Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Design and Implementation of the Plug-in Framework for an Advanced Game Engine Architecture

Degree Final Project

Degree in Informatics Engineering

Author: Vicente Eduardo Ferrer García

Tutor: Jose Ismael Ripoll Ripoll
Experimental Director: Hector Marco-Gisbert

Academic course 2014 / 2015

Resumen

En este TFG se ha abordado el diseño y la implementación de un *Plug-in Framework* en C orientado a objetos que forma parte de un motor de videojuegos. El motor, del cual soy el director y principal desarrollador, es un proyecto colaborativo que lleva 5 años en desarrollo y participan 30 personas. Este TFG es una pequeña parte de un proyecto más general el cual tiene como objetivo crear un motor avanzado de videojuegos para la industria de videojuegos orientado a juegos 2D/3D multi-jugador masivos en línea, y desarrollar juegos AAA, los mejores de la industria de videojuegos, para múltiples plataformas y/o consolas.

A medida que el código del motor empezó a crecer, apareció la necesidad de desacoplar la arquitectura del motor de videojuegos. De este modo, los programadores pueden escribir código independientemente de otros, dando como resultado menos conflictos y a su vez, siendo más efectivos en términos de coste temporal y económico. No adoptar esta filosofía o alguna similar puede traducirse en costes adicionales a la larga en el área de desarrollo.

Esta técnica o filosofía fuerza a los desarrolladores a escribir código genérico e independiente de otros módulos. La cual es especialmente apropiada cuando:

1) La aplicación está compuesta por una gran cantidad de módulos separados por capas.
2) Cada componente es implementado con una tecnología diferente.
3) La aplicación constante evolución de cada parte.
4) Está sujeto a cambios drásticos.

La mayoría de motores de videojuegos modernos están diseñados alrededor de esta arquitectura de *plug-ins*.

Palabras clave: *Plug-in framework*, motor de videojuegos, *metadata*, *memory pool*, *FFI*, *RPC*, *meta-programming*, *reflection pattern*.

Abstract

In this DFP has been addressed the design and implementation of a Plug-in Framework in object oriented C that is part of a game engine. The game engine, which I am the director and the main developer, is a collaborative project that has been developed during 5 years and in which participate 30 people. This DFP is a small part of the more general project whose aim is to create an advanced game engine for game industry, oriented to massively multi-player online 2D/3D games, and to develop AAA games, the top of the game industry, among multiple platforms and/or consoles.

As the code of the engine started to grow, there was the need to decouple the architecture of game engine. This way, programmers can write code independently from each other, which results in less conflicts, and so, more effective in terms of cost and cheaper in terms of monetization. Not adopting this philosophy or similar ones, can be translated into additional costs in the long run lost in development area.

This technique or philosophy forces developers to write generic code, independent from other modules, when we have an architecture full of modules separated by layers where each one implements a different technology targeting some functionality, growing constantly and subject to drastic changes.

Most of modern game engines use this architecture.

Keywords: Plug-in framework, game engine, metadata, memory pool, FFI, RPC, meta-programming, reflection pattern.

Table of contents

Figures index

Tables index

1. Introduction

In this project has been addressed the design and implementation of a Plug-in Framework in object oriented C that is part of a game engine. The game engine, which I am the director and the main developer, is a collaborative project that has been developed during 5 years and in which participate 30 people. This DFP is a small part of the more general project whose aim is to create an advanced game engine for game industry, oriented to massively multi-player online 2D/3D games, and to develop AAA games, the top of the game industry, among multiple platforms and/or consoles.

Because of the magnitude and complexity code has acquired during these years of development, there is the need to strongly decouple the architecture of the game engine, this way, it is possible to make independent code, to be more productive without inferring between programmers, being more effective in terms of cost and cheaper in terms of monetization. Not adopting this philosophy or similar ones, can be translated into additional costs in the long run lost in development area.

This technique or philosophy will force developers to write generic code, independent from other modules, when we have an architecture full of modules separated by layers where each one implements a different technology targeting some functionality, growing constantly and subject to drastic changes. Most modern game engines enforce to develop its architecture in this way.

A possibility to automate creation of code generic and decoupled from the engine architecture is a Plug-in Framework. With this tool is possible to develop generic and decoupled functionalities, to define a small set of rules from the point of view of the architecture, and then attach external modules to it that fulfill at least that set of rules, adding a functionality without need to know underlying implementation or technology, similar to a service based architecture. To allow that features sometimes is need to provide other functionalities where system relies on, this can be given by the game engine, but also can be provided by other third-party ones.

There is a well known projects that have been using this philosophy successfully during years, like OGRE[1] that implements a Plug-in model based on DLLs[2] . In that example, the system has been thought to extend the engine functionalities, as described before, but the system is being described also provides the extension and implementation of game-logic via plug-ins, allowing to easily develop behavior of the game without impact the engine and game architecture, providing a powerful tool for modding, create a game and customizing it without editing directly the game architecture. An example of this other kind is Unreal Engine[3] which has a more powerful system allowing to integrate plug-ins on game or engine core[4] .

The main motivation of this project is to allow faster, better and cheaper development of games, reusing as much as possible and automating the process, avoiding the time spent to teach new programmers about the whole game engine architecture. In this document is going to be described in depth all problems and possible solutions for adopting this philosophy in a game engine architecture and how to approach it.

1. 1. Discussion

Even game and/or game engine is distributed under open source or proprietary licenses, these philosophy has been successfully implemented and tested in international games, development of plug-ins can be monetized or not, and it can be distributed with or without source, or can be completely closed, in any way will generate benefits for the company, it will simplify development, and will contribute directly with money or workforce and indirectly popularizing the game engine or game that is being modded.

An example of this is Minecraft[5] , that has generated millions in a few years, and now it is played around the world. Nowadays being a freelance plug-in developer for Minecraft is a job that does not need too much knowledge about programming and it is a profitable business, which can be learned in a small period of time.

Security is also improved, from the business point of view, because any programmer has access to full source code, and from technological point of view, because plug-ins can be provided from a trusted database and contiguously deployed and updated in automated way to improve errors and back-doors, and only allow certificated plug-ins to run in the back-end of a game, controlling by this way the development and use of plug-ins between game servers.

The system by itself, it is based on the principle of code injection. This means that the system is designed to allow injection of code without knowing the source from it belongs, and how it is executed. Strong decoupling of components has the drawback that it is not possible to know most of significant information of itself.

If the source which some plug-in is provided is not trusted, that is translated into a big security problem. Anybody has the possibility to inject code, whether malicious or not, and game engine or the game itself will never release of that fact.

At the end, this system is similar to the way experienced hackers develop cheats for existing games, but instead of spending time making the reverse engineering to understand how the game and the engine is done, this system provides by means of Metadata system a way to discover the foreign interface at run-time.

If this fact is not taken into account, the system provides all the same facilities as developers will have but instead of giving it to produce valid code, it is used to produce malware. So any hacker will be able to develop cheats and in addition, the system abstracts it from difficulties as memory edition and low level difficulties.

Another problem related to the plug-ins is the overhead which adds to the whole game and engine. The system acts as a generic interface between all components, so any use of a plug-in is always encapsulated in a component which decouples it from the other tiers. Depending from the technology there will be other related problems too, because the execution of JavaScript plug-in is quite different to the efficiency of a compiled one. This fact must be taken into account when developing them.

1. 2. Objectives

The objectives of this project are derived from the actual needs of the development of a game engine, and so, the results will be directly used in production. The objectives of the following project are structured in the following list:

- To compare different strategies of solving the problem of run-time plug-in integration.
- To study and to compare the existing technologies about plug-in management.
- To design a new framework, addressing all the specific requirements of the game engine.
- To validate the design implementing a proof of concept to demonstrate the capabilities of the framework.
- To describe the system with a formal documentation in order to serve for a development group by using DocBook[15] .

Finally, in parallel, a complete documentation is being written about the system, in order to give an easy way to learn how the plug-in framework works, and allowing other programmers in the future to plug functionalities more easy, and develop faster the game engine or the logic of a game, developers will learn about the system in order to understand and use it. In the future, a big part of the architecture will be warped by the system.

2. Design

This chapter is about how a framework with specified features can be designed, what should be take into account when design it and how to fit the specs enumerated previously.

In order to define the design of the framework, it is going to be distinguished between the different modules it is formed. First of all, there is the Plug-in module, the main one. This module allows loading during run-time different kinds of code, no matter native, interpreted or compiled to an intermediate language.

It also has versioning control, and infrastructure offering, which means that it can register and offer functionalities to the plug-ins, and vice versa. By this way it is possible to provide features to plug-ins, for example, using the abstraction modules from the Operative System in order to make them platform independent.

The Plug-in module must be platform independent so, it shall rely on a System Module, which gives typical functionalities of the Operative System for loading native code at run-time, in this case, by means of dynamic library loading. For interpreter implementations or virtual machines, it can be implemented inside the plug-in loader, by means of extending the loader interface, and that can be encapsulated inside another plug-in too, making a plug-in that loads other plug-ins in a hierarchical fashion, recursively.

By other hand, there is the need to have a unique interface, to give support to the plug-ins to easily register and make communications between them in a generic way, as explained in the introduction. To allow this there is another module called Metadata. This module is able to reflect code and do introspection in other to populate functionalities; in addition it can make dynamic bindings and allow new features to the language as foreign function interfaces that it will be explained later.

Finally there is the Preprocessor module, which allows preprocessor meta-programming techniques, giving support for Metadata and Plug-in modules, in order to make automatic stub generation, and make transparent the trickery is done inside the system, and simplifying the front-end for the final programmer.

2. 1. Related work

Nowadays there is a variety of open source alternatives that can be used to implement a plug-in framework, or plug-in frameworks by themselves, and also there are other kind of libraries that can also fit the problem, but they use a different philosophy to approach the problem. The last type of libraries act as a wrapper to a desired library, framework or application, and them allow to externally extend the existing functionalities.

In the first case, there are general purpose approaches implemented in multiple types of applications. An example of an application can be Mozilla Firefox[6] , which has a system of custom add-ons for extending functionalities. When looking into the area of game development, the efficiency becomes a priority, so the system must be designed with other requirements, and a general purpose is not enough.

For this second kind of frameworks, there is a existing open source alternatives. Most of them implemented in C / C++ as a base, and plug-ins usually are developed in that language too for reasons of efficiency but there are also others that are designed to load and interpret scripts as a plug-ins, in this second case it is also used the same language to implement the interpreter but there is an overhead due to the new level of abstraction introduced, although it gives more flexibility and simplicity.

In the case of native and compiled plug-ins, the techniques used to load it and inject them are provided by the operative system. In the case of plug-ins loaded by script, there are multiple options available, Python C API[7] , Mozilla Firefox's Spider Monkey Engine[8] or Google V8 Engine[9] for JavaScript, Lua C API[10] . All of them are powerful tools that can solve the problem of loading scripts and they are also well known and easy to learn.

The third case is the most unknown because of its nature. In this case the library is designed to act as an upper layer of the system which is going to be extended, encapsulating with a technique which is capable of describe the system in a higher level of abstraction, and it gives the possibility to add new functionalities, to extend the existing ones, to create mixings. Usually these libraries are designed using meta-programming, and taking the advantage of compile-time features, making the system robust and efficient. A drawback of this feature is the higher level of abstraction. Most of times becomes difficult to understand and use it, and provides more complexity for the system instead of simplifying it. But in concrete cases, it has advantages over normal technologies, because it can provide run-time features that are not achievable with other techniques.

An example of this kind of library is Boost.Extension[11] , which uses template meta-programming to generate a new level of abstraction where functionalities are registered and then it provides a way to easily discover, use and extend them. Template meta-programming is a technique that implements a Turing Complete language for generating code at compile-time, in this case, it has been used to implement a pre-build step that can generate C++ code, implement logic statements with it in an upper level, and introduce techniques as code introspection and reflection or automated code generation.

Ultimately there is another approach which has not considered because of its nature. This case is commonly used by hackers to allow run-time code injection to a desired process or binary edition of a desired executable; although in this last technique the demand of code loading on the fly is not satisfied because it must be done before the execution. Both techniques need to do reverse engineering, a hard task, and also need knowledge about the low-level.

An example of this approach is the AMX Mod X[12] for the Counter Strike[13] videogame, a project which provides by reverse engineering a plug-in system for CS that can extend, modify and add new capabilities without having the source code of the application. Most of times these techniques are critically unsafe and illicit, so although it is possible to solve the problem with them, they are automatically discarded because of they are out of this context.

2. 2. Limitations in the current technology

The previous approaches use different philosophies, inclusive there are differences in each implementation of the solution. Except from Boost.Extension, all have in common the same problem, all plug-ins usually are coupled to the system, and the system by itself is designed to a concrete technology.

A clear example are game engines like Unity[14] or Unreal, which force you to learn them script language or use the technologies that they want, C#, Boo, UnityScript, an extension of JavaScript, or UnrealScript respectively. In addition these architectures are not designed thinking in generic code injection, or game engine extension. They only offer a script language with the API of the game engine ported to that language, so instead of coding in C / C++ they give the possibility to code in another language.

In this case this is a great approach, but it is not enough. The main problem of these solutions is that it is only a port of the game engine API, which it is defined over the versions by the development team. That is a strong dependency, which normally forces new developers to learn a concrete language, in some cases proprietary, then learn the API that the engine offers, which most of times is not backward compatible, and in last instance, that interface cannot be extended or manipulated, and the game engine cannot rely on external implementations.

The technique used in Boost.Extension is a better approach, which allows making a weak API, strongly decoupled from the game engine and plug-ins; it warps the programs and makes generic the use between them. In addition, this layer gives flexibility without scarifying the performance, because of the abuse of compile-time features that exists in C++.

An initial decision was to develop the system using this library, and create a framework more flexible than the existing ones using the features it provides. After thinking about this solution, some problems appeared in relation to the library. As it has been written in C++, a language which not defines a standard for their ABI, there is no binary compatibility between different compilers, each one implements C++ features in a different way, and the same happens with name mangling.

In order to avoid these problems it was mandatory to move to the C language. Although system still has been designed with object oriented features, there are multiple limitations because of the structured nature. The main problem is that template meta-programming does not exists, and it has to be done by another approach called preprocessor meta-programming, which is a technique that abuses of macro expansion in order to generate C code, but in a more limited way than template meta-programming does.

In addition, function binding and variable template arguments also become more difficult, reflection mechanism cannot be done at compile-time as the same way as with templates. There are multiple problems that arise when moving to C but still it is a good option, better than implementing a port of the software for each compiler to make it cross-platform and cross-compiler.

2. 3. Features and specs

In this system, requirements are focused on developer end point, and in order to improve the architecture, that also it is better for developers. Starting at this point there, features are related to needs of developers and the architecture, detailed in the following list:

- Decouple particular functionalities from the game engine architecture; generate a low dependency between the game engine core and plug-in functionalities.
- Outsource development of game engine or the game itself, allowing developers to write code faster without inferring between them. Each plug-in can be seen as a separated program by itself attachable to the plug-in system by small set of rules that must fulfill.
- Automate creation, implementation, registration and communication between plug-ins and the architecture.
- Allow to load and unload on the fly desired functionalities independent from underlying technology.
- Improve the process to debug and encapsulate errors in reduced packs of code avoiding contaminating the execution flow of the architecture.
- To facilitate development of game engine or game itself, make easy, fast and cheap to develop it.
- Enable full modding of the system with automated and secure deployment.
- Flexible and independent from architecture, platform and compiler or environment execution.

2. 4. Requirements

Based on previously defined features, the following requirements for the system are defined:

- It must implement a generic and extendable plug-in loader independent from operative system, static library linking or dynamic library loading mechanism and run-time environment.
- It has to provide a mechanism to easily register and unsubscribe plug-ins into a manager at run-time, and handle it efficiently.
- It should have a run-time self-description framework to give possibility of define interface rules during the execution and to provide metadata information to allow discovering and communication between core and plug-in implementation components.
- It needs a storage manager which implements an efficient memory pool manager to provide memory resources to the self-description framework, and provide a valid inheritable scope where plug-in metadata and interfaces live and can be queried or modified and executed.
- Provide serializable channels to marshall metadata or the full scope, to provide a transparent way to communicate between plug-ins or core efficiently and in a secure manner.
- It must be context safe, plug-ins must be run in a separated execution flow and they cannot infer in core execution, and this must be transparent and safe for communications, allowing to run asynchronously.
- It must have a dependency system and versioning to allow an easy methodology to maintain properly the dependencies between plug-in versions and core interface.

2. 5. Decoupling and encapsulation

Whole system must provide a generic way to strongly decouple functionalities over the game engine, including the system itself. What this means is that plug-ins will be loaded at run-time, the system does not know about its existence meanwhile it is not inserted in it. When system is launched it can ask for a list of functionalities that it needs, without knowing at compile-time what is going to be executed or even if that functionalities are available.

To provide that strong decoupling sometimes is need to do some trickery or introduce complex systems to allow run-time metadata definition even if language where is implemented does not support this kind of feature, but it worth; by this way it is possible to make communications between plug-in and existing engine architecture trivial allowing to decouple from architecture, platform, language and location or technology.

The context where plug-ins are implemented forces them to be designed in with the same principle. The plug-in receives a instance of the plug-in framework that is warping the whole Metadata system, holding the scope and the memory manager where the metadata lives, and gives it to allow run-time registration, so the dependencies between each plug-in and the game engine does not exists, they are constructed on the fly.

But apart from that, the system itself must be decoupled from its implementation, when looking from the game engine point of view, and the integration of the plug-in framework. This means that system must be easy to use by a programmer hiding the implementation details, allowing also to easily extend system just by deriving abstract interface of some functionality, and for example, give multiple loaders and run-time environments for plug-ins for different technologies, for example, implementing different virtual machines over the same language in order to make performance tests.

A possible pattern which provides that encapsulation is the Opaque Pointer Pattern[16] also known as PImpl idiom, which can be used no matter structured or object-oriented language. By this way plug-in framework can be decoupled from game engine architecture, and should be implemented without knowing tools it is using, so plug-in it can be seen as external library easily implementable by wrapping game engine architecture modules, but making the use of them opaque from the external point of view. This is the example of the PluginLoaderLibrary which uses the module System in order to implement dynamic library loading, but this is not visible in the front-end of the interface.

The implementation of plug-in framework is not going to impact into game engine architecture, it can be seen as upper layer that wraps functionalities of engine providing a mechanism to register and implement other new functionalities.

The way this pattern is implemented will be reviewed after, but understanding that pattern provides a better idea about how the architecture is designed, always enforcing genericity, encapsulation and designed in a way that it can be extended in the future easily.

2. 6. Component based architecture

In this framework, plug-ins can be seen as components with defined properties, with a dependency graph between them, which does not exist meanwhile system is not loaded and executed. This means that each plug-in can be compiled with no dependencies and the same happens with the complete game engine.

Plug-ins only depend from others, or the game engine infrastructure, in the moment they are going to be used, all of them can be developed as standalone packages, in addition, it can be run without having dependencies solved but result will not be the expected.

This architecture gives multiples benefits, by this way it is possible to make a robust and fault tolerant system, with error encapsulation that avoids propagation to other components o the game engine infrastructure itself, and the engine still can run in case of error, invalid loading or if dependencies are not solved. In addition it is easy to develop and extend by other programmers, because they do not need to learn full architecture, only this framework.

With components it is also possible to avoid problems of hierarchical models, in addition, the dependency graph is allowed to be more complex, without any restriction, this fits perfectly with plug-ins because it can be treated as independent packages of code, so choosing a component based architecture it is a good design choice for this, furthermore its dependencies are strongly decoupled and they are weak and dynamically loadable.

So from now each plug-in is going to be designed as a component, and in the plug-in framework there will be a Factory Pattern[17] which is the entity that generates the instances of each plug-in, as a component. Each component encapsulates a generic handle related to the real plug-in implementation loaded at run-time by the plug-in loader, which depends on platform, language or environment where plug-ins are being loaded and executed.

The factory is related to the top module, which has a higher level of abstraction, but internally the same design is used for threat the metadata, the code and interface of the plug-ins, it discovers the package of code which contains each component. So, Metadata system has internally implemented another factory which generates the entities that represent literally the code, in fact it has an array of factories, because by this way it is more efficient, each component is managed individually in an array of instances of the same type. By this way accessing to that memory begins more effective with less cache fails, and it can be optimized to avoid memory fragmentation, with a constant cost most of times over the access and insertion.

The whole system can be seen like a table of components in the upper row, that intersect between multiple functionalities creating a concrete plug-in instances at the left column, that is, make the dependencies between components, so the union between components represent a plug-in. By other hand, each component(s) that represent a plug-in, it is a container of multiple components that represent the metadata representation of the data, functions, classes and methods, definitions or enumerations and constants.

All components can be joined during run-time dynamically, so it is possible to add and remove full packages of code from a plug-in, or inside of a plug-in is possible to add, in this case, extend the current application programming interface on the fly, or modify the existent ones.

2. 7. Meta-programming

Meta-programming is a difficult concept to define. To understand it we need to know what programmer does. What we usually do is to develop some kind of tools to automate some specified process, sometimes this process is really complex so we must develop the tool by parts, beginning form small pieces that they will help to make the final tool, like a factory will do when building a drill, which automate the work of making holes.

Eventually programmers not just develop tools for doing some specific task; they also make tools for designing code faster and better with less cost. There are different kinds of tools, from text editor to debuggers, but there is a concrete type that is strongly related to meta-programming. In multiple programming languages, always there is a some specific sugar, framework or design patterns, that it is widely used and considered good practice. For example, in C++ has STL[18] , a powerful standard library that helps to make code, giving basic algorithms, that simplify tasks and abstract from common problems.

The role of meta-programming in those friendly tools is to give an upper level of abstraction to automate code generation. It can be embedded inside a language, in this case the language is able to auto-generate itself, it can sound very strange, but it is possible to do. By this way is possible to develop code that generates other code, allowing programmers to increase productivity writing less lines of code.

Python[19] implements features of meta-programming embedded in the language, Reflection[20] for example, it is a mechanism that allows self-discovering of code structure of an application. This technique is useful when generating code, because the generator script can know the structure of some code, and with that information it can generate code in a more intelligent way, not just using text expansion, like the macro expansion mechanism of the C preprocessor.

Unix Bash[21] is a good script language to illustrate what meta-programming is in an example.

```bash
#!/bin/bash

# make an echo of the header of the script, but redirect it to
# a file we will use to execute (the autogenerated script)
echo "#!/bin/bash" >> metaprogrammed_script.sh

# insert a blank line
echo >> metaprogrammed_script.sh

# make an echo of a Hello World and and redirect it to our script
echo "echo 'Hello World'" >> metaprogrammed_script.sh

# give full permissions for the script
chmod 777 metaprogrammed_script.sh

# run the script that will show "Hello World"
./metaprogrammed_script.sh
```

Figure 1: Example of bash script that generates a hello world script.

The result auto-generated script called will look like:

```bash
#!/bin/bash

echo 'Hello World'
```

Figure 2: Generated hello world bash script.

After script execution it will generate a new script called metaprogrammed_script.sh with an echo of 'Hello World' inside, then will give permissions and then execute it, showing the message in the terminal. The script first generates at run-time the second. This is the key, run-time code generation.

In this example it is difficult to see the potential of meta-programming, it is just a practical example to understand how it is possible to do it. For understand a practical example, imagine that we are interested on run multiple task on the Kahan Cluster[22] to test the performance of some application in parallel. Each task has their own parameter configurations, which changes for each test. For launching a script to the cluster, we must have a Bash script in a file, to let the cluster configure the context and execute the program.

Without meta-programming, what we may do, could be to implement a script that accepts parameters, and create a new script that calls the other script passing the parameters, and then inserting that parameters to the configuration, but this trick is not possible to Kahan's queue system because of it is design nature. It must have a unique script giving configuration and the line to execute the program.

With meta-programming, this can be feasible and much more effective. With Bash is possible to generate a script from a script. By this way, it is possible to make multiple tests, with different parameters, each one generated at fly, and then launched into the Kahan's queue with the arguments implicit in each script. In addition the generator script, for each iteration in the tests, it waits until the task in the Kahan is finished, and then generates the new script and continues the process of launching them until all tests are finished and all outputs are generated. This technique, which was originally implemented by my classmate Fernando Vañó García, helps us when doing exhaustive tests about parallel algorithms, because of the automation it provided.

Other languages like C++ do not give that features but it is possible to implement them, in this case by using some tricky template patterns hard to implement and sometimes, to use. This is the case of template meta-programming[23] , a technique that abuses from templates in order to implement a Turing Complete language that allows to generate and execute code at compile-time, as a first step, and then compiler is able to from that output, build the program, making the code generation transparent for the programmer.

Although, in the case of C++, at the last versions of the standard C++11[24] and C++14[25] , it implements new features like constexpr[26] that allow to create expressions that will be executed during compile-time. An example of that use is compile-time hash generation from an string. It allows to improve the constant hashes, replacing the string literal of the key by the integer number of the hash, inclusive, if keys belong to a known set a priori, then it is possible to implement a O(1) hash function that will be executed at compile-time.

