UNIVERSITY OF TARTU

Faculty of Science and Technology

Institute of Computer Science

Computer Science Curriculum

Mateus Surrage Reis

# Profiler Improvements for the Godot Game Engine

Master's Thesis (30 ECTS)

Supervisor(s): Jaanus Jaggo, MSc

Tartu 2023

## Profiler Improvements for the Godot Game Engine

**Abstract:**

Godot is an open-source game engine, a large, complex and growing open source software project. As one, it's in constant need of improvement and volunteers to contribute to the codebase by adding features and fixing bugs. The goal of this thesis is to improve the current built-in code profiler in Godot. The main improvement made was the repair of a major bug in the profiler. This bug caused all internal Godot functions to fail to be reported on, and the time spent in them to disappear from the profiler. The actual cause of the bug was the lack of a mechanism to profile these functions. The fix was accomplished by adding such a mechanism to the existing profiler, enabling previously unavailable information gathering. In addition, two related profiler usability improvements were contributed: sorting entries by column and plot zoom/pan. These improvements were completed and contributed back to the project as pull requests.

**Keywords:**

Game engines, Godot, debugging, profiler, open source

**CERCS:**P170 Computer science, numerical analysis, systems, control

## Godot mängumootori profileerija täiendamine

**Lühikokkuvõte:** Godot on avatud lähtekoodiga mängumootor, mis oma suuruse, keerukuse ja kiire kasvu tõttu vajab palju vabatahtlikke, kes seda täiendaksid ja vigasid parandaksid. Käesoleva töö eesmärgiks on täiustada Godot mootori sisseehitatud koodiprofileerijat, parandades selles esinev oluline viga. Selle vea tõttu ei raporteerita aega, mis kulub mootori sisemiste funktsioonide käivitamiseks. Kuna Godot mootoril puudus varem mehanism selliste funktsioonide profileerimiseks, on töös lisatud funktsionaalsus puuduva informatsiooni kogumiseks. Lisaks on täiendatud profileerija kasutatavust, lisades sellele võimalus kirjete sorteerimiseks ja graafiku suumimiseks ning nihutamiseks. Lõpuks tehti Godot projekti pull-päringud, et valminud täiendused sisse viia.

**Võtmesõnad:**

mängumootor, Godot, silumine, profileerija, avatud lähtekood

**CERCS:**P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

# Contents

# 1 Introduction

The Godot engine is a free and open source engine for development of video games, managed by the Godot Foundation. It's currently growing rapidly in popularity and undergoing active development. In contrast to its most significant competitors, it's licensed with the permissive MIT open source license, allowing developers to use it with no limitations and very few requirements. In combination with the open, active, community-driven development and feature set competitive with leaders in the market, this makes Godot an attractive engine for new and hobbyist developers interested in open source.

Being open source means anyone is free to modify and contribute code improvements, after a thorough review process. This thesis contributes to the development with one major and two minor code contributions. The larger, main contribution fixes a major bug wherein the built-in profiler only profiled user-written functions. The functionality of the profiler is then expanded to include most built-in Godot functions.

The Godot engine, like most game engines, offers users a large set of functions, its API. These functions are built to allow game developers to interact with the internals of the engine, such as applying a force in the physics engine or playing a sound. On top of these functions, it has its own scripting language and environment, allowing users to write their own code from where they may call the API functions. The bug in question caused none of these API functions to be profiled, and for time spent in them to fail to be recorded in the profiler. This text goes over this bug itself, the internal issues that caused it, the processes of diagnosing and fixing it, the additional improvements, and finally describes the processes followed to contribute these changes back to the Godot Project.

Aside from the main bug fix, the two other minor contributions relate to the profiler user interface. The first improves the plotting feature by adding intuitive zoom and pan, enabling users to visually pinpoint problem areas. The second adds the ability to quickly sort certain fields in the profiler's lists, allowing quicker searching. These were seen as straightforward additions to profile usability, and served as an introduction to the process of open source contribution. Finally, a short description of of open source contribution in general and how it played out in the case of this work are offered, along with some impressions regarding it.

# 2 Background

This section discusses background in two parts: firstly, for broad context, what is a game engine and a short history of them, a profiler, and Godot. Secondly, Godot's internals and the relevant parts of its architecture are explained as much as necessary to understand the improvements made. Those parts are GDScript, the GDScript VM, Godot servers and the Godot profiler.

## 2.1 Game Engines

At the outset of the medium of video games, in the early 80's era of the Atari, games were generally created from scratch, with few if any libraries or reused code. Compared to present times, games were far simpler, consumer expectations were lower and the hardware more restrictive, which allowed very small teams of programmers to create full games in short development cycles. Games fitting this description are Centipede(1981), Donkey Kong(1981), and Karate Champ(1984). In this period, the forefront of videogame technology at the time was the arcade, in which software had to be very specifically written for both the game in question and the hardware it ran on [Gre09, p. 3][And15]. This meant developers were unable to reuse code. Throughout the eighties, games remained mostly "arcade-style" even on the home console: with a fixed number of levels and rule sets, generally with only the goal of getting the best score [Wil17, p. 72]. This status quo allowed development to stay relatively simple. It was viable for games to be built from scratch each time.

However, even early in this period, companies kept around code and development tools in the same spirit of what would eventually become game engines, such as Nintendo's screen-scrolling machinery for the NES [Wil17, p.153]. Undoubtedly many other companies' developers had pieces of code they'd reuse for similar games. Another example of early attempts in this direction were "game construction kits" available for the 1982 Commodore 64 home computer, though they were limited by technology and usually specialized genre-wise in what they could produce. Game engines near to their current form only came into the fore during the 90s, with the popular first-person-shooters Doom(1993) and Quake(1996) [Gre09, p. 11], and their eponymous engines.

With the gradually increasing complexity of both hardware and software, it was practical for developers to separate the basic technical functions of their games (elements such as physics, input, graphical rendering and audio) from the game content (such as enemies, sprites, game logic and narrative). When cleanly separated, it was possible to license only the former to other developers, so they could create their own original games with significantly lowered technical burden. It became so easy that even common users were able to make such modifications, facilitating the related phenomenon of modding, a process of user creation that allowed users to add and substitute content in a game [Ung12].

Figure 1 shows Doom and Hexen, a game developed using a modified version of the Doom engine. Note the similar perspective, graphical fidelity and UI, and differences in asset and theme.

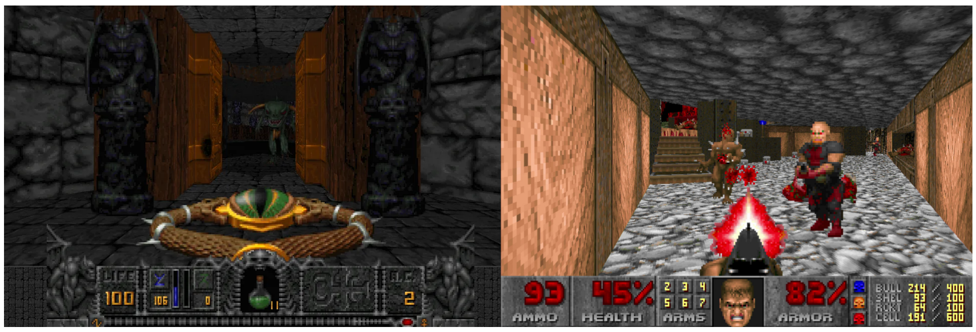


Figure 1. Doom and Hexen, built on the Doom engine.

Since the 90s and the above examples, game engines have grown greatly in complexity, popularity and market influence, but from this short history alone it's possible to extract a minimal working definition for the essence of a 'game engine': some piece of software that composes part of a game and facilitates development. The purpose is to be able to reuse code and indeed chunks of entire software engineering infrastructure in the development of games. A game engine may be narrow in scope, serving to create only games in a specific genre. But the popular and widely-used game engines discussed here are general-purpose, designed to be able to assist in the creation of any kind of game.

Some of the basic functionalities such a game engine commonly provides or assists with include [Gre09, Chapter 1.6]:

- A graphics rendering system and its connection to the hardware.
- Managing the 'main loop' of the game: the system that calculates game events every frame.
- A GUI development framework, often a hierarchical scene graph.
- Interaction with the operating system's input and windowing system.

• A physics and collision calculation engine.

Accordingly with their growing importance in the game development sphere, the scope of the most popular game engines have progressively grown wider. Most notable has been the inclusion of full Integrated Development Environments (IDE), including code editing facilities and full-featured Graphical User Environments (GUI) for managing all of the engine's respective components synergistically. These are sometimes referred collectively to as Software Development Kits, or SDKs. Not all game engines include SDKs, but the most popular ones (and Godot) do.

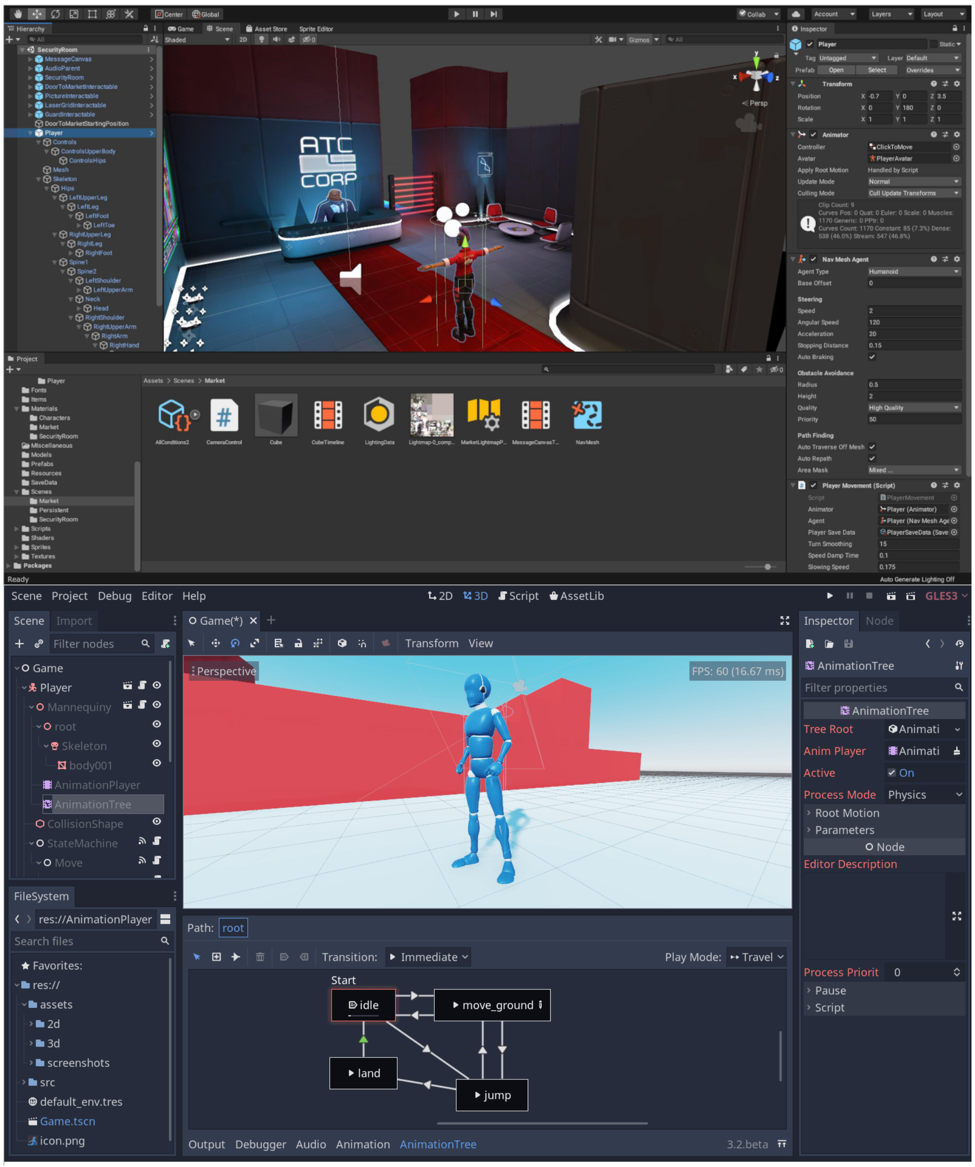


Figure 2. Main development workspaces of Unity and Godot.

