Consensus in Game Engine Architectures: an Overview of Subsystem Coupling in Game Engines

Layan Etaiwi
Polytechnique Montréal
Montréal, Canada
mashael.etaiwi@polymtl.ca

Gabriel Ullmann
Concordia University
Montréal, Canada
ullmanng162@gmail.com

Yann-Gaël Guéhéneuc
Concordia University
Montréal, Canada
yann-gael.gueheneuc@concordia.ca

Sylvie Hamel
Université de Montréal
Montréal, Canada
hamelsyl@iro.umontreal.ca

***Abstract*—The video game industry is one of the most innovative, competitive, and rapidly growing industries. The industry's successes along with the increasing gamers' expectations result in always larger and more complex games. These games thus must be developed with game engines, which have become correspondingly more sophisticated. Today, game engine developers find game engine development challenging. To support the process of creating and maintaining game engines, we propose COSA, an approach based on applying a consensus algorithm to a set of game engine architectures. Our approach generates a model that suggests the most commonly used subsystems in game-engine architectures, ranked by their degree of coupling. The model can be used by developers as a starting point when deciding what subsystems to include when building a game engine and points out the most coupled subsystems, which can play an important role towards higher subsystems' maintainability and reusability. We evaluate the approach by comparing the results of our approach against predefined ground truth data. The result of our approach matches the subsystems defined in the ground truth data and it shows that the most coupled subsystems are core, low-level rendering, third-party SDKs, and world editor. Additionally, when comparing game engine architectures, we observe that most architectures are composed of nearly the same set of subsystems. Our approach COSA thus helps game-engine developers fairly compare their engines and focus their attention on the "important" subsystems.**

***Index Terms*—Consensus Algorithms, Rankings, Game Engine, Game Engine Architecture, Game Engine Development.**

I. Introduction

The history of video games began around six decades ago with computer scientists developing Spacewar in 1961 [15]. Since then, the video game industry rapidly grew and has become one of the most profitable in the market. By the end of 2022, the global gaming industry will reach over 3 billion people and generate over $196.8 billion in revenue [23].

Video game development was traditionally done by a small development team writing code from scratch with rarely any reusable game subsystems. However, as the industry experienced fast growth and its environment became highly competitive, users' expectations have grown, calling for advancements in game development techniques. As a result, the process of video game development became more complex and the development teams became much larger and more specialized. These changes have created the need for a framework that can facilitate and shorten the development time by providing generic, reliable and reusable software subsystems such as a rendering engine, physics engine, audio system, *etc.*, so the developers could focus their effort on developing the game mechanics, which are the game's logic. Today, such a framework is known as a game engine. Examples of popular game engines are Unity engine[1], Unreal engine[2], and CryENGINE[3]. Most game engines are written primarily in C++ programming language [6], however the internal implementation varies from engine to engine.

Game engine architectures differ from other software systems architectures because their subsystems are structured as a software stack of multi-layers of abstractions increasing layer by layer until the game mechanics are described [20]. The importance of such an architecture lies behind the necessity to manage the constantly changing requirements of games, recurrent releases, complexity of game engines and their libraries and APIs [29].

Nevertheless, in game engines, as in other types of systems, developers do not always create an architectural model before coding, so architectural decisions are only represented in the code and not readily available for analysis and comparison. Such comparisons could be useful for identifying commonalities and suggesting ways to improve existing engines [7].

In this study, we aim to reach three primary objectives: determining architectural commonalities between game engines, creating a model that presents a consensus of fundamental subsystems, and identifying the degree of coupling of these subsystems. To achieve these objectives, we perform architectural recovery on a set of selected open-source game engines. Recovered architectures help us identify subsystems that are part of each engine. We measure the degree of coupling of subsystems in each architecture by generating an include graph, and calculating coupling between objects (CBO) metric. Finally, we apply a consensus algorithm to the set of subsystems in the recovered architectures to obtain a model that provides a consensus fundamental subsystems ranked by their degree of coupling. We evaluate the model by comparing it to predefined ground truth data. This model can be used by developers to decide what subsystems they will develop when building a game engine, as well as the

[1]www.unity.com
[2]www.unrealengine.com
[3]www.cryengine.com

most coupled subsystems, so they can focus their programming efforts on minimising coupling for better maintainability and reusability. Results suggest that all identified subsystems are fundamental and should be taken into account when designing an engine architecture. Moreover, results discover that engine architectures contain similar subsystems. Lastly, they show that the most coupled subsystems are core systems, low-level rendering, third-party SDKs and world editor.

The remainder of the paper is organised as follows: Background and related studies are discussed in the next Section. The proposed approach of this study is explained in Section III. Validation of the approach is presented in Section IV. In Section V, results are shown and discussed. In Section VI we present possible threats to validity. Finally, conclusions are drawn in the last section.

II. Background and Related Work

In this section, we provide background information on architecture recovery, game engine architecture and consensus algorithms as well as summarise previous related works.

A. Software architecture recovery

Software architecture recovery is the extraction of high-level software architecture information from source code entities such as files and classes [32]. Software architecture consists of “the structure of components in a program or system, their interrelationships, and the principles and guides that control the design and evolution in time” [27].

In the process of architecture recovery, we “group implementation-level entities (e.g., files, classes, or functions) into clusters, where each cluster represents a component” [16]. Some authors refer to these components as subsystems [8], which is the naming we chose for this work.

