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

Author: Vicente Eduardo Ferrer García

Tutor: Jose Ismael Ripoll Ripoll

Experimental Director: Hector Marco-Gisbert

Academic course 2014 / 2015

Table of Contents

Introduction

- 5 years ago I started developing an **Advanced Game Engine.** Now it is close to be a commercial product.
- In this project I describe one of the fundamental components of the game engine which is the **Plug-in Framework**.
- Once the core functionalities of the game engine have been implemented, the next step was to implement the Plug-in Framework.

Expected Benefits

- **Easily decouple** external libraries, game logic, artificial intelligence of the entities, ...
- **Versioning control** over existing modules of the game engine architecture.
- **Isolate bugs** and problems, avoiding its propagation over the existing architecture.
- **Externalize the development,** independent developers will be able to contribute without knowing the internals of the game engine.

Main ideas

Extend the game engine architecture to make it plug-in oriented.

The game engine is technologically agnostic about in plug-in implementation.

Requirements & Features

- **Independent** from architecture, platform and compiler.
- **Strong decoupling** of specific functionalities.
- **Low dependency** between components, the development is outsourced.
- Allow developers to code **without interfering** between themselves.
- Provides the ability to insert **“on the fly”** functionalities to the game engine upon request.
- Errors **become encapsulated** inside the plug-ins, improving the identification of the conflicting components.
- **High interoperability**, plug-ins can be written in other programming languages.

Existing Plug-in Technologies

- Compiled plug-ins:
  - Minimum overhead.
  - Problems derived from platform and compiler (C++ ABI compatibility).
  - Implementable with C and C++ static / dynamic libraries.
- Interpreted plug-ins:
  - Maximum flexibility and simplicity, plug-ins can be modified during run-time and reloaded.
  - High overhead.
  - Implementable with scripts as Python or Lua.
- Intermediate code plug-ins:
  - Intermediate solution between compiled and interpreted.
  - Usually depending on a Virtual Machine.
  - Implementable with Java or JavaScript VMs.

Design Elements

- **Plug-in Loader**: independent from the type of technology or language which it is implemented, providing transparent code injection.
- **Registration Mechanism**: generic entry point to populate the contents of a plug-in.
- **Stub Generation**: simplification of connection between components, which can be provided by meta-programming techniques.
- **Communication Protocol**: to invoke methods, to obtain and modify data, to proxy and to extend services.
- **Asynchronous** loading and execution (thread-safety).
- Production **automation** and **deployment** of plug-ins.

Implementation

- Plug-in Framework is divided into three main modules.

  1) **Plug-in** module: Provides the plug-in management, loading & unloading and exception handling.

  2) **Metadata** module: Provides a protocol to discover and reflect the anonymous code injected by the plug-in module, and to communicate between the tiers.

  3) **Preprocessor** module: Provides meta-programming techniques in order to simplify the protocol, the stub generation and the signatures for the metadata system.

