

Game Engine Comparative Anatomy

Gabriel C. Ullmann¹, Cristiano Politowski¹,
Yann-Gaël Guéhéneuc¹, and Fabio Petrillo²

¹ Concordia University, Montreal QC, Canada
g.cavanh@live.concordia.ca, c.polito@encs.concordia.ca,
yann-gael.gueheneuc@concordia.ca

² École de Technologie Supérieure, Montreal QC, Canada
fabio.petrillo@etsmtl.ca

Abstract. Video game developers use game engines as a tool to manage complex aspects of game development. While engines play a big role in the success of games, to the best of our knowledge, they are often developed in isolation, in a closed-source manner, without architectural discussions, comparison, and collaboration among projects. In this work in progress, we compare the call graphs of two open-source engines: Godot 3.4.4 and Urho3D 1.8. While static analysis tools could provide us with a general picture without precise call graph paths, the use of a profiler such as Callgrind allows us to also view the call order and frequency. These graphs give us insight into the engines' designs. We showed that, by using Callgrind, we can obtain a high-level view of an engine's architecture, which can be used to understand it. In future work, we intend to apply both dynamic and static analysis to other open-source engines to understand architectural patterns and their impact on aspects such as performance and maintenance.

Keywords: Game Engines · Game Engine Architecture · Game Development.

1 Introduction

Game engines allow developers to create games in an agile and standardized way. While there is a wide array of free, open-source engines, many large studios choose to develop proprietary solutions to be used internally. Famous examples of this approach are Frostbite³, id Tech⁴, and RAGE⁵. There are also popular closed-source engines available publicly, such as Unity and Source [11].

When writing a proprietary engine, developers are able to customize it to fulfill performance or feature requirements of a certain game or game genre, which may give them an edge on competitors. As a downside, however, this development process hinders exchange of information that could be beneficial to

³ <https://www.ea.com/frostbite>

⁴ <https://arstechnica.com/gaming/2018/08/doom-eternal-reveals-new-powers-puts-hell-back-on-earth/>

⁵ <https://www.inverse.com/gaming/gta-6-leaks-new-rage-engine>

the entire game development community. According to [12], “almost all relevant game engines are closed source which means that particularities on implementation of certain functionalities are not available to developers. The only way for a developer to understand the way certain components work and communicate is to create his/her own computer game engine”.

Developers may also find obstacles when creating a game engine from scratch, especially due to a lack of standards on how high-level architecture components should be created and related for this kind of system. Even though books have been published on the topic of game engine architecture [4–6], according to [1], often these publications “tend to only briefly describe the high-level architecture before plunging straight down to the lowest level and describing how the individual components of the engine are implemented”. Furthermore, they mention that “such literature offers an excellent source of information for writing an engine, but provides little assistance for designing one when the requirements are different from the solution described”.

We believe game engines would benefit from experience exchanges with other projects of the same kind. A comparative analysis would allow us to identify commonalities and propose points of improvement to existing engines. In this work in progress, we compare the call graphs of two open-source engines, Godot and Urho3D, considering these aspects.

The Callgrind profiling tool is used to generate these call graphs, which reflect the execution of a base game project produced with each engine. By comparing the call graphs, we aim to observe the main components of each engine and how responsibilities are divided among them.

While static analysis tools could provide us with a general picture without precise call graph paths, we chose to use a profiler because it allows us to view the call order, frequency, and the number of CPU cycles taken by each method. As a result of this analysis, we will answer the following questions: Are game engine designs similar? If so, how similar are they? Our hypothesis is that game engines follow a similar design and architectural structure.

We show that producing a high-level architecture view of the engine is possible using a profiling tool and that it is a way to get insights into an engine’s design. In future work, we will compare these architectures with those proposed by researchers, both through static and dynamic analysis. We will study how design choices made in each case may influence performance, maintenance, and the range of games that can be produced with a given engine.

2 Related Work

Several works compare game engines concerning ease of use [3], available tools and target platforms, and also to determine which are more suitable for a given platform [9] or game genre [10]. However, these comparisons are made from a game developer point of view and do not encompass details related to engine design or implementation. Other works focus on proposing architecture to fulfill

a specific requirement, such as low-energy consumption for mobile devices [2]. Novel distributed architectures have also been proposed [7,8].