Hierarchical, density-based and distribution-based clustering are popular approaches [32], applied in automatic or semiautomatic fashion. Entities can be clustered based on similarity regarding different attributes, such as number of references to variables, types or other entities [30], textual content [25], naming conventions or number of dependencies [11].

In this paper, we cluster the files into subsystems using their naming conventions, information provided by code comments, and documentation, as described in Sections III and IV.

While architecture recovery is usually mentioned in the context of understanding legacy enterprise systems, researchers have also applied it to a range of popular open-source codebases such as the Linux kernel [8], the Chromium browser [26], the Bash Unix Shell and the CVS version control system [30]. In this work, we do architecture recovery on open-source game engines.

Researchers and developers use architecture recovery techniques to retrieve lost architectural knowledge that guided the development of a software system in the past [12]. They can then use this knowledge to plan the system’s evolution and ease its program maintainability, understanding and knowledge transfer [31, 22].

B. Game Engine Architecture

The game engine architecture is the implementation and organisational structure of subsystems. This architecture varies from engine to engine, and there is currently no widely used standardised game engine architecture. However, we show in Figure 1 a high-level architecture created by Gregory. It lays out the runtime subsystems that make up a typical game engine into multiple layers of abstraction. The bottom layer interacts directly with the hardware and operating system. Starting in the resources subsystem and going up, all generic and re-usable game logic is represented, such as physics simulation, graphical rendering and audio playback. Finally, on top, the game-specific layer represents a specific game logic with limited reusability beyond the scope of the game being developed, such as in-game camera systems, artificial intelligence, weapon systems, *etc.*

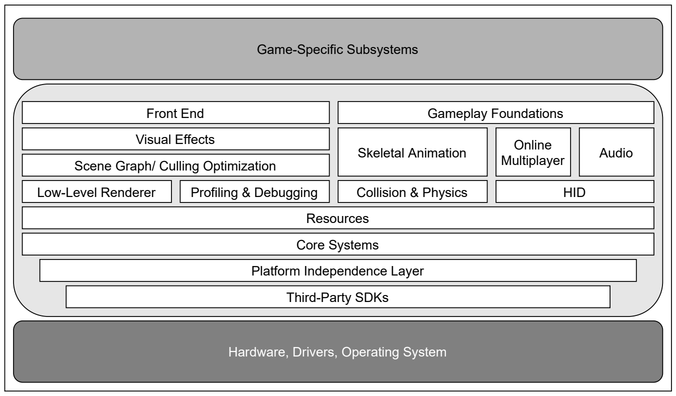

Fig. 1. Summary of a High-Level Game Engine Architecture Adapted from Gregory [17].

There are few studies on game engine architecture. While there are books on this subject [17, 14, 19], 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 [3]. Furthermore, while such literature is an excellent source of information for writing an engine, it is of little help when the requirements differ from the solution described. Similarly, according to [21]: “The current literature deals with the engine components, such as the behaviour specification, the scene render or the networking. Nevertheless, the game engine architecture connecting all these is a subject that has been barely covered.” Seeking to cover this gap, we aim to provide more insight into the importance of high-level game engine subsystems through our study.

C. Consensus Algorithms in a Nutshell

A consensus ranking is defined as aggregating a set of N different ranked lists of n elements into one ranking that orders the n elements closest to all of the N rankings within a specified distance [2]. Rankings can be incomplete: not all the n elements are ordered in every ranking or strictly ordered; some elements are ranked at the same position, *i.e.,* tied.

There are different distance measures, however many studies [5, 4] proposed using the generalized Kendall-τ distance [18] when measuring the distance between incomplete and rankings with ties. The generalized Kendall-τ distance, G, between two rankings r and s is computed as follows:

$$G(r,s) = \#\{(i,j) : i < j \wedge$$
$$((r[i] < r[j] \wedge s[i] > s[j]) \vee (r[i] > r[j] \wedge s[i] < s[j]) \vee \textbf{(1)}$$
$$(r[i] \neq r[j] \wedge s[i] = s[j]) \vee (r[i] = r[j] \wedge s[i] \neq s[j]))\} \textbf{ (2)}$$

That translates as the sum of the number of times elements i and j are ordered differently in the two rankings (1), or (2) the number of times the two elements are tied (in the same bucket) in one ranking, and not tied in the other.

The generalized Kemeny score K is defined as the sum of the generalized Kendall-τ distance between a ranking and every ranking in the input set R. It is computed as follows:

$$K(r, \mathcal{R}) = \sum_{s \in \mathcal{R}} G(r,s).$$

An optimal consensus ranking $r*$, for a set of rankings R is then:

$$\forall r \in \mathcal{R}_n : K(r^*, \mathcal{R}) \leq K(r, \mathcal{R}).$$

Anonymous *et al.* [4] studied and extensively compared 14 consensus algorithms on small and large datasets, both real and synthetic. Findings of their experiments showed that BioConcert algorithm [5], on both real and synthetic datasets, outperforms the other algorithms and produces results of the greatest quality. The KwikSort algorithm [1] placed in second place after BioConcert in terms of performance, especially when the dataset is exceedingly vast. The BioConcert and KwikSort algorithms produce the highest-quality outcomes, we decide to use them to generate the results of our study (see Sections III-D, IV-D) and compare their results.

