scene = new PMScene();
PMRenderer* render = scene->getRenderer();

render->setLightingState(true);
render->setMaterialState(true);
render->setAmbientColor(PMColor(0.0f, 0.0f, 0.0f, 0.0f));
```

To optimize a scene, the first step is to find a way to organize the graphical scene into a data structure that is able of storing both the objects and the relationships between them. It is essential that the chosen data structure is capable of providing the means to render the objects in the correct order and preserving the connections and relations that tie these objects to each other. The most commonly used structure that is used to organize a scene is called a scene graph. It consists in a graph where each node represents an object or scene entity and each edge of the graph represents the relationship between two of these objects. This is an adequate structure to represent a scene, rather than a tree for example, because the relationships between entities cannot always be represented in a hierarchical form. As a result, a graph is more adequate to represent it and it is capable of rendering a scene without compromising the order of the objects, its relationships and without drawing the entities more times than what is necessary. In this engine, a scene graph is implemented through the *PMSceneGraph* class. This is the class that holds all of the information that defines a scene through a graph. The scene graph itself is composed by a set of children of type *PMSceneNode* which store both a *PMObject* that represents the scene entity that is contained in the node and an array of the node relationships.

A scene graph is very useful to organize a scene and is also able of optimizing the rendering up to a certain point. Still, as it was mentioned above, the problem of optimizing a scene is very complex and due to its nature it does not present one unique solution. For example, the optimization method for one graphical scene that occurs in an external environment such as an open sky arena may be drastically different from one that occurs inside an apartment composed by a set of rooms. There are a great number of optimization methods that could be used to optimize the drawing depending on the scene to render, such as octrees, BSP Trees, Portals amongst many others.

To solve this problem, the solution was to implement the concept of a scene manager. A scene manager, represented by the class PMSceneManager consists basically in an object that takes a scene graph and optimizes it through some chosen method to be drawn afterwards. Every class that inherits from the *PMSceneManager* must implement a *draw* method and is able to use the provided scene graph, passed in the constructor, to pre-load any necessary data and render the optimized structure. However, the main goal behind this class is to provide a way of changing the scene manager dynamically. The *PMScene* class possesses a *PMSceneManager* pointer that indicates which scene manager shall be used. This pointer is initialized to *NULL*, indicating that scene management is inactive and only the scene graph should be used.

Otherwise, if a scene manager is specified the system will use that one instead and render it according to what is dictated by the specific implementation that was provided.

An implementation of this class is included in the engine, in the form of an octree scene manager. These octrees are represented by the *PMOctree* class whose goal is to subdivide the space into cubes that are organized into trees. The algorithm starts by dividing the scene space into eight equal cubes. If a cube contains an object, then it is subdivided once again in eight equal cubes and so on. As a result, the cubes that define volumes of the original space are stored in a tree-like structure and the algorithm stops when a specified depth limit is reached. Then, the final result can be used for both viewport culling and collision testing between objects, augmenting the performance of both the rendering and the collision system. It is also important to mention that, in order to increase the performance of the algorithm, the recursion was removed from the octree creation, using a queue of unprocessed nodes instead of calling the build function for each node.

6.2. Utilities

To support the scene graph loading for the engine, an utility module was created that supports direct loading of scene graphs from XML files. The basic component of this module is the *PMXMLReader* class that uses the *PMFileManager* class to perform the file operations and is able to read and parse XML data directly to its internal representation. The data is stored in *PMXMLNode* objects that save the name of a node, its value and its attributes in the form of *PMXMLAttributeStruct*. This structure represents the attributes that a tag in XML can have by storing its name and value.

So, in order to load a scene graph from a XML file, it is merely necessary to call the *PMLoader* class and the respective *loadSceneGraphFromFile* function. The *PMLoader* class is responsible for aiding in the loading of certain components such as the scene graph, textures and meshes, thereby representing a unified entry point of the assets that support any application that is built on top of the engine. This class intends to provide a simple way of loading content, similar to the one that is presented in XNA, namely through the *ContentManager.Load<>()* methods, which greatly simplify the loading process of meshes, textures, audio and other asset types.

After the scene graph is loaded from the XML file it is automatically set to the scene and ready to be displayed. In a matter of fact, it can be loaded through the call of a *PMScene* function that automatically calls the *PMLoader* functions with the correct parameters. If the *PMScene* function is called, then no more work is required as the graph is automatically set to the current scene and can be immediately shown in the screen as a result.

7. Application Layer

The top modules layer consists in a set of features that enrich the engine as a global application, but are not necessary for it to work. These features can be easily detached from the engine, thereby acting as plug-ins that are used only when necessary. However, although they can be removed, some of the classes that are implemented in this layer are actually very important to the engine final results and capabilities, such as the classes responsible for navigation.

