SyDRA: An Exploratory Approach to Game Engine Architecture Recovery

Gabriel Cavalheiro Ullmann

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of

Master of Applied Science (Software Engineering) at

Concordia University

Montréal, Québec, Canada

December 2023

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Gabriel Cavalheiro Ullmann**

Entitled: **SyDRA: An Exploratory Approach to Game Engine Architecture Recovery**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Software Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

Joey Paquet — Chair
Dr. Joey Paquet

Anne Etien — External Examiner
Dr. Anne Etien

Juergen Rilling — Examiner
Dr. Juergen Rilling

Yann-Gaël Guéhéneuc — Supervisor
Dr. Yann-Gaël Guéhéneuc

Fabio Petrillo — Co-supervisor
Dr. Fabio Petrillo

Approved by Joey Paquet
Joey Paquet, Chair
Department of Computer Science and Software Engineering

December 4, 2023

Mourad Debbabi
Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

SyDRA: An Exploratory Approach to Game Engine Architecture Recovery

Gabriel Cavalheiro Ullmann

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. In order to understand, develop and maintain game engines, developers need access to architectural information to make informed decisions. However, architectural information is not always readily available and is often overlooked in this kind of system. In this work, we propose, implement and evaluate an approach for software architecture recovery we call the Subsystem-Dependency Recovery Approach (SyDRA, pronounced *SEE-dra*). We apply it to 10 popular open-source game engines, demonstrating how the resultant architectural models can be used to visualize and grasp subsystem coupling patterns unique to each game engine. Additionally, we describe and discuss emergent architectural characteristics shared by all these game engines. Through a qualitative evaluation and a user study, we show the architectural models and visualisations we produced with SyDRA help developers understand game engine architecture more swiftly and correctly.

Acknowledgments

(Portuguese version follows / Versão em português em seguida)

Following the old Brazilian cliché, I will begin with a football analogy: I strive to give my best on the pitch in every match. In every project, every class, every moment I spent with the people I love and in every line of this thesis, you can find every last drop of my dedication. However, no player ever won a championship alone. This is why I would like to thank everyone who accompanied me along the way, starting with my mother Loreni, my father Paulo and my aunt Cristine. They taught me the importance of education from an early age and have motivated me to pursue knowledge and happiness.

I would like to thank my supervisor, Dr. Yann-Gaël Guéhéneuc, and co-supervisor, Dr. Fabio Petrillo, for the years of joint work and continuing support. I learned a lot with you and I hope to keep learning. I would also like to thank my friend, Dr. Cristiano Politowski, for introducing me to academia and, most specifically, to video game research. He transmitted to me his interest and enthusiasm for this research topic, and this is what brought me to Montreal.

Finally, I would like to give my special thanks to all my colleagues from the Ptidej team and all my friends, many of whom participated in the user study presented in this thesis. At a bar table in Lille, a barbecue in Santa Rosa, an icy street in Montreal or connected via videoconference anywhere in the world, it was great to play the beautiful game of Science (and life) with you. May we have many more matches ahead of us.

Como bom brasileiro, vou começar com uma analogia futebolística: em cada partida que disputo, dou o meu melhor em campo. Em cada projeto, cada aula, cada momento que passei com

pessoas queridas e em cada linha desta dissertação me dediquei plenamente. Porém, jogador sozinho não ganha campeonato. É por isso que eu gostaria de agradecer todas as pessoas que me acompanharam ao longo do caminho, começando com minha mãe Loreni, meu pai Paulo e minha tia Cristine. Foram essas pessoas que me ensinaram desde cedo a importância da educação e me motivaram a buscar o conhecimento e a felicidade.

Gostaria de agradecer ao meu supervisor, Dr. Yann-Gaël Guéhéneuc, e ao meu co-supervisor, Dr. Fabio Petrillo, pelos anos de trabalho conjunto e contínuo suporte. Aprendi muito com vocês e espero continuar aprendendo. Gostaria também de agradecer meu amigo, Dr. Cristiano Politowski, por ter me apresentado ao mundo acadêmico e, mais especificamente, à pesquisa na área de vídeo games. Seu interesse e entusiasmo por este tópico me contagiaram e me trouxeram até Montreal.

Por fim, gostaria de deixar meus agradecimentos especiais a todos os colegas do time Ptidej e a todos os meus amigos, muitos dos quais participaram do estudo apresentado nesta dissertação. Em uma mesa de bar em Lille, um churrasco em Santa Rosa, nas ruas geladas de Montreal ou via videoconferência em qualquer lugar do mundo, foi incrível jogar o jogo da Ciência (e da vida) junto com vocês. Espero que tenhamos ainda muitas partidas pela frente.

Contents

List of Figures ix

List of Tables xi

1 Introduction 1

1.1 Overview . . . 1

1.2 Research Methodology . . . 3

1.3 Research Contributions . . . 4

1.4 Thesis Outline . . . 4

2 Background 6

2.1 Software Architecture . . . 6

2.2 Software Architecture Recovery . . . 7

2.2.1 Approaches . . . 7

2.2.2 Use of metamodels . . . 9

2.2.3 Use of graph analysis . . . 10

2.2.4 Evaluation . . . 11

2.3 Game Engines . . . 12

2.3.1 Relations with software architecture recovery . . . 18

3 Approach 20

3.1 System selection . . . 21

3.2 Subsystem selection . . . 21

3.3 Subsystem detection . . . 22
3.4 Include graph generation . . . 22
3.5 Architectural model generation . . . 22
3.6 Architectural model visualisation . . . 22

4 Implementation **23**
4.1 System Selection . . . 23
4.2 Subsystem Selection . . . 24
4.3 Subsystem Detection . . . 28
4.4 Include Graph Generation . . . 32
4.5 Architectural Model Generation . . . 34
4.6 Architectural Model Visualisation . . . 35
4.7 Results . . . 36
4.7.1 RQ1: Which Subsystems are Present in Game Engines? . . . 36
4.7.2 RQ2: Do Game Engines Share Subsystem Coupling Patterns? . . . 38
4.8 Discussion . . . 43
4.8.1 RQ1: Which Subsystems are Present in Game Engines? . . . 43
4.8.2 RQ2: Do Game Engines Share Subsystem Coupling Patterns? . . . 58
4.8.3 Unclustered Files/Folders . . . 71

5 Evaluation **73**
5.1 RQ1 - To what extent do the game engines we selected for architecture recovery with SyDRA match the architectural descriptions provided in the literature? . . . 74
5.2 RQ2: Does SyDRA help developers understand and maintain the architecture of game engines? . . . 76
5.2.1 Hypotheses . . . 77
5.2.2 Participants . . . 77
5.2.3 Experimental Materials . . . 78
5.2.4 Visual Studio Code . . . 80
5.2.5 Experimental Tasks . . . 82

5.2.6 Procedures . . . 83
5.2.7 Design . . . 84
5.2.8 Dependent Variables and Their Collection Procedures . . . 85
5.2.9 Data Analysis Procedure . . . 86
5.2.10 Results . . . 88
5.2.11 Discussion . . . 90
5.2.12 Threats to validity . . . 94

6 Conclusion **96**

6.1 Limitations and Future Work . . . 97

Appendix A Task Statements and Questionnaires **99**

Appendix B Absolute Path to Files in Results **101**

Bibliography **104**

List of Figures

Figure 2.1 Examples of *include* graphs with incoming and outgoing edges 10

Figure 2.2 Architecture adapted from Jaeyong Park and Changhyeon Park [1, p. 102] . 13

Figure 2.3 Architecture by Bishop et al. [2, p. 47] 13

Figure 2.4 Architecture adapted from Sherrod [3, p. 13] 14

Figure 2.5 Architecture by Rollings and Morris [4, p. 626] 15

Figure 2.6 Architecture adapted from Thorn [5, p. 6] 16

Figure 2.7 Summarized "Runtime Game Engine Architecture" diagram, adapted from Gregory [6, p. 33]. 16

Figure 3.1 The six steps of SyDRA. 21

Figure 4.1 "Runtime Game Engine Architecture", adapted from Gregory [6, p. 33] . . 27

Figure 4.2 Subsystem Detection 28

Figure 4.3 Include graph generation 33

Figure 4.4 FAMIX-CPP metamodel used in SyDRA 34

Figure 4.5 Steps of Architectural Model Generation on Moose 34

Figure 4.6 Architectural Map showing files containing the word "camera" from Godot 35

Figure 4.7 Average subsystem in-degree. 38

Figure 4.8 Average subsystem betweenness centrality. 39

Figure 4.9 Subsystem coupling heatmap showing aggregated coupling counts. 40

Figure 4.10 Our emergent open-source game engine architecture. 42

Figure 4.11 Folder organisation pattern we found in O3DE 50

Figure 4.12 Alternative folder organisation for O3DE 50

Figure 4.13 GamePlay3d folder organisation 53
Figure 4.14 Unreal Engine's architectural model. 60
Figure 4.15 Godot's architectural model. 61
Figure 4.16 Panda3D's architectural model. 62
Figure 4.17 O3DE's architectural model. 63
Figure 4.18 FlaxEngine's architectural model. 64
Figure 4.19 GamePlay3d's architectural model. 65
Figure 4.20 Urho3d's architectural model. 66
Figure 4.21 Cocos2d-x's architectural model. 67
Figure 4.22 Piccolo's architectural model. 68
Figure 4.23 OlcPixelGameEngine's architectural model. 69
Figure 5.1 Subsystem naming differences among architectures 75
Figure 5.2 Visual Studio Code set up for the study, showing a C++ file from Godot Engine 80
Figure 5.3 Moose set up for the study, with Architectural Map visible on the top right . 81
Figure 5.4 Correctness and completion time distribution for both study groups 89
Figure 5.5 Participants' answers for NASA TLX questions 91

List of Tables

Table 2.1 Inputs in software architecture recovery, adapted from Ducasse and Pollet [7, p. 4] . . . 8
Table 4.1 Overview of the selected game engine repositories from GitHub. . . . 24
Table 4.2 Summarized "Runtime Game Engine Architecture" subsystem descriptions, adapted from Gregory [6, p. 33] . . . 26
Table 4.3 Examples of subsystem detection using folder naming . . . 29
Table 4.4 Examples of subsystem detection using documentation . . . 30
Table 4.5 Examples of subsystem detection using source code comments . . . 31
Table 4.6 Excerpt for Godot's subsystem detection CSV . . . 32
Table 4.7 Examples of files we did not cluster into any subsystem . . . 37
Table 4.8 The top frequent subsystem coupling pairs. . . . 41
Table 4.9 Unreal Engine's *./Engine/Source* folder . . . 45
Table 4.10 Godot's root folder . . . 46
Table 4.11 Panda3D root folder . . . 48
Table 4.12 O3DE *./Code/Framework* folder . . . 49
Table 4.13 FlaxEngine *./Source* folder . . . 51
Table 4.14 GamePlay3d ./gameplay folder . . . 52
Table 4.15 Urho3d's *./Source* folder . . . 54
Table 4.16 Cocos2d-x's root folder . . . 55
Table 4.17 Piccolo's root folder . . . 55
Table 4.18 OlcPixelGameEngine's *./Extensions* folder . . . 57

Table 5.1 Demographics of the user study participants 79
Table 5.2 Original words used by Briand et al. [8, p. 527] in their task statements and how we changed them 82
Table 5.3 Experimental Design 85
Table 5.4 Two-sample T-test results for study dependent variables 87
Table 5.5 P-values obtained with normality tests of the *UndTime*, *UndCorr* and *ModTime* variables 87
Table 5.6 Wilcoxon Matched Pairs test results for study dependent variables 88
Table 5.7 Descriptive statistics for each dependent variable 89
Table 5.8 Effect size for each dependent variable 90
Table A.1 Architecture understanding tasks 99
Table A.2 Impact analysis tasks 100
Table A.3 Debriefing Questionnaire 100
Table B.1 Absolute path to files mentioned in the Section 4.7 103

Chapter 1

Introduction

In this chapter, we introduce the main concepts, objectives and contributions of this work.

1.1 Overview

While mainly focused on entertainment, video games currently play a role in many areas of society. They are used for information, education, healthcare and as art forms. Video games are also part of an industry which reached 3.2 billion players and generated an estimated revenue of $184.4 billion in 2022[1].

Much like movies or other works of entertainment, video games have a high production cost. For example, GTA V, one the most acclaimed video games released in the past decade, cost $266 million to develop and market[2]. Fortunately for the developer Rockstar Games, it also generated $800 million dollars in revenue in its first day of sales[3]. However, investment alone is not a guarantee of success. Even though well-funded, game development projects suffer from a lack of specialized workforce and environmental issues such as "lack of incentives, toxic behaviors (e.g., harassment or bullying), excessive or mandatory crunch times, lack of open communication, and lack of working standards" [9, p. 6].

[1] https://newzoo.com/insights/articles/the-games-market-in-2022-the-year-in-numbers

[2] https://www.businessinsider.com/the-most-expensive-video-games-ever-made-2014-7

[3] https://www.gamesindustry.biz/gta-v-is-the-most-profitable-entertainment-product-of-all-time

Along with human aspects, technical aspects such as the choice of game engines also have a large impact on video game projects. While most game developers choose to develop their projects using commercially available game engines such as Unity and Unreal, a minority still chooses to develop their own in-house solution[4]. Developing a game engine is expensive and time-consuming, but it presents advantages such as the possibility of writing optimisations focused on a specific kind of game or game genre, as well as avoiding paying licensing fees to third-party game engine providers.

