OSS Scripting System for Game Development in Rust

Pablo Silva, Rodrigo Oliveira Campos, Carla Rocha

OSS Scripting System for Game Development in Rust

Pablo Diego Silva da Silva, Rodrigo Oliveira Campos, and Carla Rocha[1]

University of Brasília (UnB), Brasília, Brasil
pablodiegoss@hotmail.com rodrigo.redcode@gmail.com caguiar@unb.br

Abstract. Software development for electronic games has remarkable performance and portability requirements, and the system and low-level languages usually provide those. This ecosystem became homogeneous at commercial levels around C and C++, both for open source or proprietary solutions. However, innovations brought other possibilities that are still growing in this area, including Rust and other system languages. Rust has low-level language properties and modern security guarantees in access to memory, concurrency, dependency management, and portability. The Open Source game engine Amethyst has become a reference solution for game development in Rust, has a large and active community, and endeavors in being an alternative to current solutions. Amethyst brings parallelism and performance optimizations, with the advantages of the Rust language. This paper presents scripting concepts that allow the game logic to be implemented in an external interpreted language. We present a scripting module called Legion Script that was implemented for the entity and component system (ECS) called Legion, part of the Amethyst organization. As a Proof-Of-Concept (POC), we perform the Python code interpretation using the Rust Foreign Function Interface (FFI) with CPython. This POC added scripting capabilities to Legion. We also discuss the benefit of using the alternative strategy of developing a POC before contributing to OSS communities in emergent technologies.

Keywords: Tool paper · OSS · Rust Language · Game Engine · Scripting System · Amethyst Game Engine · Entity Component System · Foreign Function Interface.

1 Introduction

The rapid evolution of microprocessors and computer architecture has completely changed the game industry. The advance in computing power enables increasingly complex software solutions, and with that, a wide variety of digital games emerged [5].

Game engines implement general and necessary functionalities for several digital games. The goal is to reuse as much code that is not part of the game's logic and, at the same time, provide a stable architecture for the development of games. The game industry established design standards and code reuse to

assist in the development of complex software, thus creating the so-called game engines.

One of the particular programming requirements in games is the need for performance. This performance requirement enforced popular game engines' implementation in compiled system languages, such as `C` and `C++`. They are low-level languages and have almost universal portability for any processor architecture [15]. However, `C/C++` has a more significant learning curve and tools of greater complexity, in addition to common memory management problems [11,7], resources, and multi-platform compilation for games.

Game engines usually have scripting systems to accelerate the game development cycle and shorten the learning curve. The scripting system is an abstraction layer of the engine's modules, usually in a different language, to separate the game's specific logic from the complexities of the engine [14]. Most of the competitive engines in the gaming industry have a well-implemented scripting system with their characteristics. For example, Godot [3] is an open source game engine implemented in `C++` and can execute scripts in `C#`, `D`, and its built-in scripting language, called `GDScript` [4].

The `Rust` programming language is an alternative for `C` and `C++` in performance and portability. It also delivers performance, and the latest programming standards, such as robust packaging systems, dependency management, functional programming support, and more significant memory safety handling.

Amethyst is an open-source game engine in `Rust`, which, currently, does not have a scripting system. Therefore, it requires expertise in `Rust` to develop a game within its framework. In addition to the learning curve inherent in system languages, any changes to the code require its compilation, which can take a considerable amount of time in the `Rust` environment due to the compile-time checks. To facilitate new game developers' adhesion into the engine, the Amethyst community has expressed its intent to have a scripting system using a Request-For-Comments (RFC).

Contributing to a large and active OSS community, using emergent technologies with little technical documentation to specific problems, imposes some challenges. The Amethyst game engine has frequent architectural changes, and that is expected with new technologies, defying both contributors and maintainers to add new features, solve bugs, and leave stable versions available to users. Instead of contributing to an architecture that we know will be discontinued or contribute to architecture not yet mature, we opted to develop a Proof-of-Concept (POC). With this strategy, we could anticipate the new architecture problems and develop a functionality not yet present in the engine.