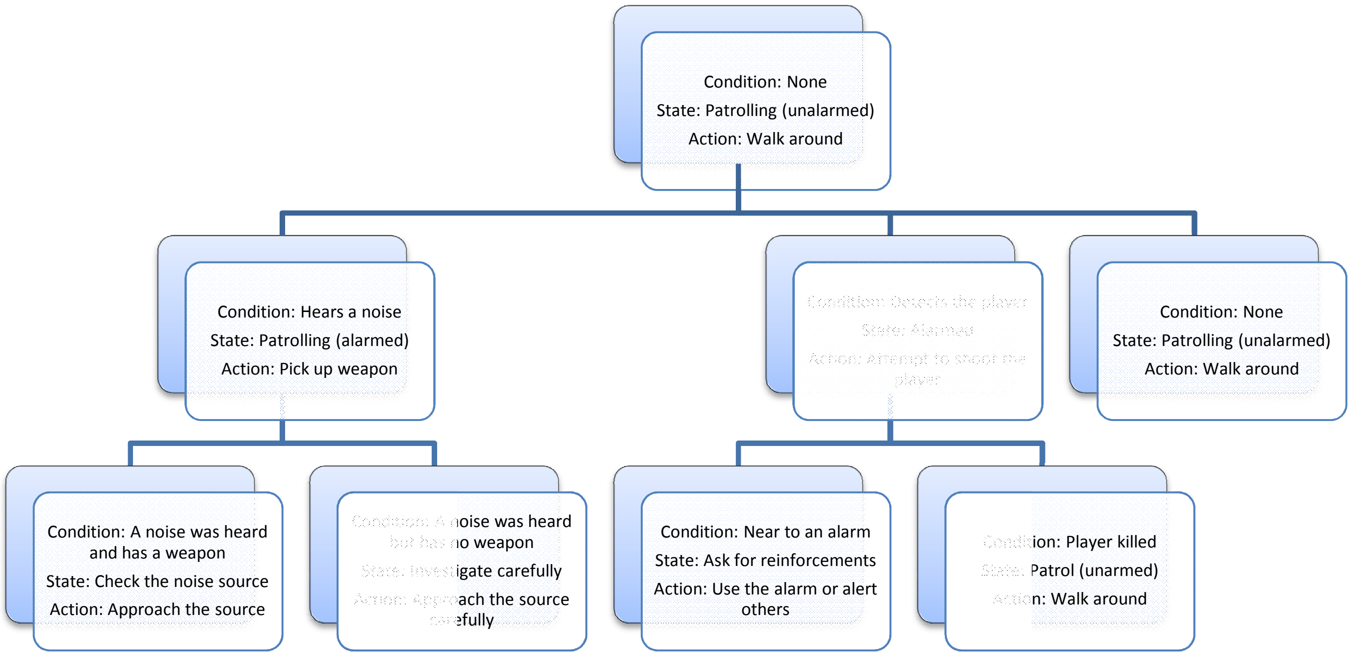
7.1. Artificial Intelligence

In what concerns a game, artificial intelligence deals not only with NPC behavior but it can also be used to control the whole game logic. So, it might be necessary to consider that the artificial intelligence system might transverse the whole object system in the engine as well as the object control system such as the transformations and so on. As it was mentioned above, every object that inherits from *PMDynamicObject* can implement a *PMExternalController* pointer that allows the object to be externally controlled by a remote script, artificial intelligence module or external component. As a consequence, almost all objects that are part of the engine can be controlled remotely by an external controller that dictates the actions that should be performed.

As it was mentioned above, a *PMExternalController* is an object that can be plugged onto a *PMDynamicObject* in order to provide the actions that should be performed by it at a certain moment. The actions and the conditions upon which these actions must be performed depend on the specific implementation of the *PMExternalController* that was set in the object itself. When an external controller is set inside an object, the later can choose to use it or not and when to use it, which may be useful for security purposes. If the external controller is used, it is able of partially control the object by specifying the actions that it should perform. An action is merely an object of type *PMExternalControllerAction* which relies upon the pseudo-reflection system to work. Basically, an action specifies the object, function and its parameters that are to be used when the action is to be executed. So, when an action is run, the object performs a lookup in itself to find the object and function that is to be executed and orders the pseudo-reflection system to run it, if it exists. Consequently, the object performs the required order based on the external controller request.

So, it should be clear that this system presents a lot of expansibility possibilities, as an object can be extensively controlled by using these facilities. Also, as most objects inherit directly or indirectly from the *PMDynamicObject*, almost everything may be remotely controlled by using this technique, thereby providing a coherent and safe way of controlling a scene and its contents through external scripts or with artificial intelligence modules.

To implement and test these capabilities, a top-level artificial intelligence layer was built, consisting of an algorithm that uses Behavior Trees to control the actions of an object based on certain conditions. A Behavior Tree is similar to a finite state machine algorithm but instead of resembling a graph it is fully contained inside the tree that maintains a state. That is, the behavior tree is at a node and will only change to some other node if the transition condition is verified. If no conditions are verified that allows the transition to another node, two things may happen. One approach is simply ignore that cycle and nothing will happen, thereby maintaining the same state and executing the action that is associated with that specific state. Another solution is to execute a default action that is specified by the programmer. Both of these cases are very similar, with the exception that in the former case the behavior tree can change its state if told to do so in the default behavior.
Considering a more specific example, this algorithm can be applied to a NPC enemy soldier in a FPS game. The behavior tree for the soldier could be similar to the next example:


Condition: None
State: Patrolling (unalarmed)
Action: Walk around
Condition: Hears a noise
State: Patrolling (alarmed)
Action: Pick up weapon
Condition: Detects the player
State: Alarmed
Action: attempt to shoot the player
Condition: None
State: Patrolling (unalarmed)
Action: Walk around
Condition: A noise was heard and has a weapon
State: Check the noise source
Action: Approach the source
Condition: A noise was heard but has no weapon
State: Investigate carefully
Action: Approach the source carefully
Condition: Near to an alarm
State: Ask for reinforcements
Action: Use the alarm or alert others
Condition: Player killed
State: Patrol (unarmed)
Action: Walk around

The example above considers that a soldier is initially patrolling a certain area, and as it is not alarmed, he is just walking around. If nothing happens, then he continues doing so, and this is his default behavior. If a noise is heard, then he becomes alarmed and picks up his weapon to become safe. Consequently, if he has a weapon he will investigate the noise source but safely armed with his gun. Otherwise, if he doesn't have a weapon, he will investigate the noise carefully and possibly call for reinforcements later on. If the player is detected, then the guard becomes alarmed and will attempt to shoot the player. Also, if he is near to an alarm, he may try to ask for reinforcements. Contrariwise, if he is able to kill the player, just return to its patrolling state. Obviously, a default state for this tree level could be keep opening fire while the player is alive and in sight. This resumes the behavior tree presented above and it is quite clear that is able of specifying the behavior of an object in a very complete and simple way. Also, it should be clear that this tree could be transformed into a graph to save some repeated nodes, but it was implemented as a tree to preserve the original algorithm used by many companies, such as Crytek.