In contrast, re-writing game engine code to adapt it to different game genres increases development time. For example, developers at Electronic Arts reported that adapting Frostbite, a first-person shooter game engine, to the requirements of the role-playing game Dragon Age: Inquisition "took up about a third of the project's development time"[5]. Similarly, developers from Bethesda added multiplayer support to Creation Engine, originally made for single-player games only. During the development of Fallout 76, this newly added feature caused several bugs and "put additional time pressure on the schedule"[6].

Existing software architecture and development tools and techniques can help address issues in game engine development. However, there are few studies assessing the effectiveness of these tools and techniques, as well as the existing gaps in our understanding of game engine architecture. This situation is further compounded by the relative absence of collaborations among major proprietary game engine developers, which work in isolation and do not follow shared industry standards or practices. We believe that breaking this isolation and establishing shared best practices and reference game engine architectures is fundamental to the improvement and evolution of the video game industry, a viewpoint which is also supported by other researchers in the field [10, p. 231].

[4] https://mma.prnewswire.com/media/1880817/Game_Development_Trends.pdf
[5] https://gamerant.com/dragon-age-mark-darrah-bioware-problems/
[6] https://kotaku.com/bethesda-zenimax-fallout-76-crunch-development-1849033233

Thesis Statement

In this thesis, we propose, implement and evaluate the Subsystem-Dependency Recovery Approach (SyDRA, pronounced *SEE-dra*), an approach for software architecture recovery. We apply it to 10 popular open-source game engines, and we use the resulting architectural models to visualise and understand subsystem coupling patterns for each game engine, as well as emergent architectural characteristics common to all game engines.

By extracting, studying and evaluating the architectural models of open-source game engines, we take a step towards improving game engine design, understanding and maintenance. We believe this may positively impact not only game engine architects and developers but the video game development industry as a whole, which can reap the benefits of having better tools and techniques to support their work.

1.2 Research Methodology

Based on software architecture recovery approaches proposed by other researchers, we propose SyDRA, an approach which extracts architectural models from software systems based on their source code, documentation and folder hierarchy, as well as human expertise about its organisation.

We then implement SyDRA and apply it to 10 popular open-source game engines selected from GitHub, obtaining an architectural model of each game engine as a result. We use Moose and other tools to create visualisations of these extracted architectural models, which enable us to answer the following research questions:

- **RQ1**: Which subsystems are present in game engines?
- **RQ2**: Do game engines share subsystem coupling patterns?

We use these visualisations to observe subsystem coupling patterns for each game engine, as well as emergent architectural characteristics common to all analysed game engines. We show that developers can use architectural models to improve game engine architectural understanding and maintainability. Finally, we evaluate SyDRA by comparing the extracted architectural models to game engine architectural descriptions existing in the literature, as well as conducting a user study.

By asking 16 software developers to use the architectural models and visualisations produced via SyDRA, we determine whether this information can help developers better perform architectural understanding and impact analysis tasks in Godot, a popular open-source game engine.

1.3 Research Contributions

Based on the proposed research methodology, we make the following contributions:

- **Architectural Model Extraction**: By applying SyDRA to 10 open-source game engines, we show it is possible to extract architectural models which represent how the parts of a game engine are organized and how its parts relate.
- **Identification of Architectural Commonalities**: By studying game engine architectural models, we show game engines share architectural commonalities with respect to internal organisation and coupling.
- **Improved Architectural Understanding**: As a result of a qualitative evaluation and a user study, we show the architectural models and visualisations we produced with SyDRA help developers understand game engine architecture more swiftly and correctly.

Based on the results we show in this thesis, we wrote the following papers:

- Ullmann et al. [11], "An Exploratory Approach for Game Engine Architecture Recovery" in IEEE/ACM 7th International Workshop on Games and Software Engineering (GAS), 2023.
- Ullmann et al. [12], "Visualising Game Engine Subsystem Coupling" in 22nd IFIP International Conference on Entertainment Computing (ICEC), 2023.

1.4 Thesis Outline

This thesis is comprised of six chapters. Chapter 2 presents related work on software architecture and game engines, while also highlighting in which way this thesis is different from other studies in the same subjects. Chapter 3 provides a high-level description of SyDRA, our software architecture recovery approach. Chapter 4 shows and discusses how we implemented each

of SyDRA's steps and the architectural models resulting from applying SyDRA to 10 open-source game engines. It also describes how these models can be visualised, which architectural information they provide and how it can be used by game engine developers. Chapter 5 describes a qualitative evaluation and user study we conducted to validate SyDRA. Chapter 6 presents the conclusion, limitations and future work.

Chapter 2

Background

In this chapter, we describe the main research topics in this thesis, *Software Architecture* and *Software Architecture Recovery*. We also describe the family of software systems we study, *Game Engines*, which are video game creation tools.

2.1 Software Architecture

Software architecture is the highest-level breakdown of a system into its parts [13, p. 1]. Through the study and application of this discipline, software architects can create architectural models, which are formal, high-level descriptions of a system. Models support reasoning about a system, the relations between its parts and the properties of both [14, p. 57]. Through reasoning, software architects and developers can decide the best course of action when designing, maintaining and extending systems.

A core principle of software architecture is the separation of concerns, which is "focusing one's attention upon some aspect" of system design, while at the same time not ignoring all other aspects [15, p. 61]. This means that to reason effectively about the system, we must hide from the model any information that is not useful for reasoning. Therefore, any given architectural model represents only certain aspects of a system. It is the software architect's task to decide what to model and how.

Architectural models are often visualised as box-and-line diagrams, which are not only useful for the architect but also for communicating software design decisions to developers and other project stakeholders.

2.2 Software Architecture Recovery

Software architecture recovery is the extraction of architectural descriptions from a system implementation [16, p. 555]. The people responsible for creating and applying software architecture recovery approaches are called reverse engineers [7, p. 4], and they create such approaches because, while every system has an architecture, it is not always formalized in a model. This leads to systems that are hard to understand and, as a consequence, hard to maintain [17, p. 46]. Researchers have applied it to systems such as the Apache Web server [18, p. 150], the Android OS and Apache Hadoop [19, p. 1] as a way to improve understanding and maintainability.

Even when models exist, they become outdated over time, in a process known as architectural drift or erosion [20, p. 173681]. As a system source code changes, its architectural model should also change accordingly. Therefore, we can argue software architecture recovery is a continuous process of taking informal and scattered architectural information and codifying it. The result of this process is an architectural model. As explained by Shaw and Garlan [21, p. 11]:

> "We begin by solving problems in any way we can manage. After some time we distinguish in those ad hoc solutions things that usually work and things that don't usually work. The ones that work enter the folklore: people tell each other about them informally. As the folklore becomes more and more systematic, we codify it as written heuristics and rules of procedure. Eventually, that codification becomes crisp enough to support models and theories, together with the associated mathematics. These can then help improve practice, and experience from that practice can sharpen the theories."

2.2.1 Approaches

A reverse engineer may consider different data sources to produce architectural models. For example, Bowman et al. [16, p. 556] clustered Linux files "based on directory structure, naming conventions, source code comments, and examination of the source code". Ducasse and Pollet [7, p. 4] propose a process-oriented taxonomy which defines nine types of inputs for software architecture recovery, which we show in Table 2.1.

Table 2.1: Inputs in software architecture recovery, adapted from Ducasse and Pollet [7, p. 4]

Type	Description
Source Code	Code written in a high-level or assembly programming language.
Textual information	Text found in comments, class names, file names, etc.
Dynamic Information	System runtime events obtained through dynamic analysis (e.g., method calls, CPU usage).
Physical Organisation	Hierarchy and organisation of files, folders or packages.
Human Organisation	Developer communication structures and ownership information.
Historical Information	Source control and bug report data.
Human Expertise	Domain-specific knowledge provided by system stakeholders.
Styles	Architectural styles present in the system (e.g., pipes and filters, layered system, data flow).
Viewpoints	The stakeholder viewpoints the reverse engineer wishes to consider (e.g., 4+1 viewpoints by Kruchten [22, p.1]).

Ducasse and Pollet's taxonomy also defines three processes of architecture recovery: bottom-up, top-down and hybrid. In bottom-up processes, reverse engineers start with low-level knowledge such as source code, and based on that "they progressively raise the abstraction level until a high-level understanding of the application is reached" [7, p. 5]. In a top-down process, a reverse engineer does the opposite, matching a high-level architecture definition with source code.

Finally, hybrid processes are a combination of bottom-up and top-down processes: "On one hand, low-level knowledge is abstracted using various techniques. On the other hand, high-level knowledge is refined and confronted against the previously extracted views" [7, p. 6].

Bottom-up, top-down and hybrid processes can be performed by reverse engineers using three techniques: quasi-manual, semi-automatic and quasi-automatic. In a quasi-manual technique, a reverse engineer will search for architectural elements in the input data sources, and then use a tool to help them understand their findings. In a semi-automatic technique, a reverse engineer will input commands into a tool so it searches for architectural elements. Finally, in a quasi-automatic approach, a tool searches for architectural elements with minimal engineer input [7, p. 11].

While beneficial to maintainability, architecture recovery is tedious and costly because there is a constant need for the reverse engineer to check the consistency between the architectural model and its informal documentation [23, p.250]. Several techniques have been proposed by researchers to mitigate this issue.

In Chapter 3 we describe the data sources and techniques we use in SyDRA, our software architecture recovery approach.

2.2.2 Use of metamodels

A metamodel describes the possible structures which can be expressed in a programming language [24, p. 2]. For example, FAMIX, a metamodel used with object-oriented languages, can represent concepts such as classes, methods and calls [7, p. 7]. Metamodels are used to create models of software systems. To do so, reverse engineers write parsers, programs that read the source code, match each of its parts with concepts described in a metamodel and save them to a structured text file. Each part matched by the parser and saved to this file is called a model entity (e.g., each class in the analysed software system). This file can then be read by software analysis tools, which may provide features such as:

- **Querying**: Instead of using regular expressions to search for patterns in the source code, engineers can use a tool to filter model entities by name, type, and number of lines of code, among other characteristics [7, p. 14].
- **Visualisation**: Engineers can visualise groups of model entities as box diagrams, or visualise their relationships as graphs [7, p. 12].
- **Conformance Checking**: If an architecture specification exists, engineers can use a tool to check whether the model matches it [7, p. 4].

When defining metamodels, engineers must consider some aspects of the software systems they wish to represent and define these aspects into the metamodel. For example, if the software system is implemented in several languages, the metamodel used to represent it must contain concepts from all of them. For example, a language-agnostic model must be able to represent both the concept of method (for object-oriented languages) and function (for procedural languages).

Moreover, engineers must also consider the level of granularity they want for the model when defining the metamodel. For example, if a metamodel only contains the concept of class, it will not be able to represent methods. Therefore, while it will be able to represent class relationships (e.g., inheritance) it will not be able to represent method relationships (e.g., method calls).

In Chapter 4, we show how we use the FAMIX-CPP metamodel to create models of 10 open-source game engines. We then visualise the *include* graphs of file entities in these models to detect and understand frequent dependencies between files. We call these occurrences "coupling patterns".

2.2.3 Use of graph analysis

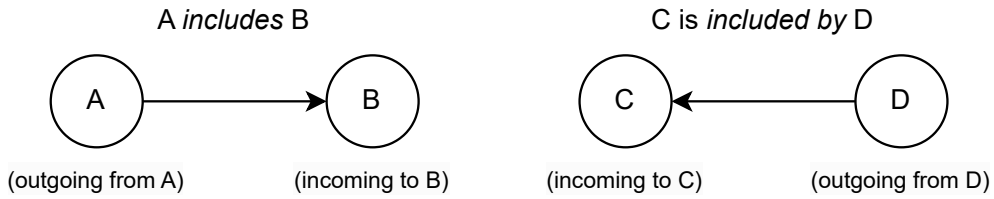


Figure 2.1: Examples of *include* graphs with incoming and outgoing edges

Software architecture recovery approaches may produce directed graphs as an output. A directed graph is a set of nodes linked by edges representing relations between nodes [25, p. 16]. When pointing towards a node, an edge is "incoming", and when pointing away from a node it is "outgoing", as illustrated in Figure 2.1. Call graphs and *include* graphs are commonly used by researchers to study system architecture and model its behaviour [16, p. 557] [26, p. 1027]. In this thesis, we use graph theory to measure the following aspects of *include* graphs:

- **In-degree**: The count of incoming edges of a node.
- **Out-degree**: The count of outgoing edges of a node.
- **Betweenness centrality**: The extent to which a node lies in the path of others [27, p. 758].
- **Density**: The proportion of possible incoming and outgoing edges in the graph that are actually present[1].

We use these measurements because they show us which nodes of an *include* graph are highly coupled. By clustering these nodes into functional modules, we can understand which modules are more fundamental to system functionality, how they work together to provide functionality and to which extent coupling between them is necessary. As explained by Fowler [28, p. 102]:

[1] https://www.ibm.com/docs/en/spss-modeler/18.0.0?topic=networks-network-density

> "Coupling also occurs when code in one module uses code from another, perhaps by calling a function or accessing some data. At this point, it becomes clear that, unlike duplication, you can't treat coupling as something to always avoid. Coupling is desirable, because if you ban coupling between modules, you have to put everything in one big module. Then, there would be lots of coupling—just all hidden under the rug."

In Chapter 4 we compute in-degree, out-degree and betweenness centrality in *include* graphs of 10 open-source game engines. We discuss coupling patterns in these systems, what they tell us about their architecture and how this information can be used by practitioners to understand, maintain and evolve software systems.

2.2.4 Evaluation

The accuracy of an architecture recovery approach can be evaluated by comparing the architectural model extracted by the approach with a ground truth architectural model. Ideally, the ground truth model is created by the original system architects and developers. A third party may also create the model and then ask the original developers to validate it [29, p. 69]. Metrics such as *MoJoFM* or *a2a* can be used to compute the difference between models. The lesser the difference between the "extracted" and "ground truth" models, the more accurate the architecture recovery [30, p. 490].