D. *Consensus Algorithm Applications*

Querying for genes possibly linked to a certain disease, for example, could yield hundreds of results. This resulted in the need for a ranking solution that is able to order the results in a consensus fashion to help scientists focus their attention on other tasks. This problem led to numerous research works studying the consensus algorithms in the field of bioinformatics [5]. Nonetheless, it has been applied in other domains. For example, in the domain of artificial intelligence, researchers used rank aggregation to aggregate multiple users' preferences into one consensus ranking to help predict future preferences [24]. Similarly, in the domain of databases, rank aggregation has been used to eliminate noise and inconsistencies from data sets by aggregating contradictory clusterings from existing data sets into one consensus clustering [1]. In addition, different rank aggregation techniques have been studied in the area of Web engines querying to combine a set of search results into one ranking [13].

III. Approach

Figure 2 shows an overview of our approach, COSA (COnsensus Software Architecture). The approach is broken down into several steps: system selection, subsystem identification and detection (architecture recovery), subsystem coupling calculation and ranking, and finally consensus algorithm application to obtain a consensus of architecture subsystems ranked based on their degree of coupling.

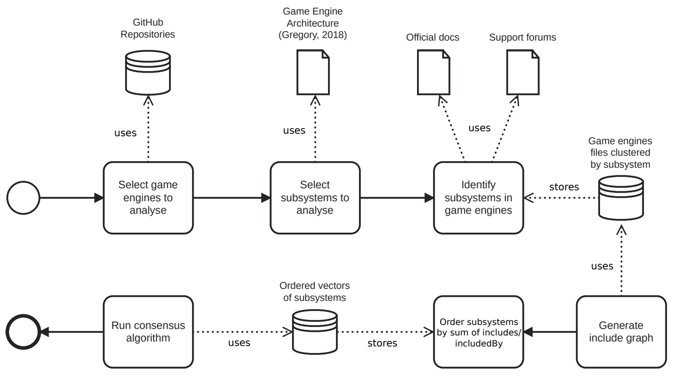
GitHub Repositories
Game Engine Architecture (Gregory, 2018)
Official docs
Support forums
uses
uses
uses
Game engines files clustered by subsystem
Select game engines to analyse
Select subsystems to analyse
Identify subsystems in game engines
stores
uses
Ordered vectors of subsystems
Run consensus algorithm
uses
stores
Order subsystems by sum of includes/ includedBy
Generate include graph

Fig. 2. A Summary of the Steps in the Proposed Approach

A. *System Selection*

In the first step of our approach, we search and select systems that can be studied. To be able to find systems that serve the purpose of the study and help apply the approach successfully, we establish a set of adequate selection criteria. Establishing criteria is different from system to system, but in this study exclusively we set criteria for game engine selection.

We define the following criteria for selecting game engines: • open source engines, • engines written in C++, • general-purpose engines, • engine repositories with the highest sum of forks, and stars and • unarchived repositories.

We decided to limit our search to engines with C++ as their primary programming language since C++ is the most widely used language for game engine development due to its performance and ability to reach low-level hardware [6]. In addition, we focused on general-purpose game engines as they target a broad range of game genres and therefore provide an overview of the features needed to make any game.

B. *Subsystem Identification and Detection*

In this step of the approach, we begin by defining ground truth data, which consists of a collection of basic subsystems that exist in any engine architecture. The ground truth data will be used later to validate the result of applying the consensus algorithm (Section V). Then, we perform system architecture recovery by determining what subsystems compose the architecture of the selected system. We achieve that by analysing each system repository's directories and files manually and clustering them according to the subsystems identified in the ground truth data.

C. Subsystem Coupling Degree Measurement

There are several metrics that can be used to measure the quality of a software system, and hence taken into consideration when developing the software, such as cohesion, inheritance, lines of codes, complexity, etc. However, in this work, besides finding the most fundamental subsystems to include in any system architecture, we opt for finding the degree of coupling of each subsystem. The underlying reason for prioritizing coupling over other metrics is that highly coupled software systems are difficult to maintain, understand, test, or even reuse. Making a change to such a system requires more effort and time due to the increased dependency between its classes, especially when new releases are frequently expected, as they are in the video game industry.

Coupling was first introduced by Stevens et al. and defined it as "the measure of the strength of association established by a connection from one module to another" [28]. In object-oriented programming, coupling is described as the dependency of one class on other classes. There are different measures that measure the degree of coupling between classes. Among these measures, *coupling Between Objects* (CBO) is the only measure that is a class-level measure, considers both import and export coupling, and determines the strength of coupling by the frequency of connections between classes [9]. CBO is a count of the number of classes that are coupled to a particular class [10]. Class A is coupled to class B if it references B and–or is being referenced by B.

To measure the degree of coupling of each subsystem in the recovered system architectures, we generate an include graph that shows the include relationships between files within each subsystem. We then calculate the degree of coupling by summing the number of include statements in each file and the number of times each file is included in other files (includedBy). Finally, we rank each architecture subsystem for each system based on their degree of coupling.

D. Consensus Algorithm Application

The final step of the approach is applying the selected consensus algorithms. As discussed in Subsection II-C, we chose to apply the BioConcert and Kwiksort algorithms. The algorithms take as input the output from the previous step, which is a set of architectures of subsystems ranked by their degree of coupling. The algorithms should output one consensus architecture of subsystems.