In our work, we compare Godot and Urho3D, not simply in terms of what features are available and in what situations they fit best, but also looking at how engine subsystems (e.g., graphics, audio, physics, etc.) are organized, initialized and in which ways they interact and relate.

3 Approach

3.1 Overview

Our approach consists of five steps. First, we selected C++ engines available on Github (more details on Section 3.2) and cloned their repositories. Following the documentation or running scripts provided by the developer, we compiled them. Using the engine’s editor, we created a new project, setting up the minimum necessary to make it run. We will hereby call this our “base game”⁶. A Godot project demands at least one scene object to run, which can be created via the engine’s editor. Urho3D, on the other hand, has no editor: one must create a .cpp file and include Urho3D’s library to create a new project. We can run the game loop by calling the *Start* method, no scene creation needed.

We then ran the base game’s packaging process, which allows us to obtain an executable that can be analyzed by the profiler. In the case of Urho3D, there is no packaging, only .cpp file compilation. Normally the code would be compiled in release mode, with all optimizations in place, but in our case, we changed the packaging settings to obtain an executable with debug symbols.

We ran Callgrind on the base game executable to obtain the list of all calls made by the program, which it saves into a log file. Using KCachegrind, we converted this Callgrind log into a visual representation of the program’s execution, a call graph. Class and method names in the graph guided our analysis since they helped us understand the responsibilities of each component.

3.2 Engine selection

We searched for all repositories on GitHub filtering by topic (*game-engine*) and language (C++). We chose to consider only C++ since this is one of the most popular languages for game engine development [11]. After that, we selected 15 projects and ordered them by number of stars and forks. We then proceeded to clone and build each one of them, starting by those with the highest sum of stars and forks. At the time of writing of this paper, we managed to compile and produce a running executable for two of these engines: Godot and Urho3D.

⁶ <https://github.com/gamedev-studies/engine-profiling-projects/tree/master/BaseGame>

3.3 Engine compilation and analysis

We cloned the GitHub repositories for Godot and Urho3D, working with the latest version at the time for each: commit *feb0d* at the *master* branch for Urho3D and commit *242c05* at the *3.4.3-stable* branch for Godot.

Following the instructions in the documentation, we compiled Godot export templates for Linux (we used Ubuntu 20.04.4 LTS), both in release and debug modes. These are binary files containing engine code compiled for a specific target platform. Before generating an executable for any Godot project, we must link these templates to the project via the editor’s GUI. We created a Godot base game, generated the executable, and then ran it with Callgrind for 30 seconds.

4 Godot’s call graph

We verified that the first tasks Godot executes upon running a game are registering classes and initializing a window (Figure 1). Since our base game is running on Ubuntu, Godot calls the X11 windowing system. After getting a window instance, the engine then proceeds to call methods to create a scene. Once ready, a notification is sent into the engine’s message queue. According to the engine documentation, the *MessageQueue* class is “a thread-safe buffer to queue set and call methods for other threads”⁷.

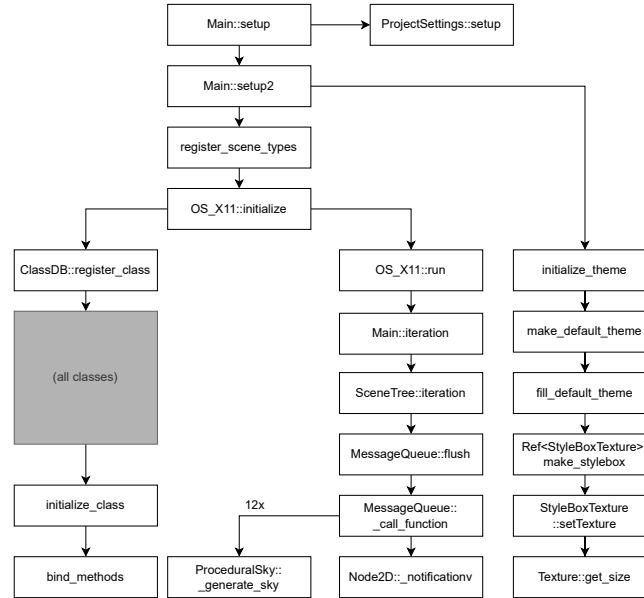


Fig. 1. Godot’s base game call graph, showing which methods are called.

