An Exploratory Approach for Game Engine Architecture Recovery

1st Gabriel C. Ullmann
Concordia University
Montreal, Quebec, Canada
g_cavalh@live.concordia.ca

2nd Yann-Gaël Guéhéneuc
Concordia University
Montreal, Quebec, Canada
yann-gael.gueheneuc@concordia.ca

3rd Fabio Petrillo
École de Technologie Supérieure
Montreal, Quebec, Canada
fabio.petrillo@etsmtl.ca

4th Nicolas Anquetil
Univ. Lille, CNRS, Inria, Centrale Lille,
UMR 9189 - CRIStAL
Lille, France
nicolas.anquetil@inria.fr

5th Cristiano Politowski
École de Technologie Supérieure
Montreal, Quebec, Canada
cristiano.politowski@etsmtl.ca

Abstract—Game engines provide video game developers with a wide range of fundamental subsystems for creating games, such as 2D/3D graphics rendering, input device management, and audio playback. Developers often integrate these subsystems with other applications or extend them via plugins. To integrate or extend correctly, developers need a broad system architectural understanding. However, architectural information is not always readily available and is often overlooked in this kind of system. In this work, we propose an approach for game engine architecture recovery and explore the architecture of three popular open-source game engines (Cocos2d-x, Godot, and Urho3D). We perform manual subsystem detection and use Moose, a platform for software analysis, to generate architectural models. With these models, we answer the following questions: Which subsystems are present in game engines? Which subsystems are more often coupled with one another? Why are these subsystems coupled with each other? Results show that the platform independence, resource management, world editor, and core subsystems are frequently included by others and therefore act as foundations for the game engines. Furthermore, we show that, by applying our approach, game engine developers can understand whether subsystems are related and divide responsibilities. They can also assess whether relationships among subsystems are appropriate for the game engine.

Index Terms—software architecture, game engines, coupling, game development

I. INTRODUCTION

Game engines are tools to facilitate game development. They are systems composed of subsystems that interact with several audio, video, network, storage, and user interface devices. Besides making these subsystems work together with high performance, developers often integrate them with other software or extend them using plugins. However, to make the right choices, developers first need to understand the game engine architecture they are working with and how its subsystems relate to one another: "[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].

Even though these are relevant concerns on both game engine and game development, the topics of game engine subsystems coupling and architectural models have not been explored extensively. The literature mostly focuses on the implementation of specific game engine subsystems and not on their design and integration in architecture. In the context of graphics, for example, "there are a lot of sources of very good information from research to practical jewels of knowledge. However, these sources are often not directly applicable to production game environments or suffer from not having actual production-quality implementations." [4, p. xiv].

Moreover, popular game engines, such as Unity [1], are closed source, which makes it harder for developers to study them [12]. When developers have no access to architectural models, the software understanding process is mostly based on trial and error: "the only way for a developer to understand the way certain components work and communicate is to create his/her own computer game engine." [11].

In this work, we propose an approach for creating architecture models of game engines. Given a game engine, we cluster its source-code files into subsystems and create an *include* graph that holds the representation of the dependency relationships between these files. After creation, we check this graph for inconsistencies, resolving manually any *include* paths that could not be resolved automatically by our graph generator. Finally, we use this graph to generate an intermediate model that can be loaded into Moose 10, a platform for software analysis [2]. Using this platform, we generate an architectural model that allows us to visualise the relationships between subsystems. With such a model we can answer the following research questions:

- **RQ1**: Which subsystems are present in game engines?
- **RQ2**: Which subsystems are more often coupled with one another?

[1] https://unity.com/solutions/game
[2] https://moosetechnology.org

- **RQ3**: Why are these subsystems coupled with each other?

We apply our approach to three popular open-source game engines: Cocos2d-x [3], Godot [4], and Urho3D [5]. Our results show that game engines share the same set of base subsystems, even though some responsibilities are grouped differently on each system. For example, files from *Profiling and Debugging* and *Human Interface Devices* subsystems can be either placed in separate folders or merged together in a single "core" folder. We also observe that the *Platform Independence Layer*, *Resource Management*, *World Editor*, and *Core* subsystems are frequently included by others and therefore are foundational for game engines. Moreover, we observe that the *Core* subsystem frequently includes graphic-related subsystems to initialise these subsystems and access debugging information. We show and discuss more examples of these and other occurrences.

We show that, by applying our approach, game engine developers can understand whether subsystems are related and how responsibilities are/should be divided. Also, they can assess if these relationships help the subsystems to fulfil their responsibilities in the overall system. We intend to apply this approach to a larger set of game engines and subsystems. We also want to explore other relationships and metrics, such as cohesion and complexity, and their effect on game engine understanding and maintainability.

The remainder of the paper is organised as follows: Section II presents related work on game engines, architectural recovery, and coupling. Section III provides a description of our game engine analysis approach. Section IV shows and discusses the architectural models resulting from applying our approach to three game engines. Section V presents threats to validity and Section VI concludes with future work.