A behavior tree is represented by the *PMBehaviorTree* class in the engine, which holds the entire behavior tree and the current state, represented by *PMBehaviorTreeNode* objects. The tree presents very simple functions that are mostly used for getting and setting the function values and to restart the tree, i.e. resetting the tree state to the root. Most of the work that is related to the behavior trees is performed by the *PMBehaviorTreeController* class that inherits from *PMExternalController*. This class receives a behavior tree as an argument in the constructor, or a filename from where a behavior tree can be loaded from an external file, and implements the virtual function *update* specified by the super class *PMExternalController*. When this function is called, the current tree state is checked and the child nodes are obtained. Then, for each child node the transition conditions are evaluated by using the pseudo-reflection system whenever necessary. If any condition is met, then the state is transferred to the respective node. Otherwise, the action for the current state is applied and the state is maintained.

The same technique can be used to implement virtually any type of artificial intelligence or remote control system, by using the pseudo-reflection system and the *PMExternalController* interface. Although some tweaking might be necessary to the implementation of the interface, the *update* method may take care of the calls to the pseudo-reflection system by itself. Therefore, creating control systems that are used to create a logic control for the scene or its enclosed objects is a very simple process by taking advantage of these architectural techniques that were implemented in the present engine.

7.2 Game creation

The *PMScene* class is responsible for centralizing the functionalities of the engine and exposing them in a carefully planned way, in order to simplify the interaction with the programming capabilities that were included in this project. However, although the class greatly simplifies the creation of graphical applications and games by putting everything together in a simple way, this does not necessarily signifies that the process of creating a game is easier. The *PMScene* deals mostly with the graphical aspects and, due to the *PMExternalControllers*, with some aspects related with the logic. Still, this is not enough to create a game, especially for people who are not very experienced in game programming. It is necessary to define a global game logic and architecture to create a game, deal with initializations, updates and the game drawing. Moreover, there are some game programming concepts which a less experienced programmer may not acknowledge, such as the main game loop and rendering.

To ease the process of creating a game, and based on what is used in the *XNA Game Studio* platform, developed by Microsoft™, some helper classes were defined in order to account for this problem. The architecture of a game involves some steps that are somewhat static across all games. These steps are usually similar to the ones presented below:

1. Initialization of the game variables
2. Loading the game assets and contents
3. Entering a main cycle
    a. Updating the scene
    b. Drawing the scene

Based on this, the Punkman engine provides a helper class that abstracts these steps, making the whole process more automatic. The *PMBasicGame* class defines a simple game class, which already provides the functions defined above, namely:

- initialize() – Initializes the game variables and required data structures
- loadContent() – Used to load the assets and contents that are associated with the game, such as models, textures, etc.
- update() – Called when the game is to be updated. Receives a *PMGameTime* object as an argument, that indicates how much time has passed since the last update and since the game was started
- draw() – Responsible for performing the drawing on screen. Such as update(), this function also receives a *PMGameTime* object.

All of these functions are virtual and can be overwritten by the programmer to fit its game needs. However, they specify a logical architecture that can be used and extended to easily produce a game. Besides this function, a *PMBasicGame* also specifies a game identifier and a name, which can be used to identify a game in the management system. It also contains two variables that indicate if a game is running and if it has terminated. Finally, the *PMBasicGame* class also contains a *PMScene* that can be used to place the assets and controlled according to the game logic. The scene pointer contained in this class can be automatically initialized with the default settings or created manually by the user.

As it was mentioned above, by using this class, when a game is created it automatically calls the initialization, loading, updating and drawing methods. This is done by using the *PMGameManager* class. It consists in a *Singleton* class that is capable of storing and managing the created games. When a game is created through the *PMBasicGame* class, it automatically registers itself in the *PMGameManager* and is added to an array of games. When desired, the user can change the currently active game to another one that is stored within the *PMGameManager*. This may be very useful to create a repository of games that can possibly be interrelated. A good example could be the case of a game in which the main character has to walk through a city performing some tasks and quests. The quests could be represented by mini-games such as a small driving game, in which the player could earn some money or points. Clearly, although the storyline relates the two games, the same doesn't happen with the in-game logic, that is drastically different between an open-world adventure and a racing game. Therefore, the main game could simply switch its context to a racing game, which would have a completely separate logic. The game would then be presented, and when necessary it would return to the main game, back to the original logic and storyline.

When a game is registered in the *PMGameManager*, this class automatically calls the *initialize*, *loadContent*, *update* and *draw* functions, by this order. It also starts the game loop inside itself, removing that responsibility from the programmer. Therefore, the user only needs to create the logic of the game and leave the rest to the engine. It will automatically call the base class methods or the ones that were overwritten by the programmer according to the game needs.

Finally, and given the current project goals, an additional helper class was created, named *PMBasicNavigationGame*. This class inherits directly from the *PMBasicGame* and adds the functionalities of the navigation module to the game. By doing so, the process of creating games that are able to use the navigation capabilities of the engine and the data provided by the location-based services is greatly reduced in terms of complexity and required time.

8. Navigation

In what concerns the present work, the navigation sub-layer was one of the fundamental problems to address. One of the major architectural decisions that appeared during the development of this engine was related to this sub-layer. This is not a common component of a game engine and is not directly related to the graphical part or even the logic layer, although it may be connected to both. Due to the fact that the engine intends to be a multiplatform software base for game development and since not all platforms are able of using location-based services, this sub-layer was not included in the core components. However, since it is a separate component that is connected to the graphical and logical engine, it is included as a separate top-level sub-layer. Consequently, it is able of using every necessary component of the underlying layers and still interoperates with the logical and graphical components whenever it is necessary.