Other quality metrics can be used to evaluate architecture recovery algorithms, such as the number of iterations and arbitrary decisions [31, p. 240]. The number of iterations refers to the number of times the algorithm has to iterate over the input data to create a model. For an algorithm with a lower number of iterations, the computational cost tends to be lower as well. The number of arbitrary decisions refers to situations where the algorithm cannot determine the best way to cluster a model entity and then makes a random assignment. The larger the number of random assignments, the less deterministic the algorithm and, as a consequence, the extracted model.

We can also evaluate how much the extracted model helps developers understand a system's architecture. Several researchers make this type of validation via user study [8, 32]. In this kind of study, a set of architectural understanding tasks is given to a group of developers. By measuring how swiftly and correctly they can perform tasks with and without a supporting architectural model

or documentation, we can assess the benefit this model brings to system understanding. We perform a user study based on the work of [8], as explained in Chapter 5.

It is also possible to assess the usefulness of extracted models via field study, observing how developers and architects use them in real scenarios [33, p. 1]. However, this kind of study requires collaboration with companies, and considering the closed-source nature of video game and game engine development, that would not be a viable option for this work.

2.3 Game Engines

Game engines are tools to facilitate video game development. While allowing customisation, they also provide a foundation for video game developers to create many different types of games without the need for major modifications [6, p. 11]. They provide a structure which "separates execution of core functionality by the game engine from the creative assets that define the play space or 'content' of a specific game title" [34, p. 1]. This structure is generally designed for reuse [4, p. 269] and compatible with multiple operating systems [35, p. 164].

The first game engines were introduced in the early 1990s and originally provided only 3D graphics rendering features. Over time they expanded to encompass physics, animation, sound and scene management, among others [36, p. 43]. This expansion was motivated by the needs of the video game industry. As video game developers worked to create games that allowed new forms of interaction and that looked more realistic in terms of graphics and physics simulation, their development tools had to become more flexible to accommodate these demands. This is why currently most widely used game engines can be integrated with other software or extended via the use of plugins, also known as "add-ons" or "extensions"[2].

However, before writing a new plugin, video game developers need to understand the game engine architecture they are working with and how its parts 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." [37, p. 1]. Even though these are relevant concerns on both game engines and game development, literature on this subject is limited [38, p. 1] and mostly focused on game engine implementation and not architecture.

[2] https://www.cocos.com/en/post/how-to-bring-your-extensions-to-cocos-creator

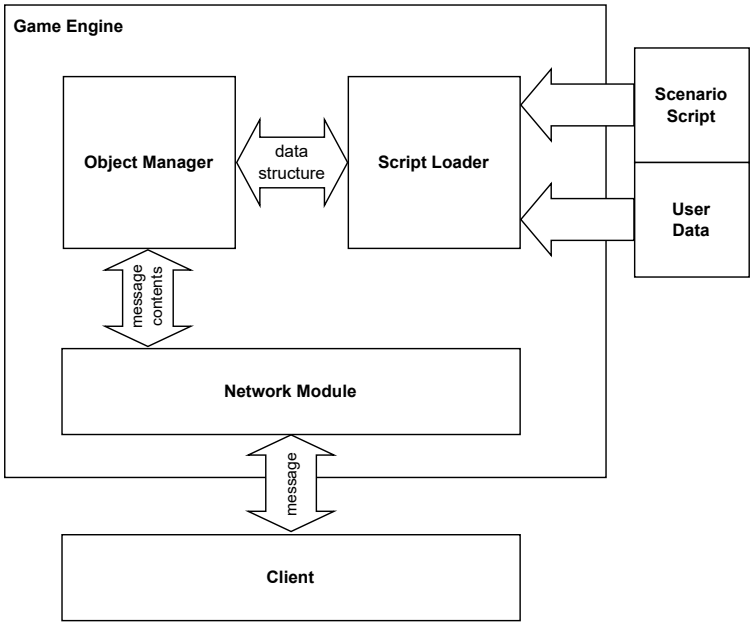


Figure 2.2: Architecture adapted from Jaeyong Park and Changhyeon Park [1, p. 102]

Several game engine architecture studies focus on a specific game engine. For example, Jaeyong Park and Changhyeon Park describe their game engine as a "TCP/IP client/server" divided into three subsystems: object manager, network model and script loader [1, p. 102]. A scripting system is "used to describe the game scenario and facilitat[e] the development of a new game"[1, p. 102]. However, this game engine was created specifically for the creation of MUDs[3] and the authors do not argue whether the architecture of their game engine is applicable to other game genres.

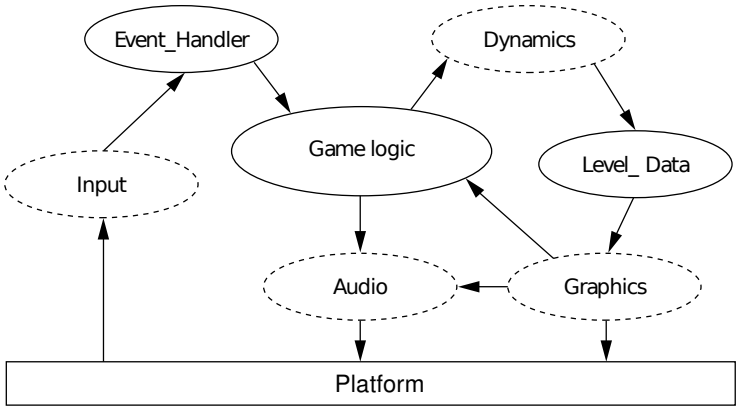


Figure 2.3: Architecture by Bishop et al. [2, p. 47]

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

Bishop et al. [2, p. 1] start their paper about the NetImmerse game engine stating: “we outline here the requirements of a 3D game engine, illustrated by describing a particular engine’s component”. As we show in Figure 2.3, the authors present audio, graphics and input as subsystems (solid ellipses) that relate but remain separated from game-specific logic and assets (dashed ellipses). However, as they focus on the implementation aspects of NetImmerse, they are not concerned with creating a generalisable game engine architecture, or investigating how subsystems relate to each other in more detail.

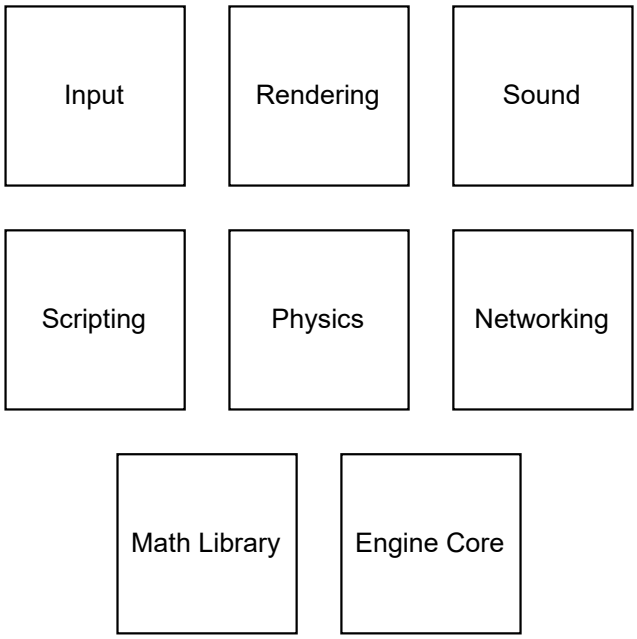


Figure 2.4: Architecture adapted from Sherrod [3, p. 13]

Describing the subject with more breadth than Bishop et al. [2], the work of Sherrod [3] is dedicated to a specific game engine called Building Blocks, but also relates it with a broader idea of game engine architecture. According to the author, a game engine “can be a complex system of programming engineering” and Building Blocks is a set of eight game development libraries “designed to be small with the capability to be expanded upon” [3, p.13].

However, as we show in Figure 2.4, the author represents the parts of the game engine as disconnected boxes, which do not accurately reflect the reality of a complex system such as a game engine. To create a playable video game, several activities must happen continuously and this implies a dependency between different parts of the game engine. For example, if in a video game, we

want to play a sound every time the player presses a button in the joystick, a dependency between the input management and sound management must be created.

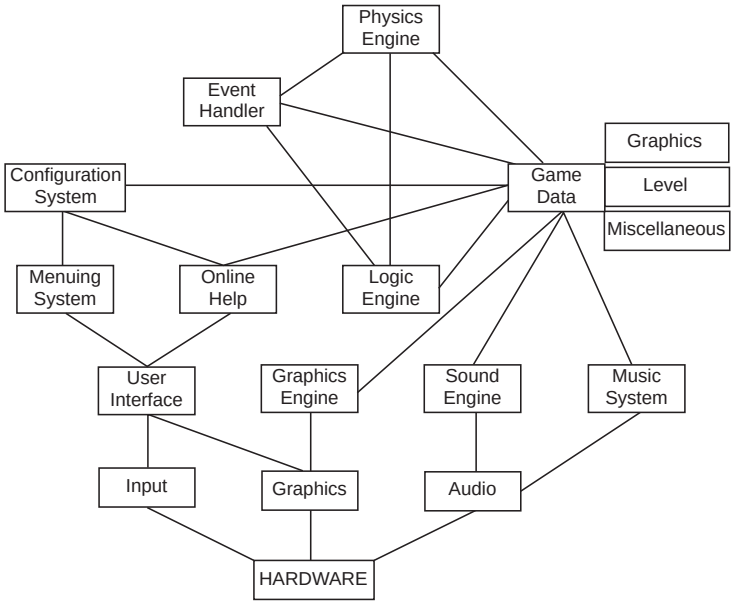


Figure 2.5: Architecture by Rollings and Morris [4, p. 626]

Different from Sherrod [3], Rollings and Morris [4, p. 626] presents a diagram of subsystems they consider the most important in video games in general, along with their relationships (Figure 2.5). While the author explains the diagram represents the architectural structure of a video game, they argue video games tend to be "similar in internal design" [4, p. 625] and their subsystems "can be implemented as a component that can be reused from project to project" [4, p. 626]. This implies the subsystems they show in their architecture are not specific to one video game, but rather usable to create multiple kinds of video games, much like in a game engine.

Also with generalisation in mind, Thorn states that game engines "contain almost all the generalizable components that can be found in a game" [5, p. 6] and proposes a tree-view of the most important game engine subsystems, as we show in Figure 2.6. Gregory proposes a layered "Runtime Game Engine Architecture" [6, p. 33], as we show in Figure 2.7. The author also provides a summary of the responsibilities of each subsystem, which is omitted from Figure 2.7 for brevity. To the best of our knowledge, this is the most comprehensive game engine architecture and therefore we use it as a reference architecture in this work.

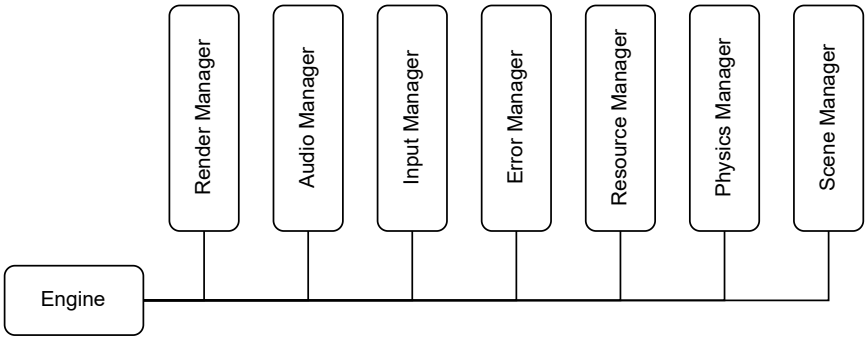


Figure 2.6: Architecture adapted from Thorn [5, p. 6]

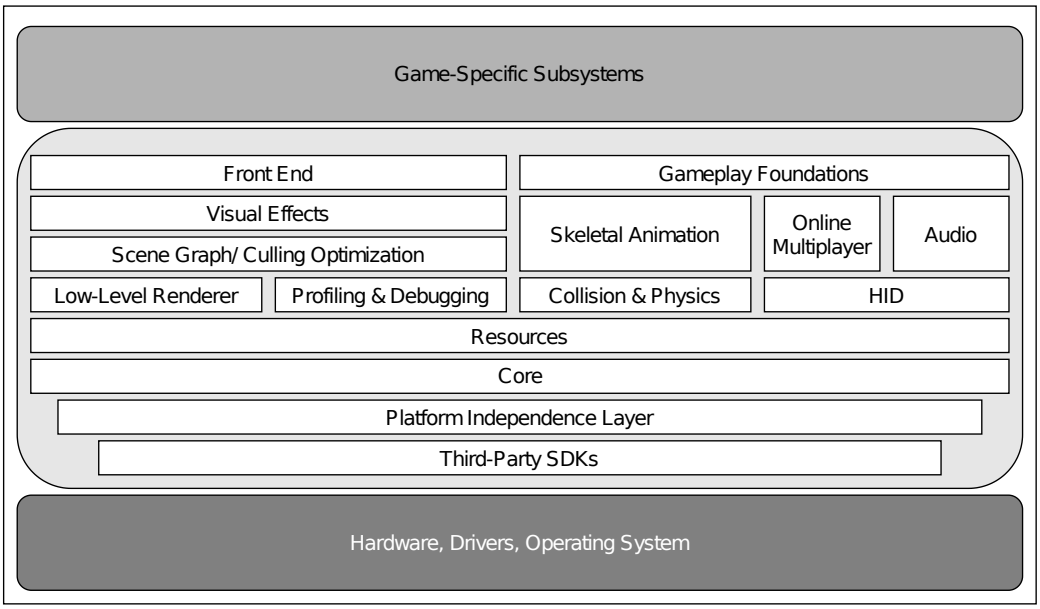


Figure 2.7: Summarized "Runtime Game Engine Architecture" diagram, adapted from Gregory [6, p. 33].