In the other hand, there is a powerful tool of meta-programming, typically used on CORBA[27] for stub generation, or in Google Protocol Buffers[28] for simplifying serialization and marshaling, called IDL[29] . This kind of tools provide a intermediate language for interface definition which can be compiled into an existing language in a transparent way, and that language can also be extended implementing the translator for each language. By this way it is possible to reduce the amount of lines of code and time spent when developing because these tools give you all the bloat code in just one automated step, in addition, avoiding coding errors.

A drawback of these kind of tools is that usually they are not standard, and all of them always need external preprocessors or generation tools, that are not implicit embedded in the main language it is being used, as a difference to template meta-programming, that it is implicit in C++. So, it opens the door to new problems like version compatibility, non standard approaches, implemented in different ways, making difficult to unify the whole process and always depending of external tools that must be maintained for third parties.

After studying these both techniques, I discarded them because of its drawbacks and incompatibilities in the case of the template meta-programming, which is not allowed in plain C. So my final decision was to use preprocessor meta-programming[30] , this technique abuses from C preprocessor in order to implement a Turing Complete language which depends on the number of deferring expansions we implement in order to prevent macros painted blue, thus allowing macros to be recursive.

To understand the potential of this technique, there are multiple frameworks nowadays which implement this kind of techniques and that are really powerful for providing meta-programming in a native way over C and C++ using the preprocessor. A different examples of these frameworks are chaos-pp[31] and Boost.Preprocessor[32] , the second is the formalization of the first one, and both are developed by Paul Mensonides and Vesa Karvonen. In addition, after developing these both frameworks, they continue developing order-pp[33] , a meta-language based on lambda calculus implemented on the top of C preprocessor.

At the end, my final design decision was to use preprocessor meta-programming, and design and implement a module based on the preprocessor, to allow to simplify stub generation for plug-ins and metadata related code. A meta-programming technique is necessary in the framework, in order to simplify stub generation and code introspection or reflection, which will provide a weak application programming interface. This is the main reason of the need for a meta-programming environment, it will hide the registration mechanisms of the run-time registration and code discovering, making transparent the use of the metadata module and plug-in registration for the programmer.

The preprocessor module must allow to implement techniques as repetition, recursion, conditionals, arithmetic operations, enumerations, also it has to provide varidic arguments support, for each standard C because C89 and C99 has different specs for these features. These techniques will be used for generating the plug-in stubs, implementing the registration of the signatures for data, functions, classes and methods, and also providing a way to implement foreign function interfaces without needing to depend on the compiler, because foreign function interfaces can be implemented with the assembly language for each compiler or architecture, but it will introduce more complexity, so having a mechanism which is able to generate stubs embedded in the compilation step without external dependencies it is the best option for the multiple problems that appear over plug-in framework and metadata system.

2. 8. Metadata design

Metadata is a module that holds functionalities for describing the internal structure of the data and code of the program where the system is being used. By the use of the reflection pattern it allows to register a signature that defines the code and data of the program, it creates a new level of abstraction done at run-time that gives all information of the program structure.

In addition it can populate this information to allow self-discovering to other components, it is able to register and to name that information, then assigning it to a specified context and scope. Any entity in the system can be understood by other tiers and it can be referenced easily and interpreted, and in the case of the data it can be also reconstructed or copied, holding the original signature.

The system also must provide an easy way to use the functionality of the components without knowing the implementation at compile-time. This means that functionalities can be registered and populated to the system at run-time and the code can be compiled with the references not solved, and when the program is launched all of them can be referenced and then being ready to be used, and the execution of all become opaque for the front-end. This mechanism is designed by means of the foreign function interface pattern, a functionality that does not exist natively in the C language but it can be implemented in a different ways.

The system also is able to do the reverse process, from a signature create an instance, instead of registering an instance with a specified signature. So it is possible to artificially create signatures, related to a metadata information, and then from this signatures create instances of an non-existent data declarations, function definitions or class and method definitions. This is an effective technique which gives multiple possibilities, in order to extend at run-time functionalities or data declarations in different systems.

The reflection process, registering the signatures of existent code or creating them artificially, gives support for implement foreign function interfaces, allowing to call functions or methods without knowing the real code signature at compile-time. Furthermore, signatures can be grouped in namespaces and the instances referred to that signatures, into scopes, giving different levels of access in order to protect memory when it is unavailable to be modified. It also it is used to serialize metadata information to allow sharing it between components making a message passing decoupled from the medium.

There are other design decisions that have been taken into account, related to the implementation, like the Object Pool Pattern[34] , which is a wrapper of the memory pool used to generate signatures and data in the Metadata system or the Singleton Pattern[35] for managing the current instance of the system top module and the PImpl idiom to hide implementations.

The result is a flexible way to identify, reproduce, and manage the code of it is own program, the Metadata system would be able inclusive to reproduce itself but it does not have sense because the system it is commonly used as a wrapper for other existing programs, so it can be linked directly to the main application. Any application, having this system linked and used for register itself, can be understood by another one with the same properties. It introduces a protocol which allows learning how programs are implemented by themselves.

2. 9. Serialization and marshaling

Serialization[36] is a mechanism which allows a program to convert data from a program into a generic format independent from platform, technology or medium. This packed format can be managed, and inserted into a stream that can be sent over any medium capable of transport bits of information. By this mechanism the application can save the state its state and construct it from the stream doing the inverse process no matter the technology where is going to be output the data.

This mechanism is going to help us in the communication between different tiers of the plug-in framework. By this way data that is need to be transmit between tiers can be compact in a packet that can be transmitted via memory, file or network. In the other hand this mechanism also can help us avoiding problems as ABI compatibility, and being an intermediate format for different languages and systems which have different data interpretations.

Most of the times, format used by serialization is standard, portable, generic, human readable and common used. These properties are the key and most formats widely used in the world fit that properties. This provides an easy programmable interface that can be shared no matter technology, architecture or medium. Most of modern distributed systems use this kind of formats, including REST[37] , a common software architecture.

Although in the plug-in framework I have chosen a binary format for the specification of the data serialization format. This means that data which is in memory, is converted into another binary format, which is defined a priory and has always the same rules. By this way the serialized format is not human readable but it is more efficient, it can be read or write really fast, it can be interpreted an packed on the fly without using parsers with a minimum overhead, and the space used is the most compact as possible.

Using a binary format can have drawbacks, such as making difficult to understand easily data contents, but this can be avoided by using a intermediate interface description language, as XML Schema[38] does, but using this schema as human readable, and the other as the compiled version of the original one.

An example of previous technology is Google Protocol Buffers. An excellent approach of this technique that uses a compiler to generate stubs for any language implementing serialization of previous interfaces defined, that is converted to binary format when serialized, fitting the schema defined.

The binary output format for plug-in framework is described as a header and body. Header is a table which index by offset and type, the data stored, it also saves metadata information, to give support for reflection and metadata mechanisms, as run-time registration and discovering.

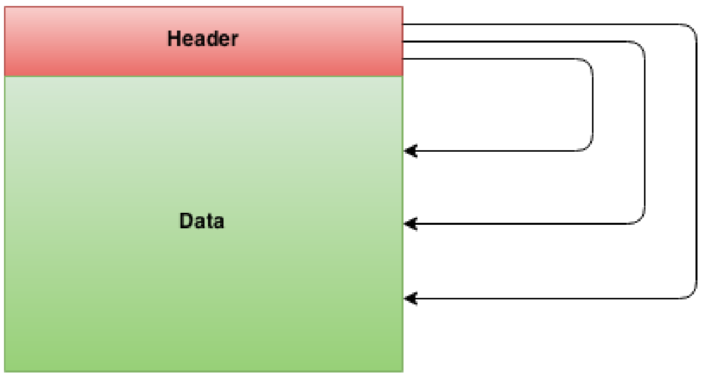


Figure 3: Diagram of serialization format.

As shown in the Figure 3, header of the data is a table which makes the specification, saving additional information, metadata, type and position where data is serialized. For convention data is always transformed to little-endian. Types supported are given by metadata module, which gives basic primitives and a mechanism to mark them and to know the internal structure.

Endianness[39] is important for serialization because data representation is different between platforms. So when serializing an object is important to know source and destination endianness which has to be serialized data. A common format allows to have an intermediate where the source platform does not need to know the destination ones, it just makes the conversion to the format specification, and the same happens in the other tier, it does not need to know what is the original format from data belongs to, it just has to know their own format and convert from the generic ones to a valid representation.

By using preprocessor technique, it is possible to detect in compile-time the endianness of the platform where program is being compiled, in this case, there is a macro block construction checking against platforms detecting what endianness is being used. At the end, this conversion is going to be done automatically if needed, to preserve the same endianness when serializing data, avoiding problems between SO and architectures.

Strings are converted into ASCII format to make them more compact, currently only it is allowed one byte by character, but it can be extended. This makes the conversion from a C string trivial and really faster, just copying it, but with the difference that null char termination is not saved, it just know size of the string at first, saved in a 32-bit length unsigned integer to read them in bursts.

Floating point values are converted always to single precision if possible, 32-bit length, and always using IEEE standard although fixed point values are cheaper to encode, it is preferable to maintain flexibility and compatibility with most of architectures. For integer and boolean, which they are also integers, the conversion to the binary format is done directly, unless it is need endianness transformation.

Marshaling[40] is a technique similar to serialization, but it adds some extra functionalities, in order to determine the source of the data, naming and registration details, and data share can be done by value, copying the instance of the data by means of the typical serialization process of data reconstruction, or it also can be done by reference, with the information is maintained that refers the source and target tiers.

By this way, with marshaling it is possible to serialize an object, pass it to the other tier by reference and in the other side, reconstruct the object instance, with data automatically bound to the source, and by this way it is possible to execute remote procedure calls. The data can be also bound and by this way it is automatically modified in the source tier too.

To allow this technique, the Metadata system has to provide a signature system which will be registered and serialized, to give for the other tier, it will be reconstructed on the other with the binding, so by this mechanism plug-ins can communicate between them and the game engine in a transparent way, without depending on the medium it is being used, memory, file or networking.

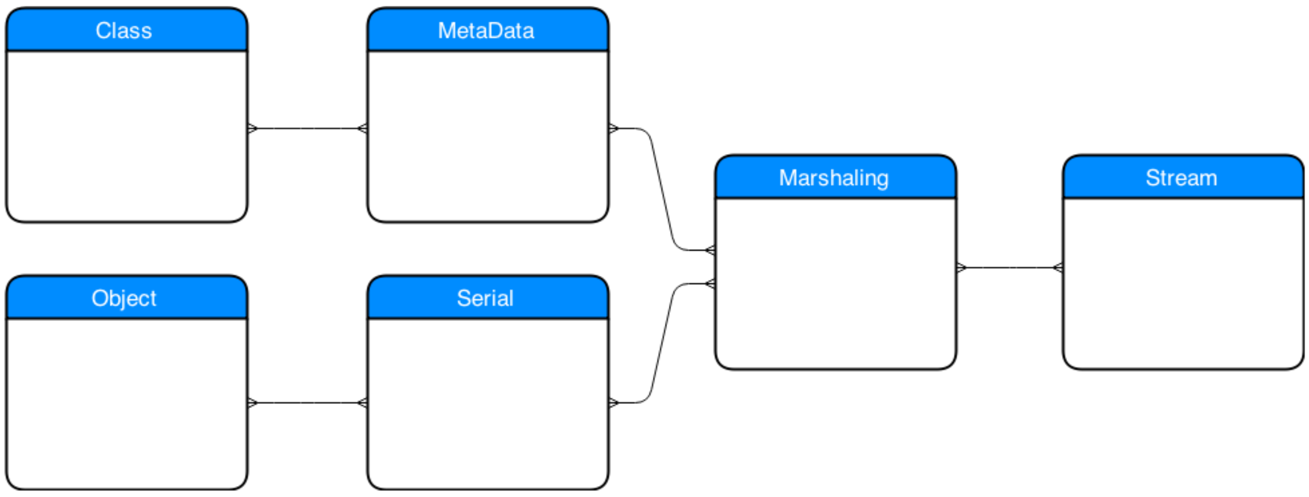


Figure 4: Complete diagram of communication process.

As shown in Figure 4, the process consist in obtain from the code, called Class in the diagram, the signature reflected by the Metadata system, specified in signatures. The instance of what is being bound, in this case called Object, but it can be a function, data, or any referenceable entity, and it is converted into a serial, which is a stream of bytes with a predefined format, by means of serialization process. Then with the metadata information, that describes the real code, and the serial instance, both of them are referenced and named by the marshaling process, which enables a context where this representation belongs and where the real data and code is. The output process is a packed stream of bytes in a binary form that can be passed to a function by reference, or can be send through network or written into a file.

When doing the inverse process, everything can be reproduced and reconstructed, except from some cases like, for native code, which is compiled into a binary, the original Class entity cannot be reconstructed, in the case of scripting it is possible to reconstruct it, so it is able to recreate functionalities copied by value from the other tier, and execute it more efficiently than if a remote produce call is done. But it is not always a possible process. The Object instance can be inferred, by means of the metadata signature, and the deserialization process is able to reconstruct that instance in a correct format for the current platform.

2. 10. Run-time registration and infrastructure offering

One of the properties of a plug-in is that it can be loaded or unloaded at run-time, and the host program does not know after plug-ins are scanned and loaded what kind of functionalities are going to be provided for him.

Those functionalities can be provided by different techniques, but all of them must fit protocol provided by plug-ins that relies on metadata. When registering plug-ins is important to track and initialize all metadata needed, by this way host application can understand functionalities are being provided.

First of all, host application loads scripts into a context where it can be executed, could be a process on memory which executes binary instructions (in the case of native compiled plug-ins) or a run-time environment, where a script is attached and can be interpreted or executed.

For each context, the plug-in framework must rely on a different implementation which allows to dynamically load and execute code, no matter the language or platform where the system nor plug-in is being executed. By this reason, the plug-in loader has been designed in order to hide the internal implementation of the plug-in. Externally a generic handle is provided, which holds the instance of the concrete loader and makes transparent the loading process for the front-end.

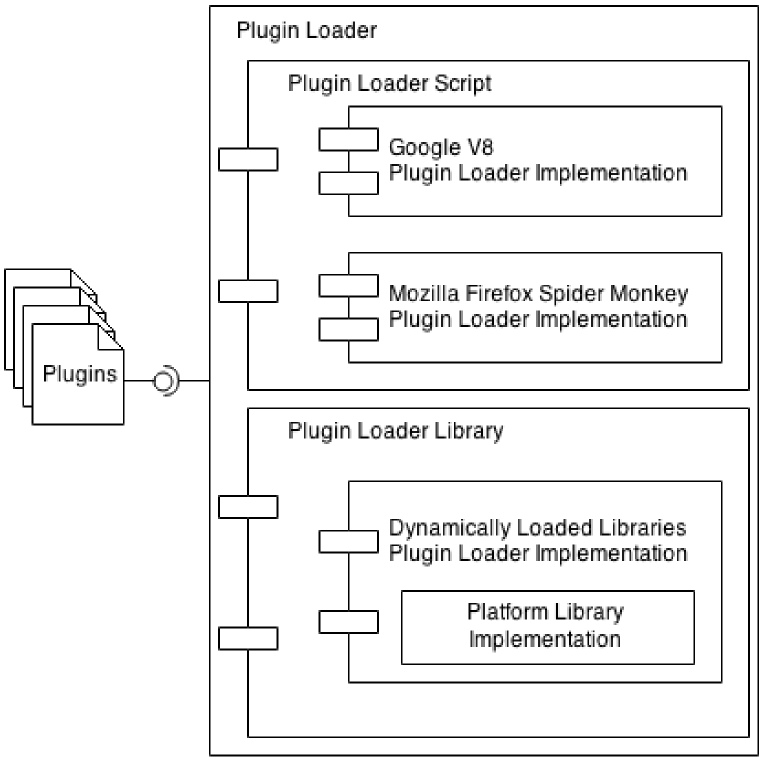

Plugin Loader
Plugin Loader Script
Google V8
Plugin Loader Implementation
Mozilla Firefox Spider Monkey
Plugin Loader Implementation
Plugins
Plugin Loader Library
Dynamically Loaded Libraries
Plugin Loader Implementation
Platform Library
Implementation

Figure 5: Plug-in loader diagram.

A generic interface of the plug-in loader wraps the implementations that can rely on multiple technologies. The system is designed in a hierarchical structure with the PImpl idiom in order to hide implementation details. Each loader module relies on different technologies, so in the case of the PluginLoaderLibrary module, which implements the loading for native plug-ins, it depends on the System module in order to use the correct system calls for each platform supported.

In the case of PluginLoaderScript, this is a generic interface that can be derived in order to support multiple scripting languages. In addition, for each scripting language it is possible to implement it by means of different software implementations. So it is possible to choose between the loader is going to be used for the same language. In the Figure 5, it is possible to see the example of JavaScript language, which can be loaded from Google's V8 Engine or Mozilla Firefox's Spider Monkey Engine in a transparent manner.

When plug-in is already loaded, then all of them have an entry point, which depends on plug-in protocol specification, that host application uses to provide the infrastructure and services to allow after to the plug-in register its functionalities and to be used. This entry point and the protocol specifications are hided to the front-end by means of meta-programming tools provided, so this is not need to be learnt by the programmer although he is interested in the implementation. For front-end users of the system this is easily wrapped and can be used minimizing the programming errors.

After using the meta-programming tools provided, the result would be that plug-ins will be wrapped by the Metadata system in a transparent way. From outside of the plug-in, the Metadata system provides an easy way to execute functionalities of the plug-ins, all of them using the same protocol, the simplest possible to the front-end user. From inside of the plug-in, the structure is registered in automated way so programmer only has to rely on functions without taking care about metadata related issues.

Plug-in framework and the Metadata system can be easily integrated into the game engine, and the infrastructure offering in both directions begins really easy to use, and stubs needed to generate communications between plug-in functionalities or game engine services become transparent to the both tiers.

The Figure 6 illustrates how the process is. In order to simplify the drawing, in the bottom of the image there is the image of the plug-in loader shown in the Figure 5, but this part of the system is wrapped by a top module, the plug-in framework. In addition, the connection point shown in the loader, it is not only one, there is one by each language or technology is used to load the plug-ins. Finally, the preprocessor module has been obviated, and both modules, the metadata and the plug-in framework, have been drawn outside of the game engine architecture for clarity, but both are included inside as a part of the game engine, like a separated libraries.

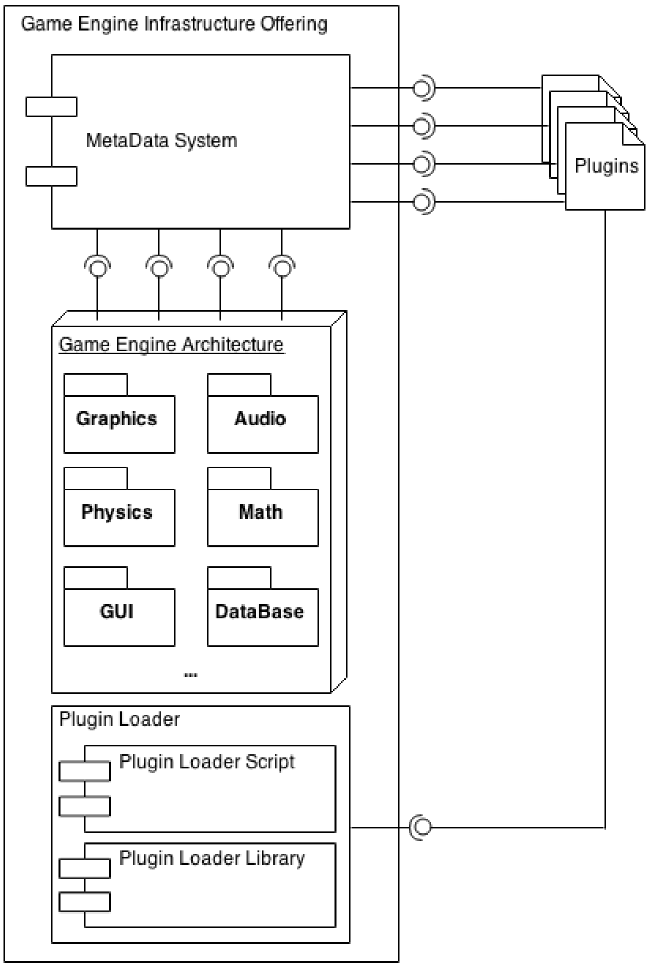


Figure 6: Diagram of game engine infrastructure offering.

When a plug-in has been registered and populated in the system, control is returned to the host application. Now, it is possible to execute any procedure without knowing anything about it, but as it would be compiled in the same binary.

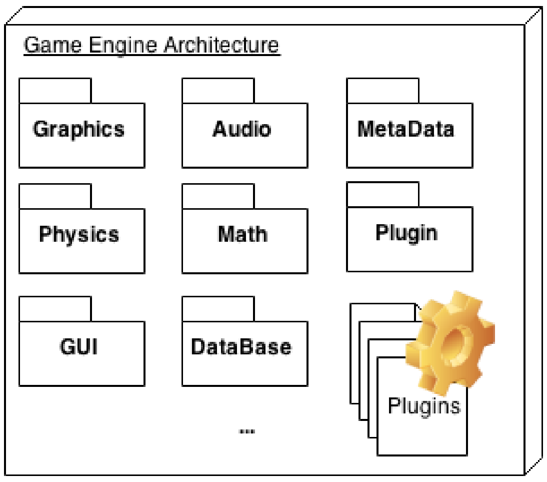


Figure 7: Diagram of game engine with plug-ins already set up.

The result, as seen in Figure 7, it is like each functionality was inside of the game engine, but the fact is that the game engine does not know how and who is being executing the plug-ins with the minimum possible overhead. The plug-ins become transparent for the game engine, they can be seen as libraries with weak interface and dependencies.

2. 11. Language agnostic and platform abstraction

In order to make generic the system, and not dependent of the technology, language of platform where it is being executed, there are multiple options available. In the last instance all of them are going to pass through the back-end of the system, which is coded in C and compiled to a binary, and it has been designed for maximize the efficiency with the minimum overhead.

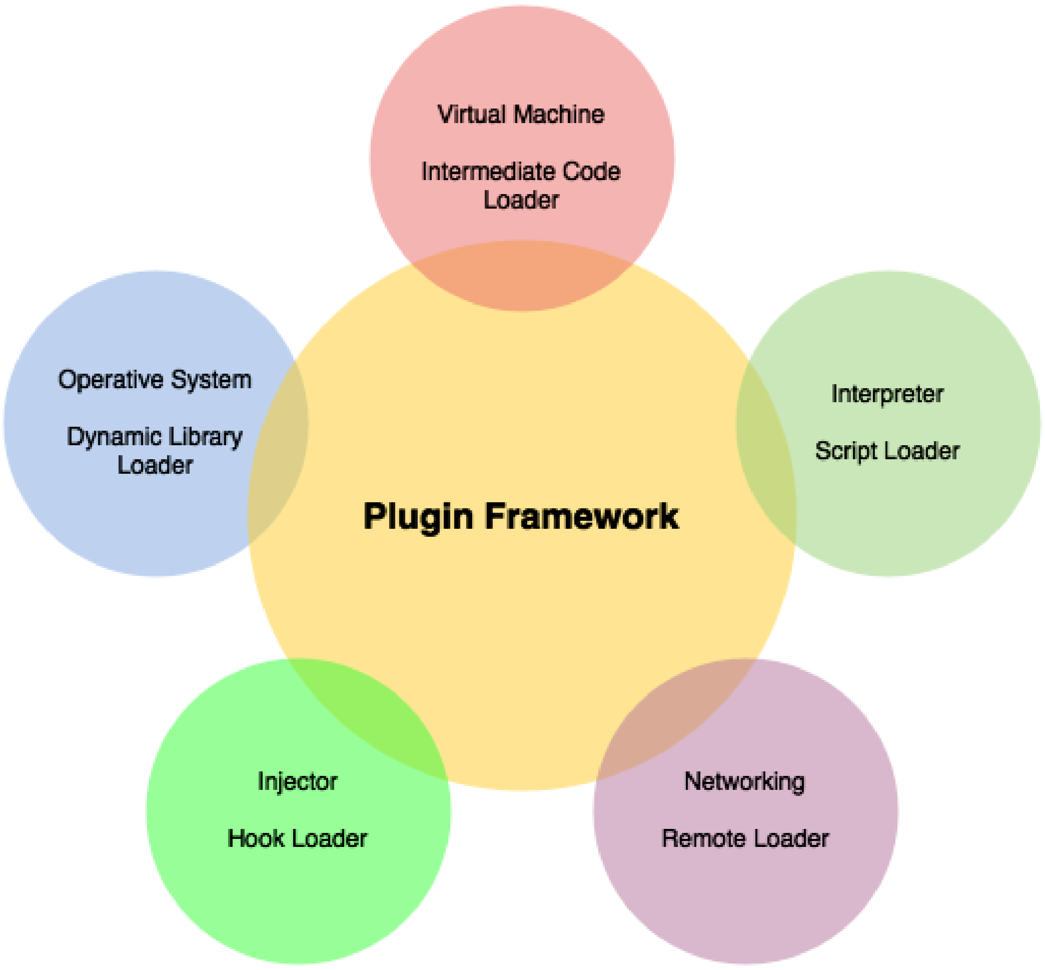


Figure 8: Diagram of Plug-in Framework embeddable loaders.

