Visualising Game Engine Subsystem Coupling Patterns

Gabriel C. Ullmann[1][0000−0002−3274−0789], Yann-Gaël Guéhéneuc[1][0000−0002−4361−2563], Fabio Petrillo[2][0000−0002−8355−1494], Nicolas Anquetil[3][0000−0003−1486−8399], and Cristiano Politowski[2][0000−0002−0206−1056]

[1] Concordia University, Montreal QC, Canada
g_cavalh@live.concordia.ca, yann-gael.gueheneuc@concordia.ca
[2] École de Technologie Supérieure, Montreal QC, Canada
fabio.petrillo@etsmtl.ca, cristiano.politowski@etsmtl.ca
[3] Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 - CRIStAL, Lille, France
nicolas.anquetil@inria.fr

Abstract. Game engines support video game development by providing functionalities such as graphics rendering or input/output device management. However, their architectures are often overlooked, which hinders their integration and extension. In this paper, we use an approach for architecture recovery to create architectural models for 10 open-source game engines. We use these models to answer the following questions: Which subsystems more often couple with one another? Do game engines share subsystem coupling patterns? We observe that the Low-Level Renderer, Platform Independence Layer and Resource Manager are frequently coupled to the game engine Core. By identifying the most frequent coupling patterns, we describe an emergent game engine architecture and discuss how it can be used by practitioners to improve system understanding and maintainability.

Keywords: Game Engines · Coupling · Game Engine Architecture

1 Introduction

Game engines are tools made to support video game development. From the perspective of Software Engineering, game engines are systems composed of subsystems, each providing functionalities essential for any video game, such as 2D/3D graphics rendering or input/output device management. However, the versatility of game engines also makes them architecturally complex and often difficult to understand. The lack of architecture understanding hinders software integration and extension, which is important in the context of plugin-extendable game engines such as Unreal, Unity and Godot. Therefore, studying game engine architecture is necessary: "[a] prerequisite for integration and extension is the comprehension of the software. To understand the architecture, we should identify the architectural patterns involved and how they are coupled." [1].

In this paper, we apply the approach for game engine architecture recovery described in our previous paper [11] to 10 popular open-source game engines. By

following this approach, we obtain *include* graphs tagged by subsystem, which we call architectural models for the sake of simplicity, for each game engine.

By studying these models' nodes and relationships, we answer the following research questions:

- **RQ1**: Which subsystems more often couple with one another?
- **RQ2**: Do game engines share subsystem coupling patterns?

The remainder of the paper is organized as follows: Section 2 presents related work on game engine architecture and architectural recovery. Section 3 describes our game engine architecture recovery approach. Section 4 shows the architectural models resulting from applying our approach and Section 5 discusses lessons learned from frequent coupling patterns. Section 6 presents threats to validity and Section 7 concludes with future work.

2 Related Work

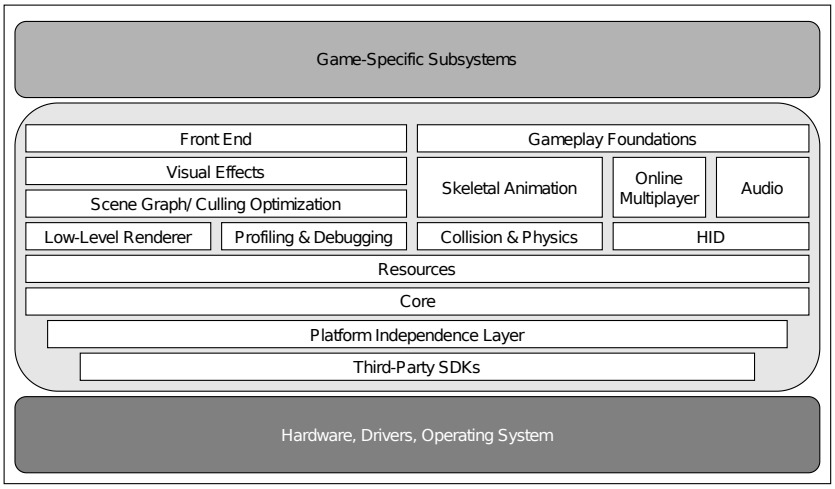


Fig. 1: Runtime Engine Architecture, adapted from Gregory [5, p. 33].