The proposers of game engine architectures also demonstrate they are aware of their limitations. Rollings and Morris states that their list of subsystems "is not all-inclusive" [4, p. 626]. Thorn states that "there is great variation among the many engines in circulation today" and that the "kind of features a developer choose to put into an engine reflects their professional experience, design preferences and business intention" [5, p. 6].

While most research is driven by existing architecture, there are researchers who propose novel architectures. For example, Marin et al. proposes a game engine architecture based on software

agents and mentions that the research of common architectural elements can help "define a genre-independent reference architecture and the recognition of the best practices on game engine development" [39, p. 27]. Maggiorini et al. propose a distributed game engine, where independent subsystems communicate to each other "via a microkernel-like message bus" [38, p. 1]. The author mentions the proposal was motivated not only by the lack of papers on the subject but also because "the majority of the literature seems to be focused on optimising specific aspects or services, such as 3D graphics or physics".

Like game engine books, game engine comparison papers are also implementation-driven. Comparisons frequently concern aspects such as ease of use [40, p. 70], available features and target platforms [41, p. 70]. They seek to aid the developers in choosing which engine best suits a given game genre [42, p. 728] or platform [43, p. 21]. These comparisons are all tabular in nature, listing game engines in columns and relating them to features in rows. The authors aim to determine which game engines have the largest number of features, which features are more commonly implemented and what benefit they bring to developers. But while discussing function, these comparisons do not mention the architectural structures that support functionality.

Moreover, game engine books mostly focus on providing developers with an overview of game engine subsystems and their implementation rather than architectural aspects [10, p. 228]. 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." [6, p. xiv]. Similarly, Eberly explains that his book's goal is "to cover a wide range of topics regarding engines" [44, p. 507] and "discuss the mathematical concepts" [44, p. 7] related to 3D graphics rendering and physics.

In this work, we recover and compare different open-source game engines in search of common architectural patterns. We believe the results of this comparison can motivate the exchange of architectural knowledge between game engine projects and as a consequence improve the current state of practice. We do not intend to propose a new architecture but rather recover, understand and compare architectures that already exist. We want to determine whether these architectures follow similar patterns.

One of the main challenges to the search for common architectural patterns in the game engine domain is that many popular, widely-used game engines are closed-source. Some examples are Unity[4], Frostbite[5] and Id Tech[6]. The impossibility of accessing the code makes it harder for developers to understand their architecture [45, p. 103]. In a situation where developers do not have 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." [46, p. 1717].

However, open-source engines exist and are our choice for this work. Unreal Engine, for example, is currently among the most popular in the world[7]. Other well-known names in this category are Godot[8] and O3DE (formerly Amazon Lumberyard)[9].

2.3.1 Relations with software architecture recovery

Software architecture recovery is rarely applied to game engines, but attempts have shown positive results. For example, Munro et al. [47, p. 247] used Doxygen[10], a popular documentation generation tool, to extract dependency information from an open-source version of the IdTech engine. This data was then used to create dependency graphs, which aided "in the process of identifying suitable improvements and enhancements to a specific engine and have supported implementing these in an appropriate manner".

Agrahari and Chimalakonda [37, p. 1] used AC^2, a software analysis tool[11] developed by authors "to generate call graphs and collaboration graphs across three releases" of Unreal Engine. The use of these visualisations helped them identify architectural patterns in components, observe their evolution and " aid in better comprehension of this complex and widely used game engine for researchers and practitioners".

[4] https://unity.com/solutions/game
[5] https://www.ea.com/frostbite
[6] https://www.idsoftware.com/
[7] https://survey.stackoverflow.co/2022/#most-popular-technologies-tools-tech
[8] https://github.com/godotengine/godot
[9] https://github.com/o3de/o3de
[10] https://www.doxygen.nl
[11] https://github.com/dheerajVagavolu/AC2

More broadly, the application of software architecture recovery to game engines and the comparative study of the resulting architectural models also enables the "identification of components that are common to all types of game engine, allowing the definition of a genre-independent reference architecture", and "the identification of a 'best practice' for the development of game engines" [10, p.230]. We explore insights obtained by comparing game engine architectures in Section 4.7.

Chapter 3

Approach

In this section, we describe the Subsystem-Dependency Recovery Approach (SyDRA), a six-step approach for software architecture recovery composed of the following steps: system selection, subsystem selection, subsystem detection, *include* graph generation, architecture model generation, and architectural model visualisation. According to the process-oriented taxonomy proposed by Ducasse and Pollet [7, p. 4], SyDRA is defined as follows:

- **Goals:** Re-documentation, Analysis and Evolution. The objective of SyDRA is to provide architects and developers with an architectural model they can use to understand software architecture swiftly and correctly.
- **Processes:** SyDRA employs a hybrid architecture recovery process, which uses source code dependencies and a reference high-level architecture and subsystem definition to create architectural models.
- **Inputs:** SyDRA creates architectural models based mainly on source code input. Supporting inputs come from textual information (comments, file names and existing documentation), physical organisation (folder structure) and human expertise (as described in Section 3.2).
- **Techniques:** Researchers implementing SyDRA can choose to perform its steps in quasi-manual, semi-automatic or quasi-automatic fashion. In this thesis, we implement steps 1, 2 and 3 quasi-manually, step 4 quasi-automatically, and steps 5 and 6 semi-automatically. We describe the implementation in more detail in Chapter 4.

- **Outputs:** SyDRA provides visual and architectural outputs, as described in more detail in Section 3.6 and Section 3.5 respectively.

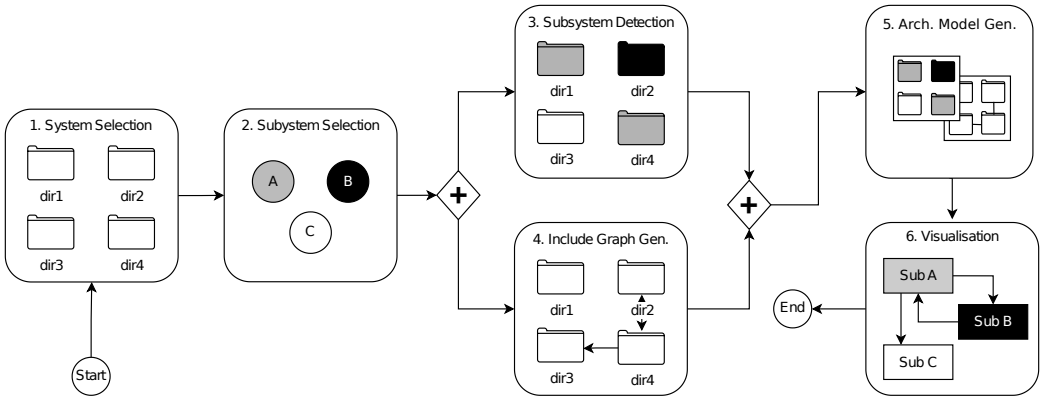


Figure 3.1: The six steps of SyDRA.

3.1 System selection

We start by selecting source code repositories to analyse. The selection process must follow criteria which will change depending on the intended goals of the analysis. Examples of selection criteria are popularity among users, frequency of usage in industrial or academic settings, programming language or available features.

3.2 Subsystem selection

In SyDRA, we cluster source code entities such as files and folders. In this step, we must define clustering criteria. Examples of such criteria are: naming patterns, functionality, disk size, number of lines of code or computational complexity.

We use the word “subsystem” to describe a cluster of files and folders in a software system. While Gregory [6] uses the terms “component”, “module” and “subsystem” interchangeably in their work, we choose to use the “sub” prefix because it emphasizes the separation of concerns we show in Figure 2.7: we can see a game engine as a whole or as a series of evermore specialized subdivisions.

3.3 Subsystem detection

In this step, we cluster files and folders from the selected systems into subsystems. The clustering process must follow the criteria defined in Section 3.2. For example, if we decide to cluster the source code files by functionality, we have to read each file, determine which functionality it provides and then cluster it accordingly.

3.4 Include graph generation

In parallel to detecting subsystems, we generate an *include* graph that encompasses all files in each of the selected systems. Given the large number of *include* relationships between files, we recommend the use of a semi or quasi-automatic process in this step. The *include* graph must be saved in a file so it can be used in the coming steps.

3.5 Architectural model generation

For each selected system, we merge the data obtained from Section 3.3 and Section 3.4 to generate an architectural model. This model is an *include* graph where nodes are clustered by subsystem. To perform this step, researchers can use software analysis tools or code a custom-made solution.

3.6 Architectural model visualisation

We generate visualisations and apply graph analysis to each generated architectural model to identify architectural similarities and frequent coupling patterns. To perform this step, researchers can use software visualisation tools or code a custom-made solution.

Chapter 4

Implementation

In this section, we describe our implementation of the Subsystem-Dependency Recovery Approach (SyDRA), which is available on GitHub[1]. For example, we describe the tools and techniques we employed in each step and why we chose them.

4.1 System Selection

We chose popularity as the main criterion for system selection because we wanted to analyse game engines that are widely used and therefore produce results that could help a large number of game engine developers. However, as explained in Section 2.3, many popular game engines are closed-source. Considering we need access to the systems' source codes to apply SyDRA, we also considered public code availability as a system selection criterion.

We chose GitHub as our source because it is public, free and used by millions of developers and researchers. We chose to measure GitHub repository popularity by using stars and forks because they indicate "there is some form of collaboration and cooperation involved in the development of the software system, which is partial evidence for the repository containing an engineered software project" [48, p.7]. Moreover, the idea that repositories with a high number of stars contain software that people like and use has been supported by several studies [48–50].

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

We started by searching for the term "game engine" on GitHub. We filtered the results to keep only repositories where C++ was the predominant programming language due to its relevance to game engine development [51, p. 10]. We then sorted the repositories by the sum of their stars and forks (as of May 2022) in descending order. The result was a list of hundreds of game engine repositories, from which we selected the top 20 for the sake of brevity.

We then filtered this list to keep only general-purpose game engines. For example, we filtered out the *minetest* game engine[2] from the list because it is limited to creating games in the style of Minecraft. Finally, we selected the top 10 remaining game engines, as we show in Table 4.1. We chose to analyse the source code on the default branch of each game engine repository, considering its state at the most recent commit at the time we performed system selection (May 2022).

Table 4.1: Overview of the selected game engine repositories from GitHub.

Repository	**Branch**	**Commit**	**Forks + Stars**	**Files (.h, .cpp)**
UnrealEngine	v4	f1b664d974	64,100	66,390
Godot	3.4	f9ac000d5d	59,200	5,603
Cocos2d-x	v4	90f6542cf7	23,300	1,601
O3DE	development	21ab0506da	6,400	7,278
Urho3d	master	feb0d90190	4,956	4,312
GamePlay3d	master	4de92c4c6f	4,900	688
Panda3D	master	2208cc8bff	4,100	5,344
OlcPixelGameEngine	master	02dac30d50	3,963	81
Piccolo	main	b4166dbcba	3,892	1,572
FlaxEngine	master	7b041bbaa5	3,613	2,134

4.2 Subsystem Selection

We used the 15 subsystems described in the "Runtime Engine Architecture" proposed by Gregory [6, p. 33]. As explained in Section 2.1, we chose this architecture as our reference because it describes subsystem responsibilities in detail, shows relationships between subsystems and is cited in several game engine research works [38, 39, 51, 52]. We excluded from our analysis any subsystems described by Gregory [6, p. 33] as game-specific, given our objective is to study characteristics of game engines and not video games.

[2] https://github.com/minetest/minetest

We added the "World Editor" to the list of selected subsystems, totalling 16 subsystems, because while not part of the "Runtime Game Engine Architecture", it is described as "a tool that permits game world chunks to be defined and populated" [6, p. 857]. Also, Petridis et al. [53, p.6] argue that many commercial game engines currently provide users with a visual world editor, level editor or integrated development environment, and therefore we cannot ignore this subsystem.

Identifier	**Subsystem**	**Description**
AUD	Audio	Manages audio playback and effects.
COR	Core	Manages engine initialisation and contains libraries for math, memory allocation, etc.
DEB	Profiling & Debugging	Manages performance stats, debugging via in-game menus or console.
EDI	World Editor	Enables visual game world-building.
FES	Front-End	Manages GUI, menus, heads-up display (HUD), and video playback.
GMP	Gameplay Foundations	Manages the game object model, scripting and event/messaging system.
HID	Human Interface Devices	Manages game-specific input interfaces, physical I/O devices.
LLR	Low-Level Renderer	Manages cameras, textures, shaders, fonts, and general drawing tasks.
OMP	Online Multiplayer	Manages match-making and game state replication.
PHY	Collision & Physics	Manages forces and constraints, rigid bodies, ray/shape casting.

PLA	Platform Independence Layer	Manages platform-specific graphics, file systems, threading, etc.
RES	Resources	Manages the loading/caching of game assets, such as 3D models, textures, fonts, etc.
SDK	Third-Party SDKs	Enables interfacing with DirectX, OpenGL, Havok, PhysX, STL, etc.
SKA	Skeletal Animation	Manages animation state tree, inverse kinematics (IK), and mesh rendering.
SGC	Scene Graph/ Culling Optimizations	Computes spatial hash, occlusion, and level of detail (LOD).
VFX	Visual Effects	Enables light mapping, dynamic shadows, particles, decals, etc.