The most immediately relevant examples of game engines for comparison with Godot are Unity and Unreal engine. Figure 2 contains screenshots of the main development spaces for

Unity (on top) and Godot (on the bottom). The user develops the game by a combination of programming in the available languages and managing their relationships and properties through the GUI. These engines are similar in that they're all intended for general-purpose creation of games without specific genres or platforms in mind, and are also available for wide release for amateur developers without up-front costs. As such, they can be considered direct market competitors. As part of the SDK, popular engines also include support for more specialized, less essential features, such as:

- Building and distributing a game for multiple platforms.
- Networking and multiplayer.
- Translation and localization.
- In-game advertising (usually for mobile games).
- Rudimentary asset creation/editing (e.g. models and sprites).
- Profiler and debugging facilities.

## 2.2 Profilers

Debugging is difficult [PSTH17], having been the subject of much research and attention since the earliest days of computer science. Accordingly, the types of debugging tools are many and varied, often specific to a field, a tool, or a programming language.

"Profiler" is a general term for a subset of these tools which targets a program and collects its information during its execution. The exact nature of this information varies significantly depending on the profiler. It may measure run time, CPU usage, subroutine call counts, memory use, or other resource usage. This may happen at a program level or at a subroutine level.

One common purpose for a profiler is as an aid to diagnose speed and performance issues. When it comes to game software, this type of analysis is highly relevant: games have particular performance needs when compared to most other software. They must be calculated and rendered in real time due to the requirements of interactivity. Framerate (that is, the number of times the game is recalculated and displayed in one second) has a severe impact on player enjoyment, and can render a game unpleasant or unplayable in cases where it's too low [CC07].

Following in the footsteps of the early gprof(1982) [GKM82], profilers often have the means to produce a full call graph, a graph detailing the relationships between all function calls happening in the program. Such a feature is important enough that profiles that include it are termed call graph profilers. The alternative, a profiler that doesn't record a full call graph but only the time of each function successively, is called a flat profiler.

Though a profiler is non-critical in game development, it's telling that all of the predominant game engines on the market (e.g. Unity, Unreal, Godot, GameMaker) offer them out-of-the-box. Aside from being useful for performance improvement in game development in general, one consideration is that game development could attract a high amount of inexperienced developers compared to other types of software. This would lead to a higher frequency of performance-degrading programming mistakes, such as inefficient loops or subroutines being called more often than they need to. Use of a profiler would ease the diagnosis of these issues for new users.

Profilers have significant variety, due partially to the corresponding variety in potential application areas, target profiled software and types of issues to be diagnosed. Two examples of profilers and the diverging visual way they can display information are found in figure 3. On the left is one component of Valgrind, a popular and full-featured stand-alone kit of debugging tools, including a full call graph profiler and memory analysis tools. On the top right is the built-in profiler for the Unity game engine, Godot's main competitor, showing cpu usage and function call graphs. They may also be specialized to target only specific languages or frameworks, as is the case with Godot's profiler and Unity's profiler.

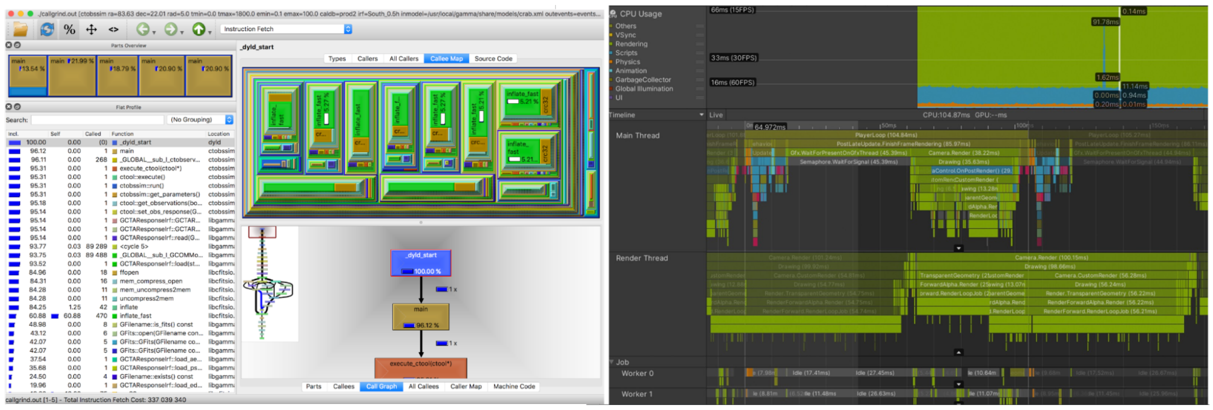


Figure 3. Valgrind memory profiler (left) and Unity's built-in profiler (right).

## 2.3 Godot

Godot is an open source game engine. As mentioned, it's a set of complementary tools and a code framework that eases the development of video games, and includes many of the components mentioned in the game engine section above.

Godot was originally started in 2007 by Juan Linietsky and Ariel Manzur for several companies in latin america. Its source code was released on GitHub under the permissive MIT license in 2014, and it has seen slowly accelerating growth since then. In 2015 Godot joined the Software Freedom Conservancy, a non-profit organization supporting open source projects.

As of this writing, Godot lags behind its main competitors in some features. However, its main distinguishing point is the fact that it's open source with a permissive license. Its source code is available and can be freely used, built upon and modified without any licensing restrictions. This could be a strong motivator for its growth.

Motivations for developers joining an open source project have been written about at length: *intrinsic* non-monetary motivations, such as enjoyment, a challenge, or the desire to give something to 'the community' are often explored [BSS07]. These motivations could be particularly strong in the case of a game engine. This is because video games have been widely consumed cultural products for decades, and are likely to have been a strong presence in the lives of aspiring game and game engine developers.

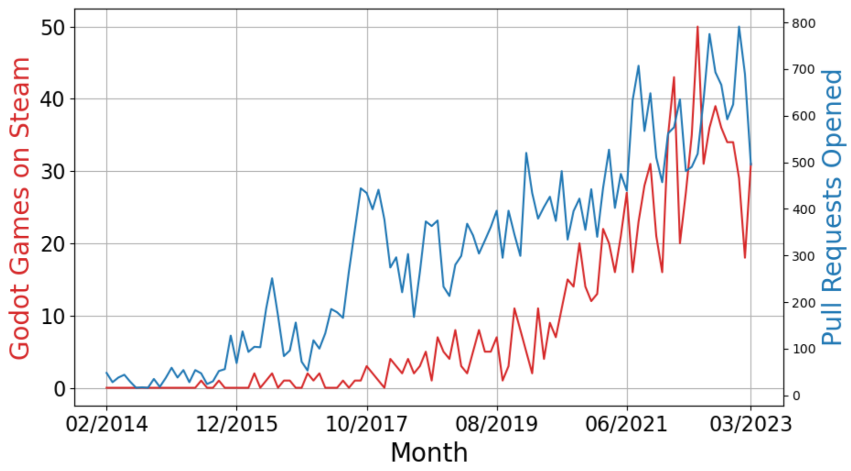


Figure 4. Pull requests opened on Godot's Github repository and games created using Godot engine released each month. Game release data from SteamDB, pull request data retrieved from GitHub.

As the most visible open-source offering in the competitive field of game engines, Godot has undergone much growth in the past few years, as seen in figure 4. It's seen a 60 to 80 percent year-over-year growth in number of games built with it in the Steam distribution platform [Lin23]. Similar levels of growth are happening in the activity around the engine development itself, as measured by GitHub pull requests and issues. The engine has recently reached a development milestone (4.0) which added many new features, and is now pivoting to focus on usability and performance improvements.[1]

### 2.3.1 Godot Internal Basics

Godot achieves its general-purpose nature with a collection of abstractions that encapsulate arbitrary data, operations, and even objects like whole scripts. These abstractions are such that, for example, a function written in Godot's native GDScript, in the supported C# language, and a built-in Godot C++ function all execute in the same internal environment, and must be treated similarly by the engine. Scripts and operations are marshaled into Godot's own internal representation and ultimately executed by the same machinery. Figure 4 Illustrates this idea: The middle layer of boxes consists of tools that transform some outside code into objects compatible with Godot.

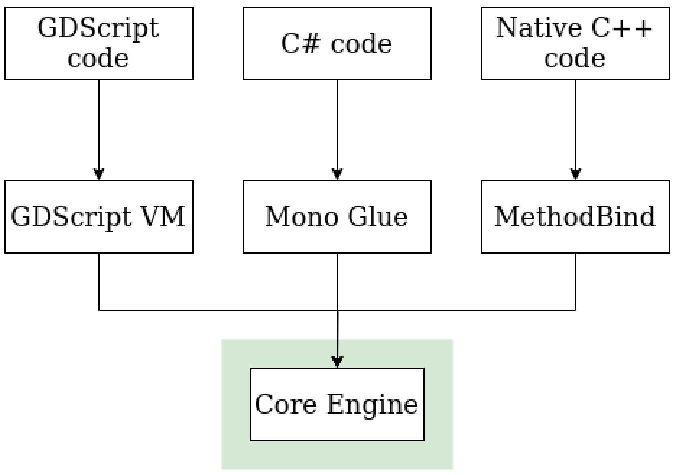


Figure 5. Illustration of the flow needed to adapt 'outside' code into Godot-compatible operations.