Besides this, due to its nature, the navigation sub-layer follows a different logic than the rest of the engine components. The main component is a class named *PMNavigation* which specifies all of the functions that are relevant to the operation of this sub-layer. The purpose of this class is to act almost like the interface of a common GPS device, allowing the user to know what the current position is, and translating it to the respective street, address and country, getting the nearby Points of Interest and finding a route from point A to point B. However, this class also represents a concept that may clearly differ between platforms and consequently it merely works as an interface that must be implemented according to the underlying software and hardware constraints. Nevertheless, it requires some basic components that are not dependent of the platform, namely a coordinate system and POI support.

The coordinate system uses mainly the usual latitude and longitude values, specified in the degrees/minutes/seconds form and is represented by the *PMNavigationCoordinate* class. This class has methods that allow coordinates to be converted easily to a decimal degree format or to Universal Transverse Mercator coordinates. These additional formats can be useful in some cases to perform calculations upon the coordinates or even for some devices that require that the coordinates are specified in a different format.

The conversion for UTM coordinates requires the specification of the Earth's ellipsoid values to be calculated. Therefore, a table is provided with a representative set of values for a whole range of different ellipsoid formats that can be chosen to perform the calculation. The ellipsoids are represented by the *PMNavigationEllipsoid* class that requires an identifier, a name, a radius and an eccentricity value. These values are defined through the class constructor so that new definitions can be easily customized. Although more definitions can be easily added, the following ellipsoid definitions were considered and implemented in this engine:

Name	Radius (meters)	Eccentricity
Airy	6377563	0.00667
Australian National	6378160	0.00669
Bessel 1841	6377397	0.00667
Bessel 1841 (Nambia)	6377484	0.00667
Clarke 1866	6378206	0.00676
Clarke 1880	6378249	0.006803
Everest	6377276	0.00663
Fischer 1960 (Mercury)	6378166	0.00669
Fischer 1968	6378150	0.00669
GRS 1967	6378160	0.00669
GRS 1980	6378137	0.00669
Helmert 1960	6378200	0.00669
Hough	6378270	0.00672
International	6378388	0.00672
Krassovsky	6378245	0.00669
Modified Airy	6377340	0.00667
Modified Everest	6377304	0.00663
Modified Fischer 1960	6378155	0.00669
South American 1969	6378160	0.00669
WGS 60	6378165	0.00669
WGS 66	6378145	0.00669
WGS 72	6378135	0.00669
WGS 84	6378137	0.00669

Table 3 - Ellipsoid definitions

As it can be seen in the table above, the utilization of coordinates requires a wide range of decimal values. In systems that do not support floating point arithmetic this could become a very serious problem, due to the fact that the decimal numeric range is clipped to fit inside an integer value, when fixed point is used. However, coordinates are usually expressed in terms of degrees, minutes and seconds or in the decimal form. In the first case, there is almost no need of using decimal values, because the range of values that can be achieved by using this format is already extremely precise. So, in this case, the loss of precision that results from the utilization of fixed point arithmetic does not exist or can be safely ignored. In the other hand, the decimal system also does not require a big precision in terms of decimal values because it is able of expressing a position very accurately by using few decimal positions. The most common devices use four or five decimal positions to represent coordinates in the decimal format. Since the engine is using fifteen bits for the decimal part of fixed point values, it can reach a maximum precision of approximately 1.523×10^{-5}, which is clearly enough to hold the coordinate values. So, the biggest problem comes from converting coordinates to UTM format. The problem with this conversion is that the conversion requires values that can be very big, such as the earth radius or very small, such the eccentricity. Therefore, when the coordinates are converted, the intermediate values may become so small or so big that precision will be lost and the algorithm will return incorrect results. To minimize this problem, two methods were used. The first method is very easy to use and consists in the conversion of metric units to values that minimize the precision loss in the algorithm, yielding approximately accurate results, nevertheless. The second method is based on the analysis of the intermediate values, required to calculate the final UTM coordinates, which demonstrates that these values are either very big or very small. So, since the calculations are mostly based on multiplications, the small values can be multiplied by a constant large value to reduce precision loss when performing divisions, and divided by the same large constant after the operation is completed. By doing so, the intermediate calculations do not suffer so much from precision loss, resulting in most accurate results.

Also, as mentioned above in Chapter 5, the inclusion of a 2D system is very important in what concerns to a game engine. But clearly, it is also very important for the navigation layer, since most of the maps, POI and routes are usually represented in 2D. For example, POIs can be associated to a certain sprite, and consequently, POIs of a given type will always be represented by instances of the specified sprite. The 2D system can be easily linked to the navigation system, since both use a pair of coordinates to specify a position in the world. Therefore, the latitude and longitude are easily converted to X and Y positions and clipped to a drawing area on the screen of a device. Of course that the same effect can be achieved in 3D, but it would require an additional amount of work to convert the coordinate spaces from two dimensional geographical coordinates to three-dimensional world positions. Nevertheless, this transformation is quite linear if the height of the elements in the world is not considered. In this case, the problem would, once again, fall onto the 2D case and the solution is trivial. If map height is provided, then each element to draw must also provide a third coordinate, which would increase the data footprint required for the representation of the points.

Based on this, and in a game-oriented perspective, it seems clear that the navigation elements, such as POIs, and the engine objects, such as sprites and meshes, could be linked together to provide new ways of creating interactive content. For instance, if a certain POI type is represented by a sprite, the engine could benefit from providing a way of dynamically changing the underlying sprite in order to change the way that these POIs appear on the screen. For example, if the objective of a game is to find the nearest park, making the player win some points by doing so, the POIs representing parks could be replaced by money bags, for example. Also, this could lead to some interesting features, such as the implementation of entities that directly link the artificial intelligence to the navigation system. The result could be, for example, the creation of NPCs that would follow a given route, chase the player or behave in a specific way, depending on the location they are on.