The first approach, the more commonly used when coding at the C level, it is to provide a way to load native code dynamically. This technique, most of times is provided by the Operative System, but each operative system provides different tools to do it. So the mechanism for loading code dynamically is OS dependent, and it must be implemented for each OS the game engine will run. To do it, from the System module, which makes the abstraction of the system calls provided by the OS, there must be an interface for loading dynamically those pieces of code, and return a generic handle referencing the instance of that loaded packages.

With the PImpl idiom, the implementation becomes opaque in the front-end, so in the plug-in loader implementation there must be a derived module matching the interface of the loader, and implementing the loading of the libraries at run-time, with the generic function provided by the System module, which internally has the implementation for the system calls for each OS. This level of abstraction simplifies the development of cross-platform code.

Another option would be an intermediate approach, to embed a Virtual Machine[41] commonly developed in a native language, to load an intermediate language, like Java for execute ByteCode[42] , or any existing JavaScript VM, like V8 or Spider Monkey, which usually try to compile the JavaScript code into Assembly in order to speed-up the performance. Any of them can be wrapped into a derived module from the plug-in loader, and it can provide functionalities for loading a specific language, with flexibility. In addition these languages have a big code forge which is functional, exhaustively tested and implemented with the most efficient techniques.

A third approach is to embed an interpreter inside the loader, like Python C API or Lua C API, both can be implemented into the system as the same way of the VMs, but the paradigm which that tools provide it is quite different, it is not as efficient as a VM because these ones uses precompiled code more near to Assembly than the interpreted languages, but they give usually more flexibility, the code does not need to be compiled each time it is modified, so this gives a flexibility for developers, they can create the scripts, and test them directly in the engine, viewing the effects directly on the fly, just by reloading the script.

A more generic way is to introduce a new level of abstraction provided by networking, because by this way it is possible to load and execute code remotely with any technology which supports socket connections, but this option implies that there will be an high latency or overhead, although the back-end of the application was in local-host, so this option cannot be used in order to warp functionalities with high performance or real-time requirements.

The last feasible approach is to implement a code injector directly. This is a common technique used by hackers, which has a similar behavior as a debugger. The injector is able to stop a current executed process and then it can make the trampolines and hooks from the process into the injector and vice versa. By this way it is possible to insert or modify the code of the process or change the execution flow at run-time of a binary application without having the original source code. Although this technique is possible to implement it is not going to be taken into account because of its illicit nature.

3. Implementation

In this section is going to be covered all aspects and details related to the implementation phase. Although this is a proof of concept, the full application has been designed and implemented in order to fit the future production version specs. So, it could be that most of code developed does not have sense for this project, but it has been implemented in order to in the future extend easily the application, making a full functional version for directly use it to develop new functionalities.

Other design patterns have been explained, related to the implementation, that have been not explained in the design section, because of all of them are related to concrete details of the implementation level. Most of the techniques are documented, by using easy examples in order to understand them, because in the real implementation, there are multiple techniques being used at the same time, so it becomes more difficult to explain each one separately.

The whole system has been designed in order to be most efficient as possible, because in the future it will be integrated on game engine architecture, with real-time requirements. So, the framework overhead must be the lowest possible, giving the highest genericity and flexibility.

3. 1. Preprocessor module

Preprocessor module is which allows meta-programming techniques that will be later on used by the Metadata and Plug-in modules. In this module there are multiple techniques must be implemented in order to support code generation. All techniques must reside on the preprocessor context, by this reason most of times implementation becomes obscure. For automation of the process, Unix Bash script has been used in order to generate the C preprocessor code, because most of times the implementation is a repetitive task.

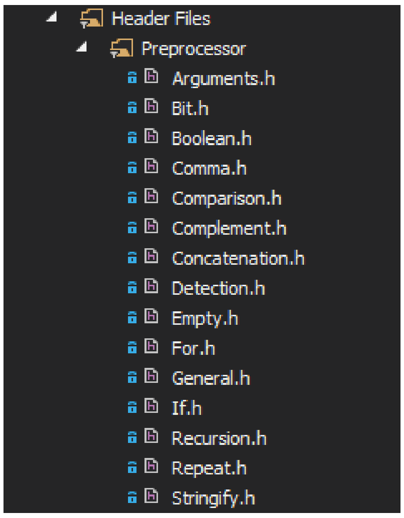

Header Files
Preprocessor
Arguments.h
Bit.h
Boolean.h
Comma.h
Comparison.h
Complement.h
Concatenation.h
Detection.h
Empty.h
For.h
General.h
If.h
Recursion.h
Repeat.h
Stringify.h

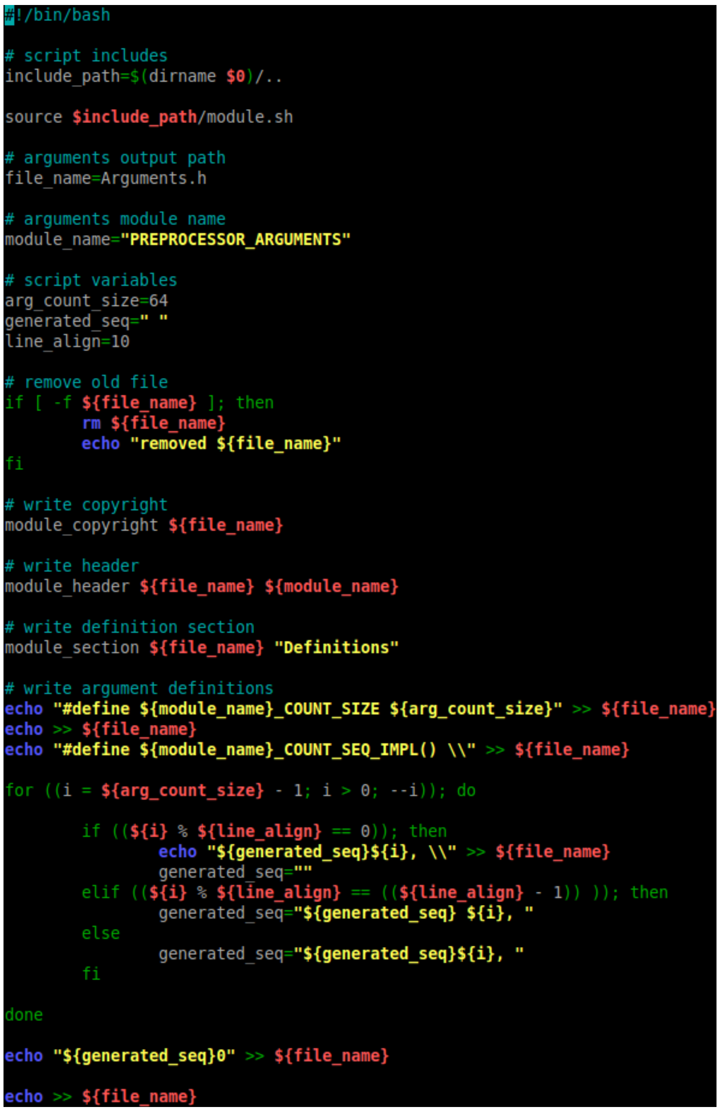
Figure 9: Preprocessor module overview.

3. 1. 1. Build environment

To maintain these scripts, another folder has been created, outside of the source, which has been called meta-programming, and there it is set up a build environment in order to configure the system and rebuild the library with the new configurations. For example, in the case of the For preprocessor implementation, there must be a limit of iterations, so in the build environment you can set the number of iterations, and then rebuild the module, in order to generate a new module with the new configuration.

In fact, what has been developed is a group of scripts that allow to generate multiple headers, implementing preprocessor techniques, in order to simplify the development of C code inside the Metadata and Plug-in modules, and to the front-end, the developer which uses the system. This means that meta-programming techniques are being used at two levels, in order to simplify, improve and make cleaner the development of the framework.

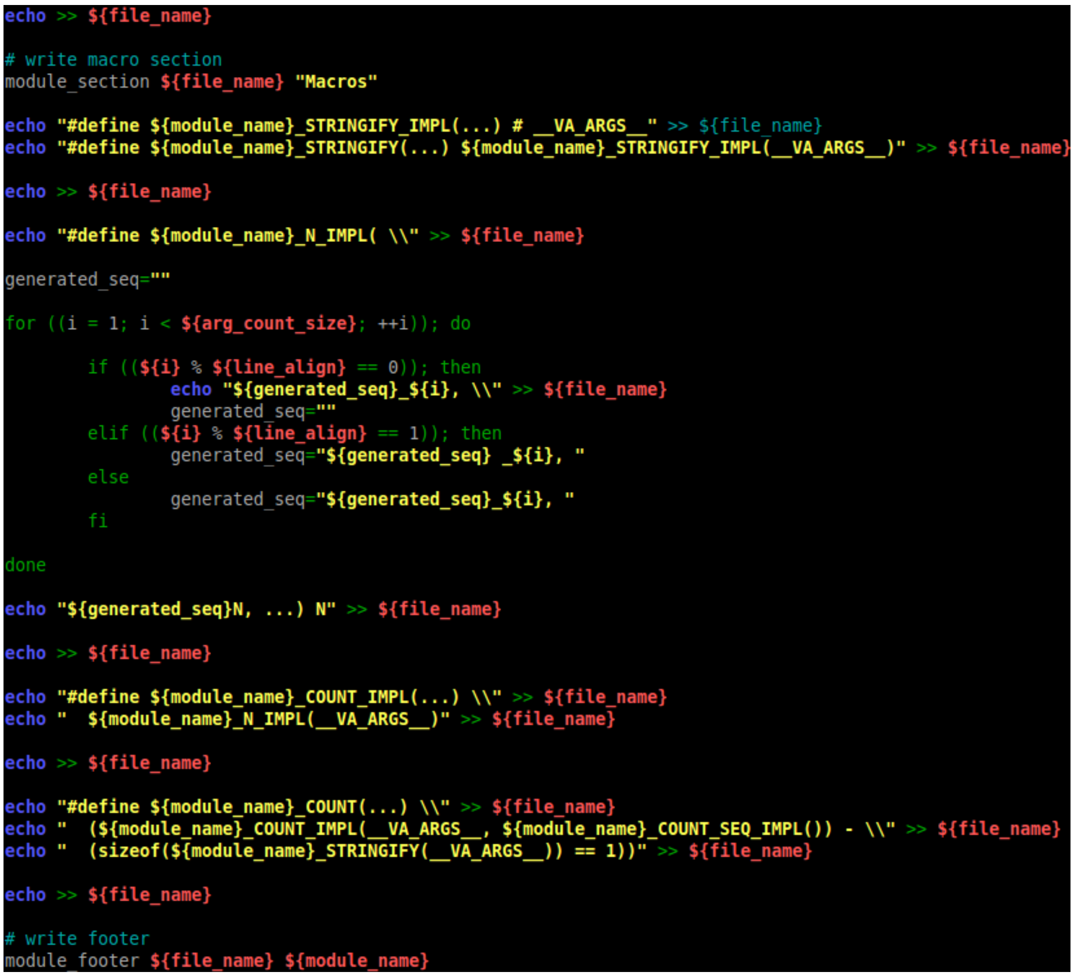
An example of this build environment is the Arguments script shown in Figure 10 and Figure 11, which generates the equivalent sub-module, which allows managing varidic arguments, and other meta-programming techniques for arguments, like argument count, which is able to replace a list of with a literal number representing the size.



```
#!/bin/bash

# script includes
include_path=$(dirname $0)/..

source $include_path/module.sh

# arguments output path
file_name=Arguments.h

# arguments module name
module_name="PREPROCESSOR_ARGUMENTS"

# script variables
arg_count_size=64
generated_seq=" "
line_align=10

# remove old file
if [ -f ${file_name} ]; then
        rm ${file_name}
        echo "removed ${file_name}"
fi

# write copyright
module_copyright ${file_name}

# write header
module_header ${file_name} ${module_name}

# write definition section
module_section ${file_name} "Definitions"

# write argument definitions
echo "#define ${module_name}_COUNT_SIZE ${arg_count_size}" >> ${file_name}
echo >> ${file_name}
echo "#define ${module_name}_COUNT_SEQ_IMPL() \\" >> ${file_name}

for ((i = ${arg_count_size} - 1; i > 0; --i)); do

        if ((${i} % ${line_align} == 0)); then
                echo "${generated_seq}${i}, \\" >> ${file_name}
                generated_seq=""
        elif ((${i} % ${line_align} == ((${line_align} - 1)) )); then
                generated_seq="${generated_seq} ${i}, "
        else
                generated_seq="${generated_seq}${i}, "
        fi

done

echo "${generated_seq}0" >> ${file_name}

echo >> ${file_name}
```

Figure 10: First part of Arguments script used for generating the equivalent sub-module in preprocessor module.

This version of the Arguments script is a simplified version, in order to allow it to be more understandable. The variable arg_count_size have been simplified; because in the original script it is configured by another script which holds all constants for configure the building of the module. In this case, it has a 64 hard-coded. The line_align variable is also hard-coded, it is related to the style of the resulting code, because if not it can be written into a line, becoming impossible to read it in a common IDE.

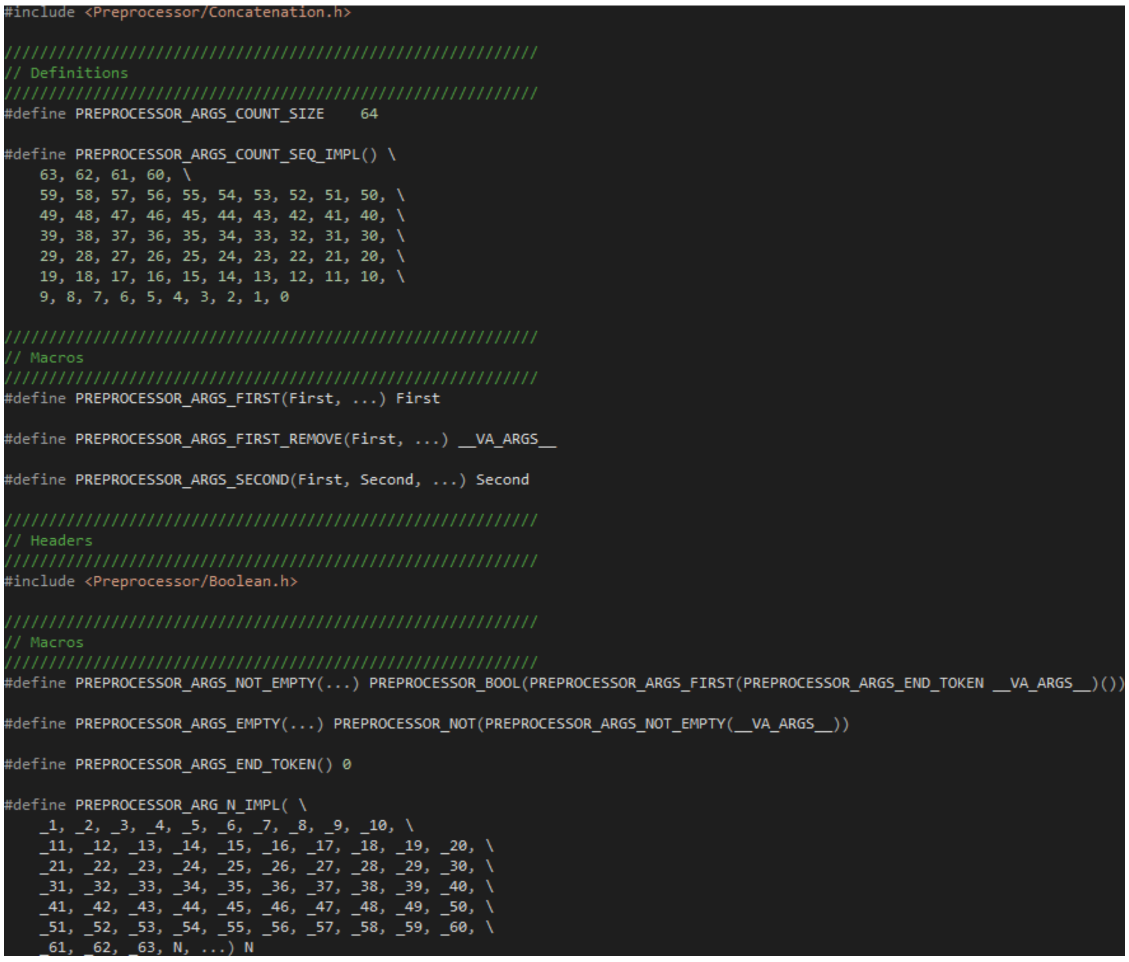


```
echo >> ${file_name}

# write macro section
module_section ${file_name} "Macros"

echo "#define ${module_name}_STRINGIFY_IMPL(...) # __VA_ARGS__" >> ${file_name}
echo "#define ${module_name}_STRINGIFY(...) ${module_name}_STRINGIFY_IMPL(__VA_ARGS__)" >> ${file_name}

echo >> ${file_name}

echo "#define ${module_name}_N_IMPL( \\" >> ${file_name}

generated_seq=""

for ((i = 1; i < ${arg_count_size}; ++i)); do

        if ((${i} % ${line_align} == 0)); then
                echo "${generated_seq}_${i}, \\" >> ${file_name}
                generated_seq=""
        elif ((${i} % ${line_align} == 1)); then
                generated_seq="${generated_seq} _${i}, "
        else
                generated_seq="${generated_seq}_${i}, "
        fi

done

echo "${generated_seq}N, ...) N" >> ${file_name}

echo >> ${file_name}

echo "#define ${module_name}_COUNT_IMPL(...) \\" >> ${file_name}
echo "  ${module_name}_N_IMPL(__VA_ARGS__)" >> ${file_name}

echo >> ${file_name}

echo "#define ${module_name}_COUNT(...) \\" >> ${file_name}
echo "  (${module_name}_COUNT_IMPL(__VA_ARGS__, ${module_name}_COUNT_SEQ_IMPL()) - \\" >> ${file_name}
echo "  (sizeof(${module_name}_STRINGIFY(__VA_ARGS__)) == 1))" >> ${file_name}

echo >> ${file_name}

# write footer
module_footer ${file_name} ${module_name}
```

Figure 11: Second part of Arguments script used for generating the equivalent sub-module in preprocessor module.

3. 1. 2. Varidic Arguments

The resulting code of the meta-programming script Arguments implements a common trick which is used along different macros related to argument preprocessing. This trick is based on a varidic macro, which accepts any number of arguments, and then these arguments are placed in the beginning of a list, which contains a sequence of all literals from arg_count_size to zero. This macro goes expanded and then it is passed to another function, which accepts arguments from one to arg_count_size, then a N argument, and the rest goes to varidic arguments. So expanding this last macro to N, it is possible to return the nth element of the sequence is passed to it, in this case the sequence of literals. The resulting header file it is similar to the Figure 12.



```
#include <Preprocessor/Concatenation.h>

////////////////////////////////////////////////////////////
// Definitions
////////////////////////////////////////////////////////////
#define PREPROCESSOR_ARGS_COUNT_SIZE    64

#define PREPROCESSOR_ARGS_COUNT_SEQ_IMPL() \
    63, 62, 61, 60, \
    59, 58, 57, 56, 55, 54, 53, 52, 51, 50, \
    49, 48, 47, 46, 45, 44, 43, 42, 41, 40, \
    39, 38, 37, 36, 35, 34, 33, 32, 31, 30, \
    29, 28, 27, 26, 25, 24, 23, 22, 21, 20, \
    19, 18, 17, 16, 15, 14, 13, 12, 11, 10, \
    9, 8, 7, 6, 5, 4, 3, 2, 1, 0

////////////////////////////////////////////////////////////
// Macros
////////////////////////////////////////////////////////////
#define PREPROCESSOR_ARGS_FIRST(First, ...) First

#define PREPROCESSOR_ARGS_FIRST_REMOVE(First, ...) __VA_ARGS__

#define PREPROCESSOR_ARGS_SECOND(First, Second, ...) Second

////////////////////////////////////////////////////////////
// Headers
////////////////////////////////////////////////////////////
#include <Preprocessor/Boolean.h>

////////////////////////////////////////////////////////////
// Macros
////////////////////////////////////////////////////////////
#define PREPROCESSOR_ARGS_NOT_EMPTY(...) PREPROCESSOR_BOOL(PREPROCESSOR_ARGS_FIRST(PREPROCESSOR_ARGS_END_TOKEN __VA_ARGS__)())

#define PREPROCESSOR_ARGS_EMPTY(...) PREPROCESSOR_NOT(PREPROCESSOR_ARGS_NOT_EMPTY(__VA_ARGS__))

#define PREPROCESSOR_ARGS_END_TOKEN() 0

#define PREPROCESSOR_ARG_N_IMPL( \
    _1, _2, _3, _4, _5, _6, _7, _8, _9, _10, \
    _11, _12, _13, _14, _15, _16, _17, _18, _19, _20, \
    _21, _22, _23, _24, _25, _26, _27, _28, _29, _30, \
    _31, _32, _33, _34, _35, _36, _37, _38, _39, _40, \
    _41, _42, _43, _44, _45, _46, _47, _48, _49, _50, \
    _51, _52, _53, _54, _55, _56, _57, _58, _59, _60, \
    _61, _62, _63, N, ...) N
```

Figure 12: Arguments sub-module resulting from script module build process.

The implementation of the meta-programming module has been a good idea, because with less than 200 lines of Unix Bash code, the system generates more than 1200 lines, with an easy configuration, for extending it if necessary. An image of the resulting code showing the number of lines can be seen in Figure 13.

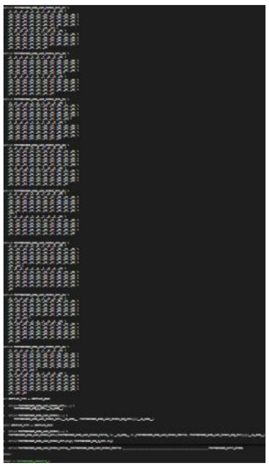


Figure 13: A global image of the output generating the Arguments sub-module.

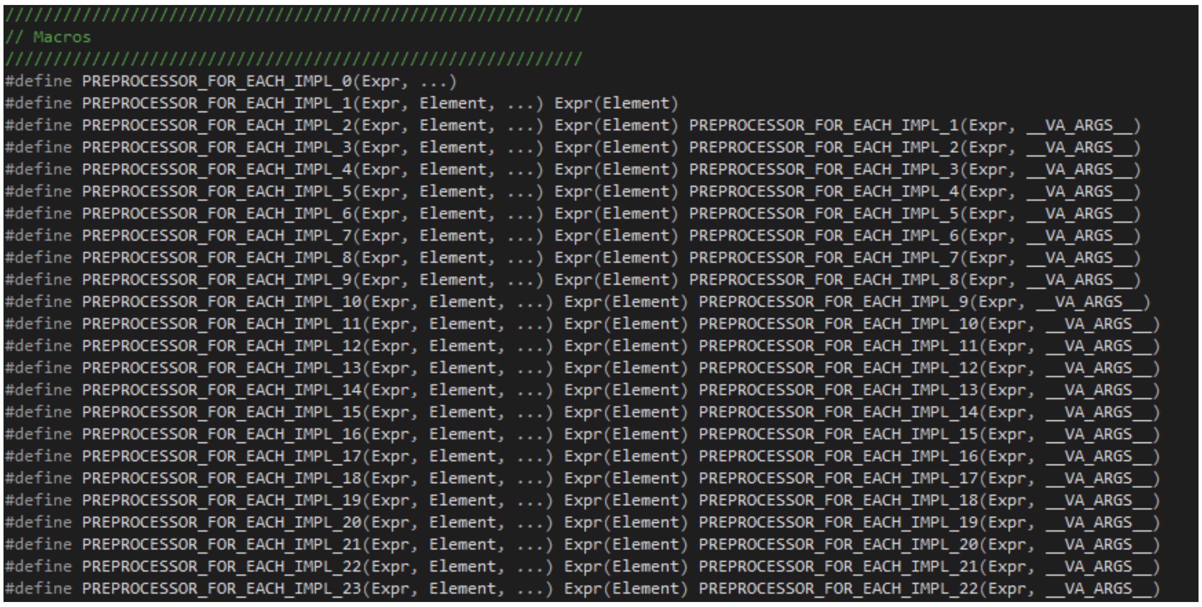
With the same trick it is possible to implement multiple functionalities related to macro arguments, like macro counting, returning the last of a parameter list, removing the first and returning the whole list, and other useful tools to simplify varidics.

3. 1. 3. Iterative and flow control statements

In this module have been implemented statements and flow control structures, which unroll the desired code in order to generate the desired structures. This is the example of Repeat, For and While sub-modules, that help unrolling the code, or the conditional sub-modules like If, to produce different code depending on the input.