Plug-in Management

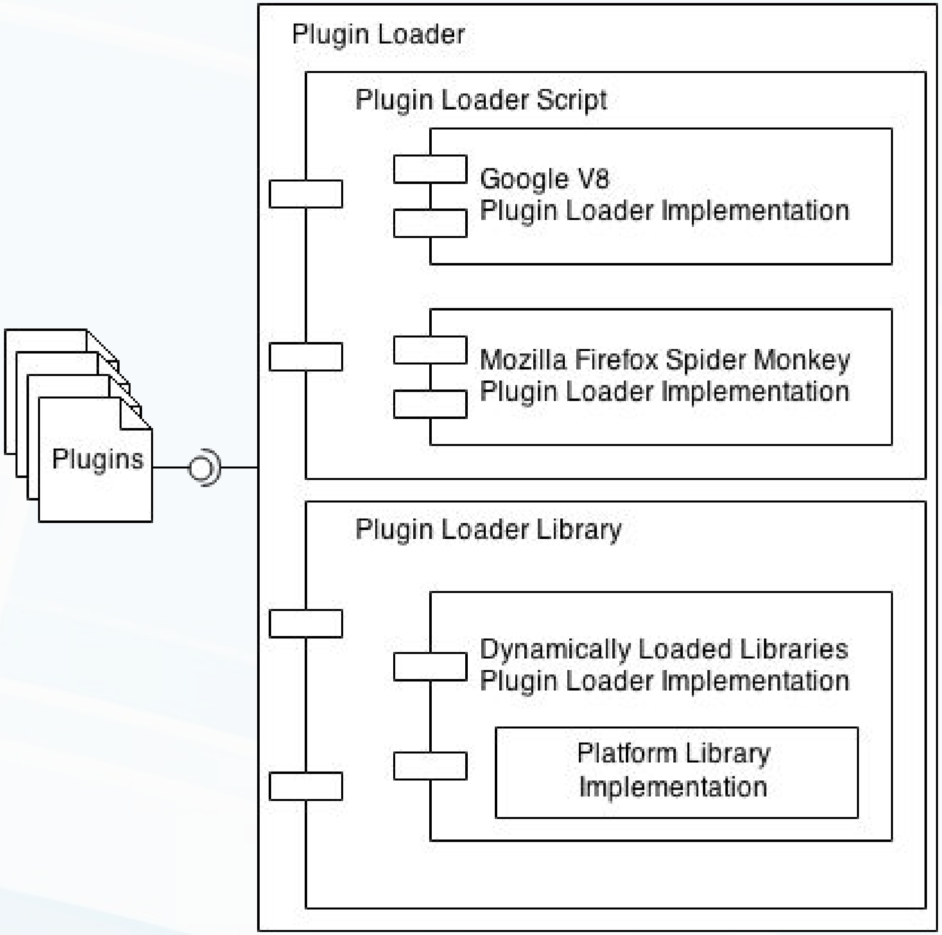
- Plug-in Framework provides a top module which is a **factory pattern,** able to manage plug-ins like generic handles, and it also **encapsulates the Plug-in Loader**.
- Each plug-in can be implemented in different technologies, but externally all of them are treated as equivalent objects with handles of the same type, so **implementation is hidden** in the front-end.
- Plug-ins can be loaded **on demand** or **statically** when the application is launch, and they can be **unloaded explicitly**, or **automatically** by the system when they are not referenced any more.
- The dynamic management of plug-ins provide an **optimal performance** and **low consumption**, without loosing **flexibility**.