In order to support the POI system the coordinate system had to include some form of finding what is the distance between two given points specified by its coordinates. This problem requires the utilization of complex algorithms to find the length of the arc over a sphere of radius R, which is the Earth itself in this case, corresponding to the distance between the two points. A relatively simple mathematical approach using spherical coordinates could have been used to compute this distance by using the angles specified by each pair of coordinates. However, as the engine is also targeted at devices that do not possess as much computational power as a normal computer, the calculations must be optimized. Therefore the Haversine algorithm [Ven10] was used in order to simplify the calculations and optimize the whole calculation process. Nevertheless, an architectural choice for this module was made so that the programmer can choose which function to use for calculating the distance between two points specified by its coordinates. Consequently, if desired, the programmer can simply swap the default function by some other method that he chooses to implement, by using the *setDistanceFunction* present in the *PMNavigationCoordinate* class. This method receives a function pointer as its argument, which in its turn receives two *PMNavigationCoordinate* points representing the two points.

A Point of Interest is a very simple object that is represented by the *PMNavigationPOI* class. This class is mainly used by the *PMNavigation* to represent the POIs that are specified in a map. One Point of Interest is therefore represented by its position, a name, a description and its type. POIs can be added to the *PMNavigation* class manually or by loading them from a database file. The *PMNavigation* class supports direct POI loading from a file, by specifying the database filename. The database file consists of a pipe ("|") separated file containing all of the relevant data that defines a POI and that was listed above. The type of each POI is represented by a PMUInt32 value which can be translated to its real type name. For example, the value 1 could represent "Restaurants" for example. It seems clear that these types of POIs cannot be hardcoded in the engine, due to its dynamic nature. For example, new types of POI can be added in a certain moment, others may have the need of being edited and some others may need to be removed. To solve this problem, there is also a database that holds the types of POI, namely the respective number and a descriptive name. This database can be loaded along with the POI database and translated accordingly by the *PMNavigation* class. As a result, the engine expansibility is once again ensured as the POI and its respective types can be dynamically changed either in memory or in the database.

After this, the navigation system can then be implemented according to the specific needs of the platform or the application to develop. For instance, it could be implemented in order to read location data from the GPS satellites, calculate the coordinates and place them upon a given map that is loaded from an external database. Otherwise, if a GPS enabled device is not available but an internet connection is, the engine can use the network to find out its approximate position and find out where it is located through a web-based location service such as Google Geocoding. Finally, if none of these options is available, the engine can simply enter a simulation mode, by using a log with coordinates that can be read in a certain interval to simulate data reading from the GPS satellites. For testing purposes, the Google Geocoding solution and the logging were implemented to support this interface. By using these classes, the engine could retrieve its location and map it to a specific address, city and country. This data could be used afterwards for any other game-related purpose that was necessary, such as drawing the current position over a map drawing.

9. Conclusions

The development of an engine such as *Punkman* brought a deeper insight over some of the issues that affect the developers of multiplatform applications, especially when mobile platforms are considered. The process of creating a suitable architecture for this engine was a hard task to accomplish due to many factors, mostly related to the specific characteristics of some devices that did not support some facilities that are available in most platforms. Also, some libraries that were used in the engine are not available or not supported by some devices that were targeted. So, there had to be a way of turning those engine capabilities on or off according to the specific device in which the application had to be deployed. Therefore, the creation of the architecture for this engine had to be carefully planned to contemplate that almost anything in the engine may be extended or overwritten in order to account for the characteristics of some devices. This was extensively tested during the development by putting it to practice. Every interface was extended at least once by some concrete implementation, although in most cases more than one implementation was provided. The engine design did not require any change when a new implementation was added, even when the new classes that were to extend existing functionalities were supposed to have a big impact on the whole engine. A good example of this was the inclusion of a DirectX Mobile renderer. Therefore, the architecture that was chosen for this engine seems to be sufficiently solid and expansible to suit the needs of the targeted platforms.

This architecture had to be based on some solid concepts that heavily rely on the abstractions level that compose the engine. To be able to cope with all the necessities that are required, it was ultimately necessary to provide a higher level abstraction over the existing types and data structures, such as numeric values and arrays. Only by doing this, it was possible to create an engine that could work on several platforms, which might specify their own implementation of the basic data types and structures. Consequently, if it is necessary, the basic definitions that are present in the engine can be changed to reflect the data types that are required by a given platform.

The *Punkman* engine has a primary focus the development of a game engine that could take advantage of the data provided by location based services. This module presented one of the biggest architectural challenges during the development, due to the fact that it is possibly a platform dependent concept, but it is not necessarily a part of the engine's core components. Also, some deployment platforms may not support the utilization of location based services and consequently it must be possible to switch the utilization of this feature. As a design choice, it was decided to place it as a part of the top modules. By doing so, the engine data and rendering cores are separated from the logical data modules, which accounts for a more logical organization. The location-aware capabilities of the engine were seamlessly integrated in the remaining features according to the engine philosophy described above. These capabilities also follow the expansible character of the *Punkman* engine, by allowing new extension points to be added very easily, which is the case of the *PMNavigationGoogle* that uses the Google Geocoding webservices to provide the necessary data. Nevertheless, this could be changed to use any kind of service, such as a physical GPS device to obtain the location data and provide it to the engine.

It was also necessary to take in consideration that some of the devices in which the engine could be deployed have very low capabilities and may not be able to support some heavier operations. So, the algorithms had to be carefully chosen in order to be as fast and lightweight as possible. In the cases where this was not sufficient to make the difference, some existing interfaces were recreated whenever necessary to account for the new platforms. A good example is the case of the window manager for both the common Windows environment and Windows Mobile. While in the first case, the windows can be created and registered in the operating system by using the *WNDCLASSEX* structure, the later does not support this extended version of the window classes and most of its definitions. So, a specific window manager had to be created in order to use the more limited *WNDCLASS* structure and some mobile-specific methods and constants. In the case of the algorithms that were used, it was necessary to take in consideration an additional detail. Some of the platforms do not support floating-point arithmetic operations. This problem is very serious and had to be accounted for when choosing and implementing the algorithms for the engine, as the numerical precision that can be achieved may be drastically affected. The solution was the creation of an intermediate layer that abstracts the floating-point values in order to make the algorithms work regardless of the current numerical implementation. However, the problem of the fixed point numerical precision could not be surpassed due to the mathematical nature of the problem.

