IMET - International Conference on Interactive Media, Smart Systems and Emerging Technologies (2023)
N. Pelechano, F. Liarokapis, D. Rohmer, and A. Asadipour (Editors) *Short Paper*

Integrating Julia Code into the Unity Game Engine to Dive into Aquatic Plant Growth

A. Lewerentz[1,2], N. Manke[3], D. Schantz[3], J. S. Cabral[2,4] and S. v. Mammen[3]

[1] Institute of Geography and Geoecology, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
[2] Center for Computational and Theoretical Biology, University of Wuerzburg, Wuerzburg, Germany
[3] Games Engineering, Faculty of Mathematics and Computer science, University of Wuerzburg, Wuerzburg, Germany
[4] School of Biosciences, College of Life and Environmental Sciences, University of Birmingham, Birmingham, United Kingdom

Abstract

Ecologists use process-based ecological models to predict biodiversity, but their complex results can be challenging to communicate. This challenge can be addressed through interactive visual simulations, which are easy to create with sophisticated game engines such as Unity. However, mechanistic models are increasingly written in the Julia programming language and Unity does not support its integration. In this paper, we present a Julia-Unity plugin that allows direct coupling of Julia and Unity codes. It was developed in a user-centred, iterative manner. The given use case of an immersive, interactive simulation of a macrophyte growth model was tailored to public outreach and science communication. The resulting, rather versatile plugin is a novel tool that features immersive visualisations of Julia-coded simulation models, ecological or otherwise. Due to the features of a game engine, they are particularly apt to engage a wider audience, facilitate collaboration and interdisciplinary work, and enable the exploration of complex systems.

CCS Concepts
• ***Human-centered computing*** $\rightarrow$ ***Visualization toolkits;***

1. Introduction

A major challenge in ecology is to predict future changes in biodiversity as a result of land-use change, climate change, and exploitation. Process-based models—rule-based or numerical representations of simplified aspects of a real system—are key tools to address this challenge [CVH17]. The output of these models can include multiple agents, parameters, and dimensions such as time and space. To better understand the multidimensional output, an interactive visual representation that goes beyond printing two-dimensional figures is very helpful [RN11]. In addition, visualisation and interactive representations also help to better communicate the model and its outputs to the public. However, building more complex and appealing applications that can visualise complex data requires expertise and appropriate tools.

Since the 2000s, game engines, i.e. comprehensive software systems that facilitate the design and development of interactive applications, have been widely used to support various scientific fields [LJ02]. Unity is a popular game engine that provides special support for the development of interactive applications [Haa14]. It offers many types of predefined data objects that can be augmented with custom scripted behaviours written in C#. However, process-based ecological models are typically based on dynamic programming tools [Pau07], that are tailored to scientific computing and give lower priority to interactive visualisation, such as the Julia programming language [BEKS17]. Julia is highly efficient, provides scientific programming functionality, allows for high-level code design, and yet facilitates fast iteration cycles due to its dynamic properties. However, there is no native support for using Julia in Unity applications and, to the authors' knowledge, no suitable plugin for tightly integrating Julia into Unity code has yet been published.

Immersive visualisations of process-based models written in the Julia language are emerging in various contexts. For example, the Julia package 'Modia3D.jl' [NO20] specifically allows the modelling of 3D mechanical systems. Dedicated packages such as 'Makie.jl' [DK21] and 'NetworkViz.jl' [JAM*16] support different scientific visualisation approaches. The 'OpenModelica' modelling framework provides both extensive visualisation functionality and tight integration with the Julia programming language [FPA*19]. Furthermore, there are already interactive applications that have successfully coupled programs written in Julia with game engines [CBL21]. One example is the serious game bioDIVERsity [LSG*21], which allows players to experience the underwater worlds of lakes. Here, Julia code is integrated directly into Unity to simulate plant growth using a Julia C# interface that utilises Julia's C++ API in the form of a dynamically linked C++ library (DLL).

In this paper, we describe the Julia-Unity bridge in detail and

DOI: 10.2312/imet.20231263

present how we have turned it into an easily accessible plugin for the Unity engine that supports not only the use case of an underwater plant growth visualisation, but also other interactive visualisation contexts. The required plugin implements an application programming interface, or API, that allows Julia code to be called and executed from within Unity, and data to be passed back to Unity.