E. Implementation

We implemented our COSA approach as a tool that can be applied to any set of subsystems and metrics. It is an open-source implementation available on GitHub[4]

IV. COSA on Game Engines

We now apply our approach COSA to game engines to identify their architectural commonalities and differences. Thus, we want to demonstrate which subsystems are present in all game engine architectures, which are only present in some architectures, how tightly the subsystems are coupled, and provide a consensus architecture of fundamental subsystems. **We hence can help game engine developers in designing their engine architecture and focusing their efforts on developing loosely coupled subsystems.**

[4]URL removed because of double-blind review.

A. Game Engine Selection

Considering that most open-source game engine repositories are stored and shared on GitHub, we chose it as our repository database. We used GitHub's search function to search and filter game engines that meet our predefined selection criteria in Subsection III-A. From the result of our search query, we selected the top 10 engines with the highest sum of forks and stars. Names of the selected engines are listed in Table I.

Engine Name	Forks + Stars	First Commit Year
UnrealEngine	64100	2014
godot	59200	2013
cocos2d-x	23300	2010
o3de	6400	2021
Urho3D	4956	2011
gameplay	4900	2011
panda3d	4100	2000
olcPixelGameEngine	3963	2018
Piccolo	3892	2022
FlaxEngine	3613	2020

TABLE I
SELECTED GAME ENGINES ALONG WITH THE SUM OF THEIR GITHUB REPOSITORIES FORKS AND STARS.

B. Game Engine Subsystem Identification and Detection

We used the "Runtime Engine Architecture" proposed by Gregory for defining the ground truth data. We chose Gregory's book since it is well-known among industry professionals and it aims to provide an in-depth discussion of the major subsystems that make up a standard game engine [17]. Besides reviewing game development and game engine foundational concepts, the author drills down into each game engine subsystem and discusses implementation details, performance issues and how the structure of these subsystems in the code impact the player and developer experiences.

Gregory structures the engine architecture into 15 layered subsystems. While he divides each subsystem into a set of tools and smaller components, we choose to consider only the subsystems in the ground truth data. Although Gregory does not include the world editor (EDI) in the architecture, he emphasizes the importance of including EDI when building a game engine, thus we add EDI to the ground truth data. Table II lists the 16 defined subsystems.

During the analysis process of directories and files contained in each engine repository for subsystem detection, we eliminated files that are not written in C++ or do not contain functionalities related to any subsystems. On the other hand, files containing subsystems-related functionalities are clustered to their corresponding subsystems. To determine

Abbrev.	Name
AUD	Audio
COR	Core Systems
DEB	Profiling and Debugging
EDI	World Editor
FES	Front End
GMP	Gameplay Foundations
HID	Human Interface Devices
LLR	Low-Level Renderer
OMP	Online Multiplayer
PHY	Collision and Physics
PLA	Platform Independence Layer
RES	Resources (Game Assets)
SDK	Third-party SDKs
SGC	Scene graph/culling optimizations
SKA	Skeletal Animation
VFX	Visual Effects

TABLE II
GROUND TRUTH DATA OF SUBSYSTEMS

whether a file contains functionalities related to a specific subsystem, we examined the followings:

- Directories, files, classes and methods naming. For instance, if a directory is named Audio, this indicates that the contents included therein are a part of the AUD subsystem.
- Source code comments that describe the semantics of a file, class or method.
- Official engine's documentation, wiki and/or support forums. Documentation can provide information on how an engine is structured, what subsystems are included, description of directories, *etc.*

When a file contains functionalities that are related to more than one subsystem, it is clustered into just one of those subsystems. We choose the most corresponding subsystem based on the engine documentation and authors' professional experiences.

We also identified several files that belong to subsystems that do not exist in the ground truth data. In these cases, we clustered the files with the most corresponding subsystems, even when a direct relationship could not always be found. If none of the subsystems specified in the ground truth data corresponds to the functions provided in the file, we exclude the file.

Lastly, we clustered together any 3rd-party libraries under the " 3rd-party SDKs" subsystem. Even while these libraries might contain some functionalities that serve a less generic subsystems, they are libraries that are not developed and maintained by the game engine developers. For example, the Piccolo engine contains no audio subsystem, but it includes *stb* libraries, which contain audio decoding and synthesizing functionalities. These functionalities, however, are not used by Piccolo, even though they can be found on Piccolo's codebase. Therefore, the *stb* file in Piccolo was clustered into the SDK subsystem, and not the AUD subsystem. The clustering of engines files into subsystems is available in a repository[5]

[5]https://zenodo.org/record/7239232#.Y1RdfdLMIUE

C. Game Engine Subsystem Coupling Degree Measurement

We generated an include graph for each subsystem in all engine architectures by using a script created by Francis Irving[6], which generates a *Graphviz* graph from the set of clustered files from the previous step. We then created a script whose pseudo-code is depicted in Listing 1. The idea is, for each engine, to select files related to each subsystem, compute coupling between objects (CBO) for the subsystem, add the subsystems to a hashmap with the name of the subsystem as the key and CBO as the value, and order the hashmap by value. The result is a set of game engine architecture of subsystems ordered by their degree of coupling, presented in Figure 3

```
# each engine will do a call to this
def get_vector(engine_files, subsystems):
    hashmap = {}
    for subsystem in subsystems:
        files_filtered = filter_files(
            engine_files, subsystem)
        calculated_metric = calculate_metric(
            files_filtered)
        hashmap[subsystem] = calculated_metric
    return sort_hashmap_by_value(hashmap, "
        descending")
```