II. RELATED WORK

Works compared game engine aspects such as ease of use [3], available subsystems and target platforms [7], and suitability for a given platform [8] or game genre [9]. These comparisons are all tabular, listing game engines in one axis and relating them to subsystems in another. They are organized in this way to determine which game engines have the largest subsystem count, which subsystems are more commonly implemented and what benefit they bring to developers. But while discussing function, these works do not mention the architectural structures which support functionality. In this paper, we present an approach to obtain architectural models from game engines, which may be useful for both game engine developers and game developers who want to integrate or extend game engines.

Outside of game engines, the recovery and study of software architecture models have been applied to different software, such as the Linux kernel [2] and the IBM SQL/DS database [14]. In their work, [2] present a model showing the relationships among Linux subsystems. This visualisation, which

they called "conceptual architecture", helped them "understand the volume of detail in the implementation" and view "relationships between subsystems that are "meaningful" to developers". Similarly, in this work, we propose an approach to obtain game engine architectural models by detecting and relating their subsystems. We present our analysis of subsystem coupling with an architectural model generated by Moose, which represents each subsystem as a node in a graph, with edges between them representing dependency relationships.

Many software quality metrics exist and we choose coupling because coupling among subsystems directly impacts understanding and maintainability [13], which are important to consider because video games are "complex, emergent systems that are difficult to design and test" [6]. Also, "the more scattered the [game engine] feature implementations are in the architecture, the more likely it is that they get tangled with features to be added." [5].

In previous work, we compared the dynamic call graphs of Godot and Urho3D to understand the similarities and differences between their initialisation processes and division of responsibilities at the class level [12]. In this work, we analyse the architecture at the subsystem level, observing the static dependencies among files and not their method calls. We analyse the same two engines, Godot and Urho3D, and add a third one, Cocos2d-x.

III. APPROACH

We divide our approach into six steps (Figure 1): system selection, subsystem selection, subsystem detection, generation of the *include* graph, Moose model generation, and architectural model visualisation. We partially automate our approach. Data and scripts are available on GitHub [6]. In this section, we explain our steps and choices in detail.

1) **System selection**: We chose the game engines to study using GitHub. This step was done manually.
2) **Subsystem selection**: We chose game engine subsystems from the game engine development literature. This step was done manually.
3) **Subsystem detection**: For each selected game engine, we clustered files and folders into the selected subsystems. This process was done manually by a single person in this paper but could be done by multiple people and the results could be combined by consensus.
4) **Include graph generation**: We generated an *include* graph that encompasses all the files of a selected game engine. This step was done automatically with a dedicated tool, described in Section III-D.
5) **Moose model generation**: We used the data obtained from Steps 3 and 4 to generate a model that can be loaded into the Moose software analysis platform. This step was done semi-automatically with dedicated scripts.
6) **Architectural model visualisation**: We loaded each game engine model into Moose and then used its "Architectural map" visualisation to generate a visual
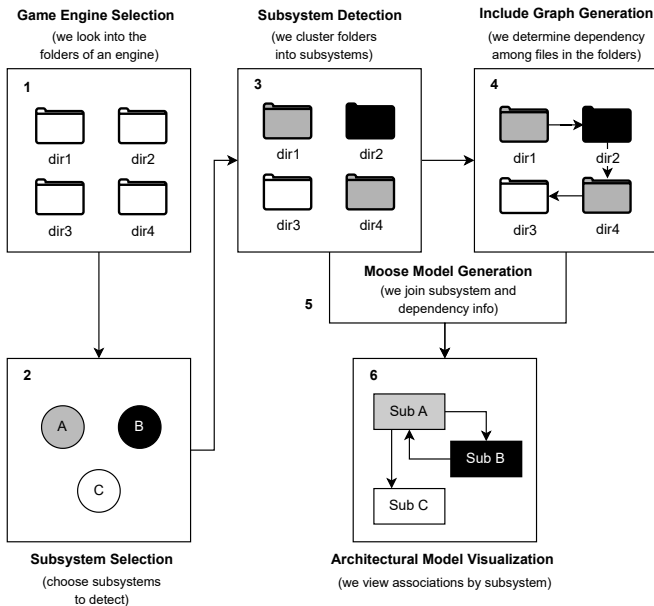
Fig. 1: Steps of our game engine analysis approach.

representation of the *include* graph and subsystems. This representation is an architectural model. This step was done semi-automatically with dedicated scripts.

A. System Selection

We searched for and selected game engine repositories in GitHub. We defined the following criteria for selecting game engine repositories:

- **Open source**: the repository must be publicly available and unarchived.
- **Game engine**: the repository must be tagged with the "game engine" tag.
- **Written in C++**: the repository must contain mostly C++ code because it is the most used programming language for game engine development [10].
- **General purpose**: the game engine must enable the creation of games of diverse game genres. We made this choice so our analysis reflects the architectural choices and needs of a broad range of games and is useful for a large set of game developers.