2. Technical realisation

The plugin and the extended support for the concrete use case were developed as part of an interdisciplinary university course between computer scientists and ecologists. To ensure an effective goal-oriented development cycle, we followed a user-centred engineering (UCE) approach [Ric14].

The technical realisation of the Julia-Unity plugin includes

1. a dynamic linked C++ library (Julia Interface),
2. the Unity package itself (Julia Unity Package), and
3. a Demo application for testing the package (MGM Simulation).

2.1. Design and Development

The 'Interface' is the software component that connects the Unity and Julia environments (Fig. 1 (a)). It allows the exchange of model data and parameters between Julia and Unity and the execution of code snippets in the respective 'other' environment. However, the user of our plugin does not interact directly with the interface realized in C++, but uses C# classes that encapsulate the functionality of the interface and are part of a Unity package. The structure of the Unity plugin and how it interacts with a Unity project and a Julia model with all its key components is shown in Fig. 1 (b).

For the interface between Julia and C++, a DLL was chosen because the Julia installation already provides an API for communicating with C++, and because Unity can use the functionalities of the DLLs as native plugins from C#. Julia's API requires Julia to be installed on the user's machine. The DLL was implemented by encapsulating Julia's API functions in newly created functions and exporting them to make them accessible in a C# language environment. More specifically, the API allows the transfer and translation of data types and basic data containers, such as arrays, between the two languages. This allows functions, modules, and data values written and handled in Julia code to be referenced and used from C++. We have also encapsulated the C++ functionality for handling errors thrown by Julia. When data is passed between Julia and C++, the data type must be properly converted to match the specifications of the receiving language.

We chose the Unity package format for the project because it makes it easy to distribute and include various assets such as DLLs, scripts, and other files (Fig. 1 b). The 'JuliaBase' class encapsulates the DLL in C#, contains 'struct' data structures for handling access to Julia functions, modules, and data values, allowing two-way communication between Julia and Unity. Julia's ANY data type for multidimensional arrays posed a problem due to its complex representation in C languages. We have resolved this issue by offering a Julia script that grants access to individual values within multidimensional arrays through the use of dissect functions, which allow for the deconstruction of said arrays into lower- and one-dimensional ones. The 'JuliaBaseManager' class ensures that these dissect functions are loaded and referenced during the start-up of a given Unity scene. It requires a 'JuliaBaseManager' instance in the scene. To use additional Julia scripts, they must be listed in the 'Inclusions.txt' file. The 'FileHandler' makes the contents of the file accessible so that the enabling 'JuliaModelHandler' can load the Julia scripts. To simplify the setup process, the 'InstallationManager' checks and sets the necessary environment variables in the Windows operating system and downloads the required version of Julia. It opens a window within the Unity editor that provides a simple user interface for triggering the appropriate installation routines and opening the documentation of the Julia Unity package. A simple C# interface called 'IJuliaPluginDebugger' is provided for logging errors thrown by Julia or the package's classes and for displaying messages, which allows the creation or extension of one's messaging and logging system.

2.2. Usage

From a user perspective, Julia models can be used from within Unity once the Julia Unity package has been installed and set up. The actual functionality of the package is made available in the 'JuliaBase' class. To use it, one needs to place a 'JuliaBaseManager' instance in a given Unity scene. This ensures that the relevant data is properly converted and passed to and from the Julia process. The 'JuliaInstallationManager' and the 'JuliaModelHandler' can be extended with an instance of the 'IJuliaPluginDebugger', allowing them to provide feedback on the state of the Julia installation and the Julia process. The 'JuliaModelHandler' provides two serialised lists in the inspector for naming Julia functions and modules. Entries in these lists are then referenced in a dictionary data structure and can be called using the 'InvokeReferencedFunction(...)' methods of the 'JuliaModelHandler' class. To make the Julia program code accessible, its files must be placed in the 'JuliaScripts' folder and be listed by full name in the file 'Inclusions.txt' file. They are loaded by the 'JuliaModelHandler' class calling the 'LoadModelDependencies()' method.