Few studies focused solely on game engine architecture and those that did focused on describing a specific game engine in detail. For example, Jaeyong and Changhyeon [7] explain the scripting and networking subsystems of a game engine for MUDs [4]. Bishop et al. [3] describe the NetImmerse engine. Both authors represent subsystems and their relationships in diagrams. However, they do not discuss why these relationships exist, how often they appear or whether they are representative of all game engine architectures.

[4] Multi-User Dungeon, a text-based precursor of MMORPG games.

Gregory [5] proposed a "Runtime Game Engine Architecture" (Figure 1), which describes common subsystems, their responsibilities, and some of their relationships. While also not focused on subsystem relationships, this is, to the best of our knowledge, the most comprehensive game engine architecture. Therefore we use it as a reference architecture in the following, as we explain further in Subsection 3.2.

Guided by a reference architecture, we apply an architecture recovery approach. Architecture recovery is concerned with the extraction of architectural descriptions from a system implementation [4]. Researchers have applied it to systems such as the Apache Web server [6], the Android OS and Apache Hadoop [8] as a way to improve understanding and maintainability.

Game engine developers can also reap the benefits of architecture recovery for their systems. For example, the information obtained through architecture recovery can be "used in the process of identifying suitable improvements and enhancements to a specific engine and have supported implementing these in an appropriate manner" [9]. This potential for assisting software architectural improvement is the main reason why we chose to apply architectural recovery to game engines in this work.

3 Approach

We now explain the six-step approach we proposed in a previous work [11], summarized in Figure 2 and used in this paper in a slightly modified version. Steps 1 to 3 are performed manually. Steps 4 to 6 are largely automated with Smalltalk and Python code, which is available on GitHub.[5]

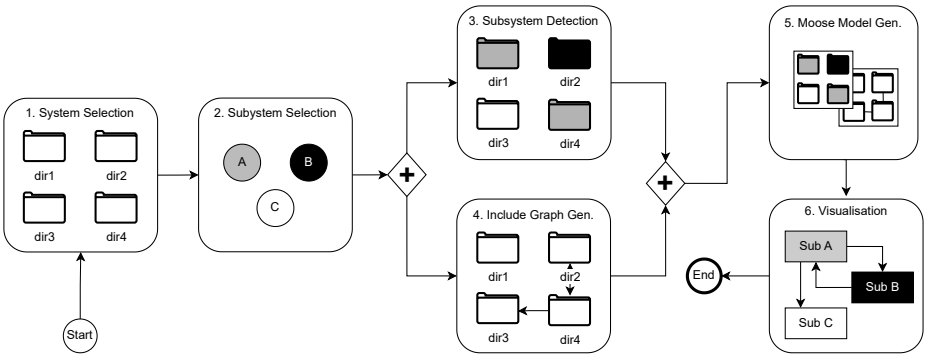


Fig. 2: Steps of our architecture recovery approach.

[5] https://github.com/gamedev-studies/game-engine-analyser

3.1 System Selection

We searched for the term "game engine" on GitHub and then selected all repositories showing C++ as the predominant programming language, given its relevance to game engine development [10]. This initial selection consisted of 20 game engine repositories. We then removed from the selection all non-general-purpose game engines. For example, we did not select the engine *minetest* [6] because it is limited to creating games in the style of Minecraft. Finally, we sorted the remaining repositories by the sum of their GitHub forks and stars (as of May 2022) in descending order. We selected the top 10 in this list, as shown in Table 1.

Table 1: Overview of the selected GitHub repositories.