Of Godot's abstractions, the most important, in the core of the engine, is the Variant. This is a class that wraps around arbitrary data in the Godot engine, allowing one class to contain one of a number of pre-defined types. It supplies fundamental low-level operations common to every Godot type. For instance, among others, Variant:

- Defines serialization and deserialization to and from bytes or strings.
- Defines which types are allowed to have gettable and settable properties.
- Defines which types are callable.
- Defines how to construct and destruct types.

[1]Official blogpost: `https://Godotengine.org/article/Godot-4-0-sets-sail/`

In Godot's own scripting language GDScript, Variant is the default type for all variables. Consider the following valid GDScript block of code:

```
var some_variable = 5
some_variable = "now it's a string"
some_variable = Object.new()
```

Here the variable some_variable seems to be initialized as an integer, and then changed to a string and then to an Object. But in fact it's a single variant type, in turn containing each of these types. It's possible to do these dynamic type operations atop an engine written in the statically-typed C++ because of the Variant type.

Variant and its large web of interconnected operations between supported types are the core of the Godot engine. Other more specific abstractions are built on it. For instance, Callable, to represent any callable entity, and Object, to represent an object-oriented style inheritable object. The rest of the necessary parts of a game engine are then written using these abstractions as a base.

### 2.3.2 GDScript and the GDScript VM

GDScript is an interpreted, dynamically typed programming language with support for object oriented programming features. It's designed to be easy to learn and to tightly integrate with the Godot editor and engine, and is built specifically to serve as a language for game development. This tight integration with the editor and engine isn't limited to surface language syntax. It goes down to the way the language is interpreted and executed, mirroring the internal workings of the Godot engine closely, for optimization and ease of development. Concretely speaking, the developers found that other scripting languages that could potentially be embedded into Godot had serious issues in their use case. Some examples of such issues include poor support for threading, inefficient garbage collectors and a lack of native vector types. Given these and other issues, they decided creating their own embeddable scripting language was a superior option.

GDScript works 'on top' of the engine, being technically an optional module. It would be possible to create a game in Godot without a single line of GDScript. However, it's treated as the default scripting language and users are encouraged to use it.

GDScript code written in the editor by a user is first parsed, analyzed and compiled into bytecode. This bytecode consists essentially of a long string of opcodes and pointers. At run time, it's fed through a large set of functions (the GDScript VM) that validates and interprets it. Following that, the operations indicated in the bytecode are steadily delegated to lower-level engine machinery and executed. An opcode tells the virtual machine what code to execute with what information present, not unlike real instruction set opcodes. Below are some examples of opcodes used in the GDScript VM.

Table 1. Examples of opcodes used in the GDScript VM and the operations they express.

| **Opcode name** | **Operation** |
|---|---|
| OPCODE_ASSIGN | Regular assignment, in the form identifier = expression. |
| OPCODE_RETURN | Return from a function, with or without a value. |
| OPCODE_ITERATE_BEGIN_ARRAY | Begin iterating through the elements of an iterable, such as a list. |
| OPCODE_CALL_BUILTIN_STATIC | Call a static function on a built-in Godot type, such as Vector3 or Basis. |
| OPCODE_CALL_ASYNC | Generic call opcode, awaited as a coroutine. |
| OPCODE_CALL_METHOD_BIND | Call with a MethodBind, a mechanism used to bind C++ functions into a Godot engine understandable object. |

### 2.3.3 Godot Server

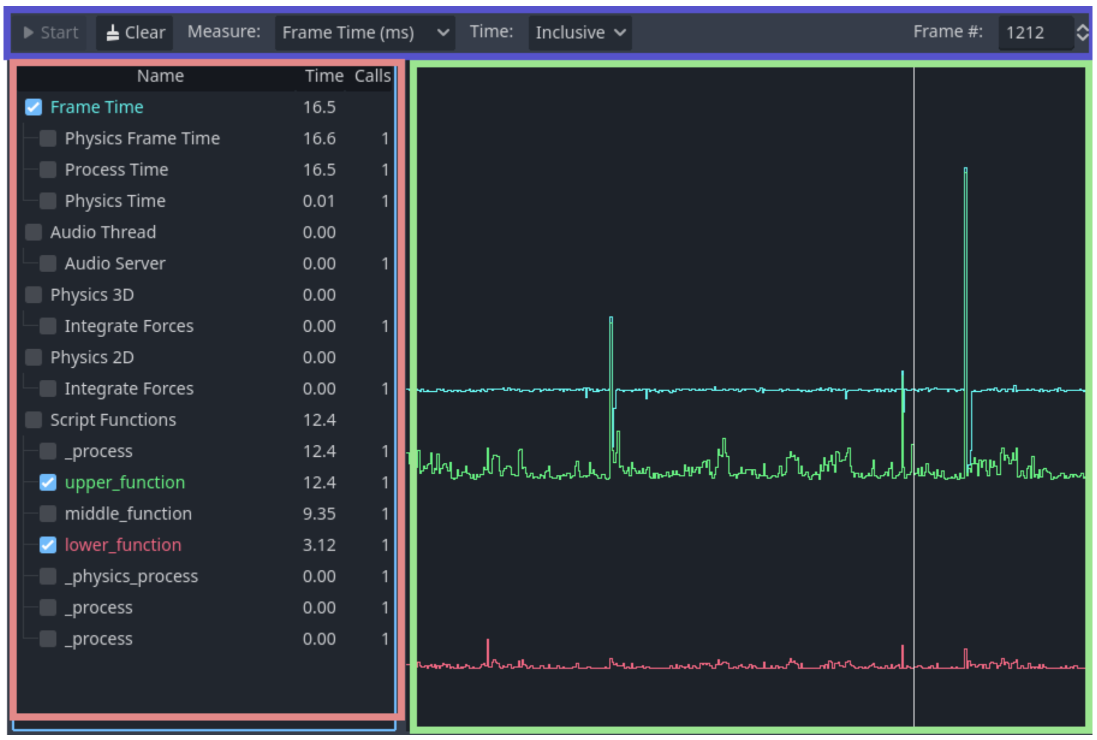
Servers are a part of Godot's abstraction, primarily as a way to enable multithreading. They're an implementation of the mediator design pattern [GHJV94]: a Server, such as a PhysicsServer or an AudioServer, is a class that contains its own set of information and state. It processes requests that come exclusively through a single command channel, runs in parallel, and doesn't allow direct user manipulation. Because their processing runs in separate threads and only involves C++ internal engine code, information about what happens inside the Servers is much less granular on the profiler, collected and displayed separately. Direct requests from the user to the server may be slow, as a call could force the server to process all pending threaded operations before getting to the user's command. Due to the potential slowness of interacting with servers, those API functions were the first to be noticed by users to be missing from the profiler, as more time was spent in them.

### 2.3.4 Godot Profiler

Similarly to the rest of the Godot development tools, the general profiler is built directly into the editor. Users can activate it when running their game from within the editor and get instantaneous ongoing time information. Godot also has a "Visual Profiler", for GPU and rendering operations, and a "Network Profiler". This thesis is focused only on the general CPU profiler, which will just be called the ‘profiler’ from here on.

Godot’s profiler is currently intended for narrow use. It primarily collects time information on the user’s subroutines. The main purpose of the design is to assist game developers in diagnosing poorly-performing sections of code and fix them. Poorly-performing, in this context, means slow code, resulting in unpleasant low framerates and slowdowns for the ultimate consumer of the game.

Unlike many common profilers including gprof, Godot’s profiler isn’t a full call graph profiler, as its machinery isn’t embedded at a low enough level in Godot to capture full call graphs. It’s specifically embedded into the GDScript VM, not in the core of the engine. As a consequence, other profilers have to be implemented separately per supported language — for instance, Godot supports C#, but a profiler for it hasn’t been written into Godot as of yet.



| Name | Time | Calls |
|---|---|---|
| ✓ Frame Time | 16.5 | |
| Physics Frame Time | 16.6 | 1 |
| Process Time | 16.5 | 1 |
| Physics Time | 0.01 | 1 |
| Audio Thread | 0.00 | |
| Audio Server | 0.00 | 1 |
| Physics 3D | 0.00 | |
| Integrate Forces | 0.00 | 1 |
| Physics 2D | 0.00 | |
| Integrate Forces | 0.00 | 1 |
| Script Functions | 12.4 | |
| _process | 12.4 | 1 |
| ✓ upper_function | 12.4 | 1 |
| middle_function | 9.35 | 1 |
| ✓ lower_function | 3.12 | 1 |
| _physics_process | 0.00 | 1 |
| _process | 0.00 | 1 |
| _process | 0.00 | 1 |

Figure 6. Godot’s main profiler interface. Blue: control area. Green: plotting area. Red: information area.

In figure 6 the Profiler display can be seen, with three distinct areas. Most important is the textual information area, in red. Of the higher-level categories ‘frame time’, ‘audio thread’, ‘physics 3D’, ‘physics 2D’ and ‘script functions’, only frame time and script functions are relevant for the purposes of this text. Physics and audio refer to measures of internal server calculations that can only be manipulated indirectly. The area under ‘Frame Time’ refers to measures that are done on an entire frame, rather than being derived from specific functions or portions of it. Finally, under 'script functions' is where data about an user's function calls are displayed.

Aside from the red information area, there is the control area in blue, where certain settings can be changed on the fly, and the plotting area in green, where the user can visually inspect the performance of their game and select specific frames. Y axis is time taken, X axis is time, with each X unit representing one frame.

As it is the main subject of this thesis, there are certain concepts within the profiler that must be explained for a clear understanding of the main bug and how it was fixed. These are self time, inclusive time, frame time, and script time.

**Self and inclusive times** The profiler display can be switched between "**self time**" and "**inclusive time**", and these refer to two alternative ways of seeing the time spent inside a function. See figure 7 for an illustration of the difference between these two time displays.

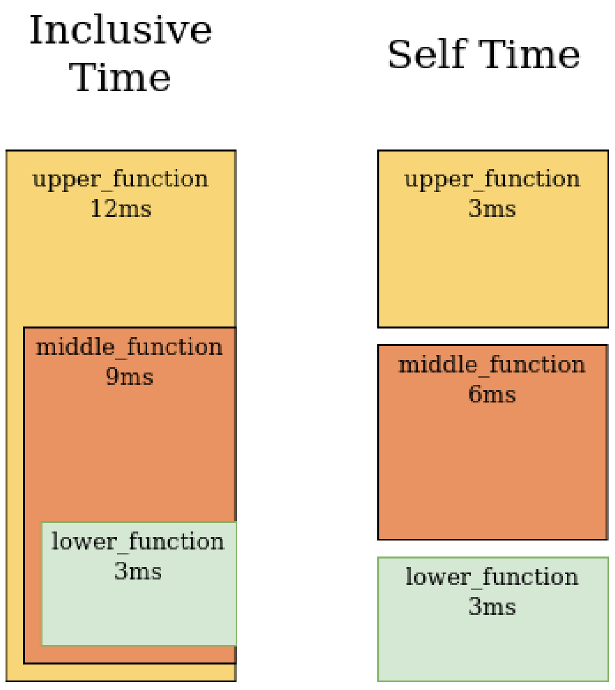


Figure 7. The difference between self and inclusive time in the profiler, illustrated. Left and right are different representations of the same 3 nested function calls.