Listing 1. Pseudo-code of Computing Coupling Between Objects (CBO)

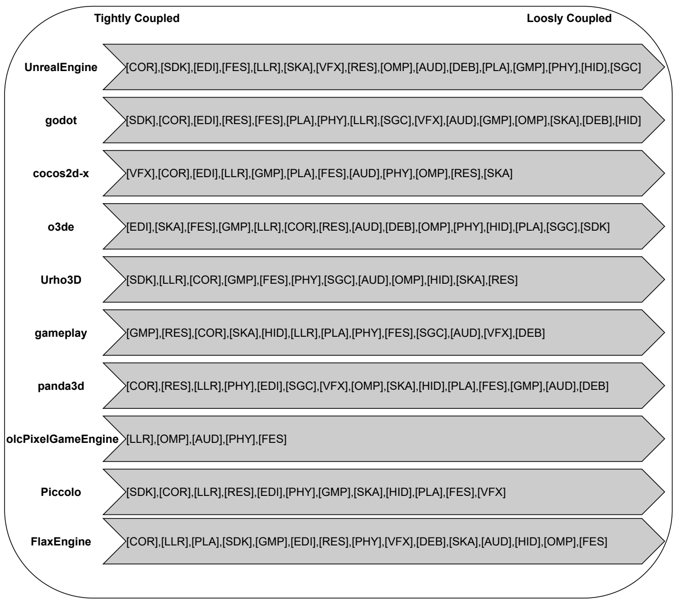
Tightly Coupled
Loosely Coupled
UnrealEngine
[COR],[SDK],[EDI],[FES],[LLR],[SKA],[VFX],[RES],[OMP],[AUD],[DEB],[PLA],[GMP],[PHY],[HID],[SGC]
godot
[SDK],[COR],[EDI],[RES],[FES],[PLA],[PHY],[LLR],[SGC],[VFX],[AUD],[GMP],[OMP],[SKA],[DEB],[HID]
cocos2d-x
[VFX],[COR],[EDI],[LLR],[GMP],[PLA],[FES],[AUD],[PHY],[OMP],[RES],[SKA]
o3de
[EDI],[SKA],[FES],[GMP],[LLR],[COR],[RES],[AUD],[DEB],[OMP],[PHY],[HID],[PLA],[SGC],[SDK]
Urho3D
[SDK],[LLR],[COR],[GMP],[FES],[PHY],[SGC],[AUD],[OMP],[HID],[SKA],[RES]
gameplay
[GMP],[RES],[COR],[SKA],[HID],[LLR],[PLA],[PHY],[FES],[SGC],[AUD],[VFX],[DEB]
panda3d
[COR],[RES],[LLR],[PHY],[EDI],[SGC],[VFX],[OMP],[SKA],[HID],[PLA],[FES],[GMP],[AUD],[DEB]
olcPixelGameEngine
[LLR],[OMP],[AUD],[PHY],[FES]
Piccolo
[SDK],[COR],[LLR],[RES],[EDI],[PHY],[GMP],[SKA],[HID],[PLA],[FES],[VFX]
FlaxEngine
[COR],[LLR],[PLA],[SDK],[GMP],[EDI],[RES],[PHY],[VFX],[DEB],[SKA],[AUD],[HID],[OMP],[FES]

Fig. 3. Game Engine Architectures of Subsystems Ordered by Coupling Degree

D. Consensus Algorithm Application

We observe that not every engine includes all 16 identified subsystems, which in turn resulted in generating incomplete rankings; not all the elements exist in every ranking. To deal with incomplete rankings when applying consensus algorithms, works proposed two normalization techniques [4]. Projection technique; keeps in each ranking only the common

[6]www.flourish.org/cinclude2dot/

elements that exist in all rankings, and unification technique; adds missing elements from each ranking at the end of the ranking in one bucket. We do not use the projection technique as it leads to the removal of some subsystems. We thus apply the unification technique to complete the rankings, which adds a bucket to the end of the rankings with missing elements. For example, completing the ranking of Piccolo engine, adds a bucket with the missing subsystems as follows:

$$\begin{aligned} Piccolo \quad = \quad & [[SDK], [COR], [LLR], [RES], [EDI], \\ & [PHY], [GMP], [SKA], [HID], [PLA], \\ & [FES], [VFX], [SGC, AUD, OMP, DEB]] \end{aligned}$$

After completing the rankings, we apply the BioConcert and Kwiksort algorithms to the set of complete rankings. While results from both algorithms were very similar, the Kwiksort algorithm generated a result with a smaller generalized Kemeny score. Given that an optimal consensus ranking is the one with the smallest possible generalized Kemeny score, we adopt the result of the Kwiksort algorithm as our COSA for game engines.

V. Results and Discussion

In this Section, we present and discuss the result of applying the consensus algorithm to a set of subsystems of 10 game engine architectures that are ranked according to their coupling degree (tight to loose). The discussion revolves around two axes; first, we discuss commonalities between architectures and the most essential subsystems; second, we explore the most coupled subsystems and the underlying cause for their tight coupling.

A. Commonalities and Consensus Architecture of Subsystems

When comparing how similar game engine architectures are in terms of subsystems (refer to Figure 3), evidently most engine architectures, excluding olcPixelGameEngine, are composed of nearly the same subsystems. All subsystems exist in two engines, fifteen appeared in three engines, while twelve of the subsystems are part of the remaining four engine architectures. This confirms the success of our architecture recovery method (Subsection IV-B) and that we were able to identify 80% of the subsystems in all architectures.