Repository	**Branch**	**Commit**	**Forks + Stars**	**Files (.h, .cpp)**
UnrealEngine	v4	90f6542cf7	64100	66390
godot	3.4	f9ac000d5d	59200	5603
cocos2d-x	v4	90f6542cf7	23300	1601
o3de	development	21ab0506da	6400	7278
Urho3d	master	feb0d90190	4956	4312
gamePlay3d	master	4de92c4c6f	4900	688
panda3D	master	2208cc8bff	4100	5344
olcPixelGameEngine	master	02dac30d50	3963	81
Piccolo	main	b4166dbcba	3892	1572
FlaxEngine	master	7b041bbaa5	3613	2134

3.2 Subsystem Selection

We use the 15 subsystems described in our reference architecture, the "Runtime Engine Architecture" (Figure 1). Given that commercial game engines provide an integrated development environment, we added the "World Editor" subsystem in our analysis, totalling 16 subsystems.

For brevity, we identify subsystems in the reference architecture with 3-letter identifiers: *Audio* (AUD), *Core* (COR), *Profiling and Debugging* (DEB), *Front End* (FES), *Gameplay Foundations* (GMP), *Human Interface Devices* (HID), *Low-Level Renderer* (LLR), *Online Multiplayer* (OMP), *Collision and Physics* (PHY), *Platform Independence Layer* (PLA), *Resources* (RES), *Third-party SDKs* (SDK), *Scene graph/culling optimizations* (SGC), *Skeletal Animation* (SKA), *Visual Effects* (VFX), *World Editor* (EDI).

[6] `https://github.com/minetest/minetest`

3.3 Subsystem Detection

In this step, we clustered all folders in each repository into the selected subsystems. When deciding which folders belong to a subsystem, we considered four pieces of information from each folder: its name, contents, documentation, and source code. We show an example of this decision process in Table 2.

Table 2: Subsystem detection example for Cocos2d-x.

Can we determine the subsystem of /cocos/editor–support/spine by:	
1) Folder name?	No, the name *spine* does not match or relate to the reference architecture.
2) Parent folder name?	No, the folder *editor-support* might be related to EDI, but we need more data to confirm.
3) Documentation?	**Yes**, according to docs: "Skeletal animation assets in Creator are exported from Spine". [7]
4) Source code?	No code analysis needed, subsystem detected on step 3.
Conclusion	*Skeletal Animation* (SKA)

3.4 Include Graph Generation

In parallel to detecting subsystems, we generated an *include* graph of each game engine using a two-pass algorithm. In the first pass, our analyser reads every source code file composing the game engine, collects all includes and outputs an *include* graph in the DOT graph description language. In the output DOT file, each row is an *include* relationship described as follows: */home/engine/source.cpp -> /home/engine/target.h*. The analyser attempts to resolve each relative *include* path into an absolute path. If the resolution fails, the analyser writes the path to another file called *engine-includes-unr.csv*.

In the original implementation of the approach [11], we read and resolved each of the unresolved *include* paths manually. However, repeating this operation for thousands of paths is time-consuming and error-prone. Therefore, we automated this step by adding a second pass to our analyser. In this pass, it loads *engine-includes-unr.csv*, iterates over each of its paths, and splits them by their folder delimiters. Then it searches for each part of the path, starting with the file name and moving towards the repository root folder. It repeats this search until it finds a match. Finally, the resolved absolute path is appended to the DOT file.

Some *include* paths inevitably remain unresolved because they refer to system or OS-specific libraries (e.g., *stdio.h*, *windows.h*) which do not belong to the game engine. In Cocos2d-x, all third-party dependencies are located in a separate repository. However, these paths do not contain code written by game engine developers, so their absence is not detrimental to the consistency of our architectural models.

[7] `https://docs.cocos.com/creator/manual/en/asset/spine.html`

3.5 Moose Model Generation

In this step, we merge the data collected in step 3 (the CSV file containing the detected subsystems) and step 4 (the DOT file containing the *include* graph) for each game engine. We build on Moose 10 [8], a platform for software analysis implemented in Pharo [9], a Smalltalk development environment. By importing the files into Moose, we create a Moose model, which is our architectural model.