In figure 7, the same function call is shown as it would be conceptualized with each type of time display, inclusive time on the left and self time on the right. Each differently-colored box represents a separate function call. In pseudo-code, these would be the example functions being represented by the figure above:

```
func upper_function() {
    //Do some operations that take 3ms
    middle_function()
}
func middle_function() {
    //Do some operations that take 6ms
    lower_function()
}
func lower_function() {
    //Do some operations that take 3ms
}
```

Inclusive time is a simple stopwatch measure from start to end of a function, and can be considered reliably correct. In 'inclusive time', all time spent in the function is represented. For instance, the inclusive time of 12 ms spent in the upper_function also 'includes' time spent in middle_function and lower_function.

Self time is a more elaborate measure, found by subtracting the inclusive times of all sub-functions from a function's inclusive time. In theory, this would show only the time spent inside the function's own code, excluding sub-calls. For instance, the upper function's self time is its inclusive time (12ms) minus the inclusive time of the middle function (9ms) = 3ms, matching the time spent in the function operations. This concept of self time is intended to pinpoint only the specific function creating performance problems.

**Frame and Script Times** There are two other relevant time metrics. Unlike the last two, they're meant to represent time spent in an entire frame, rather than in functions: script time and frame time. Also unlike the last two, they're separate measures of different processes, rather than alternate ways of seeing the same measure.

Figure 8 illustrates the difference between script and frame time. The topmost line represents an entire frame of Godot execution in the main game thread from left to right. Each coloured box along the line represents some function being executed, and its length represents the time spent in it.

Similar to inclusive time, frame time is a simple, real stopwatch measure of the time from the start to end of frame, that is, $T_{frameend} - T_{framestart}$. Due to the simplicity of the measure, frame time is a reliable measure of the real time spent in a frame. In this figure, frame time is

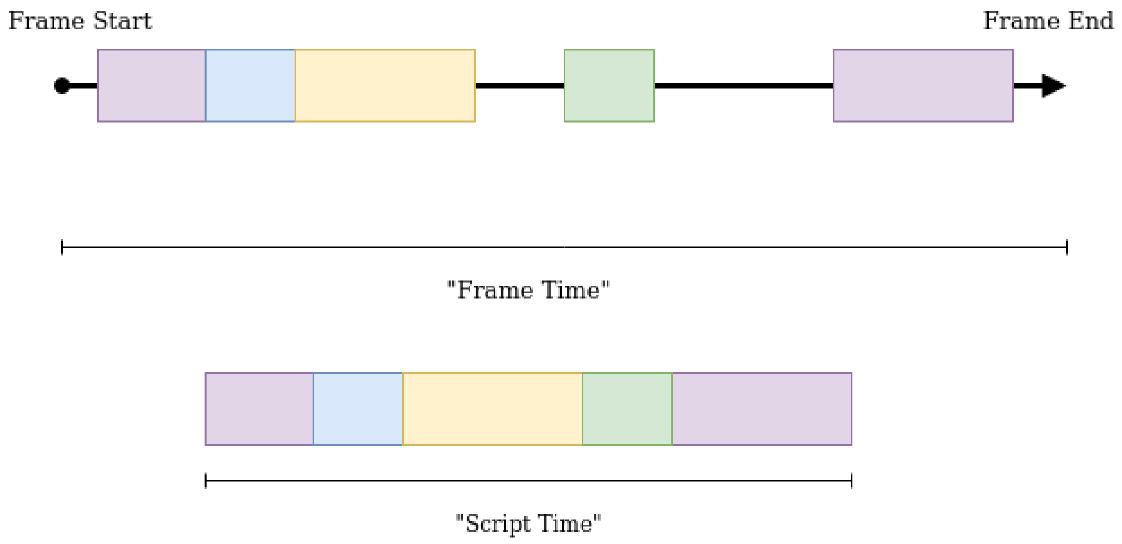


Figure 8. Illustration of the concepts of ’Frame time’ and ’Script time’

represented by the length of the frame arrow.

Script time, on the other hand, is the sum of independently calculated self times of individual functions. In this figure, this is represented by putting together all ‘function boxes’ and measuring their joined length.

In the relevant case where scripts are the performance bottleneck, these times should be equal in a theoretical bug-free profiler. The discrepancy between them was one of the ways the issue presented itself.

Note that it’s possible for there to be discrepancies between frame and script times even while the profiler is working properly, in normal operation. For instance, if very expensive internal physics calculations are happening in a separate thread and must be waited on, increasing frame time. However, this type of case will be ignored, since the issue being examined deals exclusively with script profiling. Specifically, with the case where a user’s scripts are slow and bottleneck the performance of the application.

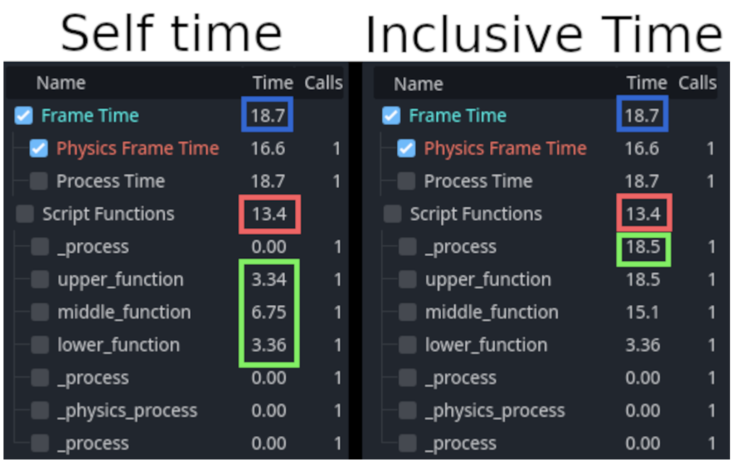
# 3 Contributions

In the following section the issues discovered and corresponding improvements contributed are described: The main issue described as seen from an user's perspective. Then, the way the issue presented internally, how it was fixed, and complications involved. Finally, a pair of related profiler usability improvements are presented.

## 3.1 Main bug description

A major profiler bug had been found and described in Godot's repository's issue section, where all issues and bugs are logged by the open source community and reviewed by the core developers. Multiple users had noted the problem since 2018.[2]

The issue as described was the loss of time when using functions that interacted with any of the Godot servers. The time spent inside those functions completely disappeared from part of the profiler, leaving it in an inconsistent state. The problem also manifested differently in self time and inclusive time display modes.



| Name | Time | Calls |
|---|---|---|
| ✓ Frame Time | 18.7 | |
| ✓ Physics Frame Time | 16.6 | 1 |
| Process Time | 18.7 | 1 |
| Script Functions | 13.4 | |
| _process | 0.00 | 1 |
| upper_function | 3.34 | 1 |
| middle_function | 6.75 | 1 |
| lower_function | 3.36 | 1 |
| _process | 0.00 | 1 |
| _physics_process | 0.00 | 1 |
| _process | 0.00 | 1 |

| Name | Time | Calls |
|---|---|---|
| ✓ Frame Time | 18.7 | |
| ✓ Physics Frame Time | 16.6 | 1 |
| Process Time | 18.7 | 1 |
| Script Functions | 13.4 | |
| _process | 18.5 | 1 |
| upper_function | 18.5 | 1 |
| middle_function | 15.1 | 1 |
| lower_function | 3.36 | 1 |
| _process | 0.00 | 1 |
| _physics_process | 0.00 | 1 |
| _process | 0.00 | 1 |

Figure 9. Manifestation of the problem in the profiler. The specific function times (in green) are consistent with script time in self mode, but with frame time in inclusive mode.

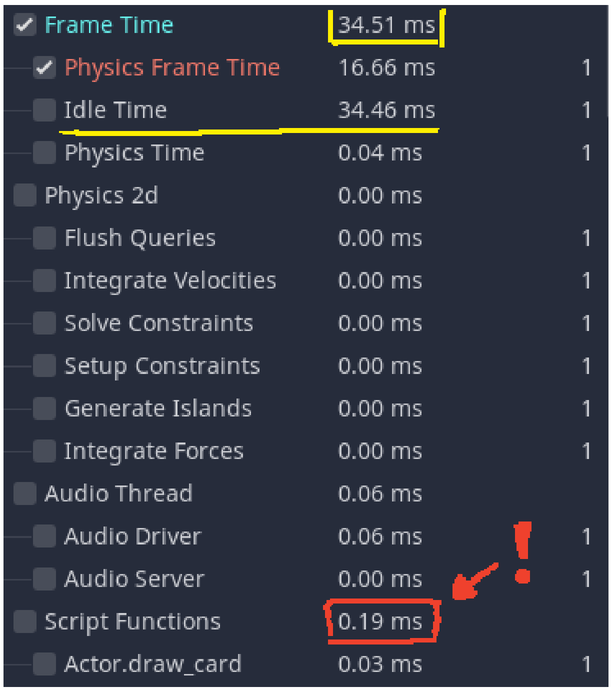
These issues are illustrated in figure 9. Consider the left side. The self time for the upper, middle and lower functions (in green) add up to roughly 13.4ms, which is correctly reflected in 'script functions' (in red). But the real time spent in the frame (in blue) is longer than that by 5 milliseconds. On the right, in the case of inclusive time display, the inclusive time for the outermost function (green) matches the real time (blue) and is correct, but the 'script functions'

[2]Links to the issues: 23715, 29849, 50251. Links are also listed in appendix II.

show a different number (red). An average user has no way to know which of these numbers is correct in which time display, and would be confused should they notice the difference. They would wonder how much time was really spent on their functions, and where this difference came from.

Information on where these 5ms were spent are lost to the user due to the bug. In this example the difference is small, but the amount of time 'lost' in this manner can easily balloon considerably. The time lost is equivalent to the time spent inside a Godot API function, and there's no guarantee those are fast. In an extreme case, the users' application can grind to a halt, with the real 'frame time' reaching the hundreds, while the script functions section shows nothing in their code taking any significant time.