The Julia Unity Package is open source and available on GitHub (`https://github.com/NicoManke/JuliaUnityPackage`). Template versions of the required text and Julia files are available in the templates folder. The package tutorial and documentation are located in the documentation folder.

3. Use Case: Macrophyte growth simulation

To demonstrate the Julia Unity Package, we present an immersive application for visualising an eco-physiological plant growth model, the Macrophyte Growth Model (MGM) [LHH*22]. Macrophytes—aquatic plants large enough to see with the naked eye—are essential for the functioning of freshwater lakes as they provide multiple ecosystem services like oxygen production, nutrient cycling and habitat for other aquatic organisms.

3.1. Macrophyte Growth Model

MGM [LHH*22] is a depth-explicit mechanistic model that simulates the life cycle of submerged macrophytes from germination

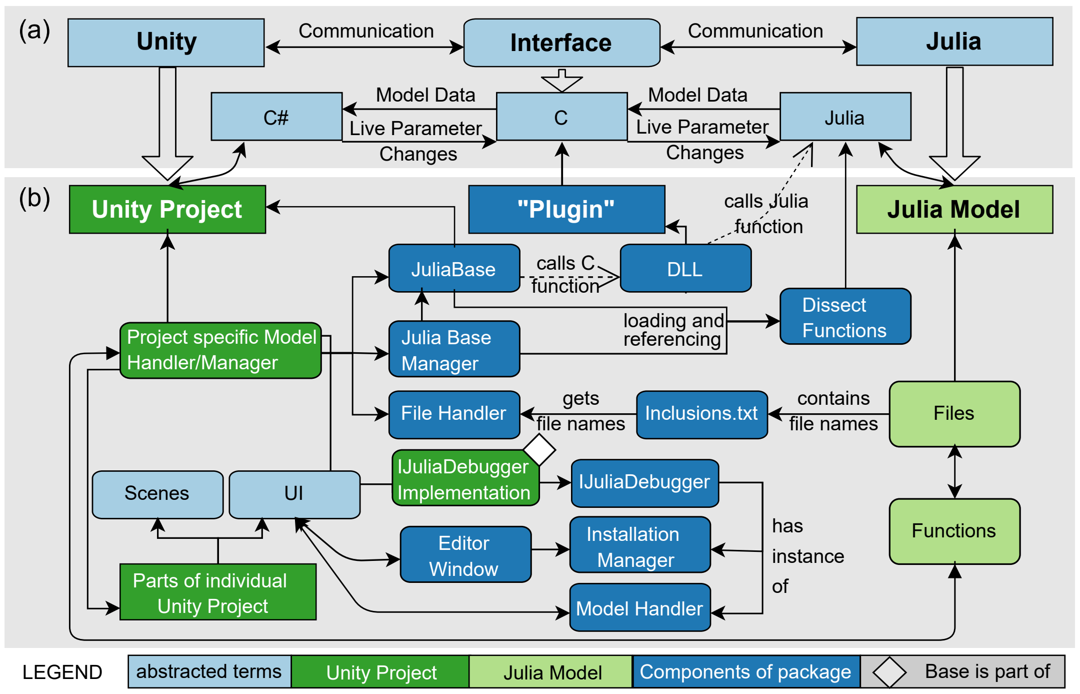

Figure 1: *The Julia Unity Package's structure and its components' function as mediators between a given Unity project and Julia model. At a high level of abstraction, the dependencies between Unity, the Interface, and Julia are shown (a). Meanwhile, at a medium level of abstraction, the technical components involved, such as the Plugin, and their interactions are revealed (b).*

through vegetative growth to reproduction. The daily growth rate of each individual is a function of photosynthesis and respiration and is primarily dependent on the light availability, nutrient availability, and water temperature. The water temperature follows seasonal changes. Light availability is a function of depth, seasonal changes in solar radiation, and turbidity of water. The model output includes time series of biomass, height, number of individuals, and individual weight of the simulated species per simulated depth. The model output is written to the console and to text files.