3.6 Architectural Model Visualisation

Finally, we use Moose's "Architectural map" visualisation, which displays all subsystems and their relationships, to visualise each of the 10 selected game engines. By compiling the information from all generated "Architectural maps", we created a heatmap which we will show and explain in Subsection 4.2. We also used Gephi [10], a graph analysis software, to compute metrics such as in-degree and betweenness centrality, which are the main point of discussion in Subsection 4.1. We chose to use Gephi instead of Moose in this case because Moose does not come with graph analysis tools out of the box.

4 Results

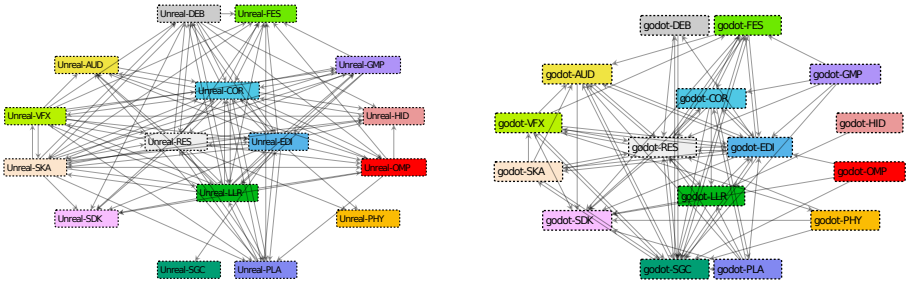
Figure 3 shows the architectural models of Godot and Unreal Engine. The high number of relationships and subsystems shows that both game engines are highly coupled and follow the reference architecture. In Godot, the most coupled subsystem is *Scene graph/culling optimizations* (SGC) because it centralizes files with diverse functionalities in its */scene/3d* folder, such as *camera.h* for *Low-Level Renderer* (LLR) and *physics_body.h* for *Collision and Physics* (PHY). In contrast, in Unreal's */Engine/Source/Runtime* folder, we observe LLR and PHY are divided in several distinct subfolders (e.g. *PhysicsCore*, *Renderer*).

While the "Architectural map" provides us with an overview of coupling in a given game engine, its density makes interpretation hard, especially because we want to understand how two or more subsystems are coupled. In the following subsections, we answer our RQs and explain how subsystem coupling can be identified and understood with the use of graph analysis and a coupling heatmap.

4.1 RQ1 - Which Subsystems More Often Couple with One Another?

We computed the in-degree for each subsystem of each game engine. Next, we computed the averages and sorted them in descending order. As we can observe in Figure 4, the top-five subsystems in average in-degree are: *Core* (COR), *Low-Level Renderer* (LLR), *Resources* (RES), *World Editor* (EDI) and, tied in 5th place, *Front End* (FES) and *Platform Independence Layer* (PLA).

[8] `https://github.com/moosetechnology`
[9] `https://pharo.org/`
[10] `https://gephi.org/`



(a) Unreal Engine (b) Godot

Fig. 3: Game engine architectural models generated with Moose 10.

The subsystems in the top-five act as a foundation for game engines because most of the other subsystems depend on them to implement their functionalities. Two subsystems in this list are graphics-related: *Low-Level Renderer* (LLR) and *Front End* (FES). We expected it because video games depend on visuals.

Same as the in-degree, we computed the average betweenness centrality, shown in Figure 5. This metric represents the extent to which a node lies in the path of others [2]. It helps us understand whether a highly coupled subsystem is an isolated occurrence, or if it consistently plays a central role within its respective system. Figure 5 shows that the top-five systems with the highest average betweenness centrality are: *Core* (COR), *Resources* (RES), *World Editor* (EDI), *Low-Level Renderer* (LLR) and *Platform Independence Layer* (PLA). For this reason, we decided to draw the top four subsystems in the centre of the architectural maps shown in Figure 3.