According to Gregory, while details of architectures and implementation differ from engine to engine, all game engines must eventually include a set of main subsystems, such as rendering engine, physics engine, audio system, etc [17]. Thus, this is an evident explanation for the commonality in engine architectures.

OlcPixelGameEngine, on the other hand, contains only five subsystems. The absence of more major subsystems occurs because olcPixelGameEngine is not a complete engine since it was developed by the YouTube channel OneLoneCoder for the purpose of teaching game engine programming. From the top 10 selected open-source game engines, this is the only project that will not be developed further into a complete engine.

Figure 4 presents the result of applying the Kwiksort algorithm; a fundamental set of game engine subsystems ordered in a consensus fashion in accordance with their degree of coupling. Comparing the result of applying the consensus algorithm to the ground truth data, the consensus result identified all 16 subsystems as essential subsystems, and developers should therefore consider them in the architecture design when developing a modern game engine. In light of the fact that the majority of the subsystems were shared by the majority of the engine architectures, this not only confirms the validity of the consensus architecture of subsystems, but also ensures the importance of each subsystem as it plays a distinct role in the architecture. Hence, the proposed approach, COSA, fulfills our objectives of determining architectural commonalities between game engines and providing engine developers with a consensus architecture of a set of subsystems.

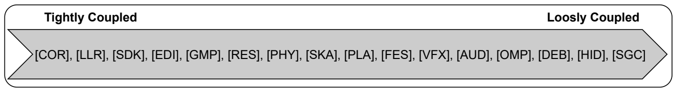
Tightly Coupled
Loosely Coupled
[COR], [LLR], [SDK], [EDI], [GMP], [RES], [PHY], [SKA], [PLA], [FES], [VFX], [AUD], [OMP], [DEB], [HID], [SGC]

Fig. 4. The Consensus Result of Applying the Kwiksort Algorithm

As mentioned in Subsection IV-B, we detected files that belong to unidentified subsystems in the ground truth data. A list of the discovered subsystems, their locations (game engines), and a relevant subsystem from the ground truth data are described in Table III. We conclude that Gregory's engine architecture is not inclusive, and it can be expanded to comprise more subsystems provided by modern game engines. In spite of the discovery of new subsystems, these subsystems can not be regarded as essential since they are found in relatively few engines.

B. Degree of Coupling

The result of applying the consensus algorithm presents a consensus of the most coupled subsystems across all game engine architectures (Figure 4). The four most coupled subsystems are core systems, low-level renderer, 3rd-party SDKs, and world editor. We now discuss the underlying reasons behind the tight coupling and draw examples from the engines' files.

a) Core Systems and 3rd-Party SDKs: As described by Gregory, these subsystems are responsible for low-level operations such as memory allocation, file I/O, system calls, as well as communication with graphic and audio APIs. Therefore, they serve as support for all other high-level subsystems such as audio, low-level renderer and visual effects. This, in turn, means that files belonging to this subsystem are included by many other files. In Unreal Engine, for example, the most coupled core system file is *CoreMinimal.h*, and it is included by 14051 files while including only 151 files.

b) Low-Level Renderer: It came as no surprise to us that the renderer subsystem is identified as one of the most coupled. It is responsible for producing 2D or 3D animated graphics we see on screen in all games. From the game objects to the UI of the world editor to everything needs to be drawn and continuously updated.

Discovered Subsystems	Game Engines	Relevant Subsystem from Ground Truth
Code editor, Multi-user synchronization, Project creation and “cooking”, CLI	UnrealEngine, o3de, panda3d	EDI
Cache, source control	UnrealEngine	RES
Cvars, graphs (data structure), Video subtitling and timecoding, Analytics, Media streaming	FlaxEngine, godot, o3de, panda3d, UnrealEngine	COR
Code hot reloading, visual scripting, assembler/compiler	FlaxEngine, godot, UnrealEngine	GMP
Virtual production (video post-production)	UnrealEngine	VFX
Screenshot capture	FlaxEngine	LLR
Foliage simulation	FlaxEngine, UnrealEngine	PHY
VR, AR, XR	godot, UnrealEngine	
Advertisement	UnrealEngine	
Cryptography	UnrealEngine, FlaxEngine	
Database	UnrealEngine, Urho3d, o3de	
Virtualization	UnrealEngine	
Cloud services integration	o3de	

TABLE III
NEWLY DISCOVERED SUBSYSTEMS

c) World Editor: We observed that the editor has a high degree of coupling because it provides a visual interface to many other subsystems. Therefore, its files include many files from other subsystems, and they are included by these subsystems as well. Observing the files names in the Godot editor subsystem, we certainly notice that this subsystem serves many other subsystems within the engine. Examples of these files are: *animation_tree_editor_plugin.cpp*, *audio_stream_editor_plugin.cpp*, *particles_2d_editor_plugin.h*, *visual_script.cpp*.

In terms of most coupled game engines, we observe that UnrealEngine, panda3d and Urho3d are the three most coupled game engines. We noticed a correlation between the game engine coupling and metrics such as the number of files and forks+stars on GitHub. While this does not imply causation, this may be evidence that as engines and teams working on them grow in size, so does coupling.