For implementing repetition and code unrolling, for iterative structures, the preprocessor trick used is to implement a macro list, where each macro expands to the previous until it reaches the last one, the number of iterations can be specified or can be obtained from the number of arguments by the Arguments sub-module functionalities.

In addition, for more complex iterative structures, there are provided a sequence of macros that expand into literals, in order to achieve variable increment and decrement during the iterations. By this way it is possible to expand, and also count the number of expansions performed.



```
////////////////////////////////////////////////////////////
// Macros
////////////////////////////////////////////////////////////
#define PREPROCESSOR_FOR_EACH_IMPL_0(Expr, ...)
#define PREPROCESSOR_FOR_EACH_IMPL_1(Expr, Element, ...) Expr(Element)
#define PREPROCESSOR_FOR_EACH_IMPL_2(Expr, Element, ...) Expr(Element) PREPROCESSOR_FOR_EACH_IMPL_1(Expr, __VA_ARGS__)
#define PREPROCESSOR_FOR_EACH_IMPL_3(Expr, Element, ...) Expr(Element) PREPROCESSOR_FOR_EACH_IMPL_2(Expr, __VA_ARGS__)
#define PREPROCESSOR_FOR_EACH_IMPL_4(Expr, Element, ...) Expr(Element) PREPROCESSOR_FOR_EACH_IMPL_3(Expr, __VA_ARGS__)
#define PREPROCESSOR_FOR_EACH_IMPL_5(Expr, Element, ...) Expr(Element) PREPROCESSOR_FOR_EACH_IMPL_4(Expr, __VA_ARGS__)
#define PREPROCESSOR_FOR_EACH_IMPL_6(Expr, Element, ...) Expr(Element) PREPROCESSOR_FOR_EACH_IMPL_5(Expr, __VA_ARGS__)
#define PREPROCESSOR_FOR_EACH_IMPL_7(Expr, Element, ...) Expr(Element) PREPROCESSOR_FOR_EACH_IMPL_6(Expr, __VA_ARGS__)
#define PREPROCESSOR_FOR_EACH_IMPL_8(Expr, Element, ...) Expr(Element) PREPROCESSOR_FOR_EACH_IMPL_7(Expr, __VA_ARGS__)
#define PREPROCESSOR_FOR_EACH_IMPL_9(Expr, Element, ...) Expr(Element) PREPROCESSOR_FOR_EACH_IMPL_8(Expr, __VA_ARGS__)
#define PREPROCESSOR_FOR_EACH_IMPL_10(Expr, Element, ...) Expr(Element) PREPROCESSOR_FOR_EACH_IMPL_9(Expr, __VA_ARGS__)
#define PREPROCESSOR_FOR_EACH_IMPL_11(Expr, Element, ...) Expr(Element) PREPROCESSOR_FOR_EACH_IMPL_10(Expr, __VA_ARGS__)
#define PREPROCESSOR_FOR_EACH_IMPL_12(Expr, Element, ...) Expr(Element) PREPROCESSOR_FOR_EACH_IMPL_11(Expr, __VA_ARGS__)
#define PREPROCESSOR_FOR_EACH_IMPL_13(Expr, Element, ...) Expr(Element) PREPROCESSOR_FOR_EACH_IMPL_12(Expr, __VA_ARGS__)
#define PREPROCESSOR_FOR_EACH_IMPL_14(Expr, Element, ...) Expr(Element) PREPROCESSOR_FOR_EACH_IMPL_13(Expr, __VA_ARGS__)
#define PREPROCESSOR_FOR_EACH_IMPL_15(Expr, Element, ...) Expr(Element) PREPROCESSOR_FOR_EACH_IMPL_14(Expr, __VA_ARGS__)
#define PREPROCESSOR_FOR_EACH_IMPL_16(Expr, Element, ...) Expr(Element) PREPROCESSOR_FOR_EACH_IMPL_15(Expr, __VA_ARGS__)
#define PREPROCESSOR_FOR_EACH_IMPL_17(Expr, Element, ...) Expr(Element) PREPROCESSOR_FOR_EACH_IMPL_16(Expr, __VA_ARGS__)
#define PREPROCESSOR_FOR_EACH_IMPL_18(Expr, Element, ...) Expr(Element) PREPROCESSOR_FOR_EACH_IMPL_17(Expr, __VA_ARGS__)
#define PREPROCESSOR_FOR_EACH_IMPL_19(Expr, Element, ...) Expr(Element) PREPROCESSOR_FOR_EACH_IMPL_18(Expr, __VA_ARGS__)
#define PREPROCESSOR_FOR_EACH_IMPL_20(Expr, Element, ...) Expr(Element) PREPROCESSOR_FOR_EACH_IMPL_19(Expr, __VA_ARGS__)
#define PREPROCESSOR_FOR_EACH_IMPL_21(Expr, Element, ...) Expr(Element) PREPROCESSOR_FOR_EACH_IMPL_20(Expr, __VA_ARGS__)
#define PREPROCESSOR_FOR_EACH_IMPL_22(Expr, Element, ...) Expr(Element) PREPROCESSOR_FOR_EACH_IMPL_21(Expr, __VA_ARGS__)
#define PREPROCESSOR_FOR_EACH_IMPL_23(Expr, Element, ...) Expr(Element) PREPROCESSOR_FOR_EACH_IMPL_22(Expr, __VA_ARGS__)
```

Figure 14: Preprocessor for each implementation.

3. 1. 4. Recursive statements

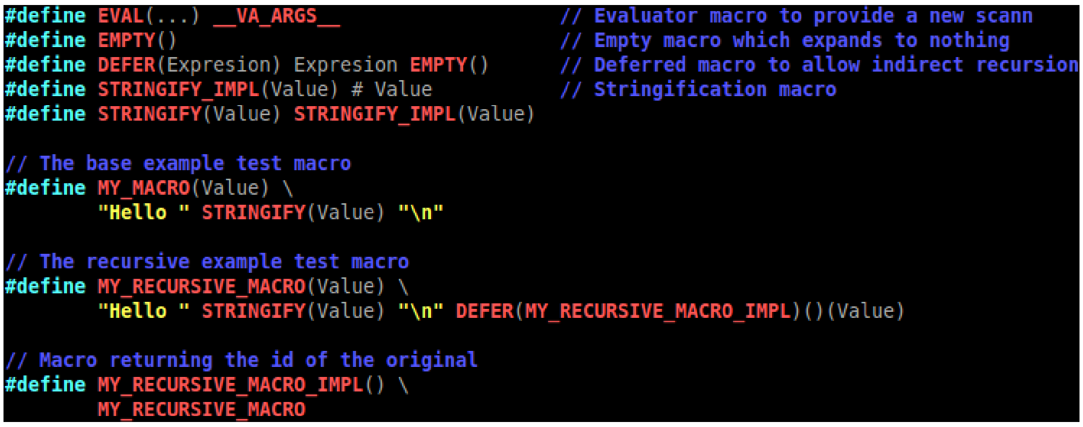
The case of the Preprocessor Recursion[43] [44] is more complex, because macros does not define that behavior by default, by this reason is why the preprocessor language is not considered Turing Complete, but there is a trick that can facility this feature, and make a hack to the preprocessor in order to partially allow recursion that can fit most of recursion problems.

When a macro is expanded, if the macro itself it is inside of the expansion, like we will do in recursion, it becomes painted blue, and then that macro cannot being expanded another time, so recursion can never happens. But, there is a way to avoid a macro becoming painted blue, but it is limited to a certain number of expansions. This number it is defined by the meta-programming module, on the configuration file that defines the rules for the building of the Preprocessor module. This constant that is set a priori is used to determine, in the implementation, how many expansions we will provide until the last macro becomes painted blue, so we are defining the maximum level of recursion.

Understanding the mechanism that preprocessor uses to avoid macro recursion, it is possible to indirectly avoid that mechanism and it can be implemented with the use of that evaluators. Each evaluator is used to defer an expression during an macro scan, when the disabling context is present, so by deferring the expression, it can be passed to another macro, making the expansion to the previous expression, where the disabling context for the original one is not present, and at the end appears another time the original expression, without being painted blue, and the second one can be another time expanded.

This process can be done as many times as the specification in the configuration. So the trick is to use as many evaluators expanding to the variable arguments, allowing new scans to the expression and giving a partial way of implement recursion. In addition, it is possible as in iterative solutions, to implement iterators, for counting the depth level or allowing to make maps, similar to the for each expression.

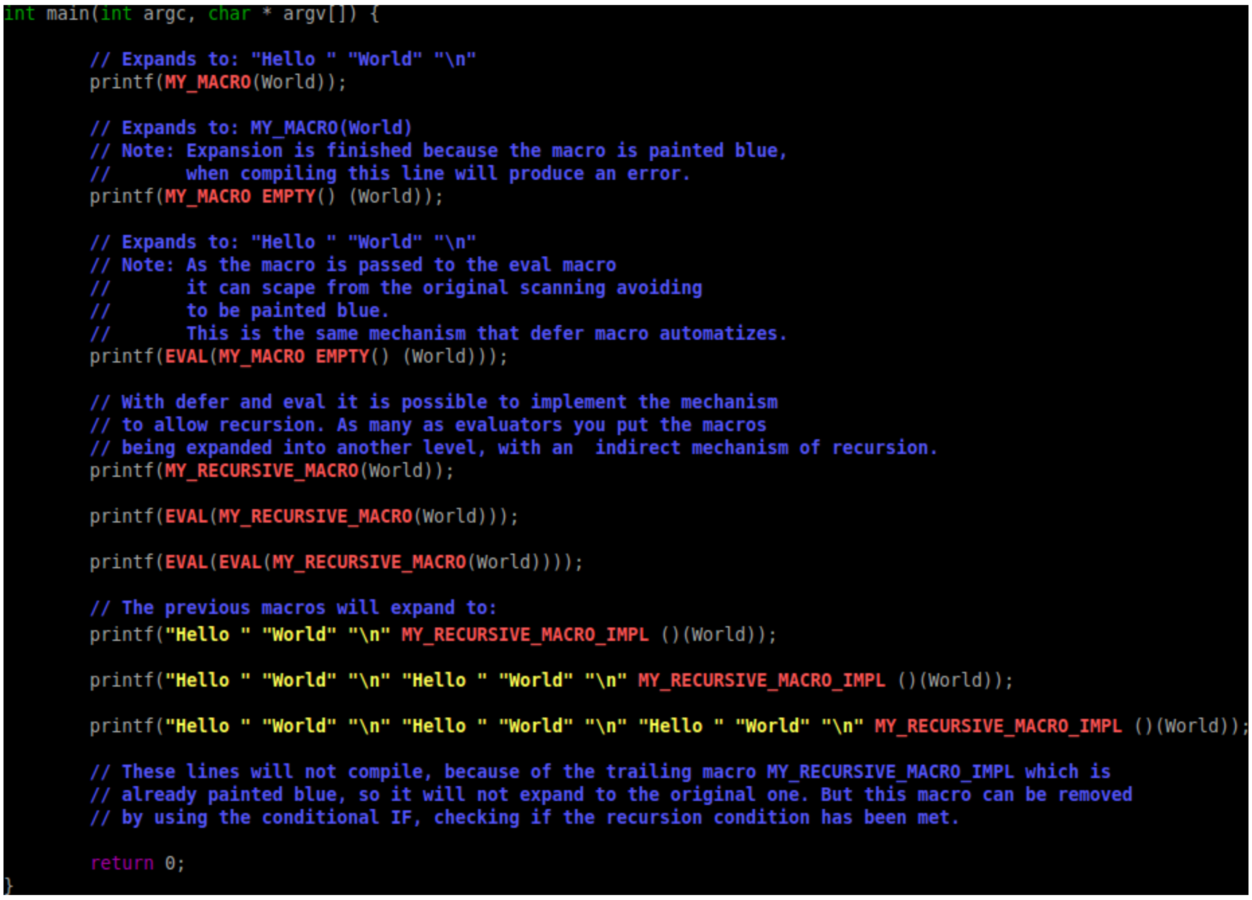
An example of the implementation is provided in the Figure 15 and Figure 16, in order to give an easy way to understand it, because the real code is really obfuscated and has other tricks to improve recursion techniques.



```
#define EVAL(...) __VA_ARGS__                   // Evaluator macro to provide a new scann
#define EMPTY()                                 // Empty macro which expands to nothing
#define DEFER(Expresion) Expresion EMPTY()      // Deferred macro to allow indirect recursion
#define STRINGIFY_IMPL(Value) # Value           // Stringification macro
#define STRINGIFY(Value) STRINGIFY_IMPL(Value)

// The base example test macro
#define MY_MACRO(Value) \
        "Hello " STRINGIFY(Value) "\n"

// The recursive example test macro
#define MY_RECURSIVE_MACRO(Value) \
        "Hello " STRINGIFY(Value) "\n" DEFER(MY_RECURSIVE_MACRO_IMPL)()(Value)

// Macro returning the id of the original
#define MY_RECURSIVE_MACRO_IMPL() \
        MY_RECURSIVE_MACRO
```

Figure 15: Header definitions of preprocessor recursion example.

In this example, the eval provides the behavior explained before, it can evaluate a new expression which has been indirect expanded by defer macro, so it is possible to avoid macro to be painted blue.



```
int main(int argc, char * argv[]) {

	// Expands to: "Hello " "World" "\n"
	printf(MY_MACRO(World));

	// Expands to: MY_MACRO(World)
	// Note: Expansion is finished because the macro is painted blue,
	//       when compiling this line will produce an error.
	printf(MY_MACRO EMPTY() (World));

	// Expands to: "Hello " "World" "\n"
	// Note: As the macro is passed to the eval macro
	//       it can scape from the original scanning avoiding
	//       to be painted blue.
	//       This is the same mechanism that defer macro automatizes.
	printf(EVAL(MY_MACRO EMPTY() (World)));

	// With defer and eval it is possible to implement the mechanism
	// to allow recursion. As many as evaluators you put the macros
	// being expanded into another level, with an  indirect mechanism of recursion.
	printf(MY_RECURSIVE_MACRO(World));

	printf(EVAL(MY_RECURSIVE_MACRO(World)));

	printf(EVAL(EVAL(MY_RECURSIVE_MACRO(World))));

	// The previous macros will expand to:
	printf("Hello " "World" "\n" MY_RECURSIVE_MACRO_IMPL ()(World));

	printf("Hello " "World" "\n" "Hello " "World" "\n" MY_RECURSIVE_MACRO_IMPL ()(World));

	printf("Hello " "World" "\n" "Hello " "World" "\n" "Hello " "World" "\n" MY_RECURSIVE_MACRO_IMPL ()(World));

	// These lines will not compile, because of the trailing macro MY_RECURSIVE_MACRO_IMPL which is
	// already painted blue, so it will not expand to the original one. But this macro can be removed
	// by using the conditional IF, checking if the recursion condition has been met.

	return 0;
}
```

Figure 16: Implementation of preprocessor recursion example.

In this example it is possible to see the approach step by step. From the base macro example until the recursion approach, showing first how defer is implemented and then how by using it, with evaluators it is possible to avoid the disabling context of the scan, providing the recursion.

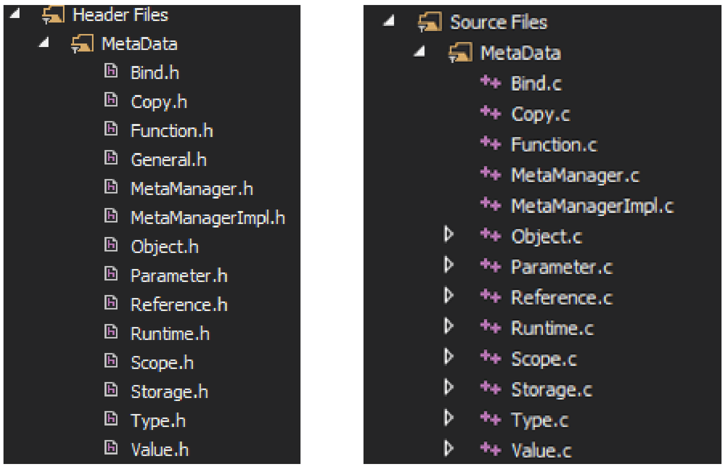
3. 1. 5. Module use cases

These iterative techniques are useful when generating the plug-in stubs for functions and methods. It is possible to expand by each parameter of the function signature, a sequence of code, and iterate between multiple parameters in order to identify and obtaining the raw result in order to call the real function. It can unroll the metadata information and write the direct call to the real function, avoiding this task to the programmer, and making transparent the foreign function call. It can be also used for the creation of the signatures, simplifying the task of creating each parameter and inserting it to the parameter list signature.

Other sub-modules as Bit or Complement and Boolean provide arithmetic and boolean operations, which can be useful for using them with the conditionals. If, Comparison and Detection provide the conditional and control operations, used later on for iteration, recursion and repetition mechanisms. Finally there are Concatenation, Stringify, Empty and Comma sub-modules, which are a sugar for the preprocessor, giving a generic basic set of tools to develop the whole module.

3. 2. Metadata module

Metadata module as defined before, it is used in order to make a wrapper for self-describing the code structure. For this, there are multiple sub-modules, which can describe, reference and bind, or create programming structures. So as it can be seen in Figure 17, there are multiple sub-modules in order to describe types, variables, references, functions or objects.



```
Header Files
  MetaData
    Bind.h
    Copy.h
    Function.h
    General.h
    MetaManager.h
    MetaManagerImpl.h
    Object.h
    Parameter.h
    Reference.h
    Runtime.h
    Scope.h
    Storage.h
    Type.h
    Value.h
```

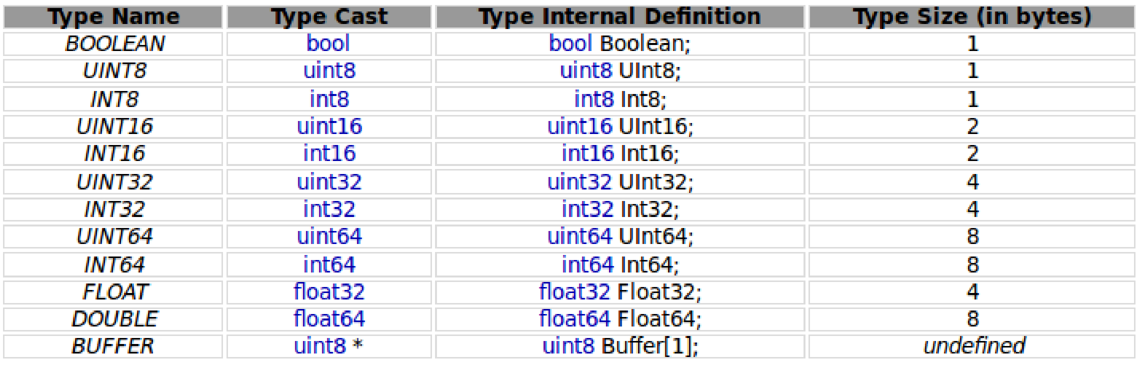
```
Source Files
  MetaData
    Bind.c
    Copy.c
    Function.c
    MetaManager.c
    MetaManagerImpl.c
    Object.c
    Parameter.c
    Reference.c
    Runtime.c
    Scope.c
    Storage.c
    Type.c
    Value.c
```

Figure 17: Metadata module overview.

From the implementation point of view, the system is going to described from fine grain to coarse grain, in order to understand layer by layer the full system, which has an estrange behavior and uses tricky techniques in order to achieve the desired functionality.

3. 2. 1. Types

The first sub-module is going to be introduced is the Type. This module describes the basic data types which the system has in order to describe the code, the properties of each type, like size, name or sing. It has other functions in order to work with them, like type casting and promotion.



Type Name	Type Cast	Type Internal Definition	Type Size (in bytes)
BOOLEAN	bool	bool Boolean;	1
UINT8	uint8	uint8 UInt8;	1
INT8	int8	int8 Int8;	1
UINT16	uint16	uint16 UInt16;	2
INT16	int16	int16 Int16;	2
UINT32	uint32	uint32 UInt32;	4
INT32	int32	int32 Int32;	4
UINT64	uint64	uint64 UInt64;	8
INT64	int64	int64 Int64;	8
FLOAT	float32	float32 Float32;	4
DOUBLE	float64	float64 Float64;	8
BUFFER	uint8 *	uint8 Buffer[1];	*undefined*

Table 1: Table description of different types in the Metadata system.

As it can be seen in the Table 1, all possible standard types are covered by the system, with multiple sizes. The Buffer type it is a raw type for internal usage, in order to simplify memory management of an instanced type.

For retrieving compile-time information about types there are a macro based methods to manage types:

- MetadataTypeName(TypeName) : Returns a literal identifying the type name.
- MetadataTypeCast(TypeName) :Casts to a valid C type.
- MetadataTypeSize(TypeName) :Gets the size of type.
- MetadataTypeGetSize(TypeName) :Gets the size of type in run-time, expects the literal obtained with MetadataTypeName as parameter.

By using a combination of function and macros, it is possible to cover in compile-time and run-time the requirements of the whole system, because type information is available in both contexts.

The Type sub-module is used among the whole system, but the most direct dependency is over the Value sub-module. This one it is the ones that provides the creation of any variable of the desired type, with the desired size, this means that it can hold any variable, from built-ins to arrays, and it also allows to create signatures, that are Values which has not are assigned to a memory block. In addition, a Value can be bound or created during the run-time, without assigning it to a concrete variable.

3. 2. 2. Values

Value is a generic wrapper to a piece of memory that has type, size, and the reference to the storage manager in order to keep the data. It can be of any type shown before in the type table, and can be directly casted, retrieved and modified, and also gives capabilities to allow type-inference to auto-describe it at run-time.

In this implementation values are treated as references to data, similar as in Java. So each value is pointing to a block of memory and when a value is copied you are copying the reference to that piece of memory, increasing the reference counter of that data. It is possible to copy the internal data representation of the value, but it needs to create a new instance and copying it explicitly.

A Value can be classified in the following types depending on its properties:

- Valid: A value is only valid if it belongs to a valid type, no matter if it is instantiated or not.
- Abstract: If a value is abstract, it means that it does not refers to any data, it is just a type definition without data implementation, used for interface definition.
- Instanced: If a value is instanced, it means that it has at least one element of desired type allocated in memory which is referring to.
- Built-in: If a value is built-in, it has one element of any type allocated in memory.
- Array: If a value is an array, it has more than one element of any type allocated in memory.

A value type hierarchy described above can be represented in the following graph on the Figure 18:

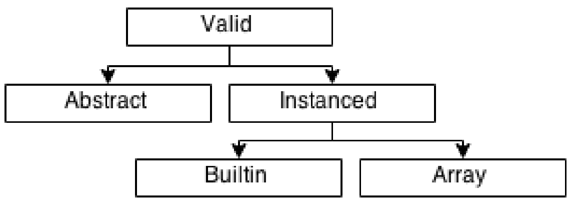


Figure 18: Graph dependency of Value properties.

The basic operations that can be performed to a Value are the following ones, in the real implementation all of them have been extended in order to provide a higher level of abstraction and other complex functionalities:

- MetadataValueCreate(Scope, Name, Type, Count, Data) : Creates a value specifying the scope where it lives, the name to reference it, type of the value, number of elements and pointer to data implementation if copy constructor is need. Returns the reference to the value.

- MetadataValueTypeOf(Scope, Value) : Returns the literal referring to type of the value.

- MetadataValueTypeSize(Scope, Value) : Gets the size of type of specified value.

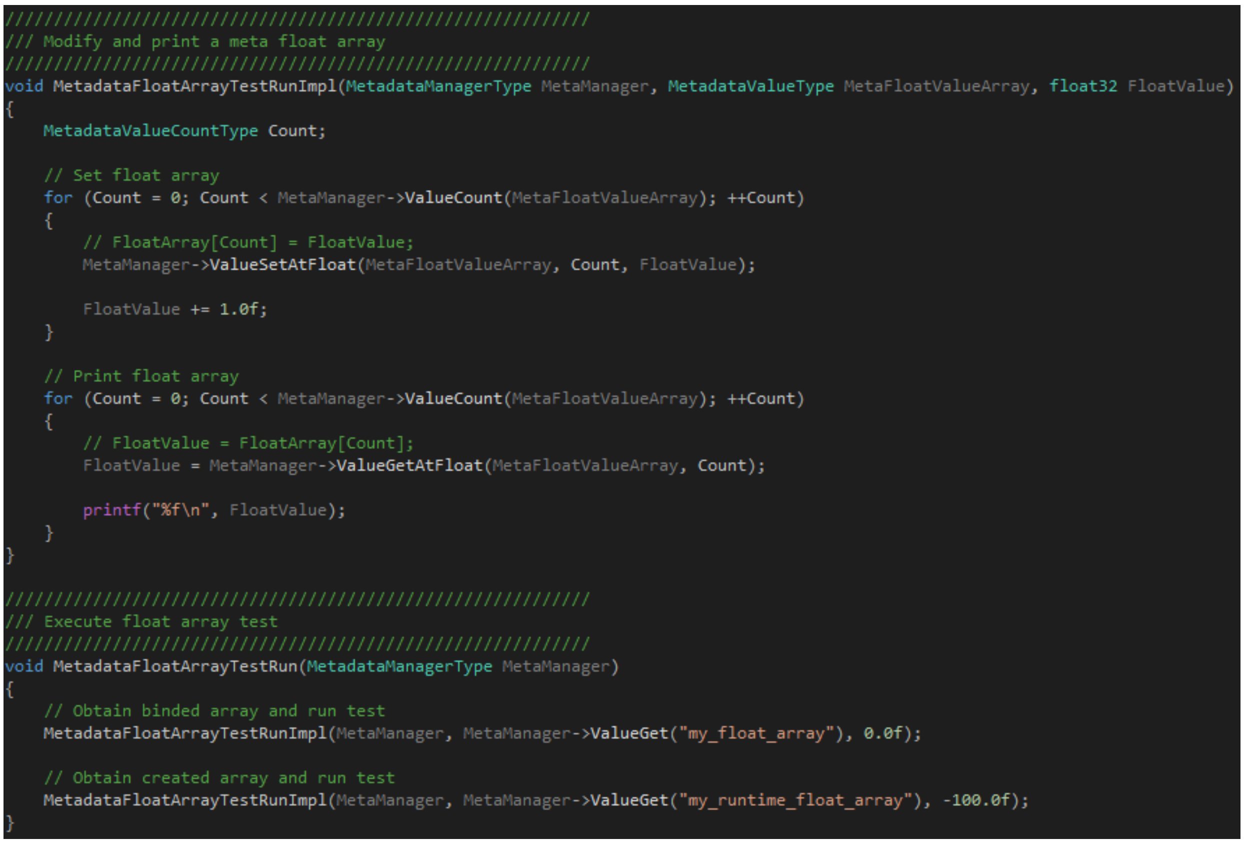
- MetadataValueCount(Scope, Value) : Gets the number of elements in memory of specified type the value is pointing to.