We observe that all subsystems with high in-degree also have high centrality, being *Front End* (FES) the only exception. While subsystems frequently depend on FES, we observe FES often depends only on *Core* (COR) and *Low-Level Renderer* (LLR) and therefore does not play the role of intermediate or "gatekeeper" between groups of subsystems. We further discuss subsystem "gatekeeping" observed in the COR subsystem in Subsection 4.2.

4.2 RQ2 - Do Game Engines Share Subsystem Coupling Patterns?

To answer this question, we created a heatmap which shows aggregated coupling counts from all architectural models (Figure 6). For example, if we take the first line from the top, we can observe the *Audio* (AUD) subsystem includes files from itself in eight game engines, and it includes files from COR in six game engines. We show the most frequent coupling pairs from the heatmap in Table 3.

While we applied our approach to 10 game engines, no square shows the value 10 in the heatmap's central diagonal. This happens because olcPixelGameEngine is fully decoupled. As an educational game engine, each of its subsystems was written in a single .h file, which is meant to be included by developers to their

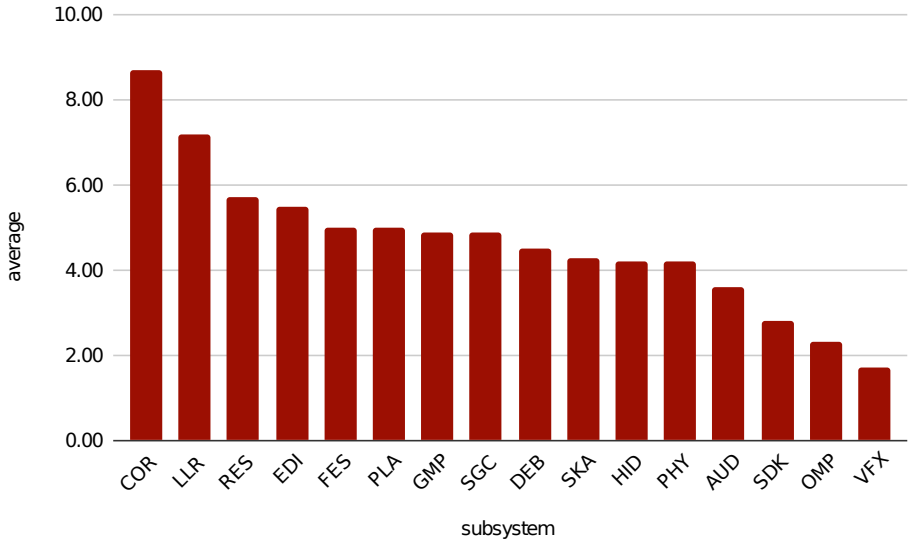


Fig. 4: Average subsystem in-degree.

own .cpp file. Also, not all subsystems were detected in all game engines, and therefore not all self-include nine times.

Table 3: The most frequent subsystem coupling pairs.

Pair	Count	Pair	Count	Pair	Count
GMP -> COR	9	FES -> COR	7	EDI -> FES	6
COR -> LLR	7	LLR -> COR	7	GMP -> FES	6
COR -> PLA	7	SKA -> COR	7	PLA -> COR	6
COR -> RES	7	AUD -> COR	6	RES -> COR	6

While the analysis of in-degree and betweenness centrality highlights which subsystems are fundamental, the heatmap shows how they work together. *Core* (COR) is the subsystem that most frequently includes others, and also the most frequently included. It is often reciprocally related to *Resources* (RES) and *Platform Independence Layer* (PLA), reflecting a "gatekeeper" role described in the reference architecture: when loading or saving game assets, RES uses PLA to interface with the OS and hardware such as the hard disk. We will further explore these frequent relationships in Section 5.