VI. Threats To Validity

This section of the paper discusses possible construct and external threats to the validity of our findings.

Construct Validity

Subsystem Identification: Our subsystem identification in the ground truth data was based on Gregory’s definition of “Runtime Engine Architecture”. The list of 16 subsystems is not exhaustive. However, other than Gregory’s book, there is no academic or technical research on game engine architecture. In the future, we intend to thoroughly examine, in collaboration with game engine developers, the source code of game engines to verify the accuracy of this list and possibly include more essential subsystems.

Clustering Bias: One of the authors manually clustered the game engine repository files and directories into subsystems. Despite the fact that the author’s judgement was based on documentation attached to the repository, the author’s judgement could be biased.

Multi-Subsystems: During the file clustering process, we encountered files that could belong to more than one subsystem. The author relied on professional experience and engine documentation to cluster these files into the most corresponding subsystem. There is also a risk that the author’s judgement could be biased.

Unidentified Subsystems: During the file clustering process, we discovered a few files that might belong to subsystems that are not part of Gregory’s engine architecture. We either discarded these files or clustered them into subsystems with comparable functionalities. As discussed in “*Subsystem Identification*”, we intend in the future to extensively study game engines to detect undiscovered or newly created subsystems.

External Validity

Generalisation: We are confident that our approach can be generalised and applied to any other systems, as well as a wider range of game engines. In this work, we investigated 10 open-source game engines. Our selection of game engines might not represent all segments of the market. Popular engines like Unity and Source were left out of our investigation because they are not open-source and therefore analysing their source code is impossible. We reduced this threat by selecting general-purpose engines that serve all genres of games. Additionally, we recognise that most game engine development is closed-source. Therefore, the results may not apply to all game engines, but ought to be valid for open-source game engines.

Reliability: To increase the reliability of our findings, we made all collected data and scripts available online in a public repository[5]. This allows other researchers to replicate and enhance our findings.

VII. Conclusion And Future Work

Video games are becoming more and more popular and rapidly evolving to provide gamers with new experiences that they have never had before. As a result, game development became more technically complex. Today, game engines are a key tool to facilitate the process of building high-quality games. However, when building a game engine, developers struggle with the lack of knowledge about engine architecture in general, which engine subsystems to incorporate into the architecture, and managing architecture complexity to meet quality standards.

This paper provided an overview of the commonalities and differences between game-engine architectures in terms of comprised subsystems. Additionally, it defined a consensus of architecture subsystems and their degree of coupling to serve as a foundation for fair comparisons and discussions among game-engine developers.

Accordingly, the main objective of the approach was to provide developers with a high-level comparison between game engine architectures, help developers decide what subsystems to include in the architecture when building an engine, and present them with the most coupled subsystem so they can focus their development work on these subsystems to improve maintainability and reusability.

We presented an approach, COnsensus Software Architecture (COSA), that provides a ranking of the subsystems of any set of software architectures, according to some chosen metrics. We described COSA and applied it to 10 game engines. To identify what subsystems compose an engine, we performed an architecture recovery on 10 open-source game engines by manually analyzing their repositories and clustering the containing files into predefined subsystems. We later built an included graph from the extracted subsystems to calculate their coupling degree and ranked each engine subsystem by its degree of coupling. To generate our model, we applied the Kwiksort consensus algorithm to the ranked lists of architectures. We finally investigated the recovered architectures and compared their commonalities.

Our review of related works showed a lack of research studies about game engine architecture. Yet we showed that game engine architectures share many common subsystems. In fact, nearly all investigated game engines include the same subsystems in their architectures (COR, RES, FES, PHY, LLR, AUD, GMP, SKA). Additionally, our model concluded that all 16 predefined subsystems are essential and should be taken into consideration when building an advanced game engine. Besides, the result of applying the consensus algorithm showed that the most coupled subsystems are core systems, 3rd-party libraries, world editors, and low-level renderers.

Game-engine developers can compare fairly their game engines and focus on the "important" subsystems relative to other game engines. For example, some developers could decide to put their effort into the core subsystem given its importance in all game-engine architectures while others could decide to implement the subsystems missing in their game engines while yet others could restructure their game engines to reduce undue coupling among subsystems and follow hard-learned design choices from other game engines.

In future work, we intend to interview game developers to get their perspectives on knowing what subsystems to include in your engine architecture and how coupled these subsystems can help them with their game engine development process. There are also two possible research interests. The first is to expand the study by investigating more game engines of various types, in different programming languages and using other software quality metrics such as cohesion and complexity. The second is to perform a comparative study of game engine architectures for the purpose of finding why not all subsystems exist in each engine architecture. Finally, we will apply COSA to other sets of software architectures, like those of Web browsers.

References

[1] Nir Ailon, Moses Charikar, and Alantha Newman. Aggregating inconsistent information: ranking and clustering. *Journal of the ACM (JACM)*, 55(5):1–27, 2008.

[2] Alnur Ali and Marina Meilă. Experiments with kemeny ranking: What works when? *Mathematical Social Sciences*, 64(1):28–40, 2012.

[3] Eike Falk Anderson, Steffen Engel, Peter Comninos, and Leigh 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. ISBN 978-1-60558-218-4. doi: 10.1145/1496984.1497031. URL https://doi.org/10.1145/1496984.1497031. event-place: Toronto, Ontario, Canada.

[4] Anonymous. Anonymous. *Anonymous*, 2015.