Table 4.2: Summarized "Runtime Game Engine Architecture" subsystem descriptions, adapted from Gregory [6, p. 33]

While the detailed description of the responsibilities encompassed by each subsystem is beyond the scope of this thesis, we summarise their definitions in Table 4.2 because they are a foundation for the next step, subsystem detection. The complete subsystem responsibility list is provided in Figure 4.1. We created 3-letter identifiers for each subsystem, which we use in diagrams from Section 4.7 for brevity.

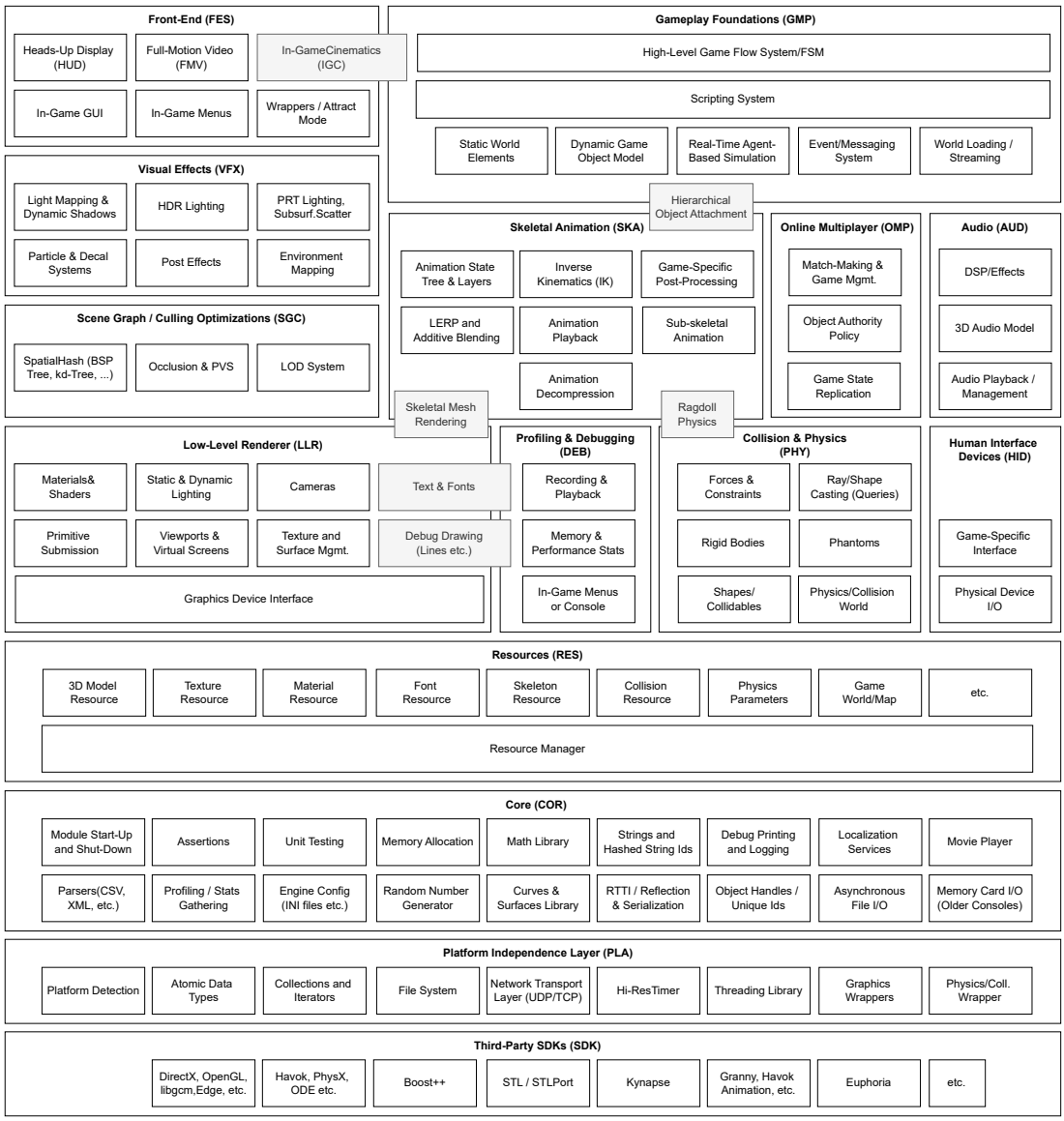


Figure 4.1: “Runtime Game Engine Architecture”, adapted from Gregory [6, p. 33]

4.3 Subsystem Detection

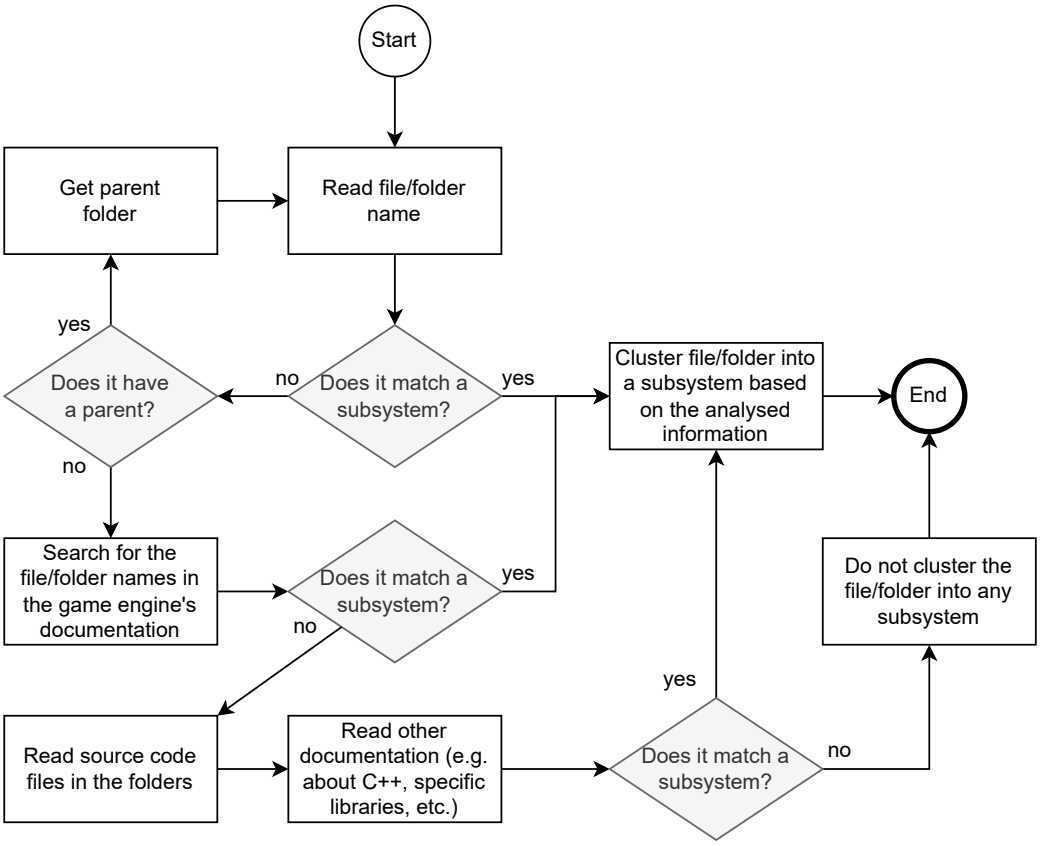


Figure 4.2: Subsystem Detection

We show an overview of the process of subsystem detection in Figure 4.2. For each file/folder in each game engine, we analysed four pieces of information in the following order: name, parent name, documentation, and source code. If we could not determine the subsystem of a file/folder after analysing all pieces of information, or if it was not described in Table 4.2, we did not cluster it into any subsystem.

The file/folder name was the first thing we considered in our decision process because it often describes the subsystem it relates to very objectively. For example, in Godot, the subsystem division can be easily understood by observing the *scene* folder, which contains subfolders with names such as *animation*, *audio*, *debugger* and *resources*. All of these names match subsystem descriptions from Section 4.2.

Some game engines also create codenames for their subsystems, which serve two purposes: internal identification (for architects and developers) and advertisement (to end-users). The Niagara particle system[3] from Unreal Engine is one of many well-known examples. Codenames are also useful in subsystem identification because, when used both in the source code and in the documentation, they allow us to quickly understand the purpose of a given file and what subsystem it belongs to. In Section 4.7 we discuss how codenames were used for subsystem identification in O3DE and Unreal Engine, and what architectural information they provide us.

When the name is not entirely explanatory we inferred the folder's subsystem by looking at the name of the files/folders it contains. For example, Godot's *main* folder contains a file named *main.cpp*, which performs several checks to determine whether a subsystem should be initialized during Godot's start-up. Given initialisation is a responsibility of the game engine core, as described in Table 4.2, we clustered the *main* folder into *Core (COR)*. In Table 4.3, we show this and other examples of subsystem detection using folder naming.

Table 4.3: Examples of subsystem detection using folder naming

Folder Path	**Belongs to**	**Why?**
/godot/servers/audio	AUD	The folder name describes its content.
/urho3d/Source/Urho3D/Audio	AUD	The folder name describes its content.
/godot/main	COR	Contains engine initialisation (main.cpp).
/urho3d/Source/Urho3D/Math	COR	Contains math functions.
/godot/platform/android	PLA	Contains Android compatibility code.
/UnrealEngine/Engine/Source Developer/Android	PLA	Contains Android compatibility code.

If the file/folder naming was not informative enough, we searched for references in the game engine's official documentation. All 10 selected game engines provide official documentation, seven of them hosted on their own website and three in GitHub Wiki (GamePlay3d, OlcPixelGameEngine and Picollo). We used the search feature on each documentation website to search for the file/folder names and obtain more information about them. We found many documentation entries that were

[3] https://docs.unrealengine.com/5.3/en-US/creating-visual-effects-in-niagara-for-unreal-engine/

empty or incomplete, especially for Unreal engine[4]. In such cases, we resorted to informal documentation, such as game engine forums and GitHub issues, as we show in Table 4.4.

Table 4.4: Examples of subsystem detection using documentation

Folder Path	**Belongs to**	**Source**	**Quote**
/cocos2d-x/cocos/ editor-support/spine	SKA	Official docs[5]	"Skeletal animation assets in Creator are exported from Spine."
/UnrealEngine/Engine/ Source/Runtime/Engine/ Classes/Kismet	GMP	Official docs [6]	"The Kismet visual scripting system puts the power in the hands of the level designer (...)."
/panda3d/panda/src/egg	RES	Official docs[7]	"Egg files are used by Panda3D to describe many properties of a scene: simple geometry, (...) characters (...), and character animation tables."
/UnrealEngine/Engine/ Source/Developer/ MaterialBaking	RES	Forum[8]	"This method allows you to bake out complex materials to a single texture which means your material would not affect your performance so heavy."
/o3de/Code/Framework/ AzToolsFramework/ AzToolsFramework/ ActionManager	EDI	GitHub Issue[9]	"The system is comprised of multiple sub-systems that handle different aspects of action management in the Editor."

Finally, if we did not find any information in the documentation, we read the source code. We tried to understand the purpose of each file by reading comments and code. In Table 4.5, we show examples of the comments found in Unreal Engine and O3DE. In a few cases, we remained unable to cluster a file/folder in a subsystem even after reading the source code because we could not understand it. This happened especially when the file/folder names referred to acronyms which were not explained in the game engine's documentation because they refer to external libraries or broader computer programming concepts which the game engine developer is expected to know. For example, O3DE has a folder called */Code/Framework/AzCore/AzCore/RTTI*, but the documentation does not explain it means "Run-Time Type information". The same goes for

[4] https://docs.unrealengine.com/4.27/en-US/API/Plugins/WarpUtils
[5] https://docs.cocos.com/creator/manual/en/asset/spine.html
[6] https://docs.unrealengine.com/udk/Three/KismetHome.html
[7] https://docs.panda3d.org/1.10/cpp/tools/model-export/egg-syntax
[8] https://forums.unrealengine.com/t/bake-out-materials/110074
[9] https://github.com/o3de/sig-content/issues/51#issue-1203873701

/Code/Framework/AzCore/AzCore/IPC, which refers to Linux's interprocess communication. In such cases, we referred to programming books and other technical documentation to understand the acronym's meaning.

Table 4.5: Examples of subsystem detection using source code comments

Folder Path	**Belongs to**	**Code comment**
/UnrealEngine/Engine/Source/ Runtime/Engine/Classes/Interfaces	GMP	(The comments mention network prediction, mesh collision and skeleton functionality) [10]
/UnrealEngine/Engine/Plugins/ Runtime/WarpUtils	LLR	"PFM/MPCDI generation."[11]
/o3de/Code/Legacy/CryCommon/ IIndexedMesh.h	LLR	"2D Texture coordinates used by CMesh." [12]
/o3de/Code/Legacy/CryCommon/ CryListenerSet.h	COR	"A simple, intelligent and efficient container for listeners." [13]
/o3de/Code/Framework/AzCore/ AzCore/RTTI	COR	"Run-time type information."[14]
/o3de/Code/Framework/AzCore/ AzCore/IPC	PLA	"Interprocess communication."[15]

We saved the clustering results in CSV files, one for each game engine. The file contains all absolute file/folder paths and the subsystem identifier we assigned to each of them. Files/folders that could not be clustered into any subsystem were recorded in the file with the OTH identifier (meaning "other"). In such cases, we added a brief description of the file/folder as a way to demonstrate why it did not fit any of the selected subsystems.