So, the present project features a wide range of native possibilities that can be freely used to create games easily which may take advantage of the fact of being location-aware. The engine was tested on both normal computers and low-specification handheld devices. The registered performance was quite acceptable in all the tested cases, what can be seen on the benchmarks presented above. For this to be possible, the engine readapts according to the platform used whenever necessary. For example, when loading textures in DirectX Mobile, the system could either lock the entire texture surface or create a separate image surface and then simply copy the data to the texture. While the first alternative is more direct and simple to use, not all the devices are able of locking the texture to write the pixels directly. The engine checks the device capabilities to make sure it can lock the surface and write the pixels. If this is not possible, then it chooses to create an alternate image surface and copy the data to the texture afterwards to make sure that this will work in every platform.

It was also concluded that if the engine features were concentrated only in the level of the *PMScene* class, it would not be sufficient to help people to create games easily. Although the *PMScene* greatly simplifies the work of creating and deploying a game, it still requires some extra work in defining the game architecture and logic. So, as mentioned above, a game class was defined in order to ease this task. A new game can be created merely by extending the existing *PMBasicGame* or *PMBasicNavigationGame* classes, which already provide a suitable architecture and means for easily creating a game. By using this method, the main simply need to create the new game, which will be registered in the game manager, initialize, load content, update itself and run automatically. This technique makes the creation of games a very simple and straightforward process that can be complemented with extra tools and the engine features to make it an even easier task.

10. Future Work

Although the engine has reached a somewhat advanced state of development and maturity, some classes and methods still lack some features and error-resistance capabilities that are necessary to guarantee full engine stability. Some definitions might also still be incomplete at certain levels, although completely specified. Therefore, these are some issues that might require some further attention in terms of future work.

Besides this, some new features could also be implemented, such as a different renderer, for example, the non-mobile version of DirectX. The implementation of extra optimization methods was considered, as mentioned above, but only the octree approach could be used due to the time limitations. However, more were considered, such as BSP Trees and Portals, but either due to their complexity or specific usage, a more generic solution was used. Also, the *PMScene* class could be completed with further management options such as direct control of the scene graph, instead of simply obtaining its pointer from the manager and modify it directly.

Besides this, a new and innovative feature was considered, although it was neither implemented nor specified in the architectural description, related to a class that deals with the possibility of configuring the engine through the network. This could be very useful in the case of multiplayer games or graphical applications that must be processed in multiple processors. By using such a class, the configuration of a scene was not necessarily done through a XML file or another local scene descriptor, but rather through an interactive remote shell that is capable of automatically creating the definitions that are received. This however, raises security problems that could not be ignored, and as this was not a central functionality, it was not yet implemented.

Also, although the engine already supports the utilization of an external scripting language to control the internal entities through the *PMExternalController* class, the system still requires some extra features and further testing to ensure it performs as required. This layer was originally designed to support the introduction of an external scripting language that was created specifically for this engine, named *Conspiracy Scripting Language*. Although the language was relatively well specified, the implementation could not be completed inside the proposed deadlines, consequently remaining a feature to be included in a further version of the engine.

No visual tools for the engine could be developed up until this point. Although this is not necessarily part of the engine, it still is a good help to have a visual aid when developing graphical applications instead of having to specify everything in code. Nevertheless, as the engine already includes networking possibilities, the creation of a network configuration layer as mentioned above can easily be used to develop external applications that are used to modify the scene characteristics and settings, by using socket communication. Some external tools were thought to support the engine, such as a small visual 3D modeling studio, a scene previewer, mesh previewer, shader creator and debugger and finally, a map editor.

Finally, given the game creation helper classes that were implemented, the engine could benefit from having a way of playing games that are specified through an external file. This could consist for example in a website or tool in which the users could create their own set of rules and objects that are present on a scene. These rules and objects could be grouped into levels that were saved into a file. This file would be submitted to the engine, which would perform the required translation operations to transform the file into a playable game. Therefore, the engine could become a game factory and player that could use external files to create and present unique, user-created games.

References

[Way05] Wayne E. Carlson, 2005, "An historical timeline of Computer Graphics and Animation" http://sophia.javeriana.edu.co/~ochavarr/computer_graphics_history/historia/ (last access on: 21/01/2010)

[Kum02] A. Kumar, 2002, "Encyclopaedia of Management of Computer Hardware". Anmol Publications. p. 1050. ISBN 9788126110308.