[5] Anonymous. Anonymous. In *Anonymous*, 2022.

[6] Anonymous. Anonymous. *Anonymous*, 2022.

[7] Anonymous. Anonymous. In *Anonymous*, 2022.

[8] 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, United States, 1999. ACM Press. ISBN 978-1-58113-074-4. doi: 10.1145/302405.302691. URL http://portal.acm.org/citation.cfm?doid=302405.302691.

[9] Lionel C. Briand, John W. Daly, and Jurgen K Wust. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on software Engineering*, 25(1):91–121, 1999.

[10] Shyam R Chidamber and Chris F Kemerer. Towards a metrics suite for object oriented design. In *Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 197–211, 1991.

[11] Stéphane Ducasse and Damien Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, 2009.

[12] Juan C Duenas, W Lopes de Oliveira, and Juan Antonio de la Puente. Architecture recovery for software evolution. In *Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering*, pages 113–119. IEEE, 1998.

[13] Cynthia Dwork, Ravi Kumar, Moni Naor, and Dandapani Sivakumar. Rank aggregation methods for the web. In *Proceedings of the 10th international conference on World Wide Web*, pages 613–622, 2001.

[14] David H. Eberly. *3D game engine design: a practical approach to real-time computer graphics*. Elsevier/Morgan Kaufmann, Amsterdam ; Boston, 2nd ed edition, 2007. ISBN 978-0-12-229063-3.

[15] Jeffrey Fleming. Down the hyper-spatial tube: Spacewar and the birth of digital game culture. https://http://www.gamasutra.com, 2007.

[16] Joshua Garcia, Igor Ivkovic, and Nenad Medvidovic. A comparative analysis of software architecture recovery techniques. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 486–496, Silicon Valley, CA, USA, November 2013. IEEE. ISBN 978-1-4799-0215-6. doi: 10.1109/ASE.2013.6693106. URL http://ieeexplore.ieee.org/document/6693106/.

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

[18] Maurice G Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

[19] Eric Lengyel. *Foundations of game engine development*. Terathon Software LLC, Lincoln, California, 2016. ISBN 978-0-9858117-4-7 978-0-9858117-5-4.

[20] Dario Maggiorini, Laura Anna Ripamonti, Eraldo Zanon, Armir Bujari, and Claudio Enrico Palazzi. Smash: A distributed game engine architecture. In *2016 IEEE Symposium on Computers and Communication (ISCC)*, pages 196–201. IEEE, 2016.

[21] Carlos Marin, Miguel Chover, and Jose M. Sotoca. Prototyping a game engine architecture as a multi-agent system. In *Computer Science Research Notes*. Západočeská univerzita, 2019. ISBN 978-80-86943-38-1. doi: 10.24132/CSRN.2019.2902.2.4. URL http://wscg.zcu.cz/wscg2019/2019-papers/!!_CSRN-2802-4.pdf.

[22] Hausi A Müller, Mehmet A Orgun, Scott R Tilley, and James S Uhl. A reverse-engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, 1993.

[23] Newzoo. Global games market report. https://newzoo.com/products/reports/global-games-market-report, 2022.

[24] David M Pennock, Eric Horvitz, C Lee Giles, et al. Social choice theory and recommender systems: Analysis of the axiomatic foundations of collaborative filtering. *AAAI/IAAI*, 30:729–734, 2000.

[25] Michele Risi, Giuseppe Scanniello, and Genoveffa Tortora. Using fold-in and fold-out in the architecture recovery of software systems. *Formal Aspects of Computing*, 24(3):307–330, May 2012. ISSN 0934-5043, 1433-299X. doi: 10.1007/s00165-011-0199-y. URL https://dl.acm.org/doi/10.1007/s00165-011-0199-y.

[26] Abdullah Saydemir, Muhammed Esad Simitcioglu, and Hasan Sozer. On the Use of Evolutionary Coupling for Software Architecture Recovery. In *2021 15th Turkish National Software Engineering Symposium (UYMS)*, pages 1–6, Izmir, Turkey, November 2021. IEEE. ISBN 978-1-66541-070-0. doi: 10.1109/UYMS54260.2021.9659761. URL https://ieeexplore.ieee.org/document/9659761/.

[27] Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice Hall, Upper Saddle River, N.J, 1996. ISBN 978-0-13-182957-2.

[28] Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[29] Alf Inge Wang and Njål Nordmark. Software architectures and the creative processes in game development. In *International Conference on Entertainment Computing*, pages 272–285. Springer, 2015.

[30] Yuxin Wang, Ping Liu, He Guo, Han Li, and Xin Chen. Improved Hierarchical Clustering Algorithm for Software Architecture Recovery. In *2010 International Conference on Intelligent Computing and Cognitive Informatics*, pages 247–250, Kuala Lumpur, Malaysia, June 2010. IEEE. ISBN 978-1-4244-6640-5. doi: 10.1109/ICICCI.2010.45. URL http://ieeexplore.ieee.org/document/5565989/.

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

[32] Tianfu Yang, Zhiyong Jiang, Yanhong Shang, and Monire Norouzi. Systematic review on next-generation web-based software architecture clustering models. *Computer Communications*, 167:63–74, February 2021. ISSN 01403664. doi: 10.1016/j.comcom.2020.12.022. URL https://linkinghub.elsevier.com/retrieve/pii/S0140366420320284.