For example, in Table 4.6 we show an excerpt[16] from Godot's clustering CSV, in which the dot symbols (.) in the path denote the repository root. The folders *./godot/bin* and *./godot/doc* were not clustered into any subsystem because binaries and documentation are not described in Table 4.2

[10] https://github.com/EpicGames/UnrealEngine/tree/release/Engine/Source/Runtime/Engine/Classes/Interfaces

[11] https://microsoft.github.io/AirSim/pfm/

[12] https://github.com/o3de/o3de/blob/21ab0506da/Code/Legacy/CryCommon/IIndexedMesh.h

[13] https://github.com/o3de/o3de/blob/21ab0506da/Code/Legacy/CryCommon/CryListenerSet.h

[14] https://learn.microsoft.com/en-us/cpp/cpp/run-time-type-information

[15] https://tldp.org/LDP/tlk/ipc/ipc.html

[16] https://github.com/gamedev-studies/game-engine-analyser/blob/8109b22/3_subsystem_detection/godot.csv

and therefore are not part of the game engine architecture. In Section 4.7.1 and Section 4.8.3, we describe unclustered files/folders for all game engines and discuss how they can supplement the reference architecture.

In the cases where we used information from the documentation or code comments in the clustering process, we also saved the related links or text excerpts into the file under the column "Related Information".

Table 4.6: Excerpt for Godot's subsystem detection CSV

Line	Subsystem	Description	Path	Related Information
2	OTH	Binaries	./godot/bin	
3	COR		./godot/core	
4	OTH	Docs	./godot/doc	
17	PLA		./godot/drivers/wasapi	https://docs.microsoft.com/en-us/windows/win32/coreaudio/wasapi
61	RES		./godot/modules/gltf	https://www.khronos.org/gltf/

While the steps we outlined in Figure 4.2 provided us with a consistent foundation for performing subsystem detection, we acknowledge that biases could influence the detection process. For example, the sequence in which these steps were carried out and the interpretation of technical terminology within the documentation can introduce bias. In Section 6.1, we address threats to the validity of subsystem detection and outline our strategies for mitigation.

4.4 Include Graph Generation

We generated an *include* graph of each game engine using a two-pass algorithm, as we show in Figure 4.3. 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*.

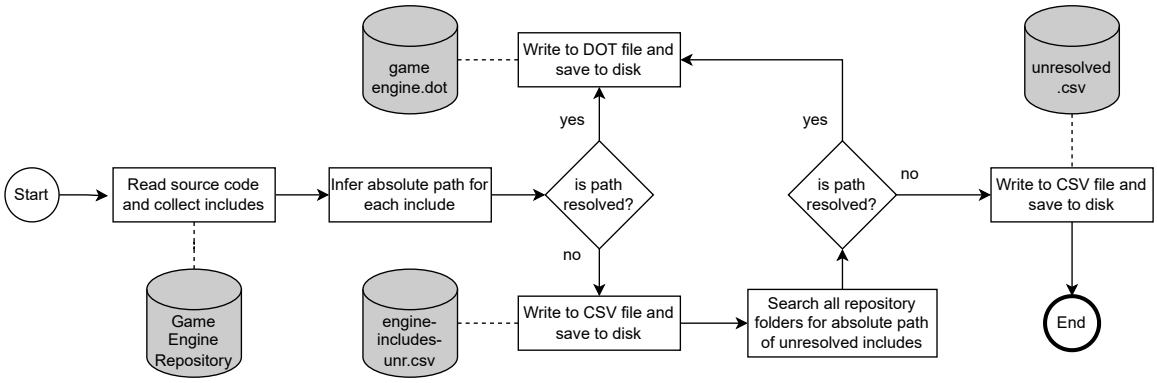


Figure 4.3: Include graph generation

In the original implementation of SyDRA [11, p. 4], 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 algorithm. 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.

4.5 Architectural Model Generation

To generate architectural models we used Moose 10[17], a platform for software analysis composed of several tools and built on top of the Pharo[18] programming language. Moose enables users to use existing metamodels or define their own. We used the FAMIX-CPP metamodel[19], which can represent the two architectural structures we want to study: folder and file, as we show in Figure 4.4. A folder may contain many files, and files might include each other.

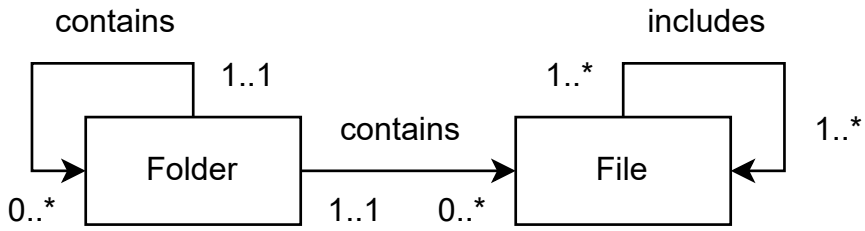


Figure 4.4: FAMIX-CPP metamodel used in SyDRA

Pharo's development environment also gives users the flexibility to write their own tools. To perform this step, we wrote a tool that loads the subsystem detection CSV file from Section 4.3 and the *include* graph DOT file described in Section 4.4. The tool uses the information contained in these files to create a FAMIX-CPP architectural model.

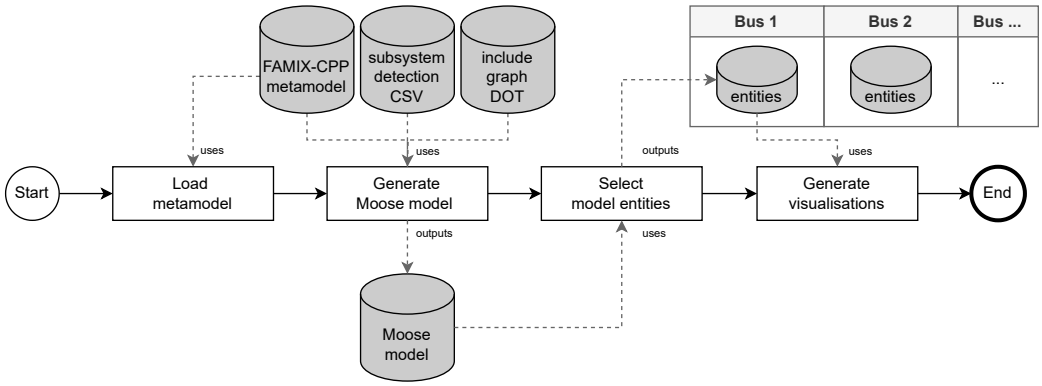
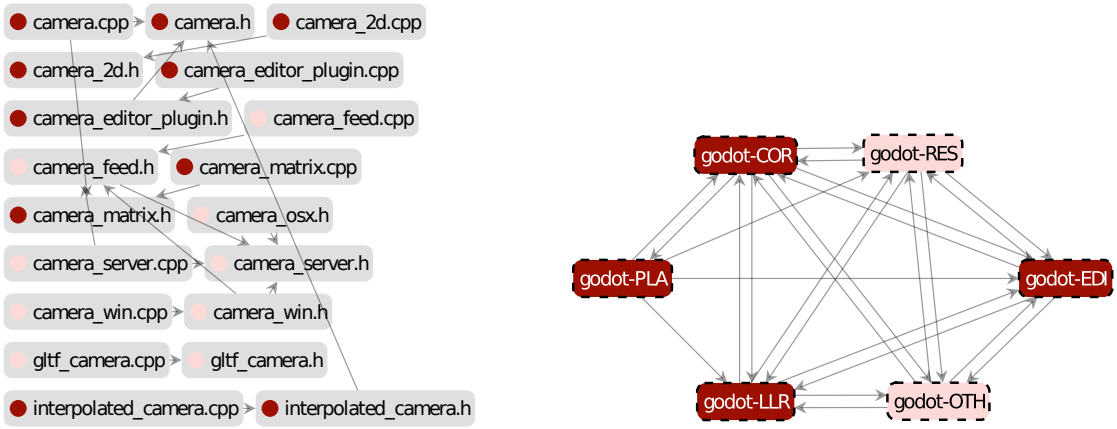


Figure 4.5: Steps of Architectural Model Generation on Moose

[17] https://moosetechnology.org
[18] https://pharo.org
[19] https://github.com/moosetechnology/Famix-Cpp

Also using our tool, we selected the model entities we wanted to analyse and wrote them (or "propagate" them, in Moose's jargon) to a bus, which is a channel of communication between tools [54, p.130]. As we show in Figure 4.5, Moose tools such as the Architectural Map can read entities from a bus and do something with them (e.g. draw a visualisation). Users may create several buses to store different groups of entities which the can select from different models.

4.6 Architectural Model Visualisation



(a) Viewing unclustered files

(b) Viewing files clustered by subsystem

Figure 4.6: Architectural Map showing files containing the word "camera" from Godot

Based on the architectural model generated in Section 4.5, we generated the Architectural Map visualisation on Moose. It is a directed graph where each file entity in the model is a node, and each *include* relationship is an edge. In Figure 4.6, we show how the Architectural Map allows us to observe the relationship between individual files and subsystems as defined in Section 4.2. If the visualisation shows an edge between subsystems, it means there is at least one file on the source subsystem that includes one file in the target subsystem.

While Moose also provides other ways to visualise model entities, we chose the Architectural Map because it represents the *include* relationship between files as a graph. By using graph analysis, we can determine which *include* relationships happen more frequently and therefore identify coupling patterns. Given Moose does not provide us with graph analysis tools out of the box, we

used Gephi[20]. For the sake of simplicity, all graph analysis visualisations were created using Google Spreadsheets and Seaborn, a Python visualisation library[21]:

- **Average subsystem in-degree**: A bar chart ordered by subsystem in-degree. It was created using Google Spreadsheets.
- **Average subsystem betweenness centrality**: A bar chart order by subsystem betweenness centrality. It was created using Google Spreadsheets.
- **Subsystem coupling heatmap**: A coloured dependency matrix showing how many times each subsystem includes another in all 10 analysed game engines. It was created using Seaborn.

We show and discuss visualisations in Section 4.7.2.

4.7 Results

In this section, we answer our RQs by showing subsystem counts and subsystem coupling patterns we found in the 10 game engines we analysed. We use visualisations to illustrate how frequently subsystem coupling patterns appear overall and what architectural information they convey.

4.7.1 RQ1: Which Subsystems are Present in Game Engines?

In Unreal Engine and Godot, we detected all 16 subsystems described in the reference architecture. In the remaining game engines, we detected 12 or more subsystems. The only exception was OlcPixelGameEngine, in which we detected only five subsystems. Therefore, 90% of the game engines we analysed contain at least 75% of the subsystems described in the reference architecture, which shows there are many similarities between the reference architecture and the actual architecture of open-source game engines.

However, the absence of a subsystem in a game engine should not be interpreted as an absence of features. For instance, in the case of FlaxEngine, the absence of a *Scene Graph / Culling Optimizations (SGC)* subsystem does not equate to the absence of scene graph functionality. Files responsible

[20] https://gephi.org
[21] https://seaborn.pydata.org/generated/seaborn.heatmap.html

for implementing scene graph features can still be found within the *./Source/Engine/Level* folder. However, we clustered this folder into the *Gameplay Foundations (GMP)* subsystem based on its naming and the content of other files within the folder. For example, files like *Actor.h* and *Level.h* encompass responsibilities related to the game object model and world loading, which align with the *Gameplay Foundations (GMP)* subsystem, as illustrated in Figure 4.1. The influence of these subsystem detection procedures on our results is further discussed in Section 6.1.

Table 4.7: Examples of files we did not cluster into any subsystem

Repository	**File/folder path**	**Justification for not clustering**
Cocos2d-x	/tools	It contains build tools only, there are no files implementing game engine features.
GamePlay3d	/gameplay/src/Terrain.cpp	Terrain modelling is not described as part of any subsystem in the reference architecture.
Godot	/modules/mobile_vr	Virtual Reality (VR) is not described as part of any subsystem in the reference architecture.
Godot	/modules/recast	Artificial Intelligence (AI) for navigation is not described as part of any subsystem in the reference architecture.
O3DE	/Code/Framework/ AtomCore/Tests	It contains unit tests only, there are no files implementing game engine features.
UnrealEngine	/Engine/Source/Runtime/ AIModule	Artificial Intelligence (AI) is not described as part of any subsystem in the reference architecture.
UnrealEngine	/Engine/Plugins/Runtime/ Oculus	Virtual Reality (VR) for Oculus devices is not described as part of any subsystem in the reference architecture.

During the subsystem detection step, we also found several files which we decided not to cluster into any subsystems because they do not fit the definitions provided by the reference architecture. For the sake of brevity, we show five examples in Table 4.7, along with a justification for our choice. In Section 4.8.3 we discuss how the architectural information we obtained by studying these unclustered files/folders can supplement the reference architecture. In Section 4.8.1, we describe the folder organisation of each of the game engines we analysed and how they map to subsystems. We also discuss whether each folder organisation could be changed to become more cohesive.

4.7.2 RQ2: Do Game Engines Share Subsystem Coupling Patterns?

As we show in Figure 4.7, the top-five subsystems in average in-degree are: *Core (COR)*, *Low-Level Renderer (LLR)*, *Resources (RES)*, *Platform Independence Layer (PLA)* and *Gameplay Foundations (GMP)*. We obtained these results by computing the in-degree for each subsystem of each game engine. Next, we computed the average of these in-degree measurements and sorted them in descending order.