⁷ https://docs.godotengine.org/en/stable/development/cpp/custom_godot_servers.html

Also during startup, Godot’s calls methods to set up its GUI theme system ⁸, such as *initialize_theme*, even though in our base game it is not being used. Similarly, a *ProceduralSky::generate_sky* method is called repeatedly even though no sky is drawn. According to Godot’s documentation⁹, this class procedurally generates a sky object, which is “stored in a texture and then displayed as a background in the scene”. Since no textures are loaded into the base game, nothing is drawn, but the sky generation method is called and updated anyway.

To validate our approach, we compared Godot’s call graph with a “layers of abstraction” diagram posted on Twitter by Godot’s creator Juan Linietsky ¹⁰. This comparison can be seen in Figure 2. Searching by class names, we found matches for all classes, even though the names were not the same (e.g *PhysicsServer* and *Physics2DServer*).

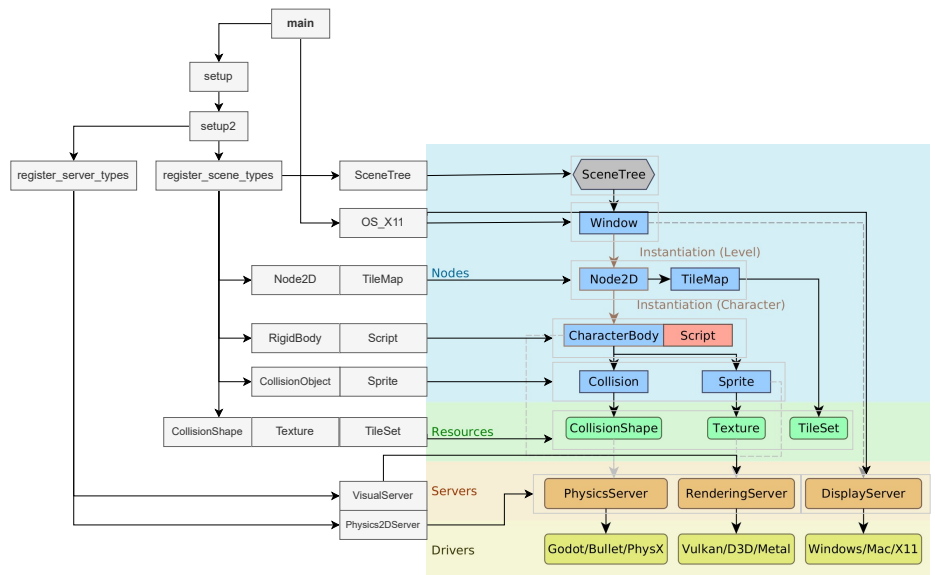


Fig. 2. Comparison between the call graph for Godot base game and diagram of “layers of abstraction” by Juan Linietsky

The most notorious difference is the class described by Linietsky as *Window*. There is no class with this name on Godot’s code base. We believe, however, that he referred to windowing systems in general and therefore decided to simply use the term *Window* since his diagram was platform-agnostic.

Also regarding the windowing system, we found no class named *DisplayServer* on the call graph. To understand what part of our call graph corresponds to this

⁸ https://docs.godotengine.org/en/stable/classes/class_theme.html
⁹ https://docs.godotengine.org/en/stable/classes/class_proceduralsky.html
¹⁰ <https://twitter.com/reduzio/status/1506266084420337666>

class, we used method names as a reference. Inspecting the source code, we determined that methods *get_singleton* and *has_feature* from the *DisplayServer* class were called by the *ProjectSettings* class, which is instantiated by the first *setup* method. In the base game’s graph, we found this same method name being called by a class named *OS*.

5 Urho3D’s call graph

Differently from Godot, Urho3D initializes its graphics object first and only then calls DSL (Simple DirectMedia Layer) and X11 to open a window, as shown in Figure 3. However, they both initialize UI-related code even though there is no UI to draw, and register all game object classes even though they are not instantiated.

Furthermore, our attention was drawn to the fact that, while Urho3D calls one initialization method per class, Godot calls *Main::setup* and *Main::setup2*. These apparently redundant calls are justified by the developer as a way to separate low and high level “singletons and core types”¹¹.