- MetadataValueGetData(Scope, Value) : Returns the data implementation which value is pointing to.

- MetadataValueSet(Scope, Value, Data) : Modifies the data implementation which value is pointing to.

- MetadataValueDestroy(Scope, Value) : Destroy a value from a scope.

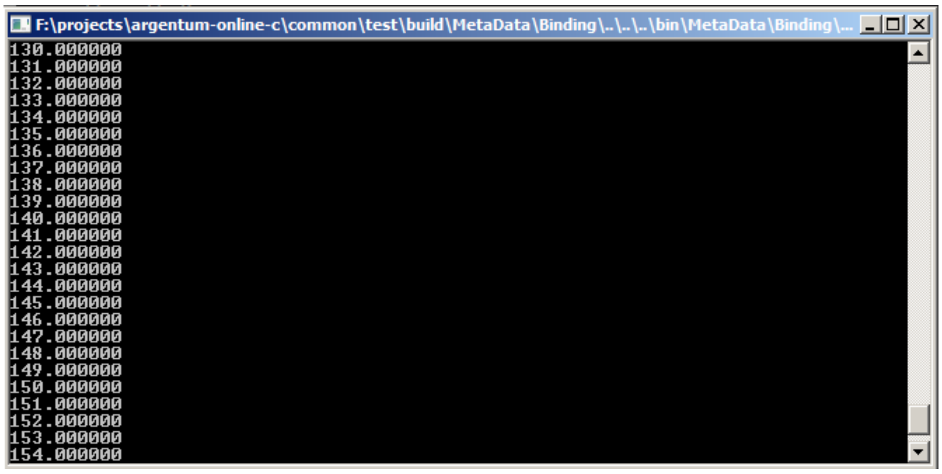
In the example provided in Figure 19, there are the two types of Values, the first one is bound to a real float array which is identified by the name my_float_array, and it is done without knowing the real array at compile-time, from the point where it is modified, the compiler does not know what kind of array is being modified. The second array, my_run-time_float_array, has been created during the execution and it is not bound to any variable.



```
////////////////////////////////////////////////////////////
/// Modify and print a meta float array
////////////////////////////////////////////////////////////
void MetadataFloatArrayTestRunImpl(MetadataManagerType MetaManager, MetadataValueType MetaFloatValueArray, float32 FloatValue)
{
    MetadataValueCountType Count;

    // Set float array
    for (Count = 0; Count < MetaManager->ValueCount(MetaFloatValueArray); ++Count)
    {
        // FloatArray[Count] = FloatValue;
        MetaManager->ValueSetAtFloat(MetaFloatValueArray, Count, FloatValue);

        FloatValue += 1.0f;
    }

    // Print float array
    for (Count = 0; Count < MetaManager->ValueCount(MetaFloatValueArray); ++Count)
    {
        // FloatValue = FloatArray[Count];
        FloatValue = MetaManager->ValueGetAtFloat(MetaFloatValueArray, Count);

        printf("%f\n", FloatValue);
    }
}

////////////////////////////////////////////////////////////
/// Execute float array test
////////////////////////////////////////////////////////////
void MetadataFloatArrayTestRun(MetadataManagerType MetaManager)
{
    // Obtain binded array and run test
    MetadataFloatArrayTestRunImpl(MetaManager, MetaManager->ValueGet("my_float_array"), 0.0f);

    // Obtain created array and run test
    MetadataFloatArrayTestRunImpl(MetaManager, MetaManager->ValueGet("my_runtime_float_array"), -100.0f);
}
```

Figure 19: Example of Value in arrays.

What this example does is to iterate through the arrays and gives values for each element of the array, begining from FloatValue, until FloatValue + MetaManager->ValueCount(...), which gives the number of elements of the array, in this case is 255. The execution result is shown in the Figure 20.


F:\projects\argentum-online-c\common\test\build\MetaData\Binding\..\..\..\bin\MetaData\Binding\...

```
130.000000
131.000000
132.000000
133.000000
134.000000
135.000000
136.000000
137.000000
138.000000
139.000000
140.000000
141.000000
142.000000
143.000000
144.000000
145.000000
146.000000
147.000000
148.000000
149.000000
150.000000
151.000000
152.000000
153.000000
154.000000
```

Figure 20: Result of Value test execution.

The most dependencies of these sub-module are related to Function, Parameter and Object, in order to provide a way to define the signatures of each entity, or generating on the fly lists of parameters. The other use of this sub-module is in order to define global variables and data, which can be accessed from the same Scope.

3. 2. 3. Parameters and functions

Functions provide a easy way to encapsulate signatures and function calls, allowing a mechanism called foreign function interfaces, which can call functions without knowing the declaration, at compilation level. This feature it is not commonly provided in most programming languages, and it is quite difficult to implement at C level, because it has not been designed to achieve this kind of behavior.

Calling a function, in C depends on the compiler, the platform and the architecture. The argument passing and the stack management is different between different compilers and architectures, so a way to implement this feature is to create an implementation of custom dynamic stack construction, for each compiler, platform and architecture.

This solution is the most used in libraries which implement this technique, like Dyncall[45] and FFCall[46] . Both libraries use the same principle, which basically implement a dynamic construction of the stack by going into a low level Assembly, using a signature can determine dynamically the foreign interface and then the stack is constructed and then the function is called, only knowing the memory address of it.

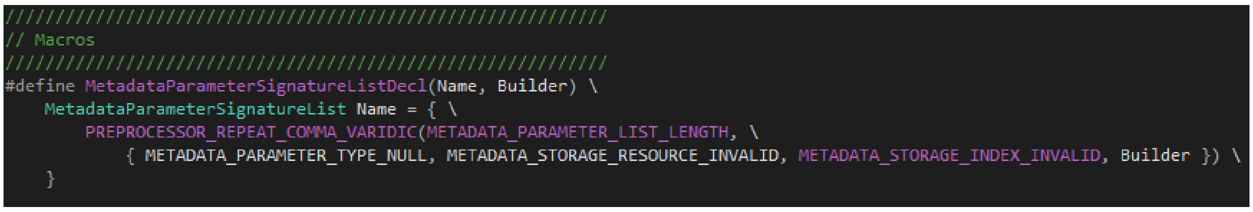
This implementation is the fastest possible, because it has the minimum overload for each function call and it is optimized at Assembly level, but the main problem of this implementation is that it has a strongly dependency on the compiler, platform and architecture. Implementing this solution implies that there must be a port for each platform, which is a difficult task to maintain and reduces the portability of the code. Prioritizing on portability over the performance, forces to avoid this solution and implement another one that instead of making dynamically the stack at run-time, this process is moved to the compile-time level.

This means that the parsing of the arguments when a function is called is done dynamically, but the function signature, and the foreign call is implemented at compile-time. This gives a powerful solution, because the stack generation can be done at compile-time, avoiding to do it during the execution. This technique is achieved by using the preprocessor meta-programming techniques described before.

To implement it, the solution resides on first, implement a generic function called Execute, which accepts first, a string that represents the name by which the function is indexed, and then accepts varidic arguments. This function parses the arguments and does the type promotion, in order to support varidic arguments generically among multiple compilers, platforms and architectures. When arguments are parsed, all of them are encapsulated into a pre-allocated parameter list assigned to the function, and built on the fly, and then this list is passed into a stub.

The stub is generated during compile-time, and it must be in the same scope which resides the foreign function, in order to be visible by the compiler, inside the stub. Here the stub is automatically created, it unroll the arguments and calls the function. This mechanism adds an overload for each call, because it has to call first to the auto-generated stub, and then to the real function. But if the code gets optimized in the compilation step, the compiler is clever enough to convert the stub into a direct call to the real function, so it can be minimized.

Parameters can be passed as value or reference, the difference is that they are copied to the destination scope and storage or not. For declaring the parameter list there is a macro provided, shown in Figure 21, which at compile-time initializes the parameters associated to a function.



```
////////////////////////////////////////////////////////////
// Macros
////////////////////////////////////////////////////////////
#define MetadataParameterSignatureListDecl(Name, Builder) \
    MetadataParameterSignatureList Name = { \
        PREPROCESSOR_REPEAT_COMMA_VARIDIC(METADATA_PARAMETER_LIST_LENGTH, \
            { METADATA_PARAMETER_TYPE_NULL, METADATA_STORAGE_RESOURCE_INVALID, METADATA_STORAGE_INDEX_INVALID, Builder }) \
    }
```

Figure 21: Macro for declaring parameter list signatures for functions.

The declaration uses the builder pattern, in order to simplify the creation of the arguments after the declaration, this means, to specify the signature, resource type, bypass method and when calling, the referenced resource. The macro is unrolled into an array initialization that declares a default parameter list.

In order to serve the function sub-module, apart from meta-programming techniques with parameters, there is a set of functionalities that parameter sub-module provides:

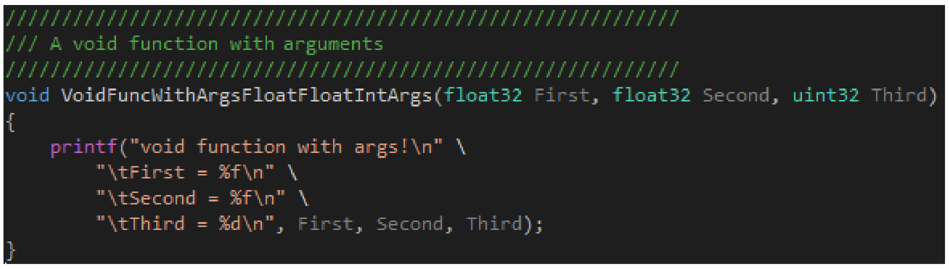
- MetadataParameterSignature(ParameterSignature, InfoType, ResourceType, Signature) : Set a parameter signature type, used to implement the builder pattern.
- MetadataParameterListCreate(Scope, Count) : Create a parameter list.
- MetadataParameterListCopy(Scope, ParameterList) : Copy a parameter list.
- MetadataParameterListSize(Scope, ParameterList) : Get size of parameter list.
- MetadataParameterListSignatureInfoType(Scope, ParameterList, Parameter) : Get the info type of specified parameter.
- MetadataParameterListSignatureResourceType(Scope, ParameterList, Parameter) : Get the resource type of specified parameter.
- MetadataParameterListGetParameterIndex(Scope, ParameterList, Parameter) : Get the index which parameter refers to.
- MetadataParameterListDefine(Scope, Signature, Parameter, InfoType, ResourceType, Index) : Define a parameter as a signature.
- MetadataParameterListUndefine(Scope, ParameterList, Parameter) : Destroy a parameter, slot is not removed from list.
- MetadataParameterListBuildSignature(Scope, Count, ParameterSignatureList) : Creates a signature.
- MetadataParameterListGenerate(Scope, Signature, Arguments) : Copy a parameter list filled by given parameters.
- MetadataParameterListDestroy(Scope, ParameterList) : Destroy a parameter list.

To provide the foreign function interfaces and dynamic calls there are the following functionalities provided in the Function sub-module:

- MetadataFunctionCreate(Scope, Signature) : Create a function.
- MetadataFunctionBind(Scope, Signature, FunctionPtr) :Bind a function.
- MetadataFunctionCopy(Scope, Function) : Copy a function.
- MetadataFunctionGetSignature(Scope, Function) : Get signature of a function.
- MetadataFunctionExecute(Scope, Function, ParameterList) :Execute a function dynamically.
- MetadataFunctionReference(Scope, Function) :Reference a function.

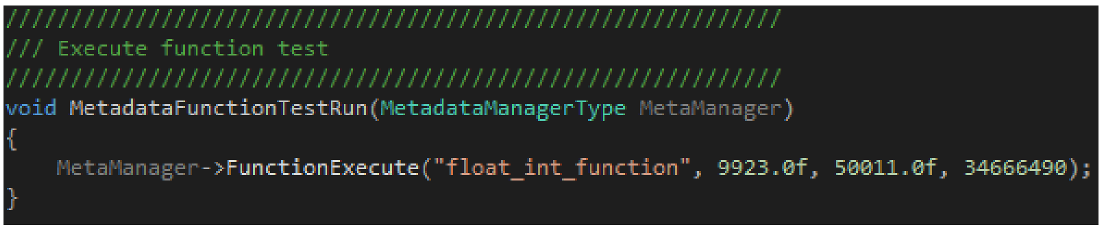
- MetadataFunctionIsReferenced(Scope, Function) : Check if function is referenced.
- MetadataFunctionDereference(Scope, Function) : Dereference a function.
- MetadataFunctionDestroy(Scope, Function) : Destroy a function.

The result of the module is the ability of executing foreign function calls, an example is given in the Figure 23, doing the call, and in the Figure 24, where the resulting call is executed. The interface for registration is not shown because it needs to be improved, it still is not easy enough for the end user, and must be simplified with the Preprocessor facilities.



```
////////////////////////////////////////////////////////////
/// A void function with arguments
////////////////////////////////////////////////////////////
void VoidFuncWithArgsFloatFloatIntArgs(float32 First, float32 Second, uint32 Third)
{
    printf("void function with args!\n" \
        "\tFirst = %f\n" \
        "\tSecond = %f\n" \
        "\tThird = %d\n", First, Second, Third);
}
```

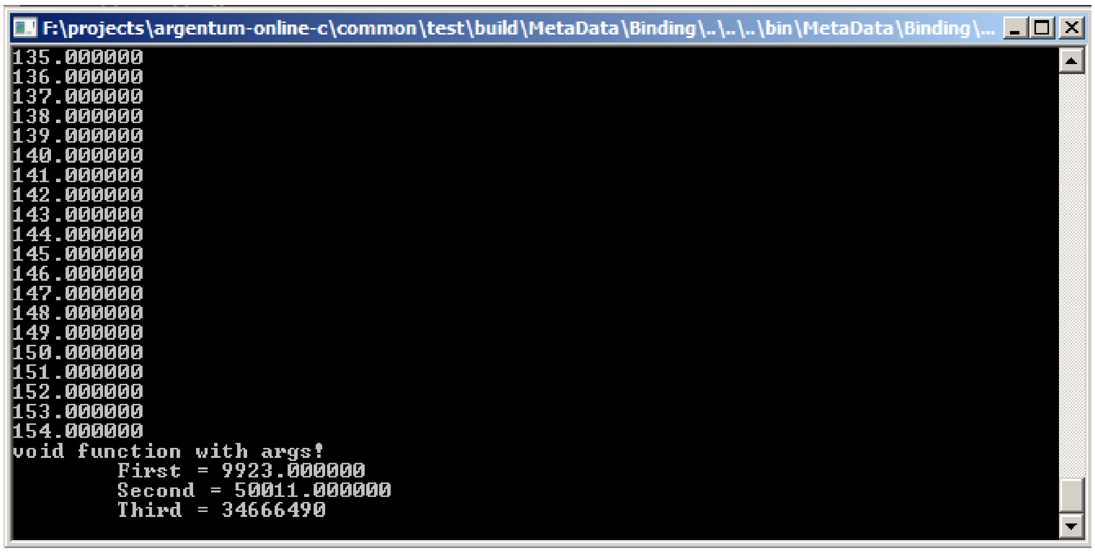
Figure 22: Foreign test function which is wrapped by the Metadata system.



```
////////////////////////////////////////////////////////////
/// Execute function test
////////////////////////////////////////////////////////////
void MetadataFunctionTestRun(MetadataManagerType MetaManager)
{
    MetaManager->FunctionExecute("float_int_function", 9923.0f, 50011.0f, 34666490);
}
```

Figure 23: Foreign function call, with VoidFunctionWithArgsFloatFloatIntArgs as a target.

As it can be seen in the example, the use of the system is really simplified. In the Figure 23, the call is done without having the target function visible for the compiler and without knowing the real signature. The metadata signature has been registered previously, and the mechanism which generates the stub and makes the call becomes transparent for the user. The result can be seen in the Figure 24, and it is the same as if the function was executed directly.


F:\projects\argentum-online-c\common\test\build\MetaData\Binding\..\..\..\bin\MetaData\Binding\...

```
135.000000
136.000000
137.000000
138.000000
139.000000
140.000000
141.000000
142.000000
143.000000
144.000000
145.000000
146.000000
147.000000
148.000000
149.000000
150.000000
151.000000
152.000000
153.000000
154.000000
void function with args!
        First = 9923.000000
        Second = 50011.000000
        Third = 34666490
```

Figure 24: Result of the execution of the foreign function call for the VoidFunctionWithArgsFloatFloatIntArgs function.

3. 2. 4. Objects and methods

An Object is a generic wrapper for structures and classes; it allows to describe a data block with different variables inside and methods. Methods are equivalent to Functions, but they pass always the first argument as the class or the structure which contains the method, commonly called this in most programming languages.

In order to describe the signature of neither the structure nor the class, which both are encapsulated by the Object entity, we cannot rely over the data offsets, because it depends on the platform, the size of the variables and the alignment. So the way to make it generic is to store directly the pointers to the data, as done with Value and Function, instead of using the offset. Another option which makes available the use of offsets is to enumerate the internal data and set local offsets for each variable, and then use an static mapping for define the positions. This is the best option, because by this way signatures always remain equal without platform dependencies.

The methods can be defined as pointers to functions, in the case of structures, and normal methods, in the case of classes. The technique used to describe it is the same as with the variables, but instead of wrapping it with a Value, all of them are wrapped with a special case of Functions, which pass the instance of the Object as the first parameter.

3. 2. 5. References

A Reference is a generic wrapper for any metadata type. It can point to a Value, other Reference, a Function or an Object. It hasn't got a primitive type as a value; the definition of itself is given by the type of metadata object.

By using a reference is possible to obtain and modify information of metadata is referring to, it is also possible to copy the reference easily but it will not affect the element which is pointing and it also will not be notified, it only will increment the reference counter of the Reference itself.

A basic set of available operations can be performed with References are:

- MetadataReferenceCreate(Scope, Name, Index, Type) : Creates a reference specifying the scope where it lives, the name to reference it, the generic index which is pointing to, and the type of the index (Value, Reference, Function, Object).
- MetadataReferenceGet(Scope, Reference) : Gets the metadata object which reference is pointing to.
- MetadataReferenceSet(Scope, Reference, Index) : Modifies the metadata object which reference is pointing to.
- MetadataReferenceTypeOf(Scope, Reference) : Gets the type of metadata object is being pointed by the reference (Value, Reference, Function, Object).
- MetadataReferenceDestroy(Scope, Reference) : Destroy a reference from a scope.

A reference is commonly used when you want to store a metadata type, or a list which you do not know exactly what type of entity is at compile-time, and in addition you want to explicitly mark them as referenced, because the handles representing the entities are references by itself, like in the Java language.

3. 2. 6. Storage

Storage is a memory manager which holds the instances of all metadata, no matter if it is an instance of a signature or a signature itself, which can be considered as an abstract metadata, sorted by type for efficient memory access.

It also has a garbage collector and gives the possibility to resize the pool if need, to increment the size if it is need and to reduce it when do not need in order to avoid memory overload. When the pool does not need to be defragmented, normally the cost of inserting and removing an element is O(1). If the pool is fragmented in lot of pieces, the cost of inserting goes to O(n) in the worst case, and the removing cost remains the same. The cost of defragment all pool it is always O(n).

In addition, it is possible to serialize the whole storage to hold the full metadata status of the application, and preserve the global state or send it over a stream. On the other side it can be reconstructed, so the metadata information of the application can be cloned, or interpreted later. This can give fault-tolerant techniques, in order to preserve information, and it also allows to write sets of signatures into files, in order to populate them and manage them in different ways, for example, making a look-up tool to identify them.

The Storage has been implemented using the singleton pattern, in order to retrieve the global instance of the storage for the application, but it is possible to create new storage dynamically, or reference them, in the case of we want to separate data, or for example, we want to access the metadata of an external plug-in which has it is own internal global storage.

The basic set of functionalities it is provided is in the following list:

- MetadataStorageGlobalInstance() : Get global storage instance.
- MetadataStorageCreate() : Create a storage manager, returning the instance of it.
- MetadataStorageAllocateResource(Storage, ResourceType, Size) : Allocate a resource in the storage manager, returning the handle associated to it.
- MetadataStorageGetResource(Storage, ResourceType, Index) : Get a resource raw instance from the storage manager.
- MetadataStorageDeallocateResource(Storage, ResourceType, Index) : Deallocate a resource in the storage manager.
- MetadataStorageClear(Storage) : Clear a storage manager, removes all data it contains.
- MetadataStorageDestroy(Storage) : Destroy a storage manager.

In order simplify the code; there are macro wrappers for the resource related functions. In the case of resource allocation:

- MetadataStorageAllocateValue(Storage, Size).
- MetadataStorageAllocateReference(Storage, Size).
- MetadataStorageAllocateParameterList(Storage, Size).
- MetadataStorageAllocateFunction(Storage, Size).
- MetadataStorageAllocateObject(Storage, Size).

In the case of resource acquirement:

- MetadataStorageGetValue(Storage, Index).
- MetadataStorageGetReference(Storage, Index).
- MetadataStorageGetParameterList(Storage, Index).
- MetadataStorageGetFunction(Storage, Index).
- MetadataStorageGetObject(Storage, Index).

In the case of resource deallocation:

- MetadataStorageDeallocateValue(Storage, Index).
- MetadataStorageDeallocateReference(Storage, Index).
- MetadataStorageDeallocateParameterList(Storage, Index).
- MetadataStorageDeallocateFunction(Storage, Index).
- MetadataStorageDeallocateObject(Storage, Index).

The index which reference the internal entities of the storage are always static. It remains equal meanwhile the resource it is not reallocated or deleted. It has the same behavior that malloc and free functions have with pointers, which are allocated dynamically but it always have the same id during its life-time. By providing static references it is possible to reallocate the whole pool without affecting the original ids of the resources, which gives a flexible mechanism that makes transparent the memory management.

Multiple scopes can reference to the same storage, each scope it is just a map to the resource references, that are implemented by the previous mechanism of the static ids, which is the same type for each entities, but logically represent different kind of structures. This means that Value, Reference, Parameter, Function and Object ids are represented by the same type at programming language level, but logically represent different types of entities.

3.2.7. Scope

Scope is a namespace where metadata lives. It gives visibility information about metadata and it is linked to a storage manager. The Scope has been designed in a similar way as Storage; it has a singleton which retrieves the global scope instance for the application.

Scope works as a map, which converts a string keyword and a type into a index, pointing to a desired Storage which is wrapped internally by the system. By this way it is possible to have a collection of metadata entities, indexed by name and type, instead of by a number, simplifying the use of them.

The list of methods which the sub-module provides is:

- MetadataScopeGlobalInstance() : Get global scope instance.
- MetadataScopeCreate(Name, Storage) : Create a new scope linked to an storage, returning the id asocciated to it.
- MetadataScopeGetName(Scope) : Retreive name asociated to a scope.
- MetadataScopeSetName(Scope, Name) : Set name associated to a scope.
- MetadataScopeGetStorage(Scope) : Retrieve storage associated to a scope.
- MetadataScopeSetStorage(Scope, Storage) : Set storage asociated to a scope.
- MetadataScopeKeywordCreate(Scope, Keyword, Index, ResourceType) : Create a keyword in the system and link it to storage.
- MetadataScopeKeywordGetIndex(Scope, Keyword) : Get storage index of the keyword.
- MetadataScopeKeywordGetType(Scope, Keyword) : Get storage type of the keyword.