3.2. MGM Simulation

The combination of the Macrophyte Growth Model and the Julia Unity Package led to the creation of an interactive demo application, the MGM Simulation, which has a main menu and a lake landscape visualisation showing the simulation results (Fig. 2 (a)). Although MGM Simulation is a visualisation tool, pure visuals were not the main focus of this implementation. Therefore, the assets used are not state of the art. Instead, it was more important to translate the data into realistic growth behaviour and to test the Julia plugin. The translation required additional functionalities to be implemented. The 'PlantGenerator' class was created to control the distribution, growth, and death of plants. The 'WaterManager' class was created to display changes in water temperature and quality. The user can set up the Julia installation path, select a species and lake from a preset list, run the model, and enter the scene. The lake landscape shows the selected species and lake output at different depths. Different species have different eco-physiological parameters that make them grow differently at the same depth, which the Unity model helps to visualise (Fig. 2, (b-d)). The daily environmental factors relevant to the simulation, such as the water temperature, are also displayed as numbers on the screen. To allow control of the MGM simulation at runtime, we have provided an easy-to-use graphical user interface, as part of the 'UI' (Fig. 1 (b)). We could not rely on the editor window of the Julia Unity Package, as it is not available after building the standalone application. The duration of the visualisation for a day can be set by the user, and ranges from 0.25 to 5.0 seconds. Advanced users can modify the input files to add other species and lakes to the simulation or adapt the model code.

4. Conclusion & Future Work

The proposed Julia Unity Package is a step towards better immersive visualisations for Julia-coded simulation models, but it has two drawbacks. It requires the user to install Julia, which makes the distribution of applications built with the package inflexible. This could be solved by using a portable version of Julia or the data models written in Julia, which do not require a Julia installation on the user's machine. Another solution could be to compile the Julia model using the 'PackageCompiler.jl' Package [CDT23]. In addition, long computation times can freeze the application, but multithreading can divide the workload and speed up computation. The second limitation did not have a major impact on the MGM simulation, as the data calculation for one lake and species only took seconds on gaming and office PCs. Therefore, graphical notification

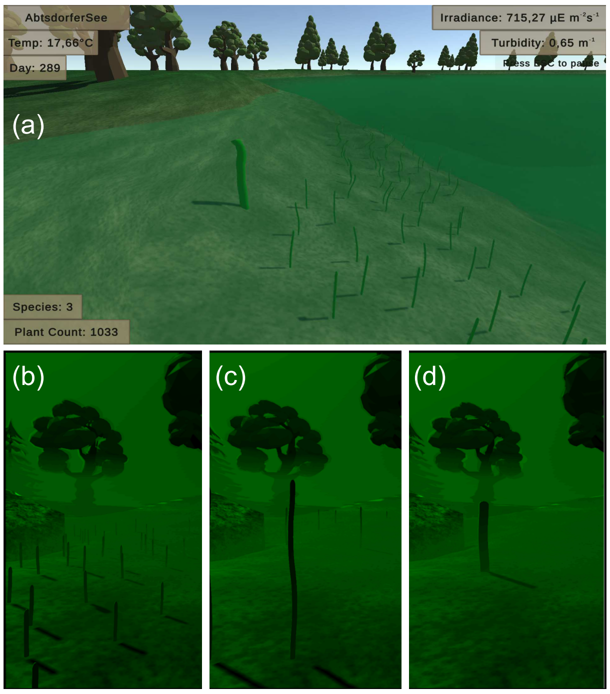

Figure 2: *The landscape employed in the MGM simulation is shown (a). The growth of three different species in the same lake is compared: a typical oligotrophic species that forms dense stands (b), a mesotrophic species with a higher growth rate (c), and an eutrophic species producing a higher biomass per individual (d).*

of the loading state was considered sufficient. The first limitation results in longer set-up and preparation times for presentations and tests on new machines, especially if Julia was not installed. This hampers the use and reduces its one-time appeal.

We propose combining Julia and Unity to create immersive visualisations of process-based ecological models. With the MGM simulation we show an exemplary use case. Compared to individual technical solutions such as bioDIVERsity [LSG*21], the plugin provides a reproducible workflow that can be used for various other applications. As Julia is a relatively young language, there is a lack of such plugins. Therefore, we hope that the proposed plugin will help to use and translate Julia models into interactive tools, for outreach, communication, education, and gaming.