This paper presents an extensible scripting system for the Entity Component System, called Legion, used inside the Amethyst game engine. This system will serve as a driver for executing different interpreters and language scripts. The scripting system allows programmers with little or no `Rust` programming skills to start using a Rust-based game engine. We developed a Proof-of-Concept (POC) with support for the `Python` language to serve as example to other languages implementations. The project is an unprecedented work that highlights the con-

cepts of implementing a scripting system in a `Rust` game engine. It defines a baseline for Legion and Amethyst contributors to continue working in scripting solutions while improving their software and features.

The rest of the paper is organized as follows. Section 2 presents the background, the necessary concepts of the Entity Component Systems developed in `Rust`. Most of the technical documentation necessary to implement the scripting system is diffuse in forums, blog posts, and other unstructured grey literature. In Section 3, we detail how we conduct this work. Finally, in Section 4, we present our results. The conclusions and lessons learned are in Section 5.

2 Background

Amethyst[1] is a data-driven open source game engine made with `Rust`, focused on being fast and configurable and maintained by the Amethyst Foundation. One of its main characteristics is the parallel Entity Component System (ECS) with user-friendly abstraction, which will manage, store and update game data using performance-focused strategies.

Amethyst community maintains an updated roadmap with the next steps for the project. The process for significant changes is based on Request For Comments (RFCs) [1] and seeks to provide a controlled, transparent, collaborative, and consistent addition of new features to the engine and its libraries.

Amethyst is a complex project organized in several modules. A scripting system gives external access to data and components through its Entity Component System. It creates a layer of interaction between the external data and the Amethyst game data. The other engine modules can be integrated in the future as the scripting becomes stable.

2.1 Entity Component System

A typical pattern in a game development project is the Entity Component System (ECS). It is a core in the engine, and it manages and organizes the objects inside a game during each iteration. This pattern favors the composition over inheritance by transforming functionalities into components, therefore, keeping each functionality self-contained and reusable. The game objects will be called Entities and will receive their behaviors through instances of the Components [6].

Entity Component Systems use a Struct of Arrays (SoA) to manage the entities and their components. Instead of having a heterogeneous array of entities, called an Array of Structs (AoS), on an ECS, each component type is stored separately, as seen in Figure 1a. It increases the performance of queries and cache optimizations when iterating over game data.

One ECS implemented in `Rust` is called Specs Parallel ECS (Specs), which the Amethyst Foundation maintains. Specs is close to the classic design of an ECS presented above. It allows for parallel system execution, with both low

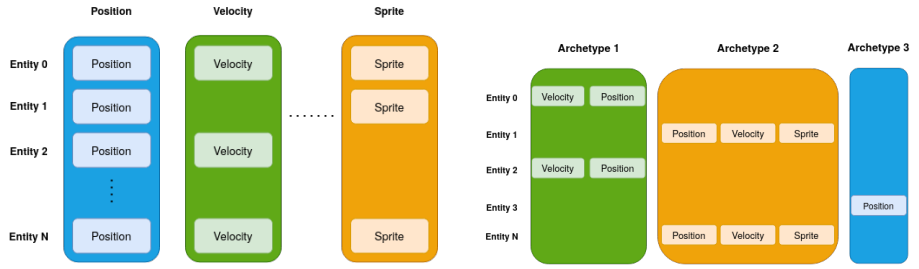
[1] https://github.com/amethyst

overhead and high flexibility [10]. The user can declare `Components`, `Systems`, `Entities`, and `Resources`, all tied to a virtual world, which becomes the main game container.

Another `Rust` ECS is Legion [2]. Legion has minimal boilerplate and presents a better performance in some ECS operations due to its abstraction over the component types, called the archetype system. A unique combination of components defines an archetype, represented in Figure 1b. Legion's archetype system stores components on tables created on-demand as new entities are inserted into the game world. It contrasts with Specs and most ECSs, with unique storage for each component. Those archetype tables create faster filtering and querying since it is done on the archetype level and not by entity iteration.

In Legion, while creating one or many entities, we need to match and find to which archetype they belong. If they do not match any of the existing Archetypes, Legion creates a new Archetype with the corresponding layout of the group of components that define the entity.

2.2 Specs x Legion



(a) Specs component data storage using arrays for each component.

(b) Legion's archetype storage for each unique component group in entities.