As a further real example, figure 10 is an image submitted by a real user alongside an issue report. Each frame in his game was taking 35ms to process, causing low framerates, and yet his actual script functions were taking almost no time at all according to the profiler. This issue made it impossible for this user to diagnose his performance problem, defeating the purpose of a profiler.



```
✓ Frame Time                34.51 ms
  ✓ Physics Frame Time      16.66 ms    1
    Idle Time               34.46 ms    1
    Physics Time            0.04 ms     1
  Physics 2d                0.00 ms
    Flush Queries           0.00 ms     1
    Integrate Velocities    0.00 ms     1
    Solve Constraints       0.00 ms     1
    Setup Constraints       0.00 ms     1
    Generate Islands        0.00 ms     1
    Integrate Forces        0.00 ms     1
  Audio Thread              0.06 ms
    Audio Driver            0.06 ms     1
    Audio Server            0.00 ms     1
  Script Functions          0.19 ms
    Actor.draw_card         0.03 ms     1
```

Figure 10. Caption: Image submitted by a real user in a Godot issue, showing a large discrepancy between real time spent on a frame (circled in yellow), and time recorded spent on user scripts (circled in red, with an exclamation mark

Users who reported this issue only noticed it when the discrepancy between given times was large and they were closely examining it. However, exploration of the bug revealed that it happened when any non-GDScript function was called from user code, not only server-related functions. Native C++-written functions tend to be much faster than script functions. Furthermore, they're already tested to be performant as opposed to just-written editor code by the user, which is more likely to be flawed. This would mean that the discrepancies would be small in the majority of normal cases, and help explain why this issue went unnoticed and unaddressed for so long.

The implication is that all profiling done in Godot displayed mistaken time to the users, to some degree. Not only was the profiler rendered useless in some cases, but every user was being fed incorrect information from a built-in debugging tool. This is a serious state of affairs, so the bug was targeted as a significant issue worth correcting.

From this point forward, functions built into the Godot engine, the ones failing to be profiled and creating the problem, will be referred to as '**native**' functions and methods. These include all Godot API calls. The user-written GDScript functions that were previously normally profiled will be called '**non-native**'.

## 3.2 Internal bug description

The issue was tracked down to the GDScript VM, at the point where the profile data collection happens. The full execution time (inclusive time) of a non-native function was always recorded correctly. The bug was instead located in the calculation of 'self time' of non-native functions.

As explained in the Godot Profiler background section, self time is obtained by subtracting the time spent in sub-calls from the correct inclusive time. However, this subtraction only makes sense if every one of these sub-calls is also profiled and shown — if they aren't, a gap happens. The bug happened because the profiling of the lower function and the subtraction weren't connected, allowing one to happen without the other.

To explain this further, consider figure 11, displaying again the same overall scenario as the right side of figure 7, with boxes representing functions' self times shown in the profiler. Supposing lower_function is a problematic native Godot function, while upper and middle are non-native functions known to profile normally, there are three potential cases shown:

1. In case 1, all functions profile normally and time is subtracted normally, creating no issues and displaying the correct self time. This is the ideal case a complete fix would aim to create.

2. In case 2, the native lower_function isn't profiled, but the middle function's self time

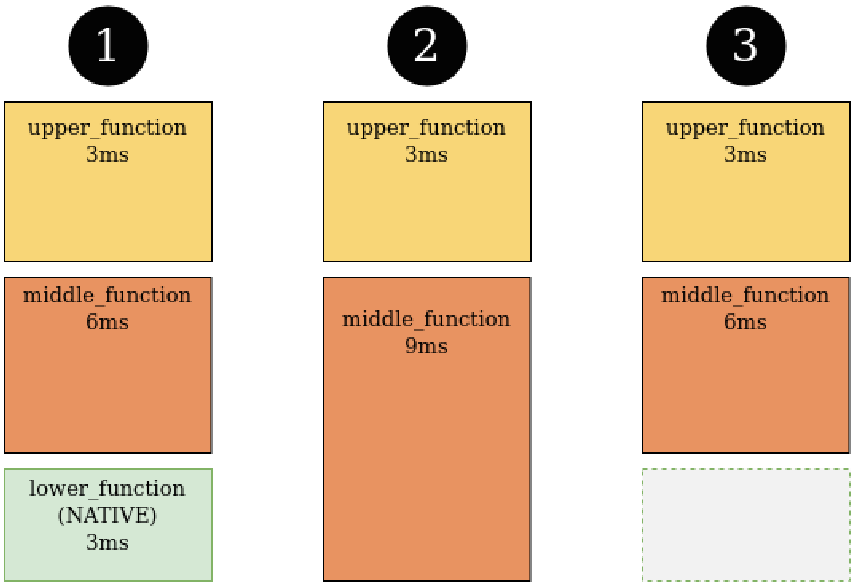


Figure 11. Three potential states of the profiler relating to this bug.

also doesn't subtract the time spent in it. This would mean the information displayed to the user would be incomplete, but also accurate. In this case, users wouldn't be able to pinpoint which exact native function is causing performance issues, but they'd be able to find where in their own code the problems are stemming from. In this case, that would be middle_function.

3. Case 3 is the actual problematic case found to happen in Godot before the fix. lower_function isn't profiled (as represented by a 'vanished' dotted grey box). However, time spent inside it still gets subtracted from the self time of the middle_function that calls it (represented by its length being shortened). This subtraction without a matching function profile creates the observed inconsistencies and problems.

Upon investigation, the reason native functions aren't profiled was that their profiling was never implemented to begin with. Native functions can be called through the GDScript VM, but only non-native GDScript functions were capable of recording their profile data. This is the source of the problem, and it's fundamental: it's the same reason the profiler doesn't record a full call graph, but only individual function calls as they happen. The profiler's data collection doesn't function at a deep enough level of the engine to collect comprehensive information on every function that gets called uniformly. It's based on the GDScript VM, which is itself a module on top of Godot. If the profiling was, for example, instead based on Variant, in the core of Godot, there'd be no such issue. This bug itself is straightforward, but fixing it the following complicating factors:

- The virtual machine at the crux of it's one of the innermost, most performance-critical portions of GDScript. The code in question is traversed as many times per frame as there are calls in the user's scripts in a single frame, and the number of frames rendered per second must be kept as high as possible. With possibly hundreds of loops per frame, the burden of any inefficient code is multiplied many times over.

- The GDScript VM is figuratively buried deep in the engine and has no real documentation. Relatively few open source contributors write code that needs to touch it, and information regarding it is mostly found inside the heads of a few core Godot developers. Information regarding its working had to be manually worked out with print statements, debugging tools, examination of the GDScript compiler and bytecode.

- The obvious, naive fix of adding profiling facilities to the different non-GDScript function types isn't feasible without large overhauls. they're structured and executed too differently between themselves, and are also in some cases highly optimized, which makes it difficult to add probes without reworks. Not to mention, it would either have to be implemented separately for each function type or, ideally, implemented at a deeper level, changing the core classes everything in the engine is built around. By itself, needing a large overhaul isn't an issue, but the aim of this thesis was to have these contributions accepted by the Godot maintainers and to act as a contributor. Such deep and sweeping core changes coming from a first-time contributor would be unwelcome.

## 3.3 Fixing the bug

Once diagnosed, two obvious fixes present themselves at a high level, patterning themselves after cases of figure 11.

The first alternative is to continue not profiling native functions, but stop the subtracting of inner function time in their particular case. This is equivalent to creating case 2 of figure 11. Without the additional subtraction from each function's self time, they'd give the accurate total time spent in them. This wouldn't exactly be 'self time' as defined, but would be suitable for the purposes of most general users of Godot. As API users, they're unable to change anything that happens further down the stack from a native API call in any case, so knowing where the bottleneck is on their code should be sufficient.

The second possibility is to actually record these missing functions, creating something equivalent to case 1 in the figure. This could be classifiable as a feature addition rather than a bug fix, and it would require creating an additional mechanism to save this information separately from normal GDScript function profiles. Native functions work differently, and as mentioned, attaching profile information to each one separately or applying a profiling mechanism to the Godot core types isn't viable. However, an extra intermediary layer at the level of the GDScript function would be a possible way of enabling this information collection.

For either fix there's a first necessary step of distinguishing native from non-native calls, and to do this in a performance-conscious way. That is, the fewer checks and operations involved, the better. This will be looked at first.

### 3.3.1 Distinguishing Native from non-native Functions

In the GDScript VM, which of the VM's code is executed correspondingly to a given script line is determined by an opcode. Given that, it's a natural first candidate to use and distinguish what type of function is being executed. It was found that the opcodes only partially separate function call types. There are many opcodes with names prefixed with OPCODE_CALL: These operations are those that happen during any kind of function call from inside GDScript. Of those, only one type (which will be called the 'generic' call opcode from here forward), can happen in the case of non-native functions. All other types of call opcode are specific and created for non-GDScript optimization cases. For instance, as shown in table 1, OPCODE_CALL_BUILTIN_STATIC only deals with static methods on builtin Godot types.

The relationship between OPCODE_CALL type and whether a function is native or non-native is as follows: A non-native call necessarily implies the generic opcode call, but a generic opcode call doesn't necessarily imply a non-native call. Given that, some of the work is already done, and a way must be found to distinguish native from non-native methods only within the generic opcode call.

This relationship is illustrated in figure 12. Red arrows represent native functions, and green represent non-native functions To distinguish native from non-native functions, only looking at those that arrive at the generic OPCODE_CALL is required. All other function calls are known to be non-native. The dotted line would represent the level at which any additional distinguishing checks must be made.