[Gar08] Gary Donovan, 2008, "TDG - 3dfx - Voodoo Graphics" http://www.thedodgegarage.com/3dfx/v1.htm (last access on: 28/01/2010)

[Ric06] Richard Thomson, 2006, "Direct 3D Graphics Pipeline"

[Ben97] Benj Lipchak, 1997, "Overview of OpenGL" http://web.cs.wpi.edu/~matt/courses/cs563/talks/OpenGL_Presentation/OpenGL_Presentation.html (last access on: 25/01/2010)

[Env09] Envision Software, 2009, – "Maslow´s Theory of Motivation – Hierarchy of needs" http://www.envisionsoftware.com/articles/Maslows_Needs_Hierarchy.html (last access on: 13/12/2009)

[Fil07] David Filip, 2007, "Interview with Joost van Dongen: author of de Blob" - http://zip-zapgames.com/115-interview-with-joost-van-dongen-author-of-de-blob/ (last access on: 28/01/2010)

[Dav05] David H. Eberly, 2005, "3D game engine architecture: Engineering real-time applications with wild magic" – Elsevier Inc., Morgan Kaufmann Publishers

[Ped01] Pedro Pires, 2001, "Dynamic Algorithm Binding for Virtual Walkthroughs" – Instituto Superior Técnico

[Nik09] Nikolaus Gebhardt, 2009, "Irrlicht Engine – A free open-source 3D engine" - http://irrlicht.sourceforge.net/ (last access on: 14/01/2010)

[Dev10] DevMaster.net (2010) - "DevMaster.net – 3D Game and Graphics Engine Database" - http://www.devmaster.net/engines/ (last access on: 14/01/2010)

[Vir01] Virrantaus, K., Markkula, J., Garmash, A.,&Terziyan, Y. V., 2001, "Developing GIS-supported location-based services. In Proceedings of the first international workshop on web geographical information systems (pp. 423-432)" – Kyoto

[OGC05] Open Geospatial Consortium, 2005. "Open Location Services 1.1".

[Bri02] Li, Y. and Brimicombe, A.J., 2002, "Assessing the quality implications of accessing spatial data: the case for GIS and environmental modelling" Proceedings GISRUK 2002, Sheffield: 68-71

[Ste06] Steiniger, S., Neun, M., Edwardes, A., 2006, – "Foundations of Location Based Services: Lesson 01" – CartouCHe

[FID09] FIDIS, 2009, "D11.2: Mobility and LBS: Future of Identity in the Information Society" http://www.fidis.net/resources/deliverables/mobility-and-identity/int-d1110001/ (last access on: 21/01/2010)

[Mic03] Michael Juntao Yuan, 2003, "Develop state-of-the-art mobile games" http://www.javaworld.com/javaworld/jw-11-2003/jw-1107-wireless.html (last access on: 21/01/2010)

[TFL06] The Free Library, 2006, – "GPS-Enabled Location-Based Services (LBS) Subscribers Will Total 315 Million in Five Years, According to ABI Research."

[GSI09] Groundspeak, Inc., 2009, "Geocaching – The official GPS Cache Hunt Site" - http://www.geocaching.com/ (last access on: 05/01/2010)

[Amo09] Amos Bloomberg, 2009, "Pac Manhattan" - http://www.pacmanhattan.com/ (last access on: 05/01/2010)

[LML08] Locomatrix Ltd, 2008, "LocoMatrix: GPS gaming for everyone" - http://www.locomatrix.com/ (last access on: 05/01/2010)

[GG09] GPSGames.org, 2009, "GPSPoker by GPSGames.org" - http://poker.gpsgames.org/ (last access on: 05/01/2010)

[MFP09] MeanFreePath LLC, 2009, "TurfWars – a massively-multiplayer GPS iPhone Mafia Game" - http://turfwarsapp.com/ (last access on: 05/01/2010)

[GWI09] Gowalla Incorporated, 2009, "Go out. Go discover. Go share. GoWalla" - http://gowalla.com/ (last access on: 08/01/2010)

[BKI10] Brightkite, Inc, 2010, "brightkite.com" - http://brightkite.com/ (last access on: 08/01/2010)

[Sha08] Sharp, A. and Vino, N., 2009, "GeoDrawing :: Create. Spread. Paint. :: Color the awareness ribbon" - http://www.geodrawing.com/ (last access on: 27/01/2010)

[Ebe05] Eberly, David H., 2005, "3D Game Engine Architecture: Engineering Real-Time Applications with Wild Magic" – Morgan Kauffman

[Lev02] Lever, N., 2002, "Real-Time 3D Character Animation with Visual C++" – Focal Press

[Len04] Lengyel, E., 2004, “Mathematics for 3D Game Programming and Computer Graphics – 2nd Edition” – Charles River Media, Inc.

[Ebe01] Eberly, David H., 2001, “3D Game Engine Design – A practical approach to real-time computer graphics” – Morgan Kauffman

[Pen03] Penton, Ron., 2003, “Data Structures for Game Programmers” – Premier Press

[Sou06] Sousa, Bruno, 2006, “Game Programming All in One” – Premier Press

[Gla90] Glassner, Andrew S., 1990, “Graphics Gems” – AP Professional

[Arv91] Arvo, James - “Graphics Gems II”, 1991, AP Professional

[Kir94] Kirk, David - “Graphics Gems III”, 1994, AP Professional

[Mul04] Mulholland, A., Hakala, T., 2004, “Programming Multiplayer Games” – Wordware Publishing, Inc.

[Wei99] Weiss, Mark A., 1999, “Data Structures and Algorithms Analysis in C++ - 2nd edition” – Addison-Wesley

[Fol94] Foley, James D., et. al, 1994, “Introduction to Computer Graphics” – Addison-Wesley

[Wil79] William M. Newman and Robert F. Sproull, 1979, "Principles of Interactive Computer Graphics", McGraw-Hill

[Fol92] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes, 1992, "Computer Graphics, Principles and Practice", second edition, Addison-Wesley Publishing Company

[Pen97] Pendleton, Bob, 1997, “Doing it Fast – Fixed Point Arithmetic” - http://www.gameprogrammer.com/4-fixed.html (last access on: 14/04/2010)

[Lau06] Lauha, J., 2006, “The neglected art of fixed point arithmetic” - http://jet.ro/files/The_neglected_art_of_Fixed_Point_arithmetic_20060913.pdf (last access on: 16/04/2010)

[Ven10] Veness, Chris, 2010, “Calculate distance, bearing and more between Latitude/Longitude points” - http://www.movable-type.co.uk/scripts/latlong.html (last access on: 05/05/2010)

[Mal89] Maling, D.H., 1989, “Measurements from Maps: Principles and methods of cartometry”, Pergamon Press, Oxford, England.