In our search result, we selected the top three engines with the highest sum of forks and stars and with the lowest numbers of header files and folders because our approach is partly manual. We thus obtained three game engines: Cocos2d-x, Godot, and Urho3D, as shown in Table I.

B. Subsystem Selection

We used the 15 subsystems described in the "Runtime Engine Architecture" proposed by [4, p. 33]. We chose these subsystems because they are well-known in the game engine development community. While Gregory uses the terms "component", "module" and "subsystem" interchangeably, we chose to use the word "subsystem" to describe a group of files and folders that contain code for a given system functionality.

Gregory states that "all commercial game engines have some kind of world editor tool", which is "a tool that permits game world chunks to be defined and populated" [4, p. 857] so we added a subsystem, which we called "World Editor", because it directly impacts game developers' work.

While the description of the responsibilities encompassed by each subsystem is beyond the scope of this paper, we summarise their definitions in the following. We also assign them a 3-letter code, which we use in the rest of this paper.

1) **Audio (AUD)**: manages audio playback and effects.
2) **Core Systems (COR)**: manages engine initialisation, contains libraries for math, memory allocation, etc.
3) **Profiling and Debugging (DEB)**: manages performance stats, debugging via in-game menus or console.
4) **Front End (FES)**: manages GUI, menus, heads-up display (HUD), and full-motion video playback.
5) **Gameplay Foundations (GMP)**: manages the game object model, scripting and event/messaging system.
6) **Human Interface Devices (HID)**: manages game-specific input interfaces, physical I/O devices.
7) **Low-Level Renderer (LLR)**: manages cameras, textures, shaders, fonts, and general drawing tasks.
8) **Online Multiplayer (OMP)**: manages match-making and game state replication.
9) **Collision and Physics (PHY)**: manages forces and constraints, rigid bodies, ray/shape casting.
10) **Platform Independence Layer (PLA)**: manages platform-specific graphics, file systems, threading, etc.
11) **Resources (RES)**: manages the loading and caching of game assets, such as 3D models, textures, fonts, etc.
12) **Third-party SDKs (SDK)**: enables interfacing with DirectX, OpenGL, Vulkan, Havok, PhysX, STL, etc.
13) **Scene graph/culling optimizations (SGC)**: computes spatial hash, occlusion, and level of detail (LOD).
14) **Skeletal Animation (SKA)**: manages animation state tree, inverse kinematics (IK), and mesh rendering.
15) **Visual Effects (VFX)**: enables light mapping, dynamic shadows, particles, decals, etc.
+ **World Editor (EDI)**: visual game world-building.

C. Subsystem Detection

In this step, we clustered all folders in each repository into the selected subsystems. When deciding which subsystem best suits a given folder, we considered all available information about the folder: name, contents, documentation, and source code. We show an example of this decision process in Table II.

D. Include Graph Generation

We generated an *include* graph for each repository using a script created by Irving[8]. This script generates a graph file in the DOT language, which contains each file's absolute path and its *include* relations with other files.

---

[7]https://docs.cocos.com/creator/manual/en/asset/spine.html
[8]https://www.flourish.org/cinclude2dot

TABLE I: Open-source game engines in GitHub which match our criteria.

Game Engine	Branch	Commit	Forks + Stars	Headers	Folders
Cocos2d-x	HEAD->v4	90f6542cf7	23,300	1,089	642
Godot	HEAD->3.4;tag:3.4.5-stable	f9ac000d5d	59,200	2,748	1,022
Urho3D	HEAD->master	feb0d90190	4,956	3,446	1,546

TABLE II: Subsystem detection example for Cocos2d-x.

/cocos/editor–support/spine	
Subsystem detected:	
1) By folder name?	No, there is no subsystem called *spine*. From the names of the files it contains, we can infer it is related to animation (SKA), but we need more data to confirm.
2) By parent folder name?	No, the folder *editor-support* might be related to EDI, but we need more data to confirm.
3) By documentation?	Yes, according to docs: "Skeletal animation assets in Creator are exported from Spine". [7]
4) By code?	No need to look at the code, subsystem detected on step 3.
Conclusion	SKA

The script attempts to resolve each relative *include* path into an absolute path. If the resolution fails, the script stores the path in a file. We then iterate over this file and resolve the paths manually. After resolution, we added the path to the script list of search paths and ran the script again (Figure 2).
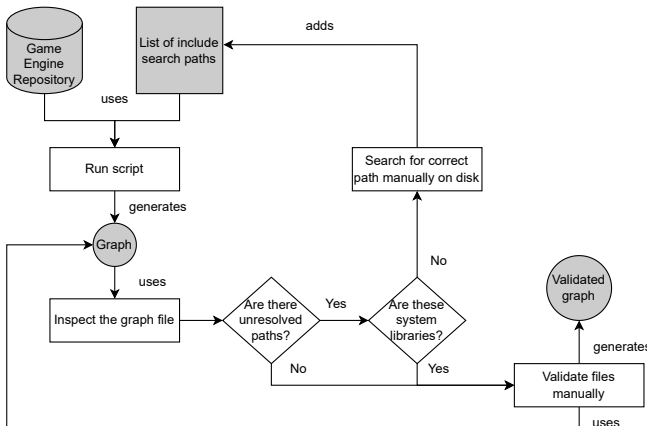

Fig. 2: Steps of *include* graph generation.