Two approaches to distinguishing native from non-native functions were attempted. The first attempted approach involved analyzing GDScript compiler code and mapping out all situations

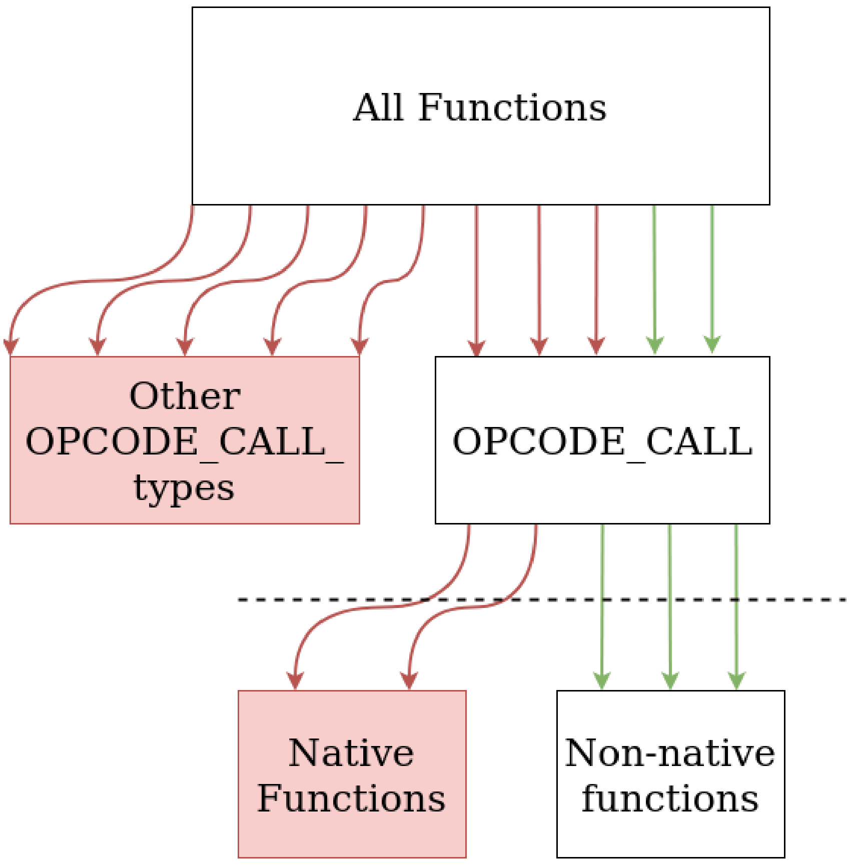


Figure 12. Flow of native (red) and non-native (green) functions relating to opcodes. The dotted line represents where the check needed to distinguish them must be made

in which calls are written into the bytecode. The compiler is the sole source of GDScript bytecode. Accordingly, analyzing it was theorized to be sufficient to find all possible situations in which native and non-native functions are called. However, this was a flawed approach. Little of the information available at compilation time was readily available during execution time. For instance, retrieving information about the arguments in the call and their types is expensive at execution time. Even if all information were available, checking for it would mean partially replicating the same complex conditional tree present in the compiler at execution time, requiring numerous checks on every function call. This approach was discarded when a more straightforward solution making use of existing Godot systems was found.

The second, better approach that was used in the final PR involves using a convenient Godot class named ClassDB. It contains metadata about all internal classes and methods. Querying this class with the name of the method being called and the name of the object it's being called from suffices to tell whether a function is native or not. This basic information is available during execution.

### 3.3.2 Problem Exception

The method of querying ClassDB works correctly for distinguishing the vast majority of commonly used API functions. However, there are a pair of important exceptions that cannot be made to work with this approach, making it a partial solution. These exceptions are functions that are technically native, but may execute non-native code.

For instance, take the user-callable Object method call. This is a native method, and its first argument must be another function, which will then be called by the engine. But this argument can either be a native or non-native function. There's no way to efficiently inspect arguments from inside the GDScript VM. This means it's impossible to tell if such functions are calling native or non-native code at that level.

There are a few other such native methods that can directly call either native or non-native code depending on the situation, such as *new* or *call_deferred.* A general solution to this hasn't been found, but since these would be a small minority of calls compared to common API calls in a normal situation, work continued as a partial fix.

## 3.4 Expansion: profiling for native calls

Once the distinction between native and non-native was made, simply deleting the time subtraction from methods detected as native is a working fix and yields correct self time in the manner described in case 2 of figure 11. However, in this state, the profiler provides no information regarding these native functions. It only folds the time that was previously lost into user functions. With this in mind, a further improvement was implemented, with the aim of including them and realizing case 1.

As previously touched on, extending the Godot profiler to be full-featured, with a complete call graph was not viable. It would've required too many significant changes to the core of the engine. However, it can be approximated by saving the information about what subroutines each function calls, using a parallel system. They can then be merged into the existing flow of profile information, with no large changes to the current functioning of the engine or GDScript VM.

Godot has GDScriptFunction objects, each of which represents one function written in GDScript and contains that function's data, including profiling information. The implemented solution was to add an an additional container to the GDScriptFunction object. With this container, each non-native GDScriptFunction stores not only their own non-native profile, but also profiles of every native function it calls. In this context the 'profile' of a function means the following collection of data: the name of the function and its base object, times it was called, and time spent inside it, for a single frame.

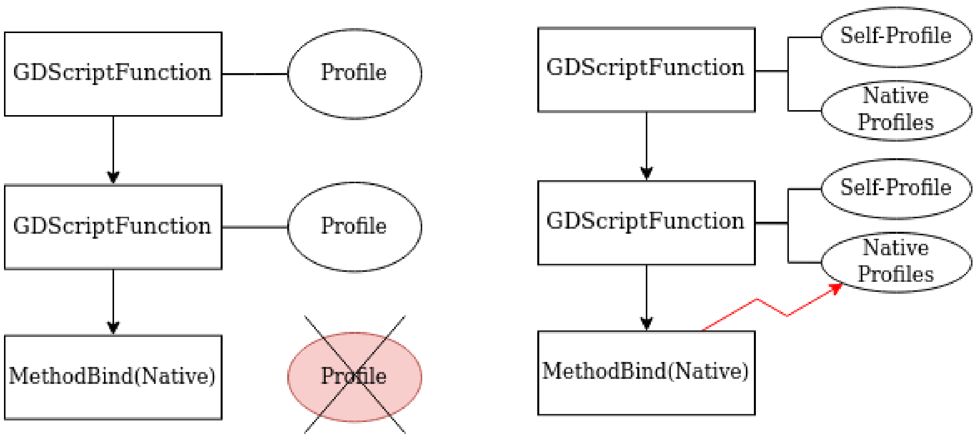


Figure 13. High level illustration of the mechanism. Left is before, right is after the fix. The red arrow represents the profile information being recorded.

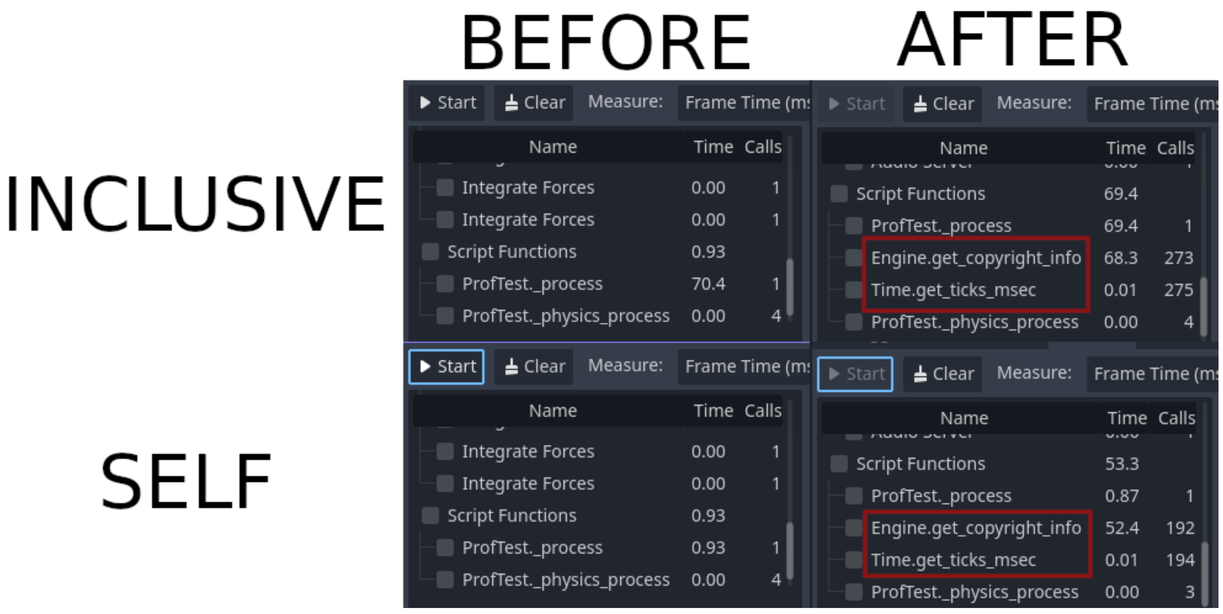
In figure 13 is a high-level illustration of the idea for the fix. On the left is how it worked previously: each GDScriptFunction object has a Profile member struct, and it fills it in with information when the function is executed in the GDScript VM. Other types of functions lack this member, causing the bug. On the right is how it works after the fix. A GDscriptFunction calling another GDScriptFunction remains unchanged, keeping the same profiling system. However, if it's determined to be calling a native function instead, it saves the information for that function (represented by the red arrow) in a separate container.

Each GDScript function can call any native functions an arbitrary number of times. Here the performance constraints become highly relevant: this information must be accessed and updated quickly. Furthermore, the same function called in multiple places must be considered only once by the profiler.

This container was Godot's hash table implementation, for fast O(1) retrieval, as this information must be retrieved many times per frame. Information saved was kept to a minimum. Only the number of calls, total time spent, name of the function and base object name are saved.

The final result of this feature, as a user would see it, is shown in figure 14. The same functions are being called in all four cases. Note the completely missing time in the bottom left. The function get_copyright_info is used for demonstration, since it's understandably unoptimized for performance and therefore shows large time losses.

The native functions circled in red, get_ticks_msec and get_copyright_info, don't appear at all in the unmodified Godot, and the time spent in them isn't shown in the script functions or in



| Name | Time | Calls |
|---|---|---|
| Integrate Forces | 0.00 | 1 |
| Integrate Forces | 0.00 | 1 |
| Script Functions | 0.93 | |
| ProfTest._process | 70.4 | 1 |
| ProfTest._physics_process | 0.00 | 4 |

| Name | Time | Calls |
|---|---|---|
| Script Functions | 69.4 | |
| ProfTest._process | 69.4 | 1 |
| Engine.get_copyright_info | 68.3 | 273 |
| Time.get_ticks_msec | 0.01 | 275 |
| ProfTest._physics_process | 0.00 | 4 |

| Name | Time | Calls |
|---|---|---|
| Integrate Forces | 0.00 | 1 |
| Integrate Forces | 0.00 | 1 |
| Script Functions | 0.93 | |
| ProfTest._process | 0.93 | 1 |
| ProfTest._physics_process | 0.00 | 4 |

| Name | Time | Calls |
|---|---|---|
| Script Functions | 53.3 | |
| ProfTest._process | 0.87 | 1 |
| Engine.get_copyright_info | 52.4 | 192 |
| Time.get_ticks_msec | 0.01 | 194 |
| ProfTest._physics_process | 0.00 | 3 |

Figure 14. View of the profiler before and after the fix, showing additional information added. *get_copyright_engine* and *get_ticks_msec* are native Godot functions.

the self time. The fix corrects this error.

### 3.4.1 Duplicated profiles

Native function profiles being saved independently in every non-native GDScriptFunction creates a potential issue: should two different non-native functions call the same native function, that native function will appear twice in profiling information. A call graph profiler wouldn't have this issue, as the calls are just edges in a graph, which would point to the same node representing a function. In Godot's flat profiler, an additional 'collation' step is necessary.