5 Discussion

By compiling the game engine coupling pattern information from Table 3 we observe a new architecture emerge. In Figure 7, we placed in the centre of the

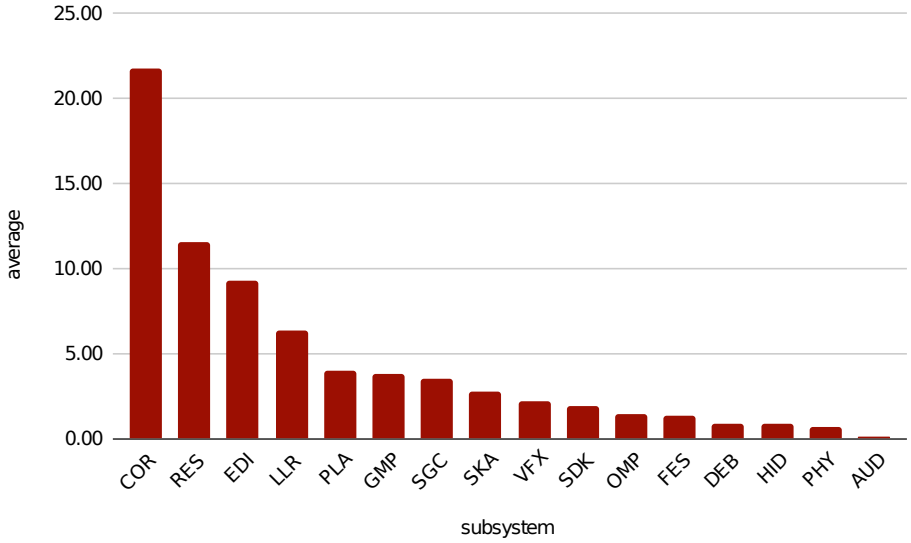


Fig. 5: Average subsystem betweenness centrality.

model the subsystems with the highest betweenness centrality, forming an inner core (dark red). Next, we placed other subsystems which appear in Table 3 in the outer core (light red). Finally, we placed the subsystems which do not appear in Table 3 in the outer core's periphery (white). All relationships shown in the diagram are among the most frequent, as shown in Table 3 and Figure 6. When there was a tie (e.g. two pairs had the same frequency), we chose the coupling pair with the highest sum of betweenness centrality.

In this emergent architecture, we observe the *Low-Level Renderer* (LLR) often inter-depends on *Core* (COR), which it uses to access functionality in the *Platform Compatibility Layer* (PLA) and the *Resources* (RES) subsystem. In Figure 1, we can observe these subsystems are also placed close to each other in the reference architecture.

While not part of the inner core, the *Front End* (FES) subsystem plays an important role. It is often included by the *World Editor* (EDI) and *Gameplay Foundations* (GMP), which are both visual interfaces between the user and the game engine. Because it manages UI elements which emit events and trigger actions throughout the system, *Front End* (FES) often depends on the event/messaging system in *Core* (COR).

More practically, the information provided by the architectural models and coupling patterns can be used by practitioners as follows:

– **Learning**: architectural model visualisations provide a friendly way for novice game engine developers to understand this kind of system and start developing their own subsystems or plugins.