Some *include* paths remained unresolved because they referred to system or OS-specific libraries (e.g., *stdio.h*, *windows.h*). These headers are external to the game engine repositories so their absence does not influence our results.

E. Moose Model Generation

We used Moose 10, a platform for software analysis, to analyse the *include* graphs. We chose Moose because it generates visualisations and is built on top of Pharo[9], a

[9]https://pharo.org/

Smalltalk development environment that gives us the flexibility to customize existing tools and also write our own.

We wrote a Smalltalk program that takes as input a list of game engine files/folders, their assigned subsystem and *include* graph files. The union of these two data sources happens in Step 4 of Figure 1. The result of this union is a Moose model, which is a Pharo object containing source-code entities (files and folders) and *include* relations. We then used Moose and Pharo methods to analyse these models, such as counting the number of entities in a model, filtering them by name and type, or displaying them with the "Architectural map" visualisation, as described in III-F.

F. Architectural Model Visualisation

Moose provides an "Architectural map" to visualise entities and relationships in Moose models. It is a directed graph where each model entity is a node, and each *include* relationship is an edge. This graph is an architectural model.

The "Architectural map" visualisation is interactive and allows us to group files/folders by tags, identified by names and colours. We created one tag for each selected subsystem and assigned one tag to each entity in each model. The tag creation and assignment process was semi-automated. We wrote a script to gather all the folder names and their respective subsystem names (see Section III-C) from a CSV file and then create the tags. We created the CSV files manually.

Once we created all tags in Moose for each game engine, we selected all entities at the root folder, propagated them to the data bus, opened the "Architectural map" and finally selected all tags and relationships (called "associations" in Moose) for visualisation. We describe and comment on the results next.

IV. RESULTS

The application of our approach on the three selected game engines produces three architectural models showing the similarities and differences among subsystems and their relations, shown in Figure 3. We discuss the architectural models and answer our RQs.

A. RQ1: Detected Subsystems

In the architectural models shown in Figure 3, all subsystems are identified by coloured boxes named with the abbreviations in Section III-B. Subsystems that were not detected are represented by a dark grey box with light grey text. The four subsystems placed in the centre of the map are the ones most often included by others. We placed the subsystems manually in a circular pattern, organized alphabetically from left to right.

We identified 12 subsystems in Urho3D, 13 in Cocos2d-x and all 16 in Godot. While this may imply that Godot
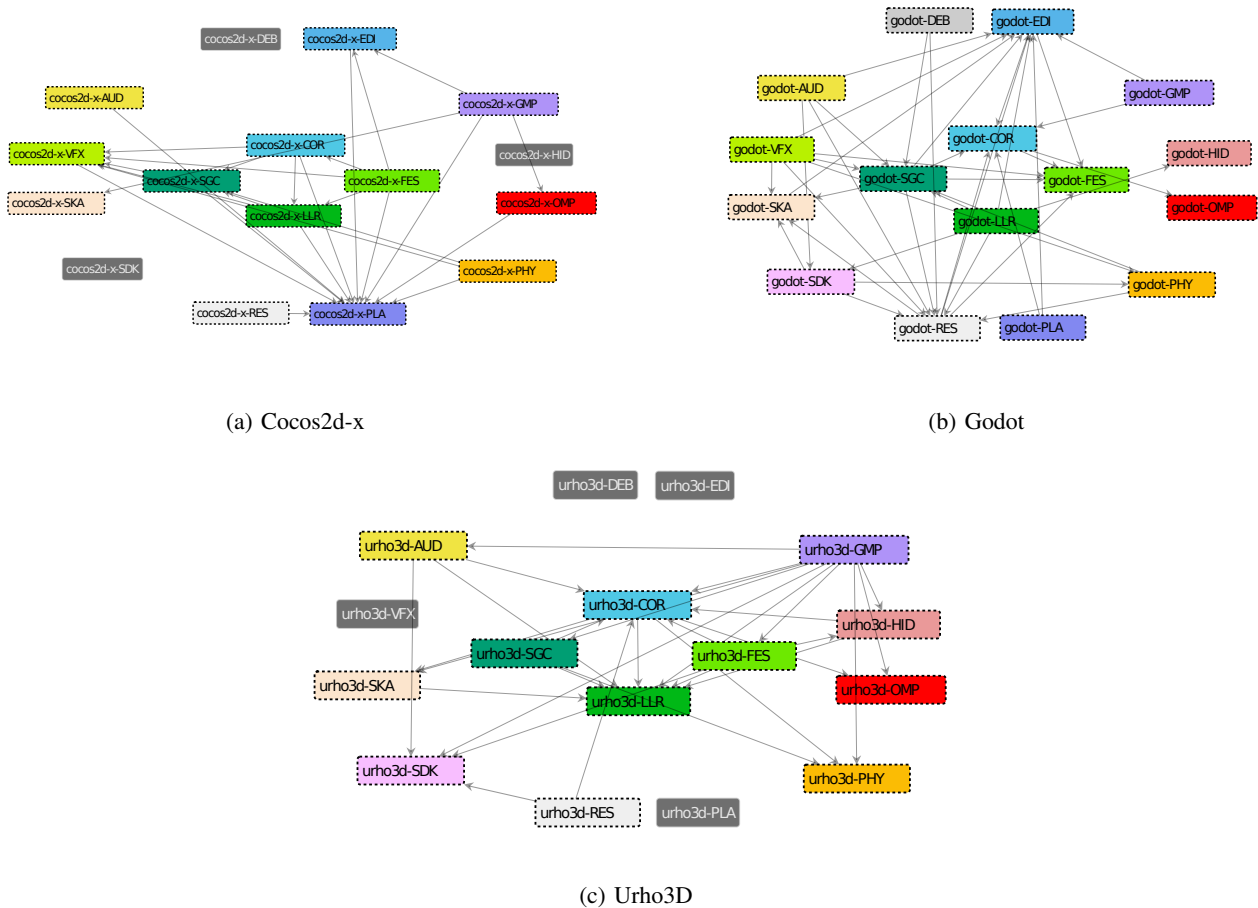
(a) Cocos2d-x