Plug-in Loader

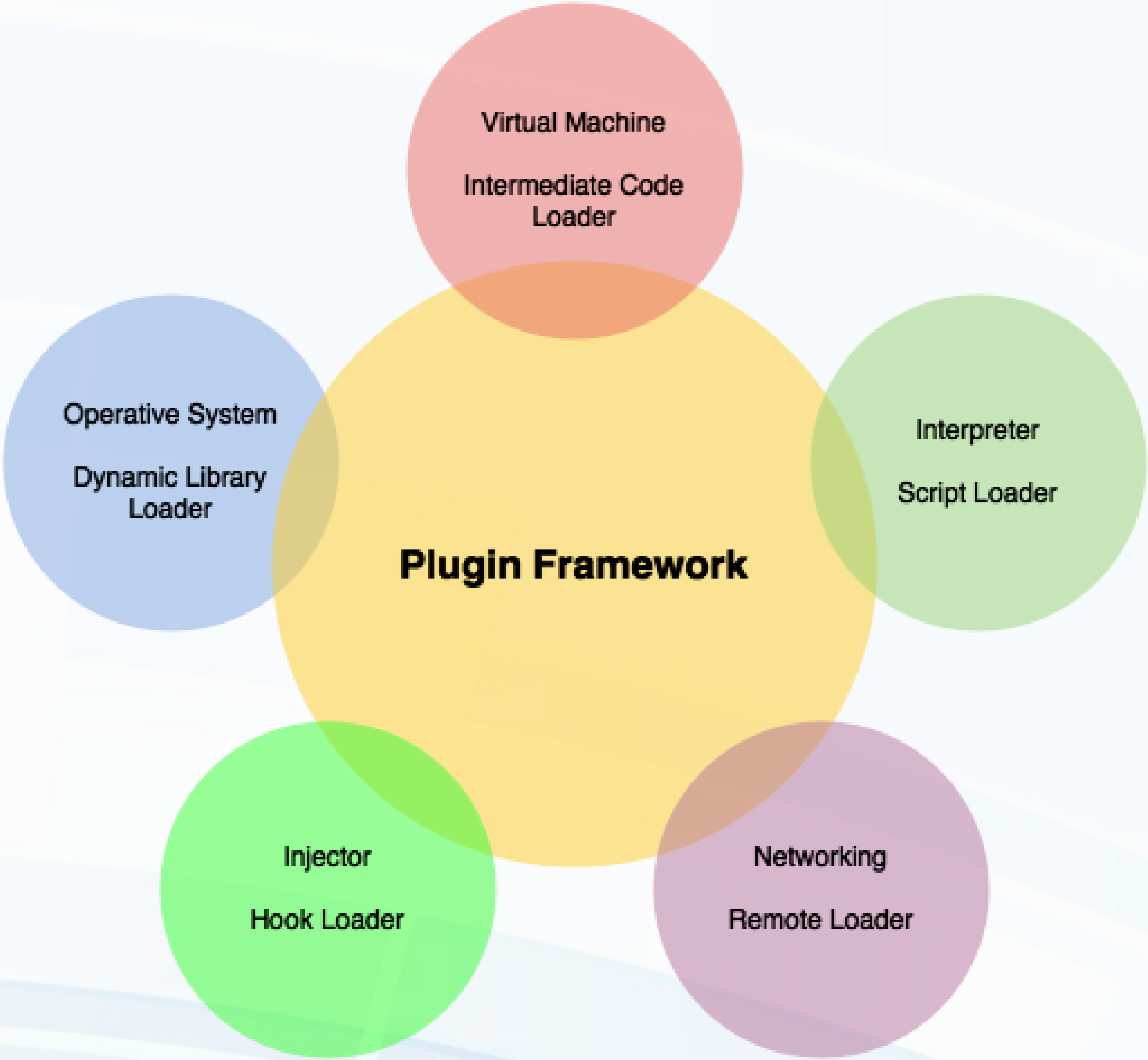
- The Plug-in Loader provides an interface which can be derived in order to provide loaders for multiple technologies.

- The implementation of the loader is hidden, when a plug-in is being loaded, each loader implementation guess if the plug-in is loadable.

- When the plug-in is loaded, the control is returned to the top module, providing a generic handle associated to the internal handle of the plug-in.