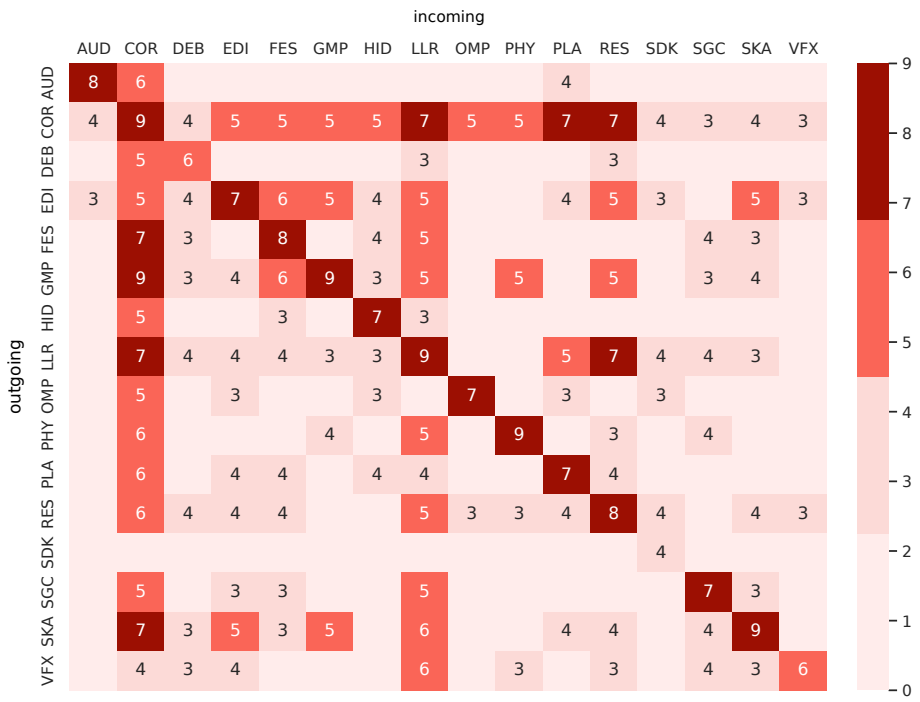


Fig. 6: Subsystem coupling heatmap showing aggregated coupling counts.

- **Refactoring**: game engine developers can refactor their code more safely by visualising how changes to a subsystem could impact the whole game engine.
- **Anomaly Detection**: a subsystem coupling heatmap can help game engine developers to find unusual or unexpected coupling patterns, and then improve the source code as necessary.
- **Reference Extraction**: game engine architects seeking to design a new engine can extract architectural models from similar systems and use them as references. This is useful both for large companies and small indie developers who develop tailor-made solutions, e.g. for performance.

6 Threats to Validity

First, the selected game engines may not be representative of all open-source game engines and the entire video game industry. Similarly, Gregory [5] is our reference architecture and we are aware other architectures exist, as explained in Section 2, even though not as detailed. Moreover, we acknowledge some modern game engine features, such as AR and VR support, were not present in the subsystem selection because they are not described in the reference architecture.

Subsystem detection was performed manually by the first author only, which may bias the detection process. To mitigate this issue, we intend to assign mul-

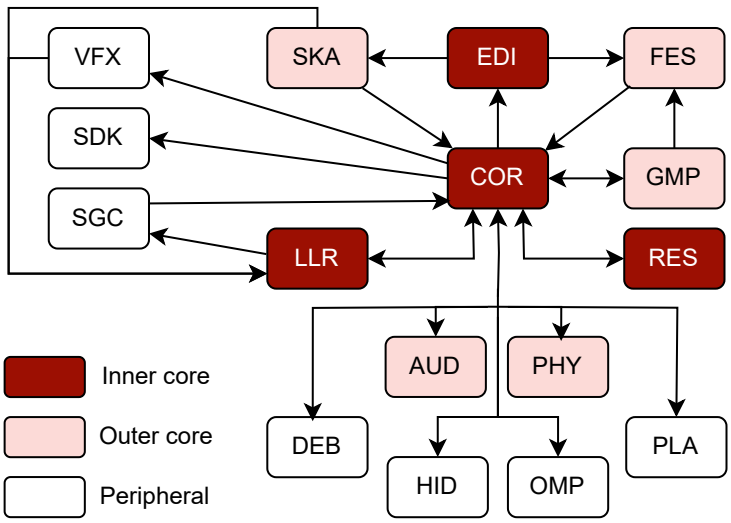


Fig. 7: Our emergent open-source game engine architecture.

tiple people to work in this step and later combine the results by consensus. We are also aware that our approach is dependent on the behaviour and metrics provided by Moose and Gephi, and changing them could also change the results and therefore our perception of these game engine architectures.