(b) Godot

(c) Urho3D

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

is more feature-complete than its counterparts, an undetected subsystem does not mean that its features are totally absent from the game engine. For example, we found out that DEB subsystem functionality is not always centralized in a file/folder, but rather spread among different subsystems. Concretely, we observed in Urho3D that both files *Renderer.h* and *DebugRenderer.h* inside the *Graphics* folder, clustered into the LLR subsystem. LLR has its own debugging utilities, no system-wide debug utilities exist.

We observed that some subsystems are present but their files are in a *base* or *core* folder, mixed with other, unrelated files. Our clustering was mainly based on folder names so these files were clustered together, even though they are not functionally related. For example, we confirmed that Cocos2d-x has HID functionality by inspecting its source files, which is backed by its documentation[10]. However, the files that contain HID-related classes are in a folder called *base*, which was clustered in the COR subsystem.

Urho3D presents several other cases where one of the selected subsystems is encompassed by another. Code for

MacOS compatibility, a PLA subsystem responsibility, can be found in folders such as *IO* (COR subsystem) and *ThirdParty* (SDK subsystem). VFX subsystem features are in files under *Graphics*, along with *Renderer.h*.

Finally, we observed cases where a subsystem was not detected because its files were in a separate folder or repository. In Urho3D, for example, the editor is represented by a single AngelScript file, *Editor.as*, which is in */bin/Data/Scripts* along with other code examples and away from */Source* in which most subsystem code resides. This subsystem is not written in C++ while, in this work, we only considered C++ files. In Cocos2d-x, third-party libraries are placed in a separate repository[11], which we excluded from our analysis.

B. RQ2 and RQ3: Coupling Among Subsystems

The architectural models in Figure 3 go beyond showing the existence of a set of subsystems or comparing their implementation with a theoretical model. Indeed, they are directed graphs whose incoming and outgoing edges we can study to understand subsystem relationships. We can also answer questions to explore and validate the architectures,

---

[10]https://docs.cocos2d-x.org/cocos2d-x/v4/en/event_dispatcher/keyboard.html

[11]https://github.com/cocos2d/cocos2d-x-3rd-party-libs-bin

such as "Should subsystem A include files from B? If so, why?".

Of the three architectural models, Godot is noticeably the densest one not only due to the number of subsystems but also edges, a total of 40. Urho3D is the second most coupled game engine, with 24 edges, followed by Cocos2d-x with 21. These numbers hint at the sizes and complexities of the selected game engines. Yet, they do not provide specific insights. In the following, we analyse the architectural models in-degree and out-degree and compare the frequencies of coupling between subsystems across game engines.

1) In-degree and Out-degree: In graph theory, the number of incoming edges of a node is called in-degree, and the number of outgoing edges is called out-degree. We noticed high in-degree nodes in all game engines, and so we decided to compute a node count for each node on each game engine to understand which subsystems support others.