Plug-in Loader

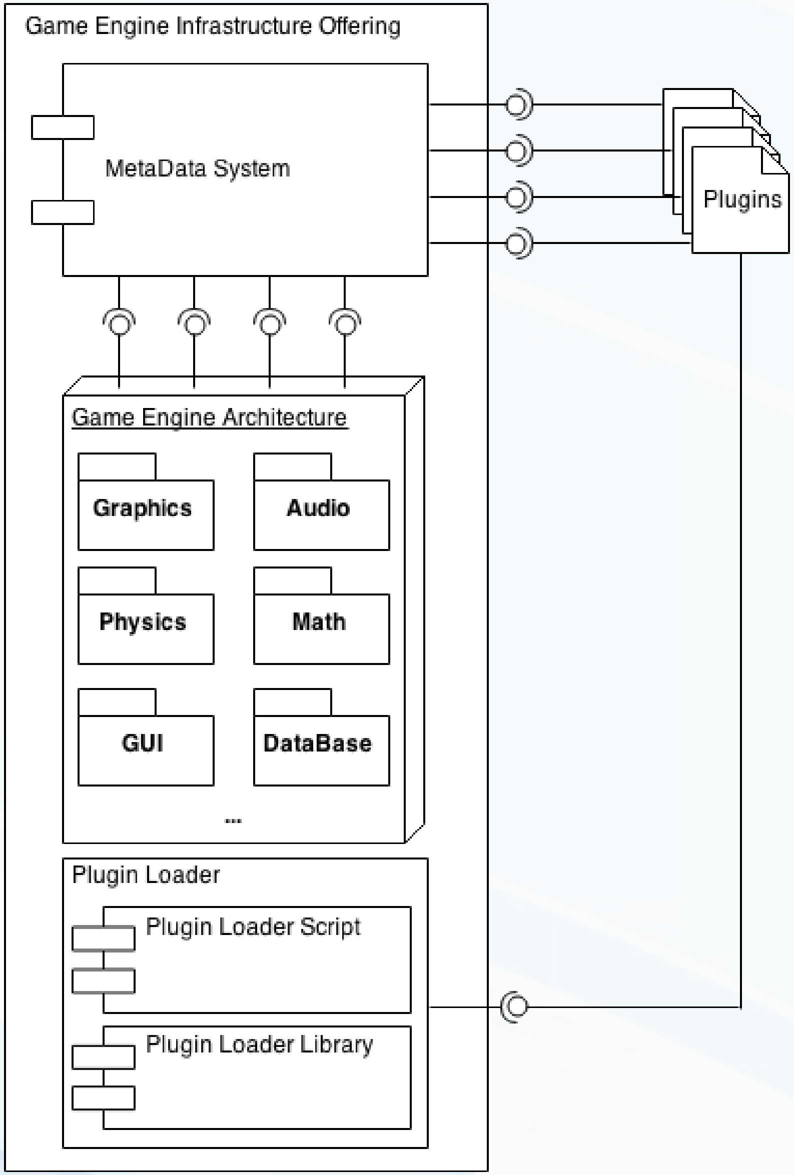


Metadata: Run-time Registration & Discovering

- Metadata module offers a upper layer for encapsulating engine architecture and plug-in functionalities as **components** or **services**.

- Using Metadata is possible to **extend** functionalities at run-time, and provide mechanism to populate **internal structure** of a code implementation.

- By this module is possible to **interconnect** different technologies with a **generic protocol**.



Metadata: Reflection

- Metadata system can reflect the code structure by means of:
  - **Values**: Which represent data.
  - **References**: Which can represent any metadata entity.
  - **Functions**: Which represent functions or methods of classes.
  - **Objects**: Which represent classes.

- With that entities is possible to **model the structure** of the code, to **hide the implementation**, and to **interconnect different technologies** with the same representation.

- When a plug-in is inserted into the system, it has an entry point which is called when it is loaded, and there is provided the top module of the Metadata, which is called **MetaManager**, in order to **populate the anonymous code** of the plug-in to the application.

- When the plug-in has registered the internal structure, the control is returned to the application, and then it can be used with the MetaManager.

Metadata: Reflection

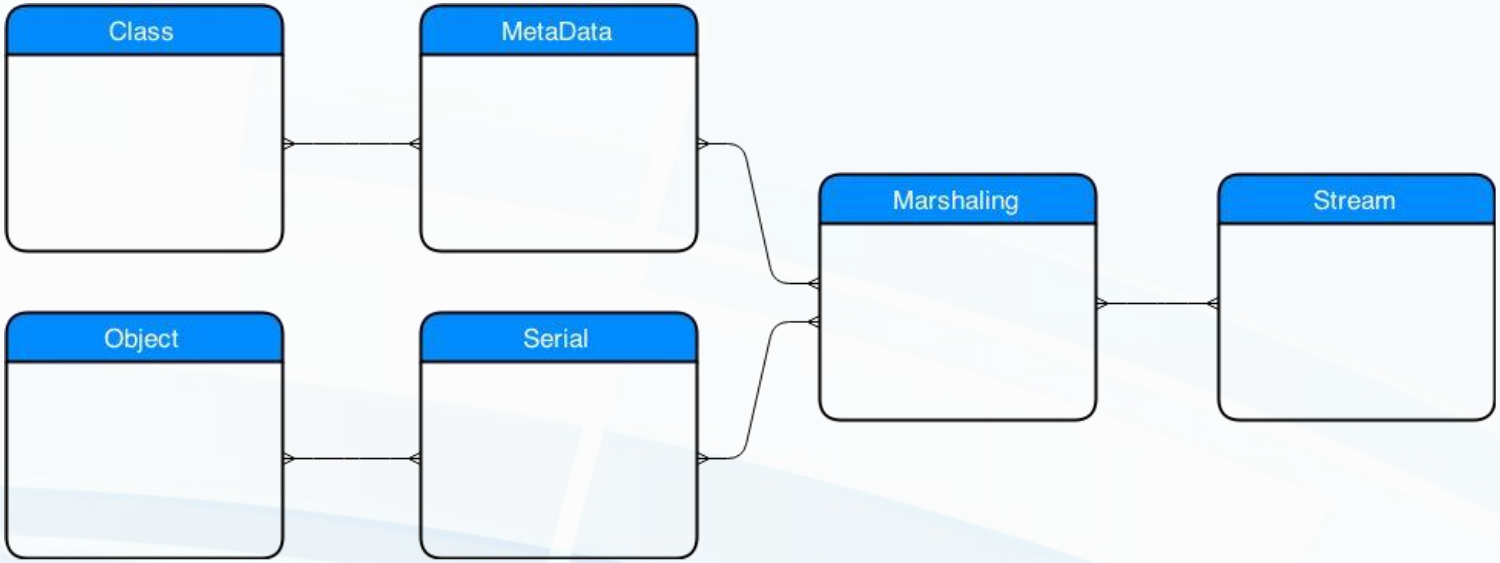
- **Values**:
  - They can be bound to existing data.
  - They can instantiate data with a specified signature.
- **References**:
  - They can be bound to existing metadata entities.
  - Used as a generic Value, when Value signature is not know a priori or it is variable.
- **Functions**:
  - They implement foreign function interfaces.
  - They can be bound to an existing functions or not, but in the second case the product is just a function signature that cannot be called.
- **Objects**:
  - They encapsulate Functions (Methods) and Values (Attributes).