[Mal93] Maling, D.H., 1993, “Coordinate Systems and Map Projections”, Pergamon Press, Oxford, England.

[Goo08] Google, Inc., 2008, “The Google Geocoding API” – http://code.google.com/apis/maps/documentation/geocoding

[Tay98]	Taylor, Chuck, 1998, "Geographic/UTM Coordinate Converter" http://home.hiwaay.net/~taylorc/toolbox/geography/geoutm.html
[Ebe02]	Eberly, David, 2002, "Fast Inverse Square Root. Geometric Tools", http://www.geometrictools.com/Documentation/FastInverseSqrt.pdf (last access on: 14/04/2010)
[Cha07]	Champandard, Alex J., 2007, "Understanding behavior trees", http://aigamedev.com/open/articles/bt-overview/ (last access on: 11/03/2010)
[Gam95]	Gamma, E., Helm, R., Johnson, R., Vlissides, John, 1995, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley

Annex A: Results

The architectural model presented in this document was fully implemented, and almost every feature was created or inserted into the resulting final product. Due to the scheduling of the project, some features could not be implemented, such as the routing. However, the inclusion of such capabilities is very easy due to the engine expansibility. The architecture revealed to be adequate to solve the problem that the present work attempts to solve, either in flexibility, efficiency and platform-independency. This can be seen clearly in the case study presented below.

A.1. A case study

To test the final results of the present project, a case study was developed. The whole architecture was implemented in C++, with the objective of testing its expansibility and the possibility of working across platforms. The project was mainly developed under a Windows Vista environment, using Microsoft Visual Studio .NET 2008. The chosen renderer was OpenGL ES, due to its portability and to the fact that it is mostly dedicated for being used in mobile platforms.

The work started by creating the base framework composed of the basic components and interfaces. After these interfaces were specified and transformed into code, some implementations were created. In this case, the engine was implemented in the same environment where it was developed, creating a W32 window and an OpenGL context for this platform. All of the interfaces were implemented at least once to test its functionalities and expansibility. However, most of them were implemented more than once, to test this concept even further, or just to add new capabilities to the engine. An example of such a case is the texture loading, which is now available for BMP, JPG and PNG files. The hardest case to implement, as stated above, was the renderer interface. Since the rendering API's are extremely different both in the available functionalities and in the paradigms used, the interface that was created only specified the most general and common features that these frameworks must share. The remaining features were enclosed inside each implementation and used only whenever necessary.

Also, since there was no GPS device available, and since the working environment did not possess any kind of location-aware device, the navigation data was obtained from a web service. However, as it used the common *PMNavigation* interface, it was simply a matter of adapting the data to this interface through the Façade Pattern, to match the data format that the client classes expect from it. This proved that the interface design and the exposed features give the engine the possibility to adapt to new circumstances and obstacles that may have not been predicted in beforehand.

Nevertheless, testing the implementation of the architecture by itself is not enough. And more than that, it was not enough to test the engine in only a desktop computer running a full-featured version of Windows. So, a simple prototype was developed as a proof of concept. This prototype consists in a simple game that joins 2D and 3D elements and runs on both a desktop or laptop computer and a low-specification PDA. The game is basically a quiz that uses location-based services for obtaining the user's current position and presenting questions that are related to the address that corresponds to that position, or to any nearby POI. The engine was deployed without any problem on the mobile platform, using the same codebase as the version that was deployed on the desktop computer. The only differences were at the *PMDefinitions* level, due to the fact that the JPG and PNG libraries were not available for the device that was being used, and therefore, these capabilities were not turned on. Also, due to the memory limitations in the PDA, the textures used had to be reduced or they wouldn't be completely loaded otherwise. For some reason, when this happened, the file managers would merely report an end-of-file, even when the file wasn't completely loaded. The figures below presents the final result obtained in the prototype:

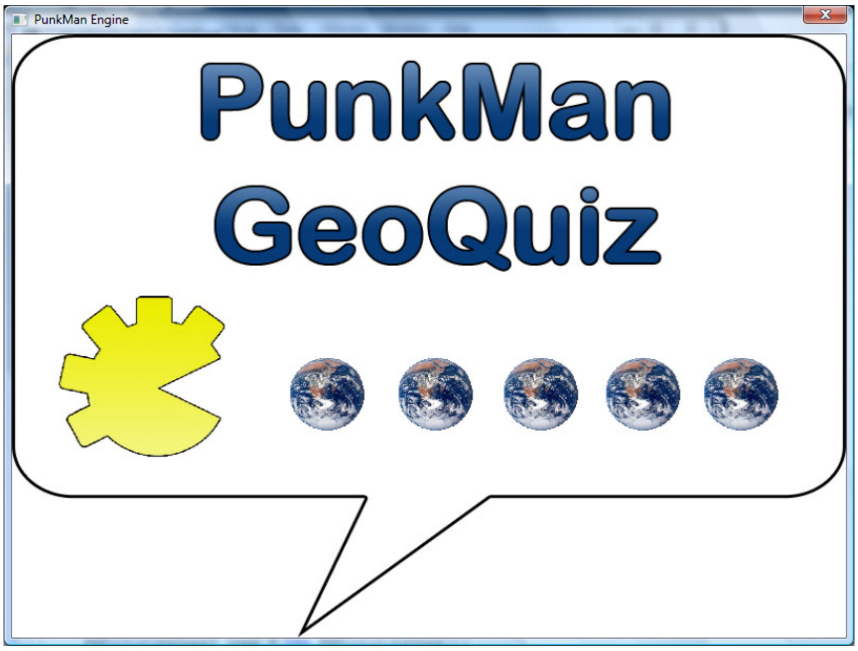


Figure 21 - Punkman Prototype

Figure 22 - Punkman PDA Prototype