On the bottom of the Cocos2d-x architectural model, we observe a node with a very high in-degree, the PLA subsystem, with 11 incoming and zero outgoing edges: 11 subsystems depend on PLA because the folder *cocos/platform* contains many files with utility classes used throughout the system. Some examples are *CCFileUtils.h* (file management), *CCImage.h* (image loading), *CCPlatformConfig.h* (checks for cross-platform compatibility), and *CCLuaEngine.h* (scripting engine). These files could be clustered into their specific subsystems but many account for OS-specific considerations (e.g., different line-breaking characters, asset root folders, etc.) so they belong to the PLA subsystem.

The RES subsystem has the highest in-degree in Godot, with seven incoming and four outgoing edges. It is followed closely by EDI, with eight incoming and two outgoing edges. These subsystems have high in-degrees, similar to what we observed in Cocos2d-x. The RES subsystem contains files with classes representing game entities from many subsystems, such as *audio_stream_sample.h* (AUD), *dynamic_font.h* (FES), *physics_material.h* (PHY/LLR), and *animation.h* (SKA).

In Godot, most files that depend on its EDI subsystem are custom engine modules, located in *godot/modules*. These modules extend editor functionality for different subsystems, e.g., GMP (*modules/gdscript/language_server/gdscript_language_server.cpp* includes *editor_log.h*) and VFX (*modules/lightmapper_cpu/lightmapper_cpu.cpp* includes *editor_settings.h*).

In Urho3D, the COR subsystem has the highest in-degree, with four incoming and five outgoing edges. We were surprised, however, to find out that besides providing functionality, Urho3D's and Cocos2d-x COR also depend on other subsystems. They relate to graphics-related subsystems, such as FES, LLR, or VFX, for the following reasons:

- **Performing initialisation.** In Urho3D, *Engine.cpp* includes *Graphics.h* (LLR) to initialise the graphics and rendering subsystems. The graphics subsystem always starts up unless the engine is run in headless mode.
- **Accessing debug information.** In Cocos2d-x, *CCConsole.cpp* contains functions to write in a command line interface, which includes *CCTextureCache.h* (LLR)

and *CCScene.h* (VFX) to print debugging information about the scene tree and texture cache to the console.

2) Coupling Among Subsystems: Besides analysing the game engines case by case, we want to understand what are the trends for game engine architecture in a broader sense. While we are aware that our selected engines may not represent the entire market and range of use cases of this kind of system, we believe that looking at the way subsystems relate throughout different systems may give us insights into how this kind of system should be built. Seeking to obtain such insights with regard to coupling, we counted the number of times each subsystem couples with each other on each selected engine. The result of this analysis is a matrix where both lines and columns represent subsystems. Zero represents no coupling, while one represents coupling exists for a given game engine. By summing up these matrices and colouring the highest values, we produced a heatmap (Figure 4).

In the x-axis of the heatmap, we can see how many times a subsystem includes another in all selected game engines. On the other hand, in the y-axis, we can see how many times a subsystem is included by another. For example, if we take the first line from the top, we can observe the AUD subsystem includes files from itself in three game engines, includes files from COR in two game engines, and so on.

The sum of values on each column (included by) and row (includes) is shown on Table III. We put the top four subsystems in the "included by" column in the centre of each architectural map from Figure 3. Since these subsystems are represented by nodes with many edges, putting them in the centre makes visualisation easier.

In the heatmap's central diagonal, we observe that not all values equal three. This happens because not all subsystems were detected in all game engines, and therefore not all self-include three times. This is the case for seven subsystems: DEB, EDI, HID, PLA, RES, SDK and VFX.

#	Included By		Includes	
1	COR	31	GMP	26
2	SGC	27	COR	25
3	LLR	26	FES	19
4	VFX	17	SGC	19
5	FES	16	EDI	18
6	RES	15	LLR	15
7	PLA	14	RES	15
8	EDI	13	VFX	15
9	AUD	12	PHY	13
10	SKA	11	SKA	13

TABLE III: Subsystems by include frequency.

By inspecting the heatmap from left to right, we can see four columns are clearly highlighted: COR, LLR, SGC and VFX, indicating they are the subsystems most often included by others. As described in Gregory's architecture, COR files are "useful software utilities" [4, p. 39] used throughout the system, and therefore it is no surprise that it is included by almost all subsystems.

