Designing an Actor Model Game Architecture with Pony

Nick Prühs
Daedalic Entertainment
Hamburg, Germany
nick.pruehs@daedalic.com

***Abstract*—Pony is an open-source, object-oriented, actor-model, capabilities-secure, high performance programming language which has made a strong public impact by being presented at Microsoft Research, Cambridge. Its high performance and its object-oriented approach already makes it interesting for game development which has been dominated by the C++ programming language for decades. By introducing *actors* at language level and a new language concept called *capabilities*, Pony prevents common multi-threading pitfalls and becomes an exciting candidate for game development and real-time application development in general.**

I. Architecture

Games require access to many different parts of their target hardware, such as the graphics device, input devices or hard disk. The number of possible hardware configurations increases every day. Thus, games access that hardware neither directly, nor by using operating system API calls. Instead, they rely on an additional abstraction layer depending on the underlying OS, most notably DirectX on Windows platforms and OpenGL, OpenAL and others on Mac OS, Linux, and most mobile platforms.

Usually, these abstraction layers provide C and C++ APIs for performing common tasks such as drawing to the screen or querying the state of a mouse button. For this reason, most game engines are written in C++ themselves in order to avoid an additional level of indirection when performing these high-performance tasks.

Games written in Pony make use of the *Foreign Function Interface (FFI)* for calling native C and C++ functions. Graphics APIs tend to be very complicated, using many propritary structures, parameters and return values. Thus, we introduce an additional C++ layer, using the *C application binary interface (ABI)* to provide an easier API for Pony games.

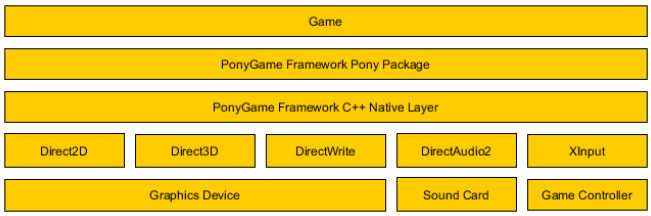

Fig. 1. Architecture of the PonyGame framework.

II. Actors

Actors in Pony are similar to classes in most object-oriented languages. However, actors can define *behaviours*, which behave like asynchronous functions. When you call a behaviour, its body is not executed immediately, but at some indeterminate time in the future. Like functions, behaviours can have parameters, but they cannot provide a return value.

With behaviours being executed asynchronously, many behaviours can run at the same time. The Pony runtime provides an own scheduler, which by default uses a number of threads equal to the number of CPU cores on the target machine. Each scheduler thread can be executing an actor behaviour at any given time, making Pony programs naturally concurrent.

Because of the real-time nature of games, Pony behaviours are ideal candidates for performing time-consuming tasks that normally would slow down overall game performance and degrade user experience, such as writing verbose log files or computing shortest paths.

While the idea of multi-threading in games isn't new, Pony's *behaviours* in conjunction with its *reference capabilities* make it easy to develop games and get all of the message passing and synchronization right. Pony introduces the concept of *isolated* data structures (`iso` in Pony) in addition to the idea of *immutable* data structures in other languages (e.g. `const` in C++ or C#). The notion of isolated data allows actors to change that data however they like and give it to another actor, which may (or may not) reside in a different thread.

III. Memory Management

Games tend to utilize system memory to the very last bit. For this reason, most games create their own memory management system. These systems usually use a combination of multiple prediction algorithms to be able to unload graphics and sounds when they are no longer needed, and preload others before they are actually needed in order to prevent hiccups during gameplay.

Others rely on garbage collection of their underlying runtime, such as Mono, which in turn can be hard to get right: Automatic garbage collection tends to kick in when least desired, causing performance hits and degrading user experience.

Pony-ORCA is a fully concurrent protocol for garbage collection in the actor paradigm. It allows cheap and small actors to perform garbage collection concurrently with any number of other actors, thus preventing the dreaded *"stop-the-world garbage collection"*. An actor may perform garbage collection concurrently with other actors while they are executing any kind of behaviour. Actors reference isolated data, only. Thus, an actor may decide whether to garbage collect an object solely based on its own local state - without consulting, or inspecting the state of, any other actor.

References

[1] S. Clebsch, *Pony: Co-Designing a Type System and a Runtime*. Microsoft Research, Cambridge, 2016.
[2] S. T Allen, S. Clebsch, S. Blessing, A. McNeil et al., *Pony Tutorial*. http://tutorial.ponylang.org, 2016.
[3] S. Clebsch, S. Blessing, J. Franco, S. Drossopoulou, *Ownership and Reference Counting based Garbage Collection in the Actor World*. Causality Ltd. and Imperial College London, 2015.
[4] S. Clebsch, S. Blessing, J. Franco, S. Drossopoulou, *Ownership and Reference Counting based Garbage Collection in the Actor World*. Causality and Imperial College London, 2015.
[5] S. Clebsch, S. Drossopoulou, *Fully Concurrent Garbage Collection of Actors on Many-Core Machines*. OOPSLA 2013.
[6] S. Clebsch, S. Drossopoulou, S. Blessing, A. McNeil, *Deny Capabilities for Safe, Fast Actors*. Causality Ltd. and Imperial College London.