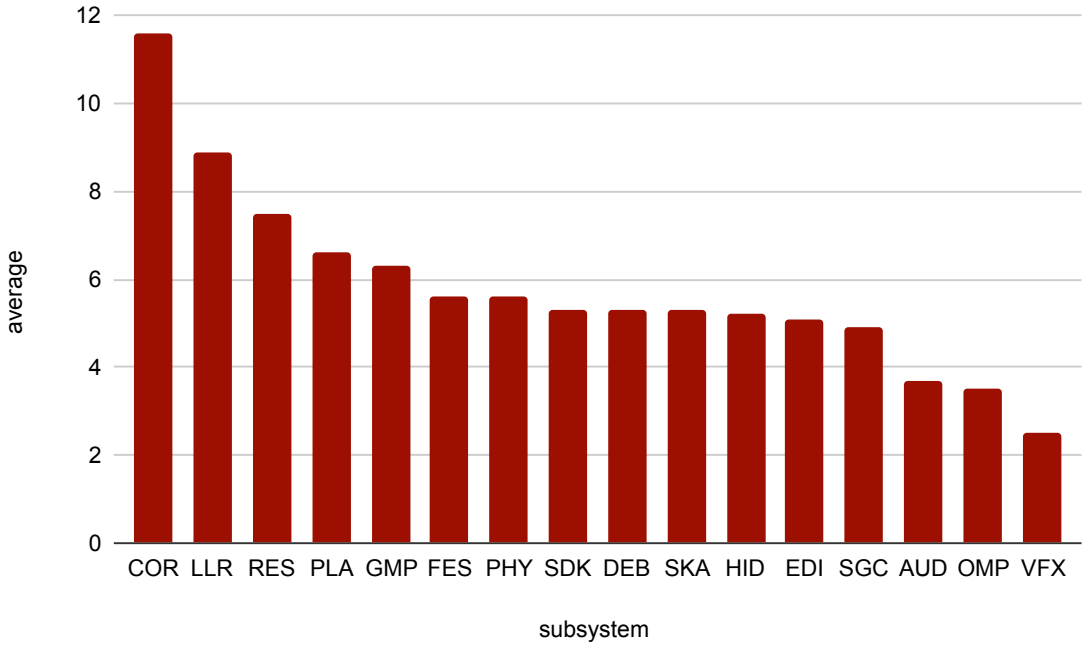


Figure 4.7: Average subsystem in-degree.

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. We expected *Low-Level Renderer (LLR)* to be one of the subsystems with the highest in-degree because video games heavily depend on visuals and the 2D/3D renderer is the part of the game engine responsible for creating visuals.

Same as the in-degree, we computed the average betweenness centrality, as we show in Figure 4.8. The top-five systems with the highest average betweenness centrality are *Core (COR)*, *Platform Independence Layer (PLA)*, *Low-Level Renderer (LLR)*, *World Editor (EDI)* and, tied in 5th place, *Gameplay Foundations (GMP)* and *Resources (RES)*. For this reason, we decided to draw

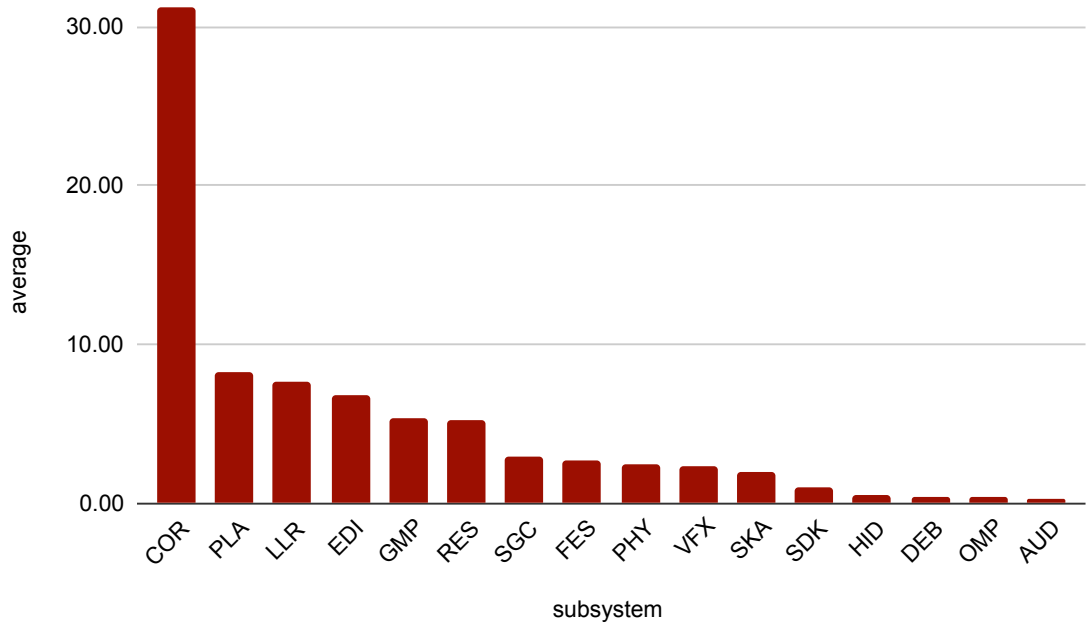


Figure 4.8: Average subsystem betweenness centrality.

only the top four subsystems in the centre of the architectural maps we show in the following subsections from Figure 4.14 to Figure 4.23.

Besides having a high in-degree, the *Low-Level Renderer (LLR)* subsystem also has a high betweenness centrality because many subsystems perform activities that result in a visual output (e.g., after the game engine runs physics computations on a particle, it must draw the particle's new position on the screen) or are triggered by a visual input (e.g., a sound is played by the game engine every time the player moves their camera towards a certain direction).

To visualise which subystem coupling patterns are more frequent across game engines, we aggregated coupling counts from all architectural models and organized them into a heatmap in Figure 4.9. The number of outgoing dependencies is noted in each of the rows of the heatmap, and the number of incoming includes is noted in each of its columns. For example, starting from the top left side, we observe the *Audio (AUD)* subsystem includes files from itself in eight game engines (line 1, column 1), and it includes files from *Core (COR)* in six game engines (line 1, column 2).

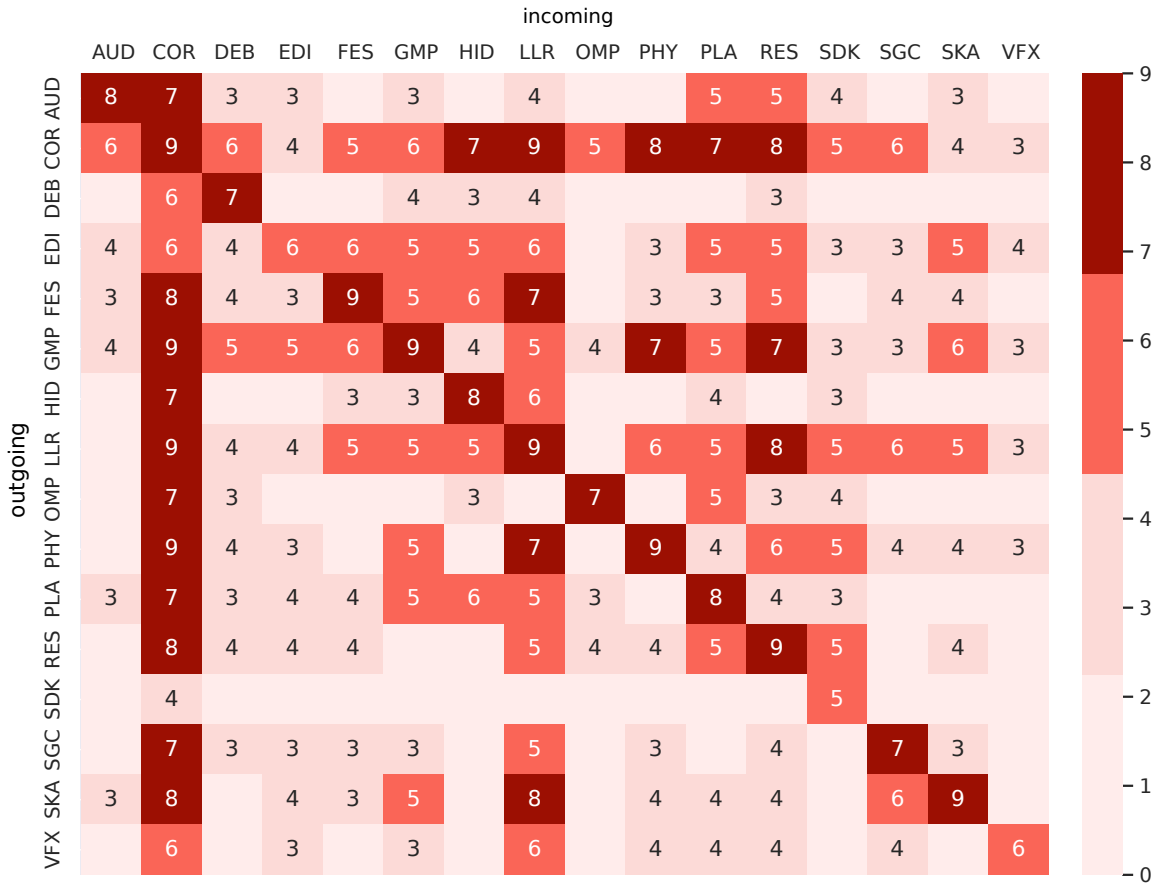


Figure 4.9: Subsystem coupling heatmap showing aggregated coupling counts.

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 explained in Section 4.8.2. Also, not all subsystems were detected in all game engines and therefore not all self-include nine times.

For the sake of brevity, we consider all coupling pairs that occurred in eight game engines or more to be frequent pairs, and we list them in Table 4.8. The *Core (COR)*, *Low-Level Renderer (LLR)* and *Resources (RES)* frequently appear among the most frequent coupling pairs. As we show in Table 4.2, these subsystems have several responsibilities, and we believe this may be the reason why they are highly coupled. Concerning in-degree and centrality, *Core (COR)*, *Low-Level Renderer (LLR)* and *Resources (RES)* also have high measurements, along with *Platform Independence Layer (PLA)*, which likewise centralizes utilities and cross-platform compatibility code.

Table 4.8: The top frequent subsystem coupling pairs.

Pair			Count	Pair			Count
COR	→	LLR	9	COR	→	PHY	8
GMP	→	COR	9	FES	→	COR	8
LLR	→	COR	9	RES	→	COR	8
PHY	→	COR	9	SKA	→	COR	8
COR	→	RES	8	SKA	→	LLR	8
LLR	→	RES	8				

While graph analysis enables us to detect coupling patterns, it cannot explain why this coupling exists and whether it could be reduced. For this reason, in Section 4.8.2, we show examples show examples which explain why certain subsystems are more coupled in certain game engines, whether these coupling patterns repeat in different game engines and what they can teach us about the architecture they are part of. Moreover, we show how observing the coupling between files from different subsystems enables understanding of how a game engine works and how its parts relate.

By compiling the game engine coupling pattern information from Table 4.8 we also observe a new architecture emerge. In Figure 4.10, we placed in the centre of the model the subsystems with the highest betweenness centrality, forming an inner core (dark red). Next, we placed other subsystems which appear in Table 4.8 in the outer core (light red). Finally, we placed the subsystems which do not appear in Table 4.8 in the outer core's periphery (white). All relationships shown in the diagram are among the most frequent, as shown in Table 4.8 and Figure 4.9. 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 2.7, 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).

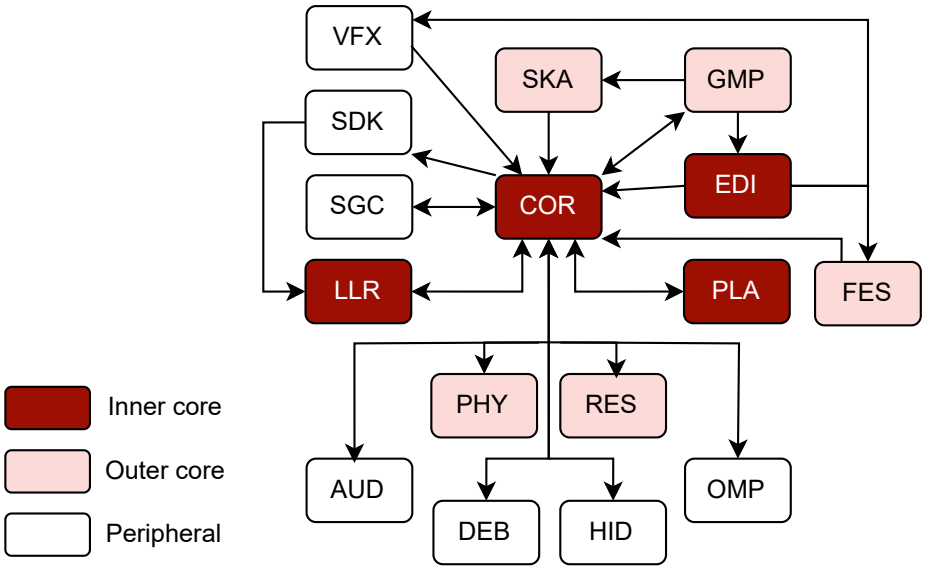


Figure 4.10: Our emergent open-source game engine architecture.

The metrics and visualisations we derived from architectural models and subsystem coupling patterns can be used by practitioners as follows:

- **Architectural Understanding**: 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. We show examples of how the Architectural Map visualisation enables exploration and learning about game engine architecture in Section 4.8.2. Moreover, we show how the use of architectural models can aid architectural understanding in Section 5.2.10.

- **Refactoring**: Game engine developers can refactor their code more safely by visualising how changes to a subsystem could impact the whole game engine. We show how the use of architectural models can aid impact analysis in Section 5.2.10.

- **Reference Extraction**: Game engine architects seeking to design a new engine can extract architectural models from similar systems and use them as references. They have the option to extract and visualize data for a single system, or they can join data from multiple systems within the same family, as we show in Figure 4.10. This is useful both for large companies and small indie developers who develop tailor-made solutions, e.g. for performance.