- MetadataScopeKeywordGetName(Scope, Index, Type) : Get name of keyword by storage index and type.
- MetadataScopeKeywordDestroy(Scope, Keyword) : Destroy a keyword in the system and link it to storage.
- MetadataScopeDestroy(Scope) : Destroy a scope instance.

The scope can be used for indexing any type of resource, because all of them have a generic handle which describes it, so it can be treated in the same way. When a signature is registered into a scope, it works as a typical namespace, in the same way as C++. When an instance of a real resource is registered, it works as a dynamic scope which can handle it.

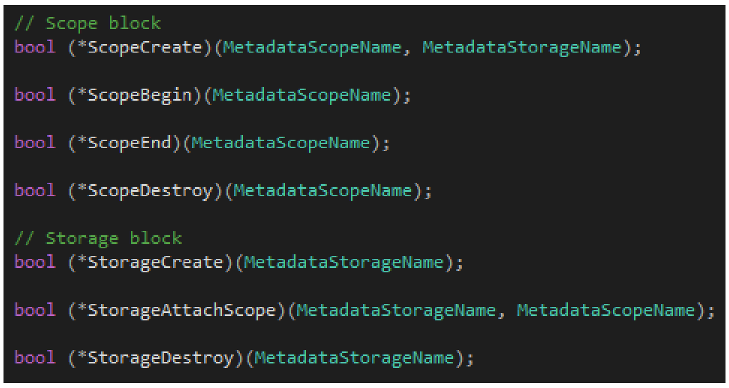
A scope can be contained by another, so it is possible to create a hierarchical structure, and manage better the access permissions to the resources. Scopes can also be shared between different tiers, so it is possible to access them from different points, for example, the game engine and a plug-in can access to the same scope in order to share information.

3. 2. 8. Metadata manager

The Metadata Manager is the top module of the system. It has been implemented as a singleton, in order to retrieve the global instance, but it also can be referenced or created dynamically, in order to manage metadata from other applications or plug-ins.

The manager is implemented as an structure, by using pointers to functions, encapsulating the functionalities of all modules. This is done by this way because of when plug-ins are created, or any kind of system, an object must be passed, as a parameter of a function, which is the entry point of the protocol. By this way is possible to access the members from an structure and use the whole system from the same point.

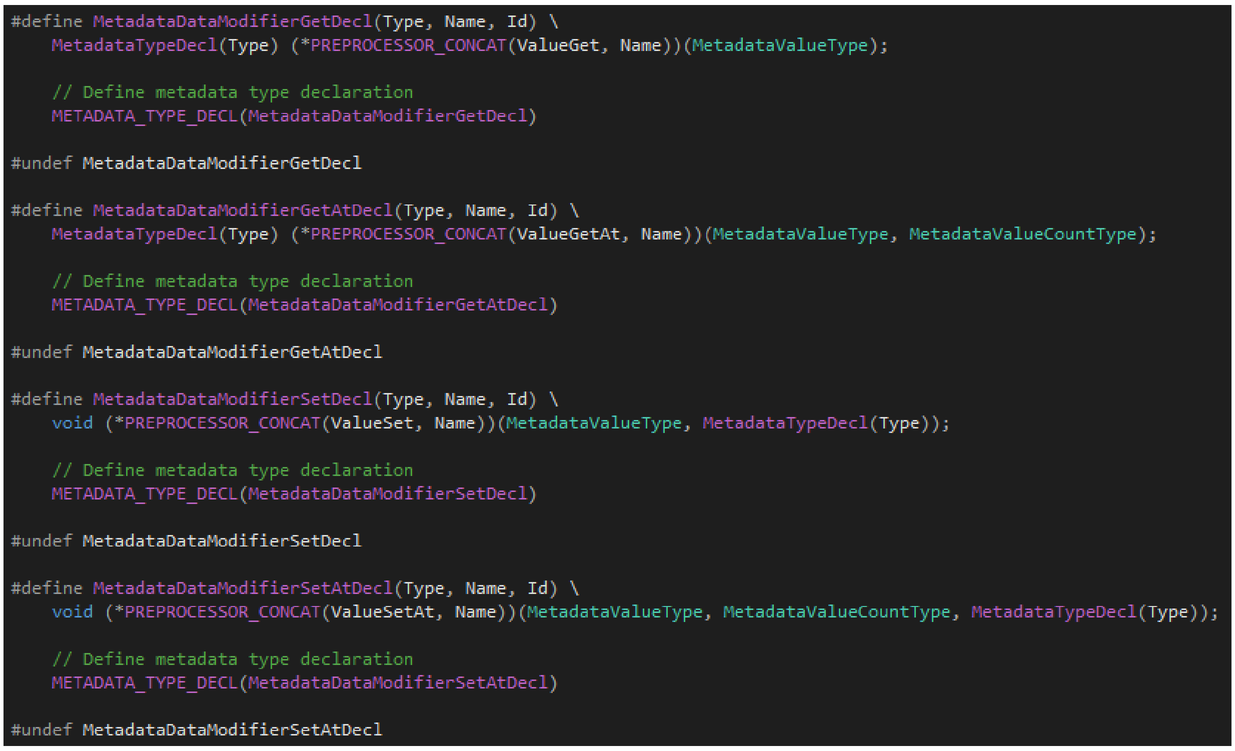
It also has been implemented by using the PImpl idiom, in order to hide and simplify the code. For example, in the all sub-modules, scope is always passed to any function, in order to identify where the entity lives. With the manager it is not need because by default, it is set to the global scope, and then it has functions in order to enable and disable other scopes, as seen in Figure 25.



```
// Scope block
bool (*ScopeCreate)(MetadataScopeName, MetadataStorageName);

bool (*ScopeBegin)(MetadataScopeName);

bool (*ScopeEnd)(MetadataScopeName);

bool (*ScopeDestroy)(MetadataScopeName);

// Storage block
bool (*StorageCreate)(MetadataStorageName);

bool (*StorageAttachScope)(MetadataStorageName, MetadataScopeName);

bool (*StorageDestroy)(MetadataStorageName);
```

Figure 25: Scope and Storage management of MetaManager.

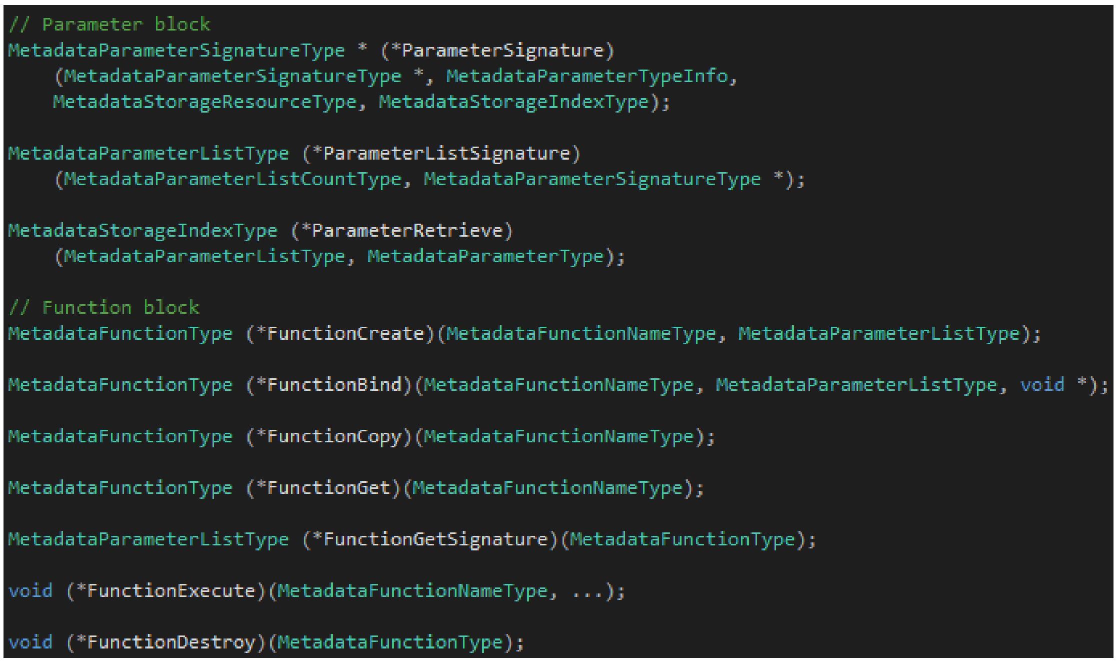
Other facilities are implemented, in the case of Values, by means of X-Macros[47] , in order to give a similar behavior as C++ templates, and provide consulters and modifiers for all types of Values, including if they are built-ins or arrays.



```
#define MetadataDataModifierGetDecl(Type, Name, Id) \
    MetadataTypeDecl(Type) (*PREPROCESSOR_CONCAT(ValueGet, Name))(MetadataValueType);

    // Define metadata type declaration
    METADATA_TYPE_DECL(MetadataDataModifierGetDecl)

#undef MetadataDataModifierGetDecl

#define MetadataDataModifierGetAtDecl(Type, Name, Id) \
    MetadataTypeDecl(Type) (*PREPROCESSOR_CONCAT(ValueGetAt, Name))(MetadataValueType, MetadataValueCountType);

    // Define metadata type declaration
    METADATA_TYPE_DECL(MetadataDataModifierGetAtDecl)

#undef MetadataDataModifierGetAtDecl

#define MetadataDataModifierSetDecl(Type, Name, Id) \
    void (*PREPROCESSOR_CONCAT(ValueSet, Name))(MetadataValueType, MetadataTypeDecl(Type));

    // Define metadata type declaration
    METADATA_TYPE_DECL(MetadataDataModifierSetDecl)

#undef MetadataDataModifierSetDecl

#define MetadataDataModifierSetAtDecl(Type, Name, Id) \
    void (*PREPROCESSOR_CONCAT(ValueSetAt, Name))(MetadataValueType, MetadataValueCountType, MetadataTypeDecl(Type));

    // Define metadata type declaration
    METADATA_TYPE_DECL(MetadataDataModifierSetAtDecl)

#undef MetadataDataModifierSetAtDecl
```

Figure 26: Consulters and modifiers of Value in the MetaManager.

Another example of the simplification it is achieved can be seen in the parameter and function methods, which provide an easy way to generate signatures, call foreign functions and work easily without adding bloat code when using it, a priority of the framework.

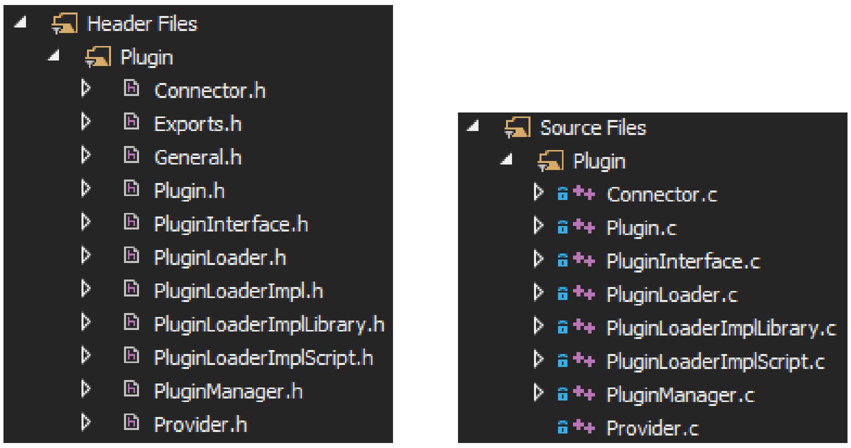


```
// Parameter block
MetadataParameterSignatureType * (*ParameterSignature)
    (MetadataParameterSignatureType *, MetadataParameterTypeInfo,
    MetadataStorageResourceType, MetadataStorageIndexType);

MetadataParameterListType (*ParameterListSignature)
    (MetadataParameterListCountType, MetadataParameterSignatureType *);

MetadataStorageIndexType (*ParameterRetrieve)
    (MetadataParameterListType, MetadataParameterType);

// Function block
MetadataFunctionType (*FunctionCreate)(MetadataFunctionNameType, MetadataParameterListType);

MetadataFunctionType (*FunctionBind)(MetadataFunctionNameType, MetadataParameterListType, void *);

MetadataFunctionType (*FunctionCopy)(MetadataFunctionNameType);

MetadataFunctionType (*FunctionGet)(MetadataFunctionNameType);

MetadataParameterListType (*FunctionGetSignature)(MetadataFunctionType);

void (*FunctionExecute)(MetadataFunctionNameType, ...);

void (*FunctionDestroy)(MetadataFunctionType);
```

Figure 27: Function and Parameter blocks in the MetaManager.

3. 3. Plug-in module

Plug-in is the module which creates the infrastructure that can load and unload components from different sources in run-time, and inject them into the existing architecture. It can provide services and interconnect in a generic way the code forge of a game engine or a game. It is based on a factory which maintains the plug-ins, and provides a flexible way to define component dependencies.



```
Header Files
  Plugin
    Connector.h
    Exports.h
    General.h
    Plugin.h
    PluginInterface.h
    PluginLoader.h
    PluginLoaderImpl.h
    PluginLoaderImplLibrary.h
    PluginLoaderImplScript.h
    PluginManager.h
    Provider.h
```

```
Source Files
  Plugin
    Connector.c
    Plugin.c
    PluginInterface.c
    PluginLoader.c
    PluginLoaderImplLibrary.c
    PluginLoaderImplScript.c
    PluginManager.c
    Provider.c
```

Figure 28: Plug-in module overview.

Plug-in architecture encapsulates all modules named before. With Metadata system is possible to define plug-in interfaces, and Preprocessor module gives the ability to automate the interface declaration and metadata registration. Metadata system also relies on Memory module, to manage efficiently memory resources, and DataType module, for the abstract data types.

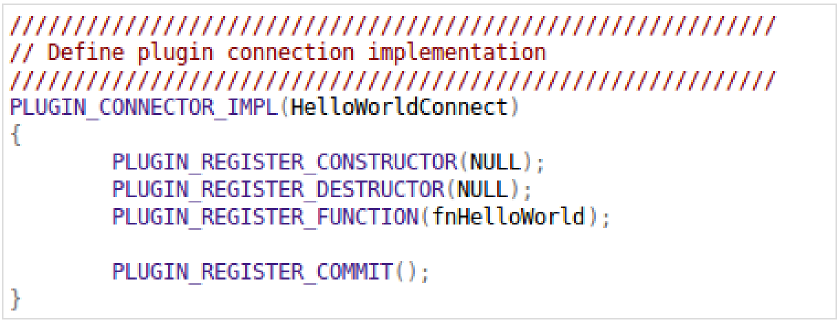
Then all of them are integrated by Plug-in module that integrates a loader which can implement dynamic load libraries, run-time environments such as script engines, or remote loaders, using any existing middleware as for example, CORBA. At the end, Plug-in module is able to load code independent from any platform, language, location, without knowing a priory anything about the implementations, and provides a two-way communication to apply plug-ins to the game engine and also, to provide services from engine to the plug-ins.

For any architecture which has a large code forge based on components, encapsulated by modules, separated by layers, like in game development, this is one of the best philosophies to adopt, from technical point of view, because it reduces the cost and effort to maintain and develop it.

3. 3. 1. Plug-in interface and connector

By using Metadata and Preprocessor modules, it is possible to encapsulate, describe the methods and generate plug-in stubs, by this way a plug-in can ask for predefined functionalities to run-time loaded plug-ins, and if the interface fits with implementation already loaded, the stub is attached to the plug-in implementation by means of Metadata module and then it can be used.

Before plug-in is loaded, the host application has the calls to stub functionalities without knowing already if that implementation is going to be provided. Plug-in management allows to easy check for concordance between the stub and the implementation provided, and then makes the connections between stubs and implementations. It is also fault tolerant, allowing to avoid using some functionality if it is not attached to the host application, throwing an exception that can be treated.



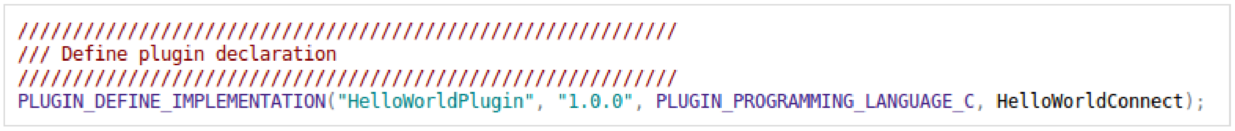
```
////////////////////////////////////////////////////////////////
// Define plugin connection implementation
////////////////////////////////////////////////////////////////
PLUGIN_CONNECTOR_IMPL(HelloWorldConnect)
{
        PLUGIN_REGISTER_CONSTRUCTOR(NULL);
        PLUGIN_REGISTER_DESTRUCTOR(NULL);
        PLUGIN_REGISTER_FUNCTION(fnHelloWorld);

        PLUGIN_REGISTER_COMMIT();
}
```

Figure 29: Macro based interface of a compiled 'Hello World' plug-in example.

As Figure 29 shows, how a compiled plug-in interface definition must be implemented. The definition of the interface is wrapped by macros, in order to simplify the programming task. In this case the macros are implemented by the Preprocessor module which allows to auto-generating the connector, a predefined entry point of all plug-ins, and generate stubs, with the metadata information, in order to describe the internal properties and methods.

In this case there is only a void function without arguments, which prints text in the standard output. The name of the function is needed in order to store the pointer address of the function. The macros stringify that name, in order to provide a string tag that references the function. By this way that function becomes registered and it can be used from the host application.

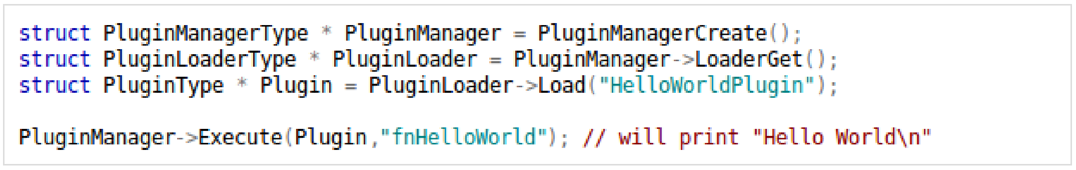
There is also need to define a line in the header, as shown in Figure 30, which is the ones that populates and exports the entry point of the plug-in, the connector. There it will be defined general properties of the plug-in, as version or language in which it was developed. This will be registered automatically into the system. This line will allow giving to the system the entry point, and executing the registration by passing the MetaManager object to the plug-in, which will register inside that manager instance, the methods and properties.



```
//////////////////////////////////////////////////////////////
/// Define plugin declaration
//////////////////////////////////////////////////////////////
PLUGIN_DEFINE_IMPLEMENTATION("HelloWorldPlugin", "1.0.0", PLUGIN_PROGRAMMING_LANGUAGE_C, HelloWorldConnect);
```

Figure 30: Plug-in declaration for providing an entry point for allow loading into the PluginManager.

As it can be seen the front-end of the Plug-in Framework provides an easy interface, which is able to be implemented with macros. This is the most important objective, because programmers should learn it in a fast way, in order to develop and inject code faster than without this system. Another priority is that all plug-ins fit the same protocol, so all of them can be treated in the same way. Internally the protocol is implemented by the Metadata system, and by this way code can be encapsulated, discovered and loaded at run-time. An example usage from the host application can be seen in Figure 31.



```
struct PluginManagerType * PluginManager = PluginManagerCreate();
struct PluginLoaderType * PluginLoader = PluginManager->LoaderGet();
struct PluginType * Plugin = PluginLoader->Load("HelloWorldPlugin");

PluginManager->Execute(Plugin,"fnHelloWorld"); // will print "Hello World\n"
```

Figure 31: Example usage of ‘Hello World’ plug-in from the host application.

3. 3. 2. Plug-in management

The plug-in management is opaque for programmers. Plug-ins are self managed and programmer only have to choose the configuration and path where plug-ins are located to load it. When plug-ins are not need any more and there is no component referencing to them, they are automatically unloaded and resources are released.

Plug-in system is also able to determine integrity between plug-ins, check towards backward compatibility, and fault tolerant, through an exception handler mechanism, which automates and simplifies the use of plug-ins.

Bindings, metadata and type-safety are also managed automatically, so definitions of stubs or implementations must be implicitly safe, or otherwise it will throw a run-time error during execution or loading.

3. 3. 3. Plug-in loading

As shown in the Figure 5 of the Design section, the plug-in loader is described by a generic interface, which can load any kind of plug-in, depending of the implementation derived from the loader. This means that it can support loading for multiple kinds of plug-ins, without taking into account the technology or the language used. Each implementation of the loader is provided by the PImpl idiom, in order to hide the underlying technology.

3.3.3.1. Dynamically loaded libraries

A technique used to load code packages, known as libraries, at run-time typically is what are used by SO. Most of SO provide techniques to load code, attach it to a process and execute it, at run-time.

An example of that is OpenGL[48] library and the well known extensions. OpenGL provides an API reference, for different versions of the library and underlying hardware. The API defines them but most of times your PC the majority are not available, just the common ones.

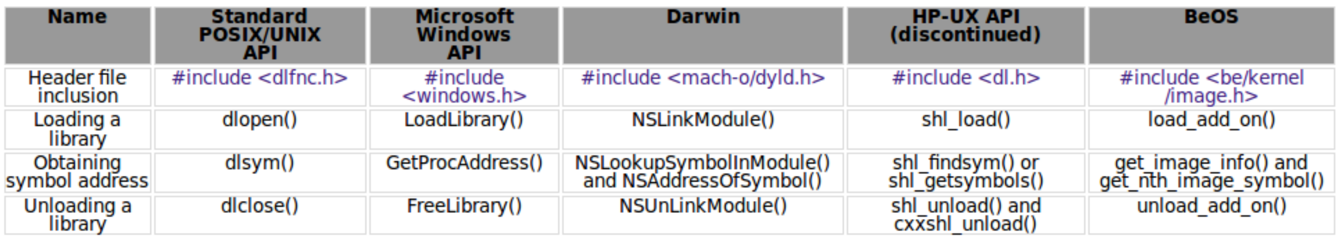
At run-time any application that needs OpenGL library can get information about it, and determine what extensions are available or not, and change the behavior to adapt it to the provided functionalities.

When the library is not need anymore then it is possible to unload it, and to free resources used. This technique depends on the OS, so everyone gives different methods to work with libraries, although at the end all of them solve the same problem. So for implementing it is need to do a version for each one. These methods are implemented in the sub-module System which provides full abstraction from the OS.

The interface of Library module has been designed with PImpl pattern, which makes opaque the real implementation used, so it is easy to extend the system or make strong changes, without modifying the public interface. The API provided is the following:

- LibraryExtension() : Returns the extension which is supported, depends on the OS.
- LibraryLoad(Name, Flags): Loads a library by name. It does not include the extension, it is applied internally to abstract from OS. Flags are used to determine properties about loading. Returns a handle identifying library.
- LibraryUnload(Library) : Unloads a library and frees related resources.
- LibrarySymbol(Library, Name, Address) : Loads a symbol from the library by name. Return true if successful and outputs the process address pointing to specified symbol, that can be after casted to any type and used or called.

The interface is simple, in order to be easy to use. Internally, it begins more complex. For each implementation depending on the OS there is a LibraryImpl module that wraps it, and it does the work related to the API of the OS. The methods used depending on the OS are:



Name	Standard POSIX/UNIX API	Microsoft Windows API	Darwin	HP-UX API (discontinued)	BeOS
Header file inclusion	#include <dlfnc.h>	#include <windows.h>	#include <mach-o/dyld.h>	#include <dl.h>	#include <be/kernel /image.h>
Loading a library	dlopen()	LoadLibrary()	NSLinkModule()	shl_load()	load_add_on()
Obtaining symbol address	dlsym()	GetProcAddress()	NSLookupSymbolInModule() and NSAddressOfSymbol()	shl_findsym() or shl_getsymbols()	get_image_info() and get_nth_image_symbol()
Unloading a library	dlclose()	FreeLibrary()	NSUnLinkModule()	shl_unload() and cxxshl_unload()	unload_add_on()

Table 2: API provided by multiple Operative Systems in order to dynamically load libraries, wrapped by the PluginLoaderLibraryImpl.

These functionalities are covered and wrapped by the system, so developers does not need to know about it, the system is completely decoupled from OS. The PluginLoaderLibrary implementation is able to load plug-ins embedded into libraries from any kind of OS, the implementation is defined by the compilation process, which detects in what OS is being compiled and then switches to the correct implementation.

3.3.3.2. Scripts and run-time environments

Another technique widely used too is run-time environments or also called virtual machines. These are kind of engines that can load scripts at run-time into memory, compact it and execute it efficiently.

There are multiple options when deciding what script is going to wrap the system, in addition, to have multiple script language handling, and for each language, different engines can be used. Some of that languages that are easy to integrate with the core are:

- **JavaScript**: Mozilla Firefox Spider Monkey JS Engine or Google V8 JS Engine.
- **Python**: Python C API.
- **LUA**: Lua C API.

Any of these script languages can be a good option for integrating, developing and deploying plug-ins. The main differences between scripts and dynamic loaded libraries are that these can be modified at run-time, and reloaded. The language used is interpreted instead of to be compiled, and it is easier to code, perfect to allow fast plug-in development.