7 Conclusion

In this paper, we show that by generating and studying game engine architectural models, game engine developers can identify which subsystems are the centrepieces of their system and therefore give them proper maintenance. Additionally, by understanding frequent coupling patterns, game engine architects can take better decisions when extending existing game engines or creating custom-made solutions for a specific kind of video game or interactive experience.

In future work, we will apply our approach to a wider variety of game engines and subsystems. We will also conduct experiments with developers to determine to which extent the visualisations produced by our approach can help improve system understanding and maintainability. Finally, we intend to explore automated approaches to architecture recovery and other software quality metrics, such as cohesion and complexity.

Acknowledgements

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

References

[1] V. Agrahari and S. Chimalakonda. What's Inside Unreal Engine? - A Curious Gaze! In *14th Innovations in Software Engineering Conference*, pages 1–5, Bhubaneswar, Odisha India, February 2021. ACM. https://doi.org/10.1145/3452383.3452404.

[2] K. Badar, J.M. Hite, and Y.F. Badir. Examining the relationship of co-authorship network centrality and gender on academic research performance: the case of chemistry researchers in Pakistan. *Scientometrics*, 94(2):755–775, February 2013. ISSN 0138-9130, 1588-2861. https://doi.org/10.1007/s11192-012-0764-z.

[3] L. Bishop, D. Eberly, T. Whitted, M. Finch, and M. Shantz. Designing a PC game engine. *IEEE Computer Graphics and Applications*, 18(1):46–53, February 1998. ISSN 02721716. https://doi.org/10.1109/38.637270.

[4] I.T. Bowman, Richard C. Holt, and N.V. Brewster. Linux as a case study: its extracted software architecture. In *Proceedings of ICSE '99*, pages 555–563, Los Angeles, California, USA, 1999. ACM Press. ISBN 978-1-58113-074-4. URL `https://doi.org/10.1145/302405.302691`.

[5] J. Gregory. *Game engine architecture*. Taylor & Francis, CRC Press, Boca Raton, third edition edition, 2018. ISBN 978-1-138-03545-4.

[6] A.E. Hassan and R.C. Holt. A reference architecture for Web servers. In *Proceedings Seventh Working Conference on Reverse Engineering*, pages 150–159, Brisbane, Qld., Australia, 2000. IEEE Comput. Soc. ISBN 978-0-7695-0881-8. https://doi.org/10.1109/WCRE.2000.891462.

[7] P. Jaeyong and P. Changhyeon. Development of a multiuser and multimedia game engine based on TCP/IP. In *1997 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, PACRIM., 1987-1997*, volume 1, pages 101–104, Victoria, BC, Canada, 1997. IEEE. ISBN 978-0-7803-3905-7. https://doi.org/10.1109/PACRIM.1997.619911.

[8] D. Link, P. Behnamghader, R. Moazeni, and B. Boehm. The Value of Software Architecture Recovery for Maintenance. In *Proceedings of the 12th Innovations on Software Engineering Conference (formerly known as India Software Engineering Conference)*, pages 1–10, Pune India, February 2019. ACM. ISBN 978-1-4503-6215-3. https://doi.org/10.1145/3299771.3299787.

[9] James Munro, Cornelia Boldyreff, and Andrea Capiluppi. Architectural studies of games engines — The quake series. In *2009 International IEEE Consumer Electronics Society's Games Innovations Conference*, pages 246–255, London, UK, August 2009. IEEE. ISBN 978-1-4244-4459-5. https://doi.org/10.1109/ICEGIC.2009.5293600.

[10] C. Politowski, F. Petrillo, J.E. Montandon, M.T. Valente, and Y. Guéhéneuc. Are game engines software frameworks? A three-perspective study. *Journal of Systems and Software*, 171:110846, January 2021. ISSN 01641212. https://doi.org/10.1016/j.jss.2020.110846.

[11] G.C. Ullmann, Guéhéneuc Y., Petrillo F., Anquetil N., and Politowski C. An exploratory approach for game engine architecture recovery. In *7th International ICSE Workshop on Games and Software Engineering*, 2023.