This collation step consists of looping through every recorded native function and joining ones with the same names and base objects together, summing their calls and time spent. This is a relatively expensive operation to have to do per frame, but it's done in a separate thread from the GDScript VM. Being in a separate thread mostly eliminates the relevant cost to the game's running speed.

### 3.4.2 Data collection toggling

The proposed fix works in nearly all cases save for those specified in section 3.3.2. However, this fix has a cost. Even keeping performance in mind and writing optimized code, more checks are performed and more information is saved with the fix. It inevitably slows down execution further when profiling is being performed. The exact amount of slowdown depends on how many and which native functions are called. In an artificial situation involving a million native calls per frame, a slowdown of up to 2x was found when profiling with this fix compiled. Such degradation of functionality wasn't an acceptable outcome, so the option to disable all collection of native profile data was also implemented in Godot's editor settings.

When collection is disabled, all additional processing, the hashmap find operation and potential information saving are replaced with a single boolean check per call per frame. This is a fast enough operation that it functionally eliminates the performance hit. When native data collection is disabled, the time that would be shown spent inside native code is instead counted as the next highest GDScriptFunction's self time, corresponding to case 2 of figure 11. This is accomplished with an additional mechanism at the point where the profile is shown, in editor code rather than engine code, so as not to cause performance issues. In this way, even with collection disabled, the original bug is fixed.

### 3.4.3 Native Data Display

Finally, as a consideration to usability, when native data collection is enabled, an additional button was added to the profiler's control interface. This button allows the enabling or disabling the display of native functions instantly, without needing to restart the profiler or the game. It was reasoned that there may be many more native API calls than common GDScript functions. Always showing all of them could severely clutter the display with unneeded information, annoying users and making it difficult to find what they actually want in the list of functions. This consideration also prompted the idea for one of the further usability improvements, the column sorting improvement described in section 3.5.1.

### 3.4.4 Testing

GDScript is a highly dynamic language, and the present issue dealt with many potential corner cases depending on the exact configuration of functions being called. Given that, systematic testing of some sort was deemed necessary. Godot provides a suite of automated unit tests for open-source contributors, but these were unfortunately unsuitable for testing the improvements.

There's a clear separation between 'Godot engine code' and 'Godot editor' code. The former is the core engine needed to run the actual games built in Godot even in absence of the editor. The editor is instead an additional component, also built using the engine. The Godot unit tests do not provide testing for the editor. Therefore, some portion of this testing would have to be done by manual testing of cases on the editor.

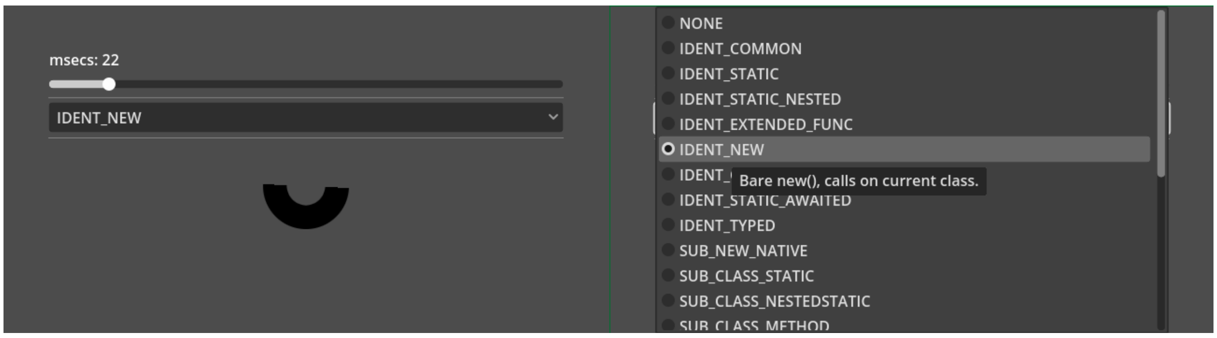


Figure 15. Views of the simple test project interface. Each item in the list is a different configuration of function calls to test.

The method used for this testing was to create a Godot project in the editor as a user would. This project consists of a menu where the developer chooses one of many alternative ‘situations’, as seen in figure 15, and a slider for how long to spend per frame. Each situation corresponds to a particular configuration of functions within the code, running in an infinite loop. While this loop is executing, the profiler can be opened to check its response. The option can be then changed in the menu to test all written possibilities. For a general idea of what was being tested, some of the options were:

- A nested static non-native function.
- An awaited native function.
- *call_deferred* on a native and non-native function.
- *new* on native or non-native classes.

More or less elaborate variations may cause slightly different data to arrive at the crucial point of the fix: the GDScript VM. The profiler’s response to these and other types of call configurations could then be watched and checked for errors. A table of the tested cases and responses can be found in appendix I.

## 3.5 Additional Improvements

Aside from the main bug fix, two additional minor usability improvements were submitted as pull requests: Column sorting, and profiler plot zoom and pan.

### 3.5.1 Column Sorting

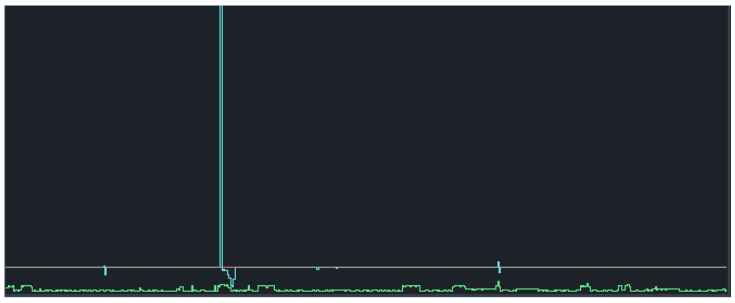
Enabling profiling of native Godot calls has the potential to increase the number of functions shown at a given time by a large amount. However, prior to this contribution, the user had no way to search for a specific function from within the list, or even narrow the search. To ease this limitation to a degree, a convenience feature was added: sorting the profiles according to a column. This feature is present in many software, such as file managers and music players. It doesn't need to be taught to experienced computer users. By clicking either the 'calls' or 'time' columns, the functions with the largest amounts of said quantity will be placed at the top. By clicking the 'name' column instead, they're sorted alphabetically, by class. Finally, clicking twice on the same column will reverse the order.

This feature is a convenient way to very quickly filter down to what the user wants to find in an interactive way, explaining why it's so widespread among software user interfaces despite lack of standardization.

The implementation of this feature was simplified by the fact that the Godot editor itself runs in Godot: the requisite signals and mechanisms for triggering a callback on clicking a column title were all already in place. In practical terms, the algorithm implementation consisted of getting profile data from the editor portion of the profiler, sorting it, and reordering the columns according to that information, with few complications.

### 3.5.2 Plot zoom and pan

The plotting feature of the Godot profiler is unrefined in a number of ways, but a major one is a lack of ability to zoom or any visual spatial manipulation. Prior to this, the maximum number of frames that could be seen at one time was fixed. It was only changeable in the editor settings (which required a restart of the profiler). Furthermore, except when changing the size of the entire editor window, the number of pixels occupied per profile frame was also fixed. This means that, for instance, if the maximum number of saved profile frames is larger than the width of the plot, it can look jagged and strange, or hide spikes of bad performance, as seen in figure 16. In the figure, both plots represent the same stretch of profiled functions, but on the bottom, the plot window has been shortened horizontally. Note how the low framerate spike (on the top) disappears. Pinpointing a single frame visually can also be difficult in this situation. The addition of zooming and panning goes a ways towards increasing usability, having been requested by users before.



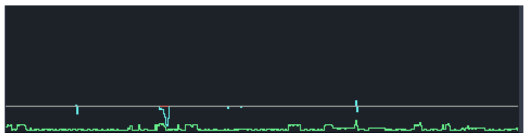


Figure 16. Illustration of one problem caused by the current profiler plotting, caused by differences in length between number of frames and width of the plot. The performance spike disappears when the plot is shortened.

Plotting and zooming don't exactly fix this type of issue, as there's no change to unzoomed plots. But they give the user the tools to examine the plot in more detail, and once zoomed in, such issues disappear. The prior plot rendering code drew the plot to a texture pixel-by-pixel, and this wasn't touched in this implementation. Instead, inputs to the previously written code were manipulated with a linear transformation. Values in pixel space, from 0 to the width of the plot, had to be translated to the correct corresponding profile frame indices, and the inverse operation. The corresponding 'space' can be determined by the magnification of the zoom and the center of the 'zoom location'.

The zooming and panning themselves aren't complex operations, but there are certain wider requirements for the improvement as a whole. Unlike the other two improvements presented, this is usability code that is directly interacted with in detail by users. Any slight awkwardness in the experience such as a slowdown of a fraction of a second when panning will be noticed by the end user and likely reflect upon Godot. Consequently, a greater share of work was spent testing and smoothing out interactivity. For instance:

- The zoom factor is used as a divisor, and therefore had to be put in an exponential function for the final zoom amount to move linearly.
- Special case considerations had to be made regarding the 'pan center' marker at the corners of the plot, to retain the ability to instantly pan in any direction without delay.
- A mechanism had to be developed to retain the ability to pan consistently even at very high magnifications, where mouse movements could cover less space than the width of a

single profiler frame.

From observation of the Godot team discussions, these kinds of usability details are some of the most paid-attention-to issues in the project, alongside anything that could create feature scope creep.

# 4 Open Source Contribution Process

Here a simplified overview of the process to get your contributed code added to the project is offered. Resources, instructions and communication channels with the core developers are looked at. Then, the procedure followed for the fixes detailed in this thesis are related, along with their current status in the Godot project.

## 4.1 General Process

The general flow is one common to many open source projects [PKMS14], using the version control system Git and website Github as a central hub for information regarding issues and proposals, as well as the mechanisms for organizing code changes. Issues and bugs are reported by anyone who finds them into the ‘Issues’ section, while feature proposals are separated into their own repository. The main medium for discussion and review of contributor code is a pull request or PR.

To create a PR, a prospective contributor first ‘forks’ or copies the original project repository, makes changes separately on their own copy, and then creates a request for the original repository to adopt their changes. This request and its associated systems are what’s called a pull request. A PR can exist for many months or occasionally years while changes are made and discussed by all parties involved. The core developers responsible for different areas of the engine will review the code, and if the changes are approved they will be ‘merged’ or adopted into the main codebase.

Each open source project has its own rules and procedures around what’s acceptable or desirable in PRs and issues. As a response to the intense growth experienced by Godot, the core development team created a tight set of procedures regarding contributions. New features especially see a lot of scrutiny before merging, and contributors are strongly encouraged to write a feature proposal before working on any code. The contributions discussed in this text were made keeping this in mind, being bug fixes or incremental improvements with little associated user burden. Their internal scope was also limited for this reason.