In order to implement a new loader based on a run-time environment, the only part should be migrated is the Plug-in and Metadata modules. The other game engine infrastructure will be offered by means of these modules. Another particularity that must be taken into account is the stub generation. With scripting languages this is not a problem, because most of them allow meta-programming techniques. So in order to simplify the implementation of the scripts, another module should be designed providing an easy abstraction and automation of the stub generation process.

The differences between other systems are notable, in this approach, the API should be the minimal possible, and we only have to migrate the part directly dependent to the plug-ins, not the whole game engine API. It becomes offered as services, instead of porting each module of the game engine. This provides better interoperability and makes easy the insertion of new loaders for different languages.

4. Evaluation

In order to evaluate the whole code, there are designed a battery of tests, for checking by separately each module implemented in the framework. From preprocessor meta-programming techniques, to metadata management and registration, and a minimal plug-in tutorial for understanding the use of the system.

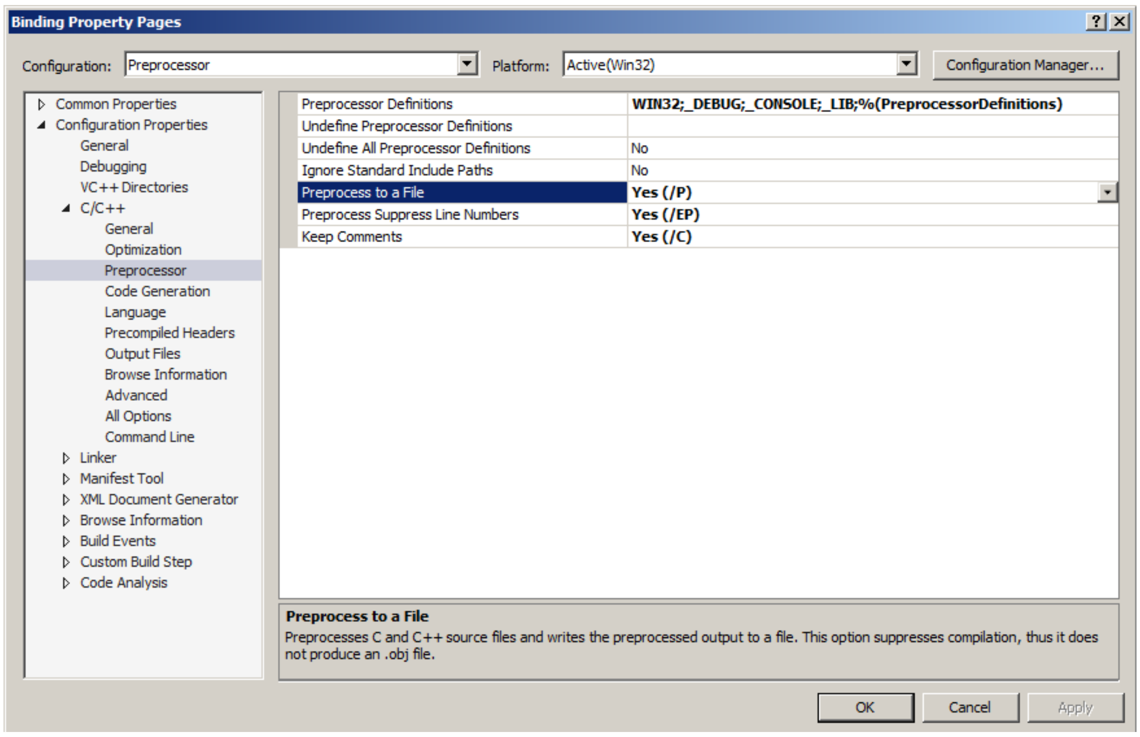
For evaluating the plug-in framework has been designed an example host application, which loads the simplest plug-in, introducing the system as a simple tutorial, in order to use it in the documentation. Both projects has been implemented in C, because the only available loader is the PluginLoaderLibrary ones.

The example is a proof of concept, demonstrating the capabilities of the framework, step by step, and how it is the front-end of the application, an important aspect that must be take into account because this system will be used by many developers, and it should be easy to learn and must simplify the work.

4. 1. Preprocessor meta-programming tests

The Preprocessor module has been tested in Microsoft Visual Studio 2013 with the MSVC compiler, and with GCC, over Windows 7 and Linux Lubuntu respectively. As the functionality only can be tested after preprocessor step, it has need to configure in a different way the compilation in order to obtain code after preprocessing step, instead of compiling directly the tests.

For debugging the preprocessor, in the first case, for the Visual Studio project has been added a new build configuration called Preprocessor, with the configuration shown in over the preprocessor step



Configuration: Preprocessor   Platform: Active(Win32)

Preprocessor Definitions	WIN32;_DEBUG;_CONSOLE;_LIB;%(PreprocessorDefinitions)
Undefine Preprocessor Definitions	
Undefine All Preprocessor Definitions	No
Ignore Standard Include Paths	No
Preprocess to a File	Yes (/P)
Preprocess Suppress Line Numbers	Yes (/EP)
Keep Comments	Yes (/C)

Preprocess to a File
Preprocesses C and C++ source files and writes the preprocessed output to a file. This option suppresses compilation, thus it does not produce an .obj file.

Figure 32: Preprocessor configuration of Visual Studio project in order to test the preprocessor meta-programming techniques.

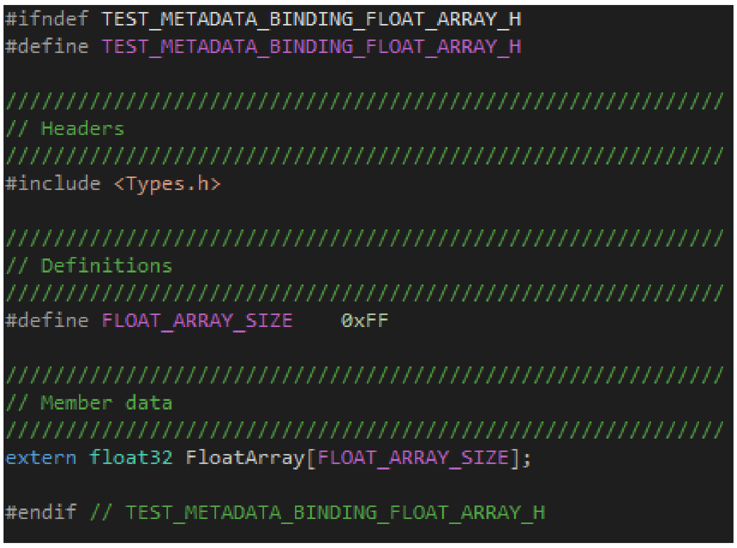
For the GCC tests, the environment used was directly the Unix Bash, so it was not need to modify any project configuration, just by passing the -E option to the gcc command, it output the code after preprocess step. For checking if the literals were directly inserted into the binary, the option -S was used, for generating the assembly output in order to see that Preprocessor has been done the work effectively.

4. 2. Metadata definition

For testing metadata definition has been created different tests, based on float arrays, which are modified and read from a point where the definition at C level is not visible. For checking the functionality, in the example is shown two ways of declaring metadata, from existing variables by a binding mechanism, and from nonexistent variables, with dynamic memory allocation.

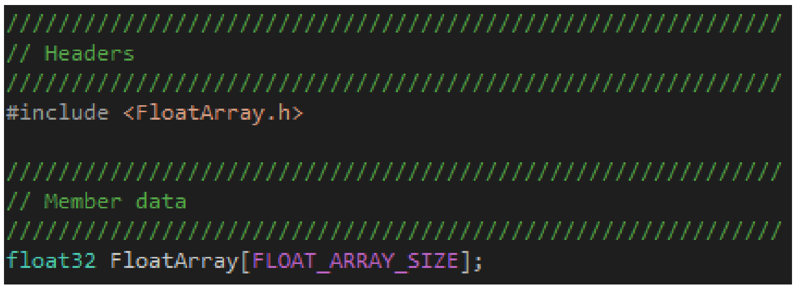
An example of Value binding and creation is shown in the section Values, inside the Implementation chapter. The code shown in Figure 19 and Figure 20 are the front-end of the Value sub-module and the result of the execution. But there is a piece of the puzzle that has not being shown, the real declaration of the array and the registration into the system.

In the Figure 19, the real data is not available for the compiler. At that level it is an anonymous array that is declared in FloatArray.h header, and it is implemented in FloatArray.c as shown in Figure 33 and Figure 34.



```
#ifndef TEST_METADATA_BINDING_FLOAT_ARRAY_H
#define TEST_METADATA_BINDING_FLOAT_ARRAY_H

///////////////////////////////////////////////////////////////
// Headers
///////////////////////////////////////////////////////////////
#include <Types.h>

///////////////////////////////////////////////////////////////
// Definitions
///////////////////////////////////////////////////////////////
#define FLOAT_ARRAY_SIZE    0xFF

///////////////////////////////////////////////////////////////
// Member data
///////////////////////////////////////////////////////////////
extern float32 FloatArray[FLOAT_ARRAY_SIZE];

#endif // TEST_METADATA_BINDING_FLOAT_ARRAY_H
```

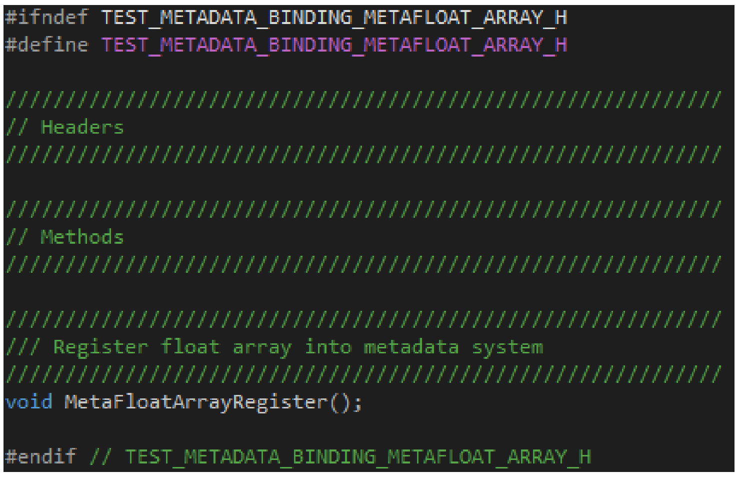
Figure 33: FloatArray metadata test header declaration.



```
///////////////////////////////////////////////////////////////
// Headers
///////////////////////////////////////////////////////////////
#include <FloatArray.h>

///////////////////////////////////////////////////////////////
// Member data
///////////////////////////////////////////////////////////////
float32 FloatArray[FLOAT_ARRAY_SIZE];
```

Figure 34: FloatArray metadata test implementation.

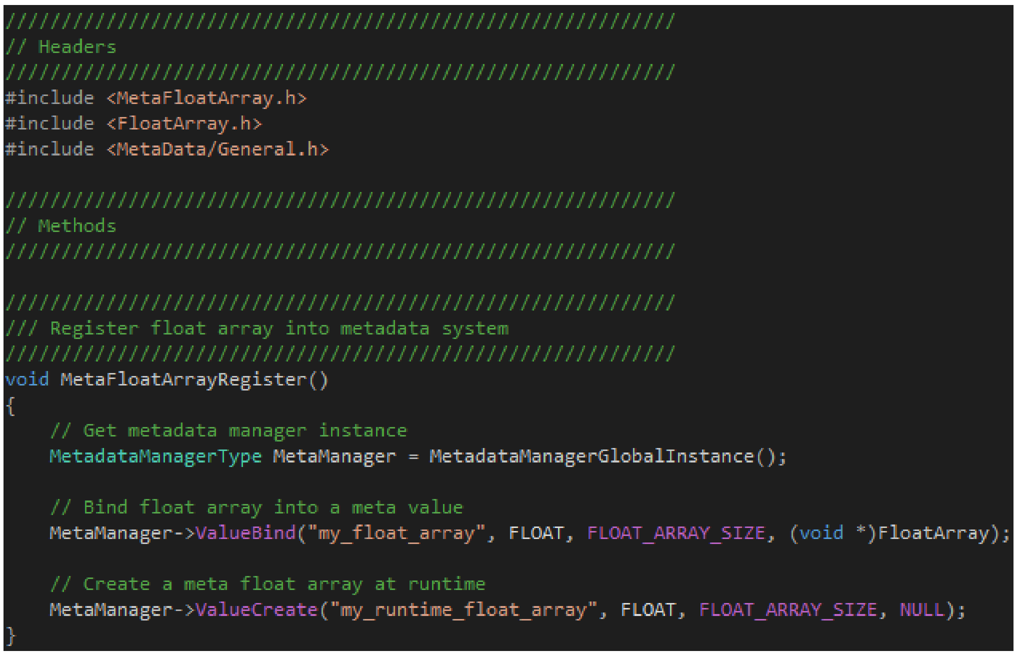
For populating the FloatArray into the Metadata system, it has to be registered with a specified signature. At this level, the declaration of the array it is visible for the compiler, so it can be registered in the system. Note that FloatArray has been designed without any relation with the Metadata system. In the Figure 35 and Figure 36 is shown how the Metadata system can wrap the FloatArray in a transparent manner. Without affecting or modifying the original implementation.



```
#ifndef TEST_METADATA_BINDING_METAFLOAT_ARRAY_H
#define TEST_METADATA_BINDING_METAFLOAT_ARRAY_H

////////////////////////////////////////////////////////////
// Headers
////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////
// Methods
////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////
/// Register float array into metadata system
////////////////////////////////////////////////////////////
void MetaFloatArrayRegister();

#endif // TEST_METADATA_BINDING_METAFLOAT_ARRAY_H
```

Figure 35: MetaFloatArray module for registering the FloatArray instance.

Note that in the MetaFloatArray declaration is not need to include any header, in this way the declaration of the real data is hided, and also the dependency with the Metadata system.



```
////////////////////////////////////////////////////////////
// Headers
////////////////////////////////////////////////////////////
#include <MetaFloatArray.h>
#include <FloatArray.h>
#include <MetaData/General.h>

////////////////////////////////////////////////////////////
// Methods
////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////
/// Register float array into metadata system
////////////////////////////////////////////////////////////
void MetaFloatArrayRegister()
{
    // Get metadata manager instance
    MetadataManagerType MetaManager = MetadataManagerGlobalInstance();

    // Bind float array into a meta value
    MetaManager->ValueBind("my_float_array", FLOAT, FLOAT_ARRAY_SIZE, (void *)FloatArray);

    // Create a meta float array at runtime
    MetaManager->ValueCreate("my_runtime_float_array", FLOAT, FLOAT_ARRAY_SIZE, NULL);
}
```

Figure 36: FloatArray metadata registration into the system.

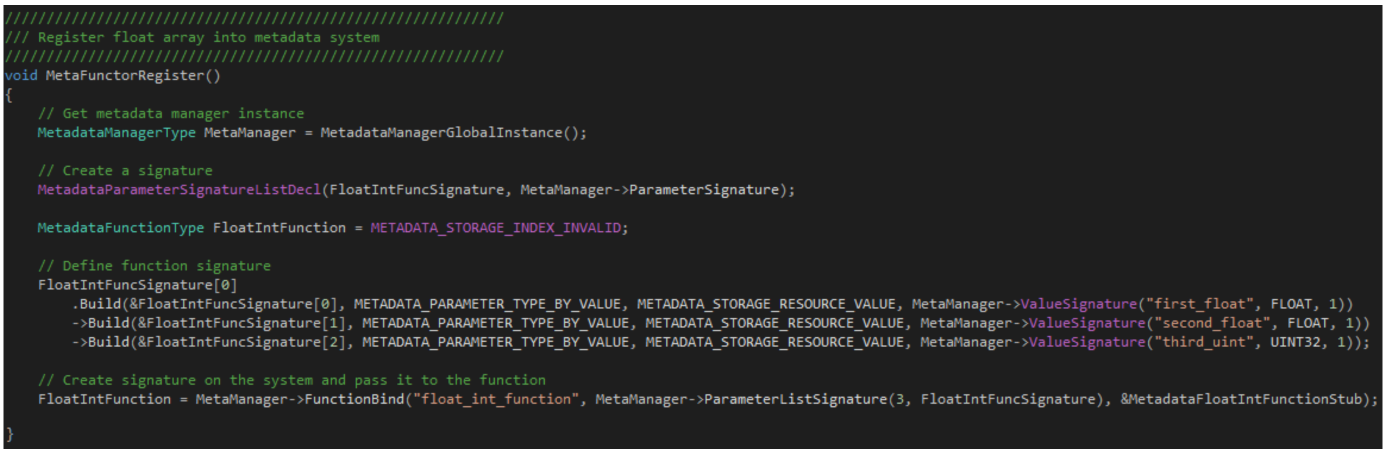
With an easy syntax, the FloatArray is bound into the Metadata system, with the tag my_float_array. By the other hand, my_runtime_float_array is allocated dynamically, and also registered into the system. At this point, the FloatArray and the Metadata system are visible, this is the only part where it is need to have both visible, because it is where the interconnection is done.

4. 3. Foreign function calls

For testing foreign function calls there have been designed some tests which register different functions into Metadata system, by means of MetaManager. The registration requires specifying the signature of the functions, and this process should be wrapped by the Preprocessor module in order to make easier to define them.

In the Figure 22 and Figure 23 is shown the front-end and the back-end of the technique respectively, how the call is done, by using the method FunctionExecute which provides a generic way for calling the foreign functions. Then the other end-point receives the call with the arguments correctly passed. In the Figure 24 is shown the result of the execution.

The registration process is hidden, and it should be wrapped by the Preprocessor module in order to simplify and automate the signature registration. In this example it has been done manually by using the Builder Pattern that is provided, which is the key for make it generic in the future.

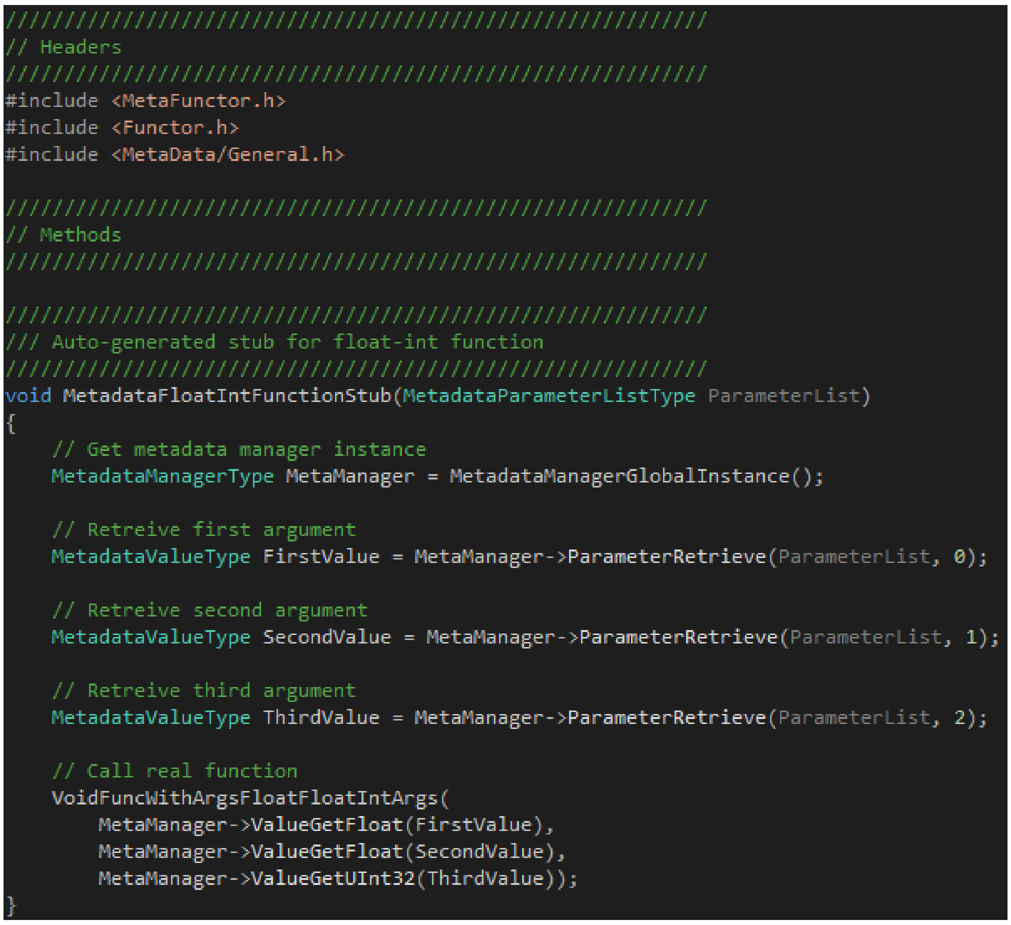


```
///////////////////////////////////////////////////////////////
/// Register float array into metadata system
///////////////////////////////////////////////////////////////
void MetaFunctorRegister()
{
    // Get metadata manager instance
    MetadataManagerType MetaManager = MetadataManagerGlobalInstance();

    // Create a signature
    MetadataParameterSignatureListDecl(FloatIntFuncSignature, MetaManager->ParameterSignature);

    MetadataFunctionType FloatIntFunction = METADATA_STORAGE_INDEX_INVALID;

    // Define function signature
    FloatIntFuncSignature[0]
        .Build(&FloatIntFuncSignature[0], METADATA_PARAMETER_TYPE_BY_VALUE, METADATA_STORAGE_RESOURCE_VALUE, MetaManager->ValueSignature("first_float", FLOAT, 1))
        ->Build(&FloatIntFuncSignature[1], METADATA_PARAMETER_TYPE_BY_VALUE, METADATA_STORAGE_RESOURCE_VALUE, MetaManager->ValueSignature("second_float", FLOAT, 1))
        ->Build(&FloatIntFuncSignature[2], METADATA_PARAMETER_TYPE_BY_VALUE, METADATA_STORAGE_RESOURCE_VALUE, MetaManager->ValueSignature("third_uint", UINT32, 1));

    // Create signature on the system and pass it to the function
    FloatIntFunction = MetaManager->FunctionBind("float_int_function", MetaManager->ParameterListSignature(3, FloatIntFuncSignature), &MetadataFloatIntFunctionStub);

}
```

Figure 37: Function binding into Metadata system in order to allow foreign function calls.

As in the FloatArray, with the functions happens the same behavior. In the sub-module where lives the function shown in Figure 22, Metadata system is not visible, and the design of that sub-module has been done without the knowledge of if that functions will be wrapped in the future, and this is transparent for the function implementations.

The stub for the function should be auto-generated by the Preprocessor module, but in order to provide a detailed view of the mechanism, in the Figure 38, the stub is provided without a macro wrapper. As it can be seen, the stub receives from the arguments, the parameter list already parsed, which is built when the FunctionExecute is called. At this point, it is possible to call the real function with the correct parameters, because at this point that function is visible for the compiler.