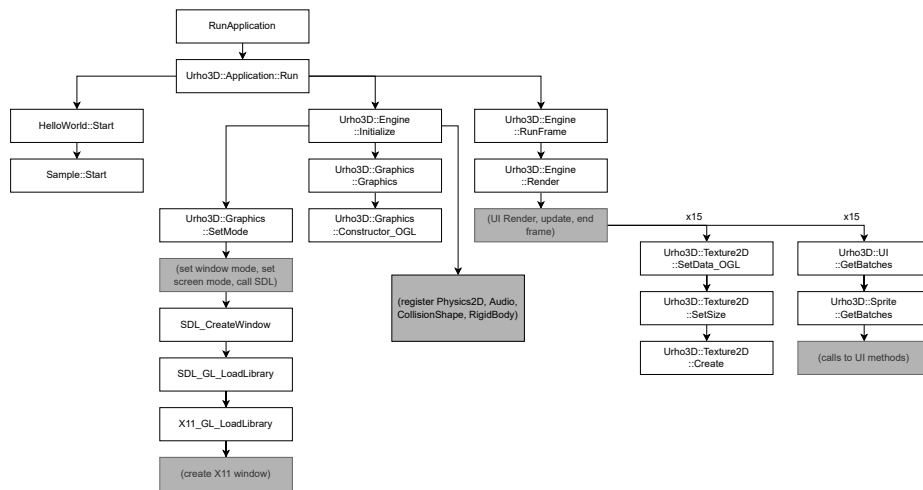


Fig. 3. Urho3D’s base game call graph, showing which methods are called.

6 Comparing Godot and Urho3D

In terms of similarities, both Godot and Urho3D have the same features present in the most widely used game engines, such as graphics, audio, and physics.

¹¹ Line 342 at <https://github.com/godotengine/godot/blob/3.4/main/main.cpp>

Both engines register all game object classes upon startup and initialize graphics systems, even though there is nothing more than an empty window to draw.

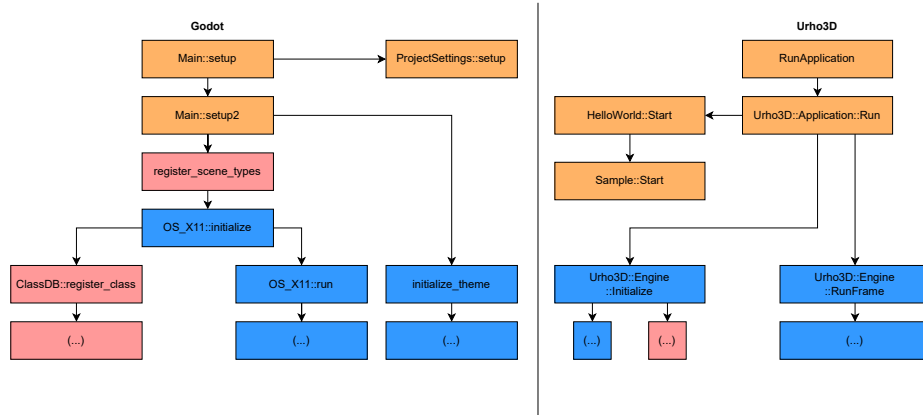


Fig. 4. Division of responsibility on Godot and Urho3D: initialization methods (orange), class registration (red) and graphics (blue). Call tree ramifications omitted for brevity.

On the other hand, the engines run graphics and window initialization in different orders. Godot creates the window first, Urho3D does the opposite and uses the DSL library to manage the interaction with X11. Godot has a scene manager and internal messaging system which are initialized even in a simple project such as the base game. Urho3D initializes only graphics and UI in this case.

7 Discussion

We can use call graphs to visualize the division of responsibilities inside the engines during initialization. They also provide an overview of the most important subsystems and at which moment they are registered and called.

However, these observations do not give us deep insights into architectural patterns and design choices for specific subsystems that could be useful for game engine developers. To obtain this knowledge, we will generate dynamic and static call graphs, as well as C++ #include graphs for each subsystem in the future.

While comparing the engines, we also identified project and runtime environment conditions that could make the profiler generate the call graph differently. These are important to note since they may change our overall view of the architecture.