Video games are highly dependent on visuals, and graphics are a cross-cutting concern. Therefore it is also no surprise that so many subsystems depend on LLR and SGC. The reasons
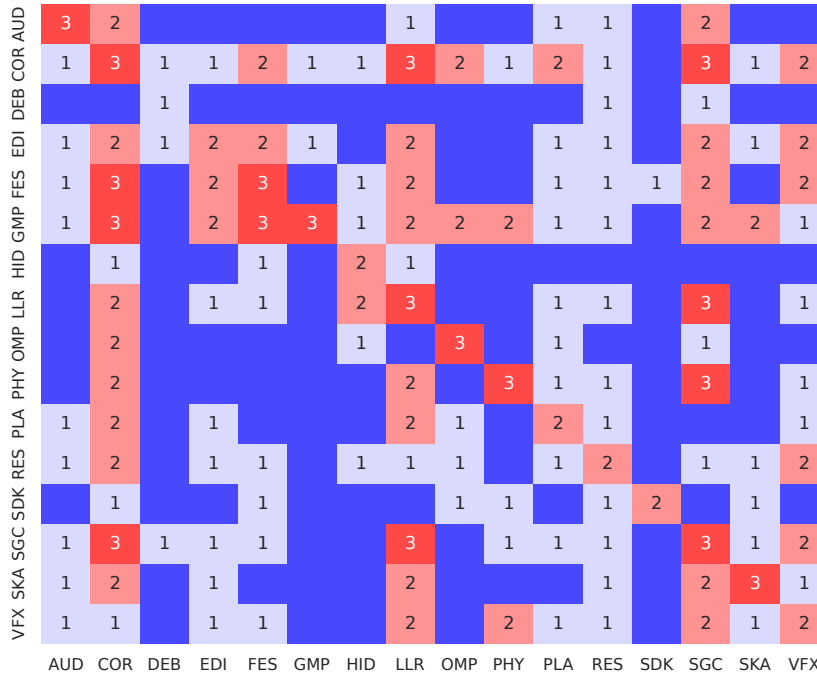
Fig. 4: Subsystem coupling heatmap for the selected game engines.

	AUD	COR	DEB	EDI	FES	GMP	HID	LLR	OMP	PHY	PLA	RES	SDK	SGC	SKA	VFX
AUD	3	2						1			1	1		2		
COR	1	3	1	1	2	1	1	3	2	1	2	1		3	1	2
DEB			1									1		1		
EDI	1	2	1	2	2	1		2			1	1		2	1	2
FES	1	3		2	3		1	2			1	1	1	2		2
GMP	1	3		2	3	3	1	2	2	2	1	1		2	2	1
HID		1		1			2	1								
LLR		2		1	1		2	3			1	1		3		1
OMP		2					1		3		1			1		
PHY		2						2		3	1	1		3		1
PLA	1	2		1				2	1		2	1				1
RES	1	2		1	1		1	1			1	2		1	1	2
SDK		1			1				1	1		1	2		1	
SGC	1	3	1	1	1			3			1	1	1	3	1	2
SKA	1	2		1				2				1		2	3	1
VFX	1	1		1	1			2		2	1	1		2	1	2

for this dependency only become clear when we look at the code. The most common example can be seen in Cocos2d-x: the FES subsystem, responsible for drawing UI elements, includes the renderer *CCRenderer.h*. Urho3D PHY and AUD subsystems include files from LLR to print debugging information. In Cocos2d-x, PHY includes *CCScene.h* because it holds a representation of the physics simulation, so it needs to add/remove physics objects.

V. THREATS TO VALIDITY

Firstly, we are aware the selected game engines may not be representative of all open-source game engines and the entire video game industry. While we used GitHub stars and forks as a metric to enable us to select systems which are relevant to the development community, we observed that the nature of the open-source game engine ecosystem itself poses challenges that do not exist in other kinds of systems.

For example, many open-source front-end web development frameworks exist and are being used in large industry-grade projects. Open-source game engines, on the other hand, are not used by large game development companies, being Unreal Engine one of the few exceptions. However, Unreal Engine has a large number of files and complex subsystems which would deserve a dedicated study.

In the context of subsystem selection, Gregory's "Runtime Engine Architecture" is our single architectural reference for game engine subsystems. We know that other authors have proposed different reference architectures.

During subsystem detection, we noticed some files and folders did not fit into any of the selected subsystems. Hence, we are aware our subsystem list is not exhaustive.

In this study, subsystem detection was performed manually by the first author only. As described in subsection III-D, we also resolved some include paths manually. Therefore, we are aware both of these steps may have inconsistencies and biases.

We used Moose for creating models of the selected game engine files, folders, and *include* relations. We used its built-in tools to query the entities in this model and generate architectural models. Other, similar tools exist and could provide different models.

VI. CONCLUSION

We proposed an approach of subsystem detection, model generation, and visualisation of architectural models of game engines. We answered the following questions on three open-source game engines, Cocos2d-x, Godot, and Urho3D:

a) RQ1: Which subsystems are present in game engines?: We detected all 16 selected subsystems [4] in the selected game engines, even though we did not detect every subsystem in every game engine. When we did not detect a subsystem, it was usually because its functionalities were provided by other subsystems (e.g., *Visual Effects* and *Low-Level Renderer*, *Profiling and Debugging* and *Core Systems*).

b) RQ2: Which subsystems are more often coupled with one another?: We observed that different subsystems play a core role in their system: in Cocos2d-x, it is *Platform Independence Layer*; in Godot, *Resource Management* and *World Editor*; in Urho3D, *Core Systems*. When looking at the

aggregated data of all game engines, *Core Systems*, *Low-Level Renderer*, *Visual Effects*, and *Scene Graph/Culling Optimizations* are the subsystems most often included by others. They are all related to either *Core Systems* or graphics. *Core Systems* often includes graphics-related subsystems.

c) RQ3: Why are these subsystems coupled with each other?: In Cocos2d-x and Godot, *Platform Independence Layer* provides functionalities used throughout the system, such as scripting, configuration and file system management. In Godot, *Resource Management* provides game object classes (e.g., animations, materials). *Core Systems* are included by many others and frequently include graphic-related subsystems for initialisation and debugging.