The main static resource aimed at new contributors mostly consists of a long section of the official documentation. It goes over ways to contribute, best practices, code style, workflow, compilation guides, writing documentation, translations, and development philosophy[3]. The Godot source code itself is well written and easy to follow in most areas.

A large portion of the support given to new contributors isn’t in static pages, but in its

[3]https://docs.godotengine.org/en/stable/contributing/ways_to_contribute.html

discussion communities. The official Godot project maintains a chat server for users of the engine and a separate chat for engine development. Many core developers and experienced contributors frequent both chats and answer questions in them. The developers' chat includes a 'new contributors' channel, designed to help people coming into the project. This communication channel was found to be prompt, well-populated and friendly, and a critical resource for a user trying to become a contributor.

## 4.2 Proceedings for the current work

The initial work was focused on the main profiler bug. After some initial diagnostic work and static analysis of the code, the main cause of the bug was discovered. The first interaction with the Godot project was a post in one of the issue comment threads dedicated to the problem. This initial post detailed the causes of the bug, and its aims were:

- To notify any developers watching the issue that there's someone actively working on it. As the last activity on the issue was some months old at this point, someone could potentially have already done some work on this problem, or it might have become obsoleted by unrelated work on the profiler. Additionally, it was thought that any developers notified this way might be prompted to share insights or knowledge that isn't available elsewhere.
- To prevent some other developer from coincidentally starting work on the same issue and accidentally doing redundant work independently.
- To present the current thinking about the issue to more experienced developers, such that had any obviously mistaken conclusions or assumptions been reached, they could be corrected.

Through the next weeks, debugging and development continued mostly independently, with the exception of two questions asked in the developer's chat regarding internal workings of the engine. These were promptly answered by experienced Godot engine developers.

The next development that involved the Godot team was the pull request for the main bug fix. Within a week of submitting it, a general provisional positive response was received on the PR from the core developer responsible for the profiler and GDScript VM, where most changes were. However this developer pointed out one major flaw in the distinguishing of native and non-native functions. This led to the exploration in section 3.3.1 and the discovery of the exception described in section 3.3.2. The core developer in question also expressed his doubt that a full fix would be possible without moving the profiler's mechanism to a deeper part of the engine, such as Variant. The exception that causes the PR to become a partial fix was communicated clearly in a post. Further review and decisions on whether to adopt the PR regardless of its nature as a partial fix are pending, as of this writing.

Shortly after the PR for the main bugfix, PRs for the other 2 improvements were also submitted. These garnered general positive responses but are currently waiting for initial developer review.

### 4.2.1 Discussion

The overall experience of contributing to the Godot engine was positive. Core developers and experienced contributors are prompt and help new users readily. Because of the current popularity of the project, there are knowledgeable people present at virtually all hours of the day to talk and answer questions. Resources and guides for new contributors were also thorough regarding the procedures involved and what was expected from issues and pull requests, making for a smooth introduction for prospective developers.

Effective development requires direct proactive communication, and this may be one potential stumbling block for developers. Particularly for those who are more used to working alone, in smaller teams, or in the more generally regimented closed-source environment. This initial communication is a common barrier to first-time contributors [SCGR15], and this was also found to be the case in this work.

Godot in particular has been developed and grown for a long time and has proven to be a viable game engine. Continued development and project maturity has attracted many peripheral developers [SRSC12], to the point that their collective contributions may exceed the core developers' capability to keep up. This would lead to slow response times, another common potential barriers to first-time contributors [SCGR15].

Prospective contributors could be made to be prepared for this by managing their expectations: in such large projects, one can't be expected to receive instant responses. It may be best to view contribution to such a project as a long-term project, or even as indefinite, as joining a community. The Godot resources also highlight non-programming ways to contribute: writing documentation, creating guides and content for Godot game developers, engaging with the wider community, or serving as word-of-mouth promotion.

## 4.3 Future work

None of the three pull requests submitted have been merged or thoroughly reviewed as of yet. The work detailed here will continue until the pull requests are either merged or closed.

The idea of removing the profiler from the higher-level GDScript VM entirely and moving it into the core of Godot was suggested during the course of development. This would be a significant change and require reworking of many of the internal interfaces involved, but it's a clear path to be explored further.

# 5 Conclusion

This work has gone over the creation of improvements for a Godot, a popular open source game engine in wide use. The aim of the project was primarily to contribute meaningfully to an open source project, and secondarily to serve as a reference for future first-time open source contributors or those searching for information regarding Godot. It described the issues being addressed, code changes and processes involved. It has explored a significant bug in the engine's built-in profiler. A fix that covers a majority of the problem has been created, tested and submitted to the project as a contribution, along with two general usability improvements. Finally, the process of open source contribution to Godot was briefly related and discussed.

# References

[And15] A. Andrade. Game engines: a survey. *EAI Endorsed Transactions on Serious Games*, 2(6), 11 2015.

[BSS07] Jürgen Bitzer, Wolfram Schrettl, and Philipp J.H. Schröder. Intrinsic motivation in open source software development. *Journal of Comparative Economics*, 35(1):160–169, 2007.

[CC07] Kajal T. Claypool and Mark Claypool. On frame rate and player performance in first person shooter games. *Multimedia Systems*, 13(1):3–17, Sep 2007.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994.

[GKM82] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, page 120–126, New York, NY, USA, 1982. Association for Computing Machinery.

[Gre09] Jason Gregory. *Game engine architecture*. Taylor & Francis Ltd., 1 edition, 2009.

[Lin23] Juan Linietsky. Godot as an open ecosystem, 3 2023. Remarks about Godot's growth and presence at the Game Developers Conference 2023, by Godot lead developer Juan Linietsky.

[PKMS14] Rohan Padhye, Senthil Kumar Kumarasamy Mani, and Vibha Sinha. A study of external community contribution to open-source projects on github. 05 2014.

[PSTH17] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. Studying the advancement in debugging practice of professional software developers. *Softw. Qual. J.*, 25(1):83–110, 2017.

[SCGR15] Igor Steinmacher, Tayana Conte, Marco Aurélio Gerosa, and David Redmiles. Social barriers faced by newcomers placing their first contribution in open source software projects. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, CSCW '15, page 1379–1392, New York, NY, USA, 2015. Association for Computing Machinery.

[SRSC12] Pankaj Setia, Balaji Rajagopalan, Vallabh Sambamurthy, and Roger Calantone. How peripheral developers contribute to open-source software development. *Info. Sys. Research*, 23(1):144–163, mar 2012.

[Ung12] Alexander Unger. *Modding as Part of Game Culture*, pages 509–523. Springer Netherlands, Dordrecht, 2012.

[Wil17] A. Williams. *History of Digital Games: Developments in Art, Design and Interaction*. CRC Press, 2017.

# Appendix

## I. Glossary

| Term | Meaning |
|---|---|
| SDK | Software Development Kit, a kit of tools designed to allow development of software with a certain platformor framework in mind. |
| GUI | Graphical User Interface, a type of user interface that allows users to interact graphically with software. |
| IDE | Integrated Development Environment, an application that offers a comprehensive set of tools aimed at software development, such as, among others, debugging tools, compilers and source code editors. |
| VM | Virtual Machine, the emulation of a computer system inside software. |
| Git | A widely-used version control system. |
| Repository | A store of source code created and managed by the version control system git. |
| GitHub | A service used to host git repositories and provide development collaboration facilities |
| Debugging | The process of diagnosis and repair of mistakes in source code, or ’bugs’. |

## II. Github Repository, Pull Request, and Issue links

- Repositories
  - Main Godot Engine GitHub repository:
    `https://github.com/Godotengine/Godot`
  - Author's forked development Godot repository:
    `https://github.com/reach-satori/Godot-prof-proj`
- Pull Requests
  - Profiler missing time fix PR:
    `https://github.com/Godotengine/Godot/pull/75623`
  - Profiler column title sorting PR:
    `https://github.com/Godotengine/Godot/pull/75931`
  - Profiler plot zoom and pan PR:
    `https://github.com/Godotengine/Godot/pull/76055`
- Issues
  - Profiler missing most of my functions. Issue #40251:
    `https://github.com/Godotengine/Godot/issues/40251`
  - Profiler sometimes properly count time but sometimes not. Issue #29049:
    `https://github.com/Godotengine/Godot/issues/29049`
  - Time spent waiting for servers not included in functions profiler (e.g. physics). Issue #23715:
    `https://github.com/Godotengine/Godot/issues/23715`
  - Profiler update suggestions #2045:
    `https://github.com/Godotengine/Godot-proposals/issues/2045`

# III. Testing results table

Table 2. Table of test results.

| **Case** | **With fix** | **Without fix** |
|---|---|---|
| Non-native method called on the same class | No issue | No issue |
| Non-native method called on the same class | No issue | No issue |
| 2 nested non-native static methods called on the same class | No issue | No issue |
| Calls native ancestor method on current non-native class | No issue | Method doesn't appear, time is missing |
| *new()* on current non-native class, which contains native functions on initialization | *new* doesn't appear, time is missing | *new* doesn't appear, time is missing |
| Non-native method awaited on the same class | No issue | No issue |
| Non-native static method awaited on the same class | No issue | No issue |
| Typed non-native method | No issue | No issue |
| *new()* on native class | No issue | Doesn't appear, time is missing |
| Non-native static method called on different non-native class | No issue | No issue |
| Non-native nested static methods called on different non-native class | No issue | No issue |
| Non-native method called on different non-native class | No issue | No issue |
| Non-native method called on different non-native class, with *call()* | *call()* itself is missing from the profiler, but time is consistent | *call()* itself is missing from the profiler, but time is consistent |
| Native method called with *call()* | Neither *call* nor the method appear, time spent inside them is missing | Neither *call* nor the method appear, time spent inside them is missing |
| *call_deferred* with non-native method | No issue | *call_deferred* doesn't appear, method being called appears |
| *call_deferred* with native method | Native method doesn't appear on the profiler, its time is missing | Neither *call_deferred* nor method appear, their time is missing |
| Native call inside _physics_process | No issue | Call is missing, time is missing |
| Non-native call inside _physics_process | No issue | No issue |
| Native call on object | No issue | Call is missing, time is missing |
| Native call on audio server | No issue | Call is missing, time is missing |
| Native call on physics server | No issue | Call is missing, time is missing |
| Awaited native method call | No issue | Call is missing, time is missing |
| Native method call with typed variable | No issue | Call is missing, time is missing |
| Static native method | No issue | Call is missing, time is missing |

# IV. Licence

## Non-exclusive licence to reproduce thesis and make thesis public