Fig. 1: Storage comparison between Legion and Specs.

Figure 1 illustrates the differences in data storage in both Specs and Legion, directly impacting the engine performance. Declaring data is the first step for an ECS-based game. In Specs, after declaring a component, one must register them into a `World` structure. It creates the storage table for each component and the synchronization logic for parallel access. However, Specs uses Rust's unique struct identifier, called `TypeId`, as an index to each registered component type.

A `TypeId` in `Rust` is the unique identifier for a type. It is only available for types from the program's static lifetime. Also, `Rust` does not allow runtime declarations of structs, making injecting externally defined components into Specs impracticable.

For Legion, a `Component` is defined as any type with static lifetime and implements `Rust` traits as `Sized`, `Send`, and `Sync`. The registering and storage will be

done later by the ECS internal code as it combines the components to create an `Entity`. It enables building a scripting system through new components capable of sorting and uniquely identifying external components inside the Legion core.

3 Method

In this paper, we present an experience report of a contribution to a growing but still recent community of `Rust` and game engines. The use of modern technologies is also a highlight for the Amethyst engine. In this emerging technology context, we have technical challenges related to implementing a scripting system in Rust without a previous reference solution and the need to adapt and include this volatile community environment into our OSS contribution workflow.

We began with community bonding, in a traditional OSS contribution process. We got familiar with the Amethyst code base during this phase, guidelines, reading closed and open issues, forum posts, and threads focused on scripting. Amethyst has a separate repository for its Request-For-Comments (RFC), keeping discussions focused on topics such as the scripting system. They have a dedicated forum [13] for communication, questions, and discussions about the engine. Amethyst also has a Discord [12] server for faster collaborations.

Since Amethyst is a continually evolving project with many modules, we decided to adopt a Proof-Of-Concept (POC) strategy. The strategy consists of developing a POC in a separate repository and implementing the minimal features necessary to validate the solution proposal. Once the community mentors review the POC, we can plan a roadmap to contribute to the Amethyst engine codebase.

In our repository, the Amethyst team could quickly review our progress and give feedback. We could do our separate version control and manage the project's risks without all the other engine modules volatile environment. The POC environment allowed us to be mentored by experienced Amethyst members. Some had participated in the scripting RFC, and some created scripting initiatives that would be incorporated into our POC. Finally, after sharing the results with the community, validating our scripting module proposal through the POC, implementing the scripting system in the Amethyst engine has fewer risks involved, and the POC serves as a guide to new contributors.

4 Results

We implemented the Proof-of-Concept during a year of research with the support of the `Rust`, Legion and Amethyst communities. It is a `Rust` module integrated into Legion. It comprises a library that adds scripting capabilities to the ECS. Our library allows running game scripts written in `Python`. As a result, it is possible to define components and entities from `Python` and still use a `Rust` ECS.

The scripting system is developed in `Rust`, `C`, and `Python`. Figure 2 depicts our architecture, using a bidirectional FFI to connect external languages to the

scripting system. The developed library supports creating and querying entities from `Python` scripts stored inside Rust's domain. It does not infringe the ECS mechanisms, and it benefits from Legion. We implement a `Python` driver for demonstration purposes, but the goal is to be extensible to any other language drivers.

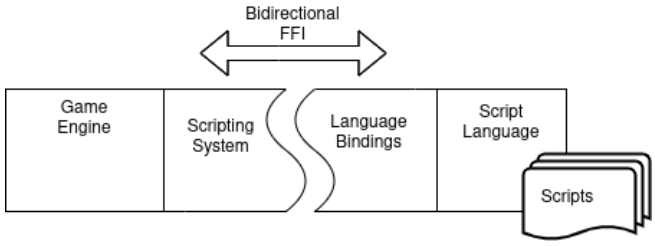


Fig. 2: Representation of the Scripting System architecture within the Amethyst Game Engine.

The `Rust` component of the scripting system is responsible for interacting with Legion and its API while also exposing this API for the `C` language driver. The language driver is written in `C`, and it is responsible for interpreting scripts and using the Rust API to inject data into Legion. Finally, in the `Python` script, the component data is appropriately defined and grouped into an entity using the `C` interpreter's API. Therefore, the game engine can directly execute our Rust code, which will execute the `C` language driver responsible for interpreting the Python script.