Resources

- Julia Unity Plugin: `https://github.com/NicoManke/JuliaUnityPackage`
- Trailer: `https://youtu.be/sJ3bboF1WCY`
- MGM simulation: `https://github.com/NicoManke/MGM_Simulation`

Acknowledgment

We thank Julian Gröh and Andreas Knote for their ideas during the development of bioDIVERsity, which inspired this project. Furthermore, we thank anonymous reviewers for their helpful advice and comments on a previous version of the article. AL and JSC acknowledge funding by the Bavarian Ministry of Science and the Arts in the context of the Bavarian Climate Research Network (BayKliF).

References

[BEKS17] Bezanson J., Edelman A., Karpinski S., Shah V. B.: Julia: a fresh approach to numerical computing. *SIAM Rev. 59*, 1 (Jan. 2017), 65–98. `doi:10.1137/141000671`. 1

[CBL21] Castelo-Branco R., Brás C., Leitão A. M.: Inside the matrix: immersive live coding for architectural design. *International Journal of Architectural Computing 19*, 2 (June 2021), 174–189. `doi:10.1177/1478077120958164`. 1

[CDT23] Carlsson K., Danisch S., Trevisani L.: JuliaLang/PackageCompiler.jl: Compile your Julia package. https://github.com/JuliaLang/PackageCompiler.jl, 2023. 3

[CVH17] Cabral J. S., Valente L., Hartig F.: Mechanistic simulation models in macroecology and biogeography: state-of-art and prospects. *Ecography 40*, 2 (2017), 267–280. `doi:10.1111/ecog.02480`. 1

[DK21] Danisch S., Krumbiegel J.: Makie.jl: flexible high-performance data visualization for Julia. *Journal of Open Source Software 6*, 65 (Sept. 2021), 3349. `doi:10.21105/joss.03349`. 1

[FPA*19] Fritzson P., Pop A., Asghar A., Bachmann B., Braun W., Braun R., Buffoni L., Casella F., Castro R., Danós A., Franke R., Gebremedhin M., Lie B., Mengist A., Moudgalya K., Ochel L., Palanisamy A., Schamai W., Sölund M., Thiele B., Waurich V., Östlund P.: The OpenModelica integrated modeling, simulation, and optimization environment. In *The American Modelica Conference 2018, October 9-10, Somberg Conference Center, Cambridge MA, USA* (Feb. 2019), pp. 206–219. `doi:10.3384/ecp18154206`. 1

[Haa14] Haas J. K.: A history of the unity game engine. *Diss. Worcester Polytechnic Institute 483*, 2014 (2014), 484. 1

[JAM*16] Jamadagni C., Anilkumar A., Mathew K. T., Mulimani M., Koolagudi S.: Dynamic 3D graph visualizations in julia. In *Proceedings of the Summer Computer Simulation Conference* (San Diego, CA, USA, July 2016), SCSC '16, Society for Computer Simulation International, pp. 1–8. 1

[LHH*22] Lewerentz A., Hoffmann M., Hovestadt T., Raeder U., Cabral J. S.: *Potential change in the future spatial distribution of submerged macrophyte species and species richness: the role of today's lake type and strength of compounded environmental change*. Preprint, unpublished, May 2022. `doi:10.22541/au.165401091.12520929/v1`. 2

[LJ02] Lewis M., Jacobson J.: Game engines. *Communications of the ACM 45*, 1 (2002), 27. 1

[LSG*21] Lewerentz A., Schantz D., Gröh J., Knote A., von Mammen S., Cabral J. S.: bioDIVERsity – a computer game to face the image problem of aquatic plants. *Mitteilungen der Fränkischen Geographischen Gesellschaft*, 67 (2021), 29–36. 1, 4

[NO20] Neumayr A., Otter M.: Modia3d: modeling and simulation of 3d-systems in julia. In *Proceedings of the JuliaCon Conferences* (2020), vol. 1, The Open Journal. 1

[Pau07] Paulson L. D.: Developers shift to dynamic programming languages. *Computer 40*, 2 (2007), 12–15. 1

[Ric14] Richter M.: *User-centred engineering: creating products for humans*. Springer, New York, 2014. 2

[RN11] Rothrock L., Narayanan S. (Eds.): *Human-in-the-loop simulations: methods and practice*. Springer, London, 2011. `doi:10.1007/978-0-85729-883-6`. 1