Operating System: There is a possibility that if we ran the profiler on a Windows or Android version of the same base games, we would obtain a different

call graph. Each platform has its own system calls, GUI and data types, so the engines account for this by adding platform-specific code. If the call graph would be largely different, this difference could indicate that code evolution and maintainability are compromised.

Active Features: Important class or method names may not be present in the graphs because they have not been called during the analyzed run. The base game only calls methods from the engine “core” and not others referring to more specific features (e.g., networking methods, which would be needed in a multiplayer game).

Other Means of Analysis: Results could vary when running our base game with other profilers such as `gprof` or `gperftools`. Furthermore, while dynamic analysis can provide us with a precise call graph, it only represents a given run of the program and is therefore only a partial view.

8 Conclusion

In this work in progress, we showed that producing a high-level architecture view of an engine is possible with the use of a profiler, such as Callgrind, which generates engine call graphs. We compared Godot 3.4.4 and Urho3D 1.8 not only by looking at the engine’s subsystems but also in which order and frequency they are called, which would not be possible by applying static analysis alone.

As for the research question “are game engine designs similar?”, we concluded that Godot and Urho3D have a similar feature set and therefore could be used to create the same types of games. Their features are the same present in most widely used game engines, but they divide responsibilities and call them in the code in different ways. Also, we identified that both engines initialize subsystems even though they are not used (e.g. UI). While this may consume resources, it seems to be the right choice since these systems are often used.

In future work, we intend to compare different open-source engine call graphs, starting with Unreal Engine 5, to understand what architectural design patterns are most frequently applied to them and how similar they are among engines. Also, we will compare these architectures with those proposed by researchers, using static and dynamic analysis, to understand the impact of design on engine performance, maintenance, and feature richness.

Acknowledgements

The authors were partially supported by the NSERC Discovery Grant and Canada Research Chairs programs.

References

1. E. F. Anderson, S. Engel, P. Comminos, and L. McLoughlin. The Case for Research in Game Engine Architecture. In *Proceedings of the 2008 Conference on Future Play: Research, Play, Share*, Future Play '08, pages 228–231, New York, NY, USA, 2008. Association for Computing Machinery. event-place: Toronto, Ontario, Canada.
2. E. Christopoulou and S. Xinogalos. Overview and Comparative Analysis of Game Engines for Desktop and Mobile Devices. *International Journal of Serious Games*, 4(4), Dec. 2017.
3. P. E. Dickson, J. E. Block, G. N. Echevarria, and K. C. Keenan. An Experience-based Comparison of Unity and Unreal for a Stand-alone 3D Game Development Course. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, pages 70–75, Bologna Italy, June 2017. ACM.
4. D. H. Eberly. *3D game engine design: a practical approach to real-time computer graphics*. Elsevier/Morgan Kaufmann, Amsterdam ; Boston, 2nd ed edition, 2007.
5. J. Gregory. *Game engine architecture*. Taylor & Francis, CRC Press, Boca Raton, third edition edition, 2018.
6. E. Lengyel. *Foundations of game engine development*. Terathon Software LLC, Lincoln, California, 2016.
7. D. Maggiorini, L. A. Ripamonti, E. Zanon, A. Bujari, and C. E. Palazzi. SMASH: A distributed game engine architecture. In *2016 IEEE Symposium on Computers and Communication (ISCC)*, pages 196–201, Messina, Italy, June 2016. IEEE.
8. C. Marin, M. Chover, and J. M. Sotoca. Prototyping a game engine architecture as a multi-agent system. In *Computer Science Research Notes. Západočeská univerzita*, 2019.
9. A. Patrasitidecha. Comparison and evaluation of 3D mobile game engines. Master's thesis, Chalmers University of Technology, 2014.
10. S. Pavkov, I. Frankovic, and N. Hoic-Bozic. Comparison of game engines for serious games. In *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 728–733, Opatija, Croatia, May 2017. IEEE.
11. C. Politowski, F. Petrillo, J. E. Montandon, M. T. Valente, and Y.-G. Guéhéneuc. Are game engines software frameworks? a three-perspective study. *Journal of Systems and Software*, 171:110846, 2021.
12. M. Sršen and T. Orehovački. Developing a game engine in c# programming language. In *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*, pages 1717–1722, 2021.