To build our **POC** we had to solve some problems like linking and compiling `C` code from `Rust`, calling external functions defined in `C` from `Rust`, and converting types between the two languages. Finally, we customize the **ECS** to manage external components from `Rust`. We also performed minimal modifications on Legion that still will be revised by the community.

5 Conclusion and Lessons Learned

The main result of our work is the POC called `legion_script` [8]. It is an extensible library that adds scripting capabilities to the Legion ECS. This project implements minor changes to Legion and creates another scripting layer as an API that language drivers can access. Since the implementation is on top of an ECS, The scripting system is not specific to the Amethyst game engine. Therefore, any other Rust project could benefit from this project.

Working in a non-consolidated area of programming, like scripting for games using `Rust`, brought some not expected challenges. These vague concepts of scripting and language interpretation generally available on the web might create the illusion that the work to be done is more straightforward than it is. It creates

a scope management problem that can be entirely out of sync with the team's capabilities to produce in time. We did not find much scripting documentation for `Rust` since the language is relatively new. Also, smaller teams and projects tend to make decisions by word of mouth, not being recorded into repositories. Mainly, investigating and understanding decisions is much harder when documentation and working examples do not exist.

Besides the problem of scope, newly created technologies are very volatile. We expected a traditional OSS contribution process, from community bonding to pull request revision. During our planning phase, the Amethyst community never manifested any obstacles concerning Specs' use as its ECS system. However, they started migrating from Specs to Legion during our research, which directly affected our contribution relied on the ECS system. It obliged us to change our OSS contribution strategy to the development of a POC.

A lesson learned was the interaction with the community. Even though we could not find working examples of the concepts we were trying to develop, we reached out to many developers who tried approaching the problem and discussed the community's solutions. It allowed us to combine many incomplete or deprecated solutions into one working **POC**.

Besides developing documentation for scripting and Legion, our project aimed to create a platform and a runnable example. This solution can help guide the community to develop new solutions based on what we provided. Our complete work was shared through Amethyst Forums [9], Discord servers, and other social media.

References

1. Amethyst Team: Amethyst rfcs (2018), https://github.com/amethyst/rfcs, Accessed 01 December 2019.
2. Gillen, T.: Legion (2020), https://github.com/amethyst/legion, Accessed 9 December 2019
3. Godot: Godot engine (2014), https://github.com/godotengine/godot,Accessed 01 December 2019.
4. Godot: Gdscript basics (2020), https://docs.godotengine.org/en/3.2/getting_started/scripting/gdscript/gdscript_basics.html, Accessed 17 September 2020.
5. Gregory, J.: Game engine architecture. Peters (2009)
6. Halpern, J.: Developing 2D Games with Unity: Independent Game Programming with C#. Apress (2019)
7. Novark, G., Berger, E., Zorn, B.: Plug: Automatically tolerating memory leaks in c and c++ applications (01 2008)
8. Oliveira, R., Silva, P.: Legion script (2020), https://github.com/redcodestudios/legion_script, Accessed 8 October 2019
9. da Silva, R.O.C.P.D.S.: Undergrad thesis on game scripting for legion. https://community.amethyst.rs/t/undergrad-thesis-on-game-scripting-for-legion/1753 (2020), accessed 22 January 2021.
10. Specs: Specs parallel ecs. https://specs.amethyst.rs/docs/tutorials/ (2020), accessed 29 November 2019.

11. Tang, Y., Gao, Q., Qin, F.: Leaksurvivor: Towards safely tolerating memory leaks for garbage-collected languages. USENIX Annual Technical Conference pp. 307–320 (01 2008)
12. The Amethyst team: Amethyst discord (2018), https://discordapp.com/invite/amethyst, Accessed 29 November 2019
13. The Amethyst team: Amethyst forum (2018), https://community.amethyst.rs, Accessed 29 November 2019
14. Varanese, A.: Game scripting mastery. Premier Press (2003)
15. Zivkov, D., Kurtjak, D., Grumic, M.: Gui rendering engine utilizing lua as script (2015)