By applying our approach, game engine developers can understand whether subsystems are related and share responsibilities and whether this sharing helps fulfil the objectives of the game engine. By generating architectural models and coupling heatmaps of their systems, developers can understand which subsystems are available, how they depend on each other, and which subsystems are centralising responsibilities.

In future work, we intend to apply this approach to a larger set of game engines and subsystems. We will ask several developers to perform subsystem detection so as to decrease classification biases. We will improve the *include* graph generation, aiming for fully automatic resolution of *include* paths to increase analysis speed and accuracy. We will explore other software quality metrics, such as cohesion and complexity, and how they affect architectural understanding and system maintainability.

REFERENCES

[1] Vartika Agrahari and Sridhar Chimalakonda. What's inside unreal engine? - a curious gaze! In *14th Innovations in Software Engineering Conference (Formerly Known as India Software Engineering Conference)*, ISEC 2021, New York, NY, USA, 2021. ACM. ISBN 9781450390460. URL https://doi.org/10.1145/3452383.3452404.

[2] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Linux as a case study: its extracted software architecture. In *Proceedings of the 21st international conference on Software engineering - 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.

[3] Paul E. Dickson, Jeremy E. Block, Gina N. Echevarria, and Kristina 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. ISBN 978-1-4503-4704-4. URL https://doi.org/10.1145/3059009.3059013.

[4] Jason Gregory. *Game engine architecture*. Taylor & Francis, CRC Press, Boca Raton, 3 edition, 2018. ISBN 978-1-138-03545-4.

[5] Victor Guana, Eleni Stroulia, and Vina Nguyen. Building a game engine: A tale of modern model-driven engineering. In *2015 IEEE/ACM 4th International Workshop on Games and Software Engineering*, pages 15–21, Florence, Italy, 2015. IEEE. URL https://doi.org/10.1109/GAS.2015.11.

[6] Chris Lewis, Jim Whitehead, and Noah Wardrip-Fruin. What went wrong: A taxonomy of video game bugs. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, FDG '10, page 108–115, New York, NY, USA, 2010. ACM. ISBN 9781605589374. URL https://doi.org/10.1145/1822348.1822363.

[7] Prerna Mishra and Urmila Shrawankar. Comparison between Famous Game Engines and Eminent Games. *International Journal of Interactive Multimedia and Artificial Intelligence*, 4(1):69, 2016. ISSN 1989-1660. doi: 10.9781/ijimai.2016.4113. URL https://doi.org/10.9781/ijimai.2016.4113.

[8] Akekarat Pattrasitidecha. Comparison and evaluation of 3D mobile game engines. Master's thesis, Chalmers University of Technology, 2014. URL https://hdl.handle.net/20.500.12380/193979.

[9] Sanja Pavkov, Ivona Frankovic, and Natasa 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. ISBN 978-953-233-090-8. URL https://doi.org/10.23919/MIPRO.2017.7973518.

[10] Cristiano Politowski, Fabio Petrillo, João Eduardo Montandon, Marco Tulio Valente, and Yann-Gaël Guéhéneuc. Are game engines software frameworks? a three-perspective study. *Journal of Systems and Software*, 171:110846, 2021. ISSN 0164-1212. URL https://doi.org/10.1016/j.jss.2020.110846.

[11] 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, Opatija, Croatia, 2021. MIPRO. URL https://doi.org/10.23919/MIPRO52101.2021.9596801.

[12] Gabriel C. Ullmann, Cristiano Politowski, Yann-Gaël Guéhéneuc, and Fabio Petrillo. Game engine comparative anatomy. In *Entertainment Computing – ICEC 2022*, pages 103–111, Cham, 2022. Springer International Publishing. ISBN 978-3-031-20212-4. URL https://link.springer.com/chapter/10.1007/978-3-031-20212-4_8.

[13] F.G Wilkie and B.A Kitchenham. Coupling measures and change ripples in c++ application software. *Journal of Systems and Software*, 52(2):157–164, 2000. ISSN 0164-1212. URL https://doi.org/10.1016/S0164-1212(99)00142-9.

[14] Kenny Wong, Scott R. Tilley, Hausi A Muller, and M-AD Storey. Structural redocumentation: A case study. *IEEE Software*, 12(1):46–54, 1995.