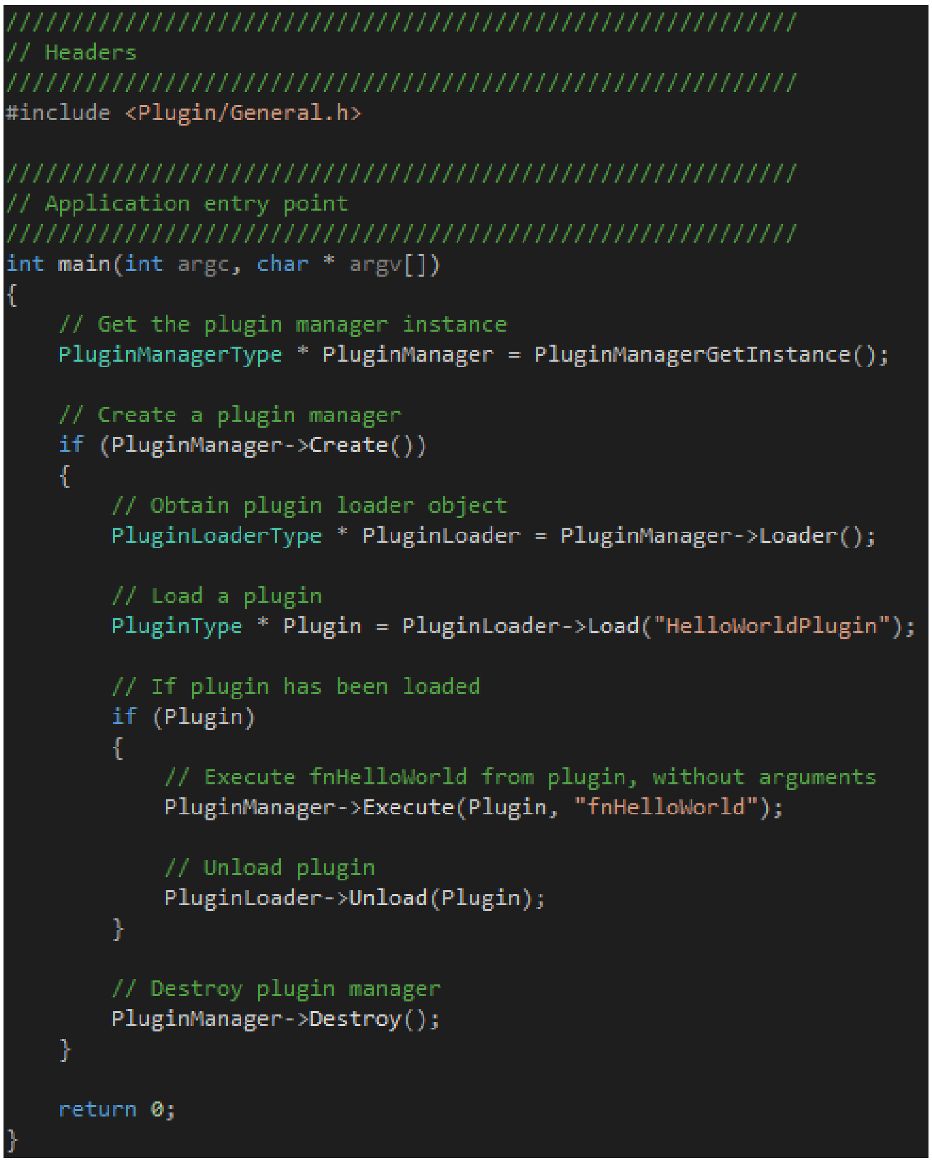


```
/////////////////////////////////////////////////////////////
// Headers
/////////////////////////////////////////////////////////////
#include <MetaFunctor.h>
#include <Functor.h>
#include <MetaData/General.h>

/////////////////////////////////////////////////////////////
// Methods
/////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////
/// Auto-generated stub for float-int function
/////////////////////////////////////////////////////////////
void MetadataFloatIntFunctionStub(MetadataParameterListType ParameterList)
{
    // Get metadata manager instance
    MetadataManagerType MetaManager = MetadataManagerGlobalInstance();

    // Retreive first argument
    MetadataValueType FirstValue = MetaManager->ParameterRetrieve(ParameterList, 0);

    // Retreive second argument
    MetadataValueType SecondValue = MetaManager->ParameterRetrieve(ParameterList, 1);

    // Retreive third argument
    MetadataValueType ThirdValue = MetaManager->ParameterRetrieve(ParameterList, 2);

    // Call real function
    VoidFuncWithArgsFloatFloatIntArgs(
        MetaManager->ValueGetFloat(FirstValue),
        MetaManager->ValueGetFloat(SecondValue),
        MetaManager->ValueGetUInt32(ThirdValue));
}
```

Figure 38: Result of the stub generation for the implementation of the foreign function interface.

4. 4. Plug-in host application

The plug-in host application is a simple program which has the plug-in framework linked and can use it for loading and executing plug-in functionalities. In this simple example, the program reads from the current folder the plug-ins available, which in this case is the HelloWorldPlugin already compiled as DLL for Windows and as SO for Linux.



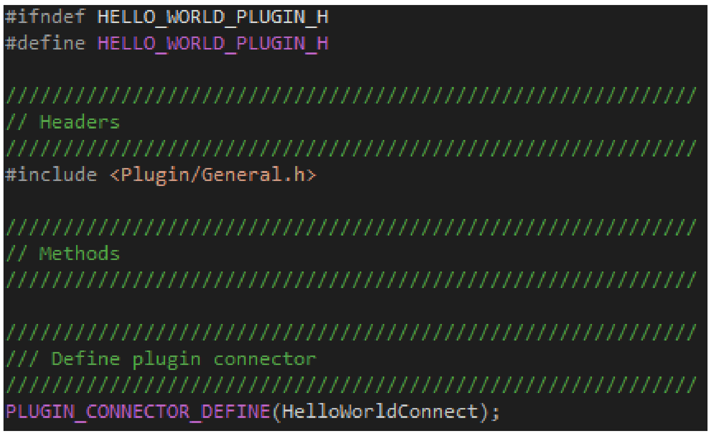
```
////////////////////////////////////////////////////////////
// Headers
////////////////////////////////////////////////////////////
#include <Plugin/General.h>

////////////////////////////////////////////////////////////
// Application entry point
////////////////////////////////////////////////////////////
int main(int argc, char * argv[])
{
    // Get the plugin manager instance
    PluginManagerType * PluginManager = PluginManagerGetInstance();

    // Create a plugin manager
    if (PluginManager->Create())
    {
        // Obtain plugin loader object
        PluginLoaderType * PluginLoader = PluginManager->Loader();

        // Load a plugin
        PluginType * Plugin = PluginLoader->Load("HelloWorldPlugin");

        // If plugin has been loaded
        if (Plugin)
        {
            // Execute fnHelloWorld from plugin, without arguments
            PluginManager->Execute(Plugin, "fnHelloWorld");

            // Unload plugin
            PluginLoader->Unload(Plugin);
        }

        // Destroy plugin manager
        PluginManager->Destroy();
    }

    return 0;
}
```

Figure 39: Plug-in host application minimal example.

The program loads the library and executes the method fnHelloWorld, with the mechanism provided by the Metadata, calling the function without knowing the real signature and implementation.

4. 5. Minimal plug-in implementation

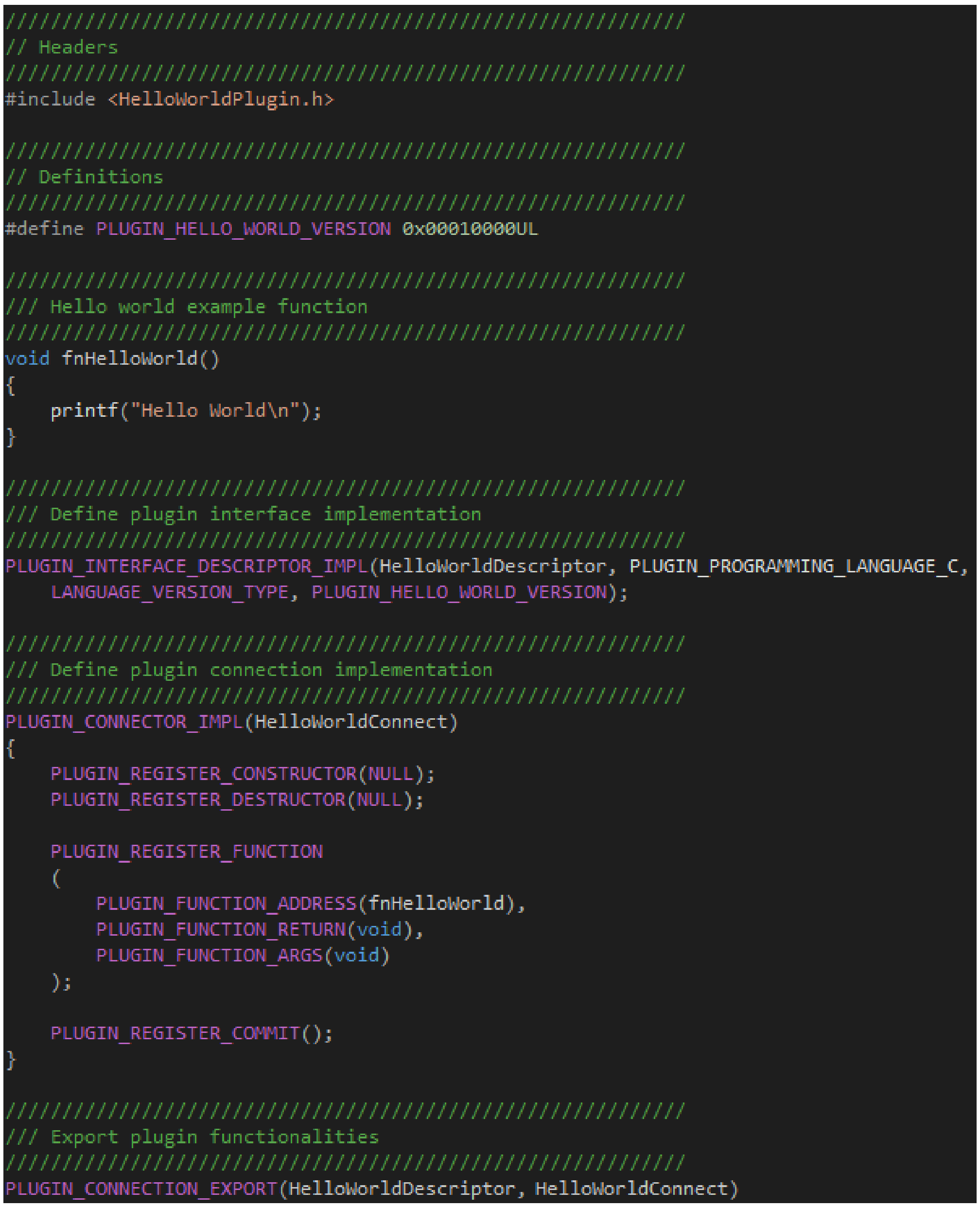
This plug-in is a minimal version developed in C, giving an easy example and a overview of how programmers should develop the components using this system. The implementation differs slightly because during the development, the front-end has been simplified in order to make more easy to use the framework.



```
#ifndef HELLO_WORLD_PLUGIN_H
#define HELLO_WORLD_PLUGIN_H

////////////////////////////////////////////////////////////
// Headers
////////////////////////////////////////////////////////////
#include <Plugin/General.h>

////////////////////////////////////////////////////////////
// Methods
////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////
/// Define plugin connector
////////////////////////////////////////////////////////////
PLUGIN_CONNECTOR_DEFINE(HelloWorldConnect);
```

Figure 40: Minimal plug-in declaration.

In the header of the plug-in, there is only a declaration for populating the connector to the host application. This connector is implemented by an structure which has always the same name hard-coded, and has a pointer to the function where metadata is registered. Note that the functionality which is going to be implemented, the fnHelloWorld function, it is not visible at this point, connector is the only entry point that the host application has visible.



```
///////////////////////////////////////////////////////////////
// Headers
///////////////////////////////////////////////////////////////
#include <HelloWorldPlugin.h>

///////////////////////////////////////////////////////////////
// Definitions
///////////////////////////////////////////////////////////////
#define PLUGIN_HELLO_WORLD_VERSION 0x00010000UL

///////////////////////////////////////////////////////////////
/// Hello world example function
///////////////////////////////////////////////////////////////
void fnHelloWorld()
{
    printf("Hello World\n");
}

///////////////////////////////////////////////////////////////
/// Define plugin interface implementation
///////////////////////////////////////////////////////////////
PLUGIN_INTERFACE_DESCRIPTOR_IMPL(HelloWorldDescriptor, PLUGIN_PROGRAMMING_LANGUAGE_C,
    LANGUAGE_VERSION_TYPE, PLUGIN_HELLO_WORLD_VERSION);

///////////////////////////////////////////////////////////////
/// Define plugin connection implementation
///////////////////////////////////////////////////////////////
PLUGIN_CONNECTOR_IMPL(HelloWorldConnect)
{
    PLUGIN_REGISTER_CONSTRUCTOR(NULL);
    PLUGIN_REGISTER_DESTRUCTOR(NULL);

    PLUGIN_REGISTER_FUNCTION
    (
        PLUGIN_FUNCTION_ADDRESS(fnHelloWorld),
        PLUGIN_FUNCTION_RETURN(void),
        PLUGIN_FUNCTION_ARGS(void)
    );

    PLUGIN_REGISTER_COMMIT();
}

///////////////////////////////////////////////////////////////
/// Export plugin functionalities
///////////////////////////////////////////////////////////////
PLUGIN_CONNECTION_EXPORT(HelloWorldDescriptor, HelloWorldConnect)
```

Figure 41: Plug-in minimal implementation.

The interface declaration and the connector implementation is wrapped by the Preprocessor techniques, encapsulated by plug-in macros. By this way it is possible to generate automatically, and avoiding programming errors, the implementation of the plug-in.

5. Conclusion

The Plug-in Framework gives a new infrastructure for the game engine and the game, that has been demanded during the time, and more recently it was mandatory because of the need for implementing new features to the game engine and for the game logic developers, in order to implement new features with plug-ins, and in the future, to be integrated with scripting in order to develop faster the game-play and provide a way to test more easily the code is being developed, without stopping the whole game server.

This framework is going to increase the productivity, and it solves problems that can annoy to the players, as long time waiting for restart the servers if there is a patch or an error that must be solved, making a poor reputation for the team and the game. From technological point of view, it also adds a new way to design the software and to understand the game engine architecture thus provides a better encapsulation and improves the software quality.

Time ago when there were design errors and the code had to be refactored, developers had to spend too much time just by rewriting functional code that does not fit correctly into the architecture because of a design error. By using this framework, now it will be possible to minimize these design errors, and all of them will be encapsulated inside each component. Each code implementation will fit always the same protocol so, connecting components between the architecture and changing its dependencies will be easier than before.

The main difficulties found in the development were to design the Metadata system, because of its nature, it is a system which implies understanding a new level of abstraction, making a program that understand itself and can provide information at run-time about how the system is designed.

Another problem found was to implement the Preprocessor module correctly. When developing it, there were big differences between the MSVC preprocessor, which has the C89 standard, and the GCC preprocessor, which fits the C99 standard. A macro developed in GCC sometimes did not worked in MSVC because of the old standard used, and it was need to refactor the macro into another version valid for the old standard.

In summary, all the initial goals have been successfully achieved:

- Several Plug-in frameworks have been analyzed and considered.
- The specific requisites needed by the game engine have been discussed.
- Taken into account the requisites and the existing technology, a new plug-in infrastructure has been designed.
- The plug-in interface is defined by means of a metadata interface model, which allows to describe the plug-in's internal structure.
- A prove of concept implementation has been created, which is able to inject new features into an existing and running game engine.

5. 1. Future work

In the short term, this framework is going to be used in order to implement a sandbox with the model loader embedded in a plug-in, using Assimp[49] library, giving the possibility to modelers to test a model with the game engine, and move around the world, checking the kinematics, physics, animations, appearance and the quality of the models as if all of them were in the game. By this way, modelers will have a tool for simplifying the integration of designed models into the game.

In the medium term, this system is going to be integrated into a deployer, in order to provide an automation tool for developing code for the game engine and the game. As the game is going to be online in the near future, this is a tool will be need for avoiding problems have been seen in this text. This tool will be extend in order to give support for game designers, in order to provide the same development flow, integrated with the game engine tools. This will give the possibility to, as programmers will do, inject new resources to the game meanwhile the server is online, being transparent for the players.

In the future the main idea is to extend this system in order to a make higher module to allow code transformations over compilation step or in run-time, so giving a high flexibility over the whole system, with the possibility of move or joint complete software layers, encapsulating plug-ins and functionalities in higher modules, so for example giving possibility of compiling the game engine in a monolithic application, or making it distributed over multiple tiers and divided into libraries, improving the architecture over genericity, encapsulation, flexibility, reliability, usability, and other good practices software design.

6. References

[1] Object-Oriented Rendering Engine. (Copyright © 2000-2015). *Torus Knot Software Ltd*. Retrieved 22:26, July 4, 2015, from http://www.ogre3d.org/

[2] OGRE, Creating A Plug-in DLL. (2010, May 01). In *Ogre Wiki*. Retrieved 22:28, July 4, 2015, from http://www.ogre3d.org/tikiwiki/tiki-index.php?page=Creating+A+Plug-in+DLL

[3] Unreal Engine. (Copyright © 2004-2015). *Epic Games*. Retrieved 22:29, July 4, 2015, from https://www.unrealengine.com/

[4] Unreal Engine Plug-ins. (Copyright © 2004-2015). *Epic Games*. Retrieved 22:30, July 4, 2015, from https://docs.unrealengine.com/latest/INT/Programming/Plug-ins/index.html

[5] Minecraft. (Copyright © 2009-2015). *Mojang*. Retrieved 22:31, July 4, 2015, from https://minecraft.net/

[6] Mozilla Firefox. (Copyright © 1998 – 2015). *Mozilla*. Retrieved 22:31, July 4, 2015, from https://www.mozilla.org/en-US/firefox/new/

[7] Python/C API Reference Manual. (May 23, 2015). In *Python Documentation*. Retrieved 22:33, July 4, 2015, from https://docs.python.org/2/c-api/

[8] SpiderMonkey (software). (2015, May 31). In *Wikipedia, The Free Encyclopedia*. Retrieved 22:36, July 4, 2015, from https://en.wikipedia.org/w/index.php?title=SpiderMonkey_(software)&oldid=664939276

[9] V8 (JavaScript engine). (2015, June 3). In *Wikipedia, The Free Encyclopedia*. Retrieved 22:38, July 4, 2015, from https://en.wikipedia.org/w/index.php?title=V8_(JavaScript_engine)&oldid=665321820

[10] Lua C API. (Copyright © 2003 – 2004). By *Roberto Ierusalimschy*. Retrieved 22:38, July 4, 2015, from http://www.lua.org/pil/24.html

[11] Boost.Extension Library. (2008, July 25). By *Jeremy Pack*. Retrieved 22:40, July 4, 2015, from http://boost-extension.redshoelace.com/

[12] AMX Mod X. (Copyright © 2003-2015). *AMX Mod X Dev Team*. Retrieved 22:49, July 4, 2015, from http://www.amxmodx.org/

[13] Counter Strike. (Copyright © 2010). *Valve Corporation*. Retrieved 22:49, July 4, 2015, from http://store.steampowered.com/css

[14] Unity. (Copyright © 2015). *Unity Technologies*. Retrieved 22:40, July 4, 2015, from https://unity3d.com/

[15] DocBook. (Copyright © 2004-2015). By the *DocBook Technical Committee of OASIS*. Retrieved 22:41, July 4, 2015, from http://www.docbook.org/

[16] Opaque pointer. (2015, March 28). In *Wikipedia, The Free Encyclopedia*. Retrieved 22:44, July 4, 2015, from https://en.wikipedia.org/w/index.php?title=Opaque_pointer&oldid=653955024

[17] Factory (object-oriented programming). (2015, June 15). In *Wikipedia, The Free Encyclopedia*. Retrieved 22:50, July 4, 2015, from https://en.wikipedia.org/w/index.php?title=Factory_(object-oriented_programming)&oldid=667035823

[18] Standard Template Library. (2015, May 7). In *Wikipedia, The Free Encyclopedia*. Retrieved 22:52, July 4, 2015, from https://en.wikipedia.org/w/index.php?title=Standard_Template_Library&oldid=661280152

[19] Python. (Copyright © 2001-2015). *Python Software Foundation*. Retrieved 22:53, July 4, 2015, from https://www.python.org/

[20] Reflection (computer programming). (2015, May 5). In *Wikipedia, The Free Encyclopedia*. Retrieved 22:54, July 4, 2015, from https://en.wikipedia.org/w/index.php?title=Reflection_(computer_programming)&oldid=661003997

[21] Bash. (2014, April 11). In *Wikipedia, The Free Encyclopedia*. Retrieved 22:55, July 4, 2015, from https://en.wikipedia.org/w/index.php?title=Bash&oldid=603775281

[22] Kahan Cluster. (2014, January 17). In *Clúster de Computación "Kahan", Universitat Politècnica de València*. Retrieved 22:56, July 4, 2015, from http://users.dsic.upv.es/~jroman/kahan.html

[23] Template meta-programming. (2015, July 2). In *Wikipedia, The Free Encyclopedia*. Retrieved 22:56, July 4, 2015, from https://en.wikipedia.org/w/index.php?title=Template_meta-programming&oldid=669685865

[24] C++11. (2015, June 6). In *Wikipedia, The Free Encyclopedia*. Retrieved 22:56, July 4, 2015, from https://en.wikipedia.org/w/index.php?title=C%2B%2B11&oldid=665701830

[25] C++14. (2015, June 2). In *Wikipedia, The Free Encyclopedia*. Retrieved 22:56, July 4, 2015, from https://en.wikipedia.org/w/index.php?title=C%2B%2B14&oldid=665224919

[26] Constexpr. (2015, May 13). In *CppReference, C/C++ Online Reference*. Retrieved 22:57, July 4, 2015, from http://en.cppreference.com/w/cpp/language/constexpr

[27] Common Object Request Broker Architecture. (2015, July 2). In *Wikipedia, The Free Encyclopedia*. Retrieved 23:12, July 4, 2015, from https://en.wikipedia.org/w/index.php?title=Common_Object_Request_Broker_Architecture&oldid=669611219

[28] Protocol Buffers. (2015, May 27). In *Google Developers, Protocol Buffers*. Retrieved 23:14, July 4, 2015, from https://developers.google.com/protocol-buffers/

[29] IDL (programming language). (2015, May 9). In *Wikipedia, The Free Encyclopedia*. Retrieved 23:16, July 4, 2015, from https://en.wikipedia.org/w/index.php?title=IDL_(programming_language)&oldid=661589242

[30] Advanced Preprocessor Meta Programming with Boost.Preprocessor Library. (Copyright © 2009). *NDS Technologies*. Retrieved 23:18, July 4, 2015, from http://zao.se/~zao/boostcon/09/2009_presentations/fri/BoostPP.pdf

[31] Chaos Preprocessor Library. (2011, Nov 4). In *Github, Chaos Preprocessor Repository*. Retrieved 23:18, July 4, 2015, from https://github.com/rofl0r/chaos-pp

[32] The Boost Library Preprocessor Subset for C/C++. (Copyright © 2002). By *Paul Mensonides and Housemarque Oy*. Retrieved 23:20, July 4, 2015, from http://www.boost.org/libs/preprocessor

[33] The Order Metalanguage for C Preprocessor Meta-programming. (2014, Mar 13). In *Github, Order Preprocessor Repository*. Retrieved 23:25, July 4, 2015, from https://github.com/rofl0r/order-pp

[34] Object pool pattern. (2015, April 29). In *Wikipedia, The Free Encyclopedia*. Retrieved 23:31, July 4, 2015, from https://en.wikipedia.org/w/index.php?title=Object_pool_pattern&oldid=659867986

[35] Singleton pattern. (2015, July 4). In *Wikipedia, The Free Encyclopedia*. Retrieved 23:31, July 4, 2015, from https://en.wikipedia.org/w/index.php?title=Singleton_pattern&oldid=669854928

[36] Serialization. (2015, June 30). In *Wikipedia, The Free Encyclopedia*. Retrieved 23:32, July 4, 2015, from https://en.wikipedia.org/w/index.php?title=Serialization&oldid=669299012

[37] Representational state transfer. (2015, June 23). In *Wikipedia, The Free Encyclopedia*. Retrieved 23:33, July 4, 2015, from https://en.wikipedia.org/w/index.php?title=Representational_state_transfer&oldid=668335174

[38] XML Schema. (Copyright © 2015). *W3C*. Retrieved 23:34, July 4, 2015, from http://www.w3.org/standards/xml/schema

[39] Endianness. (2015, June 19). In *Wikipedia, The Free Encyclopedia*. Retrieved 23:35, July 4, 2015, from https://en.wikipedia.org/w/index.php?title=Endianness&oldid=667624209

[40] Marshalling (computer science). (2015, April 2). In *Wikipedia, The Free Encyclopedia*. Retrieved 23:36, July 4, 2015, from https://en.wikipedia.org/w/index.php?title=Marshalling_(computer_science)&oldid=654601873

[41] Java virtual machine. (2015, June 26). In *Wikipedia, The Free Encyclopedia*. Retrieved 23:37, July 4, 2015, from https://en.wikipedia.org/w/index.php?title=Java_virtual_machine&oldid=668774805

[42] Bytecode. (2015, May 1). In *Wikipedia, The Free Encyclopedia*. Retrieved 23:38, July 4, 2015, from https://en.wikipedia.org/w/index.php?title=Bytecode&oldid=660227726

[43] C Preprocessor tricks, tips, and idioms. (2015, June 19). In *Cloak, a mini-preprocessor library to demostrate the recursive capabilites of the preprocessor by Paul Fultz II*. Retrieved 23:39, July 4, 2015, https://github.com/pfultz2/Cloak/wiki/C-Preprocessor-tricks,-tips,-and-idioms

[44] C Pre-Processor Magic. (Copyright © 2015). By *Jonathan Heathcote*. Retrieved 22:46, July 4, 2015, from http://jhnet.co.uk/articles/cpp_magic

[45] Dyncall, calling C functions dynamically. (Copyright © 2007-2015). By *Daniel Adler and Tassilo Philipp*. Retrieved 23:40, July 4, 2015, from http://www.dyncall.org/

[46] GNU Libffcall. (2014, February 14). In *GNU Project - Free Software Foundation*. Retrieved 23:42, July 4, 2015, from https://www.gnu.org/software/libffcall/

[47] The New C: X Macros. (2001, May 01). In *Dr.Dobbs by Randy Meyers*. Retrieved 23:44, July 4, 2015, from http://www.drdobbs.com/the-new-c-x-macros/184401387

[48] OpenGL - The Industry Standard for High Performance Graphics. (Copyright ©1997-2015). *Khronos Group*. Retrieved 23:46, July 4, 2015, from https://www.opengl.org/

[49] Open Asset Import Library. (Copyright © 2007-2009). *Assimp Development Team*. Retrieved 23:49, July 4, 2015, from http://assimp.sourceforge.net/