Writting a game engine in Rust: A study on how compile-time memory safety can enhance the workflow of writting a modular game engine

Lukas Vogl

FH Technikum Wien

Abstract. Since years C and C++ are the programming languages of choice when performance is an obligation. They allow a programmer to take care of memory management which often is needed to fit specific use cases in the field of game engineering. But they also require the programmer to have a profond knowledge about hardware and memory. Therefore bugs such as memory leaks or other undefined behaviours are common in games and engines. This paper is going to describe the new system programming language Rust. Rusts new concepts of ownership, moves and borrows shall allow the programmer to write secure and concurrent code without a performance penalty.

Key words: Rust, system programming, modular game engine, C++, memory safety

1 Rust - a new system programming language

In 2015 the first stable version of the system proramming language Rust was released. Mozilla started to support Rust in 2010 with the goal of creating a safer programming language to replace C++ in their new browser rendering engine Servo. Because performance and safety are key concerns in many system applications Rust's concepts aim to tackle two problems that have proven difficult to solve. The first one is the creation of secure code. With Rust it shall be easier and less error-prone to manage memory correctly which, when done manually and without good knowledge of underlying hardware, often led to security holes in C or C++ applications. The second one is concurrency which is a field that is approached with caution even by experienced programmers. Rust tries to completely eliminate race conditions at compile-time and allows code to exploit the abilities of current machines. [1, Chapter 1. Why Rust?]

1.1 Language features

Rust shares the vision that Bjarne Stroustrup had for C++ which he has written down in his paper "Abstraction and the C++ machine model":

> In general, C++ implementations obey the zero-overhead principle: What you don't use, you don't pay for. And further: What you do use, you couldn't hand code any better. [2]

To this definition Rust adds its goals of compile-time memory safety and simplified handling of concurrency. Rust's way of holding its word is a new system of ownership, moves and borrows. These language features are checked at compile-time and are designed to work with and complement Rust's static type system. It has to be mentioned that Rust is neither an object-oriented programming language nor a functional one. It combines useful aspects of both worlds making Rust a multiparadigm system programming language.

Ownership Regarding ownership Rust promises that the decision over a value's lifetime is up to the user and that the program, in spite of the given freedom, will never use a pointer to an already freed object (dangling pointer). While C++ allows lifetime decisions it cannot guarantee that the program does never hit a dangling pointer because it's the user's responsibility to ensure that no pointer to an already freed object is used. Some highlevel languages such as Java or C# solve the dangling pointer problem by introducing a garbage collector that controls when exactly objects are freed. This technique however also introduces an impact in performance. To cope with this problem and to avoid garbage collection Rust came up with its own concept of ownership. In C++ and other languages it is said that an object 'owns' some other value that it points to and the owner therefore has control over the value's lifetime. In Rust this implicit ownership rule was turned to an explicit one and the language enforces that a value just has a single owner that controls its lifetime. As soon as the owner is freed, or dropped in Rust terminology, the owned value is dropped too. One example of Rust's ownership model is its *Box<T>* type that is a pointer to a value of type *T* that is stored on the heap. When the *Box* goes out of scope and is dropped, it frees the space it has allocated too avoiding an otherwisely introduce memory leak. [1, Chapter 4. Ownership]

Moves In C++, when an user assigns a variable to another or passes it to a function by-value, a copy of the value is created. Rust has chosen a different approach and instead of copying values, they are moved. When a move occurs the previous owner transfers ownership to the destination and gets unintialized. The destination now controls the value's lifetime and became the new owner. Move semantics also exist in C++ and were introduced in the C++11 standard allowing the programmer to define move constructors and move assignment operators. [3] With the rule of using move as the default behaviour for assignment Rust can allow cheap assignments and keep the ownership clear as well. One tradeoff the developer has to pay is that a copy now has to be requested explicitly. To do so in Rust one can implement either the *Copy* or *Clone* trait. The first one describes its implementor as trivially copyable by a plain *memcpy* where the second option requires the caller to invoke the *clone* method that returns a new object. [1, Chapter 4. Ownership]

Borrows Beside owning pointers, such as a *Box<T>*, Rust also defines non-owning pointer types that are called *references*. The major difference to owning-pointers is that a reference has no effect on the lifetime or ownership of the

value the reference is referring to. In Rust terminology the act of referencing a value is called *'borrowing'* and the compiler enforces that no reference outlives its referent. There are two kinds of references in Rust – *shared* and *mutable* ones. A shared reference allows to read the value but not modify it and there can be as many shared references as the programmer needs. A mutable reference however is allowed to be read and modified but no other reference is allowed to be active at the same time. This concept introduces the compile-time rule of either several readers or one writter which is essential to the memory safety of Rust. One thing that shall not stay unmentioned is a big difference between C++ and Rust references. While under the hood both are just addresses, Rust references are allowed to be reseated after initialization whereas C++ references just alias the object they have been initialized with and cannot be reassigned afterwards. [1, Chapter 5. References]

2 Rust game engines

Because of the promises Rust makes in the fields of performance, memory safety and concurrency some people from the game development community already implemented game engines in Rust. None of them were compared to similar systems written in C or C++ so there is no scientistic proof on the advantages of Rust as an engine programming language. This section will shortly describe the two best known engines in the Rust community.

2.1 Piston

Piston is a modular open source game engine written in Rust. It was created in 2004 by Sven Nilsen to test 2D graphics in Rust. The current development of Piston has shown that the project has grown further than just 2D graphics. Today the community around Piston is developing 2D and 3D graphics, user interfaces, artificial intelligence and other game engine related subsystems. The engine is separated into several modules/libraries to allow for greater flexibility and a better way to avoid breaking changes in the engines source code. The community is also responsible for VisualRust, a visual studio plugin to support Rust development. The whole engine source code can be obtained at GitHub under `https://github.com/PistonDevelopers/piston`.

2.2 Amethyst

Amethyst is a game engine written in Rust that focuses on data-oriented and data-driven design. It is heavily inspired by the Bitsquid Engine that is nowdays called Autodesk Stingray. Some of its core features are a parallel architecture, an entity-component system, an optimized renderer for modern APIs such as Vulcan and a modular editor. Amethyst is also completely open source and its souce code can also be obtained on GitHub unter `https://github.com/amethyst/amethyst`.

References

1. J. Jim Blandy, *Programming Rust*, 10th ed. 1005 Gravenstein Highway North, Sebastopol, CA 95472: OReilly Media, Inc., 2017. [Online]. Available: https://www.safaribooksonline.com/library/view/programming-rust/9781491927274/ch01.html
2. B. Stroustrup, *Abstraction and the C++ Machine Model*, Z. Wu, C. Chen, M. Guo, and J. Bu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. [Online]. Available: https://doi.org/10.1007/11535409_1
3. cppreference.com, "C++ Move Semantics," http://en.cppreference.com/w/cpp/language/move_constructor, 2015, last accessed 2017-12-03.