Metadata: Serialization & Marshalling

- The Metadata entities live in the ***Storage***, and it holds the signatures and the instances.
- **Storage** is a memory pool encapsulated into an **object pool pattern**, which provides fast **creating**, **deleting** and **accessing** the Metadata entities.
- The Storage also provides **memory defragmentation**, in order to optimize the memory usage.
- The Storage can be completely **serialized** into a **binary format**, independent from the platform and architecture.
- The binary format is **efficient** and can be easily bypassed by different mediums.
- The signatures and instances can be **reconstructed** in other tiers, and then used in the Metadata system.

Metadata: Serialization & Marshaling

- The Metadata entities belong to a **Scope**, and it provides a **naming service** in order to identify them.
- Each Scope can contain others in a **hierarchical** distribution.
- By the use of the Scope it is possible to provide remote procedure calls, pass and use Metadata entities by reference.



Meta-programming: Preprocessor

- Meta-programming is need in order to simplify the **stub generation** for the foreign function interfaces and the **data or function signatures**. The available options are:
  1. **Preprocessor Meta-programming**: Implementable with C macros; portable and does not need external tools or scripts.
  2. **Template Meta-programming**: Implementable with C++, Turing Complete.
  3. **Interface Definition Language (IDL)**: It needs of external tools or pre-build steps.

- In the Plug-in Framework the **Preprocessor Meta-programming** technique has been used.

- **Simplifies the errors** when using the Metadata system, and **hides** the implementation.

Project in Numbers

- The Game Engine is about ~300.000 lines of code.
- The Game Engine has been developed completely by me, but the MMORPG game (Argentum Online C: http://aoc.dev.parrastudios.com/) is being developed in parallel using the game engine is a collaborative project of 30 people, including musicians, game designers, UI designers, writers, modelers, animators…
- The Plug-in Framework is about ~15.000 lines of code.

Hello World Example

Plug-in Framework creation.

Obtain Plug-in Loader instance.

Load "HelloWorldPlugin" implemented as a dynamic library.

Execute the void function without arguments tagged as "fnHelloWorld" which does a simple printf("Hello World\n");.

Unload the HelloWorld plug-in.

Plug-in Framework destruction.

```
////////////////////////////////////////////////////////////
// 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;
}
```

Conclusions

- Plug-in Framework has been implemented and integrated successfully into the game engine architecture.
- The benefits of he Plug-in Framework have been demonstrated in the proof of concept.
- The future is to enter into the game industry with the same successful model as Minecraft.