4.8 Discussion

In this section, we show the folder organisation and subsystem coupling patterns found in each of the game engines we analysed, listed in descending order of popularity, as we show in Table 4.1. We discuss what these patterns can teach us about a game engine's architecture and how they can be used by practitioners such as video game and game engine developers.

4.8.1 RQ1: Which Subsystems are Present in Game Engines?

In the following subsections, we describe and discuss the folder organisation of each of the game engines we analysed and how they map to subsystems. We show our results according to the following template:

- The number of subsystems we identified in the game engine.
- The absolute path to the folder where most of the source code implementing game engine subsystems is located. A dot (.) at the start of the folder path denotes the repository root.
- The names of the three largest subsystems, along with their file count, are in descending order.
- The third-party libraries used by the game engine are also listed under the *Third-Party SDKs (SDK)* subsystem definition in the reference architecture as we show in Figure 4.1.

We chose to show these four items in the template because they give the reader an overview of how much each game engine matches with the reference architecture, and what subsystems are more frequently large. Also, by showing where these subsystems are located in the folder structure, we provide a starting point for researchers who wish to delve into the source code and explore game engine architecture and implementation on their own.

While we do not discuss in detail how developers make their choices of third-party libraries for every game engine, we chose to list them in the template. For brevity, we show only those which are both present in the game engine and in the *Third-Party SDKs (SDK)* subsystem definition in the reference architecture. We include this information because it can give the reader a notion of how diverse the selection of third-party libraries across game engines is, and also specify the cases

where we did not analyse the third-party libraries because they are kept by developers in a repository separate from the game engine.

Along with the template and folder organisation, we also discuss whether each folder organisation could be changed to increase file cohesion. However, when we mention cohesion we are not referring to a quantitative metric, but rather a qualitative one. We decided not to use any cohesion metrics from the Software Engineering domain because they are better suited for the analysis of low-level design rather than high-level, as we do in this work. As Briand et al. [8, p.516] explain in their work on system coupling and cohesion:

> "Note that [in our work] there are no measures used to capture class cohesion. This is because the information required by existing measures is not usually available at high-level design, e.g., method attribute interactions, method-method interactions, and pairs of methods which reference common attributes."

In the following subsections, we consider file cohesion to mean "the act or state of keeping together"[22]. Therefore, if during our subsystem detection process, we determined files/folders to be part of the same subsystem, we consider that they have high cohesion if they are centralized under the same parent folder. Contrarily, if these files/folders are distributed in several folders under different hierarchical levels in the folder organisation, we consider they have low cohesion.

Unreal Engine

- **Subsystems Detected:** 16 out of 16 (100%)
- **Main Source Code Folder:** *./Engine/Source*
- **Largest Subsystems:** *Third-Party SDKs (SDK)* with 21,843 files, *World Editor (EDI)* with 8,500 files, and *Core (COR)* with 4,674 files
- **Reference Third-Party Libraries:** *Boost*, *DirectX*, *OpenGL*, *PhysX*

In Table 4.9, we show Unreal Engine's folder organisation along with descriptions from the official documentation[23]. While the *Editor* folder contains only files from the *World Editor (EDI)*

[22] https://www.oxfordlearnersdictionaries.com/definition/english/cohesion
[23] https://docs.unrealengine.com/4.26/en-US/Basics/DirectoryStructure/

subsystem, the *Developer*, *Runtime* and *Program* folders contain files for all other subsystems. Each of its subfolders, called modules, corresponds to one subsystem.

Table 4.9: Unreal Engine's *./Engine/Source* folder

Folder	**Files**	**Description**
ThirdParty	20,074	Third-party SDKs
Runtime	12,343	Files used by just the engine
Editor	5,888	Files used by just the editor
Developer	2,493	Files used by both the editor and engine
Programs	855	External tools used by the engine or editor

Each module is further divided into two subfolders, *Public* and *Private*. While the documentation does not explain this part of the folder structuring, according to video game development blogger Tom Looman this structure specifies which files are used only internally and which are also used by other modules: "your header files are placed in the Public folder so other modules can gain access and the cpp files are in the Private folder"[24].

The concentration of files in the *Runtime* folder results in its large size and reduced file cohesion. The mismatch between the number of files in the *ThirdParty* folder (20,074 files) and the *Third-Party SDKs (SDK)* subsystem (21,843 files) is evidence of this lack of cohesion. This difference happens because the *Runtime* folder also contains modules that act as drivers to third-party libraries such as CUDA[25], the parallel computing platform from Nvidia, which were clustered into the *Third-Party SDKs (SDK)* subsystem. To increase cohesion, we argue these drivers could be moved to the *ThirdParty* folder or a new sibling folder called *Drivers*.

Differently from other game engines, Unreal Engine provides integration with a wide range of services from *Google*, *Meta* and *Steam*, implemented as libraries found in the *ThirdParty* folder. Considering that Unreal Engine is currently the most popular open-source game engine and that it is used by several large video game development companies, this integration diversity is not only a convenience but also a need for developers who want to integrate with popular video game digital distribution services.

[24] https://www.tomlooman.com/unreal-engine-cpp-guide
[25] https://docs.unrealengine.com/5.1/en-US/API/Runtime/CUDA/

Finally, we observe that, while conforming to the reference architecture, Unreal Engine designers imply architectural divisions of their own using codenames. For example, while the reference architecture states that both visual effects and light mapping are responsibilities of the *Visual Effects (VFX)* subsystem, Unreal Engine gives these two features distinctive codenames: Niagara[26] for general visual effects and Lightmass[27] for light mapping.

Godot

- **Subsystems Detected:** 16 out of 16 (100%)
- **Main Source Code Folder:** the repository root
- **Largest Subsystems:** *Third-Party SDKs (SDK)* with 2,913 files, *Core (COR)* with 469 files and *World Editor (EDI)* with 331 files
- **Reference Third-Party Libraries:** *OpenGL* (named *Gles3* in Godot)

In Table 4.10 we show Godot's folder organisation along with descriptions we adapted from a YouTube video tutorial by Juan Liniestsky, one of Godot's creators[28].

Table 4.10: Godot's root folder

Folder	Files	Description
thirdparty	2,913	Dependency bundle for distribution (e.g., Debian packages)
modules	546	Secondary (non-core) dependencies
scene	420	Classes representing scene tree nodes
editor	308	World editor
core	274	Godot's core types and structures
servers	185	Exposed low-level interfaces (e.g., for rendering, physics)
platform	142	Code for OS compatibility (e.g., Android, iOS, Windows)
drivers	106	Core dependencies
main	44	Godot's main loop

Inside the *scene* folder, each of the subfolders corresponds to one subsystem: *audio*, *debugger*, *gui*, *resources*, *animation*. The only exceptions are the folders *2d* and *3d*, which encapsulate several audio, graphics and physics-related features. The files in these folders implement different versions of the same features (e.g., 2D collision and 3D collision).

[26] https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/Niagara
[27] https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/Lightmass
[28] https://www.youtube.com/watch?v=5RIPRlCvRAk

As in Unreal Engine, observing file counts by folder and by subsystem also reveals interesting architectural choices in Godot. For example, the difference between the file count in the *core* folder (274 files) and the *Core (COR)* subsystem (469 files) happens because the *main* and *modules* folders also contain files from the *Core (COR)* subsystem. The reasons why the *Core (COR)* subsystem is divided into three folders are both organisational and historical, and we can understand them by observing Godot's branches and issues on GitHub.

For example, by checking out branch *v2.0*, which contains the oldest version of Godot available on GitHub, we observe the *main* and *modules* folders already existed at this version. While the purpose of the *main* folder was to separate the main loop from the rest of Godot's core, the *modules* folder was created to bundle together all libraries, both those created by Godot's developers and third parties. Later, in branch *v3.0*, a *thirdparty* folder was created exclusively for keeping only "unmodified upstream versions"[29] of third-party libraries. However, the *modules* folder remained with a mix of first-party and third-party code.

The existence of the *modules* is an example of how the evolution of system architecture may leave behind "vestigial" structures, such as folders, which no longer work cohesively with the surrounding architecture. To increase cohesion, we argue that all third-party libraries should be moved to the *thirdparty* folder while libraries developed by Godot's developers should be moved to a folder with a name that describes their subsystem, such as *core* or *scene*, for example.

While the structure of the *thirdparty* folder may not be cohesive, its evolution and the choice of libraries made throughout Godot's development history are well-documented. For instance, Godot's documentation has an article dedicated to discussing the reasons why it does not use STL, a popular C++ library providing containers, iterators and a series of useful algorithms. According to the documentation, "STL templates create very large symbols, which results in huge debug binaries". The text also claims that Godot's implementation of containers "helps better track memory usage"[30].

[29] https://github.com/godotengine/godot/issues/6157
[30] https://docs.godotengine.org/en/stable/about/faq.html#doc-faq-why-not-stl

Panda3D

- **Subsystems Detected:** 13 out of 16 (81%)

- **Main Source Code Folder:** the repository root

- **Largest Subsystems:** *Core (COR)* with 1,056 files, *Resources (RES)* with 838 files and *Low-Level Renderer (LLR)* with 536 files

- **Reference Third-Party Libraries:** we did not analyse them because they are in a separate repository[31]

In Table 4.11, we show Panda3D's folder organisation along with descriptions we adapted from source code comments and official documentation[32] [33]. Most files are in the *panda* folder as we show in Table 4.11, which contains two subfolders: *metalibs* and *src*. The files in *metalibs* are C++ wrappers which implement no functionality themselves, but act as a facade to third-party libraries both outside of the repository or in the *src* folder. The files in the *src* folder implement all game engine features. Each of its subfolders corresponds to one subsystem.

Table 4.11: Panda3D root folder

Folder	Files	Description
panda	3,483	Panda3D engine core and features
pandatool	701	3D model and geometry import tools
dtool	504	Debugging tools and utility classes
direct	124	Supports "Distributed Networking", Panda3D's high-level network API
contrib	86	Features by other project contributors (not the original developers)

[32] https://docs.panda3d.org/1.9/cpp/tools/index
[33] https://docs.panda3d.org/1.9/cpp/programming/networking/distributed

O3DE

- **Subsystems Detected:** 14 out of 16 (87%)
- **Main Source Code Folder:** ./Code/Framework
- **Largest Subsystems:** *Low-Level Renderer (LLR)* with 1,892 files, *Front-End (FES)* with 1,567 files and *Gameplay Foundations (GMP)* with 1,440 files
- **Reference Third-Party Libraries:** we did not analyse them because they are in a separate repository[34]

In O3DE, most of the files implementing subsystem features are located in *./Code/Framework*. In Table 4.12, we show this folder's organisation along with descriptions we adapted from the official documentation[35]. By analysing the subfolder organisation of each folder in Table 4.12, we observe that the folders *AzCore*, *AzFramework*, *AzNetworking* and *AzToolsFramework* share the same folder organisation pattern, comprised as follows:

- **Features:** Contain files that implement features related to the folder's description in Table 4.12. This folder always has the same name as its parent (e.g. *AzCore* has a "features" folder also named *AzCore*).
- **Platform:** Contains files that implement a part of *Platform Independence Layer (PLA)* subsystem related to the "features" folder.
- **Tests:** Contains unit tests to features in the "features" folder.

Table 4.12: O3DE *./Code/Framework* folder

Folder	Files	Description
AzCore	1,139	Supports AzFramework
AzToolsFramework	1,073	CLI tools
AzFramework	519	Supports AzGameFramework and AzToolsFramework
AzQtComponents	278	GUI tools
AzNetworking	96	Networking features
AtomCore	18	Atom renderer core
AzGameFramework	4	Project runtime

[35] https://docs.o3de.org/docs/welcome-guide/key-concepts/#overview-of-the-o3de-sdk

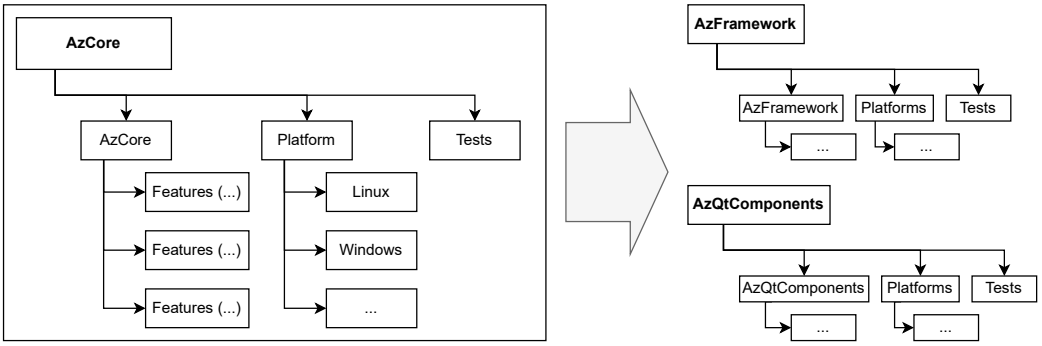


Figure 4.11: Folder organisation pattern we found in O3DE

We demonstrate this folder organisation pattern in Figure 4.11. While its rationale is not explained in O3DE's documentation, we believe it was created to break down the *Platform Independence Layer (PLA)*, a large subsystem which encompasses many responsibilities, into smaller, more specialized parts.

However, while highlighting the separation between O3DE's core and other subsystems, this organisation does not use its folder hierarchy to express the level of granularity of different parts of the game engine. For example,*AzQtComponents*, which is a part of the *Front-End (FES)* subsystem, is in the same hierarchy level as *AzFramework*, a folder which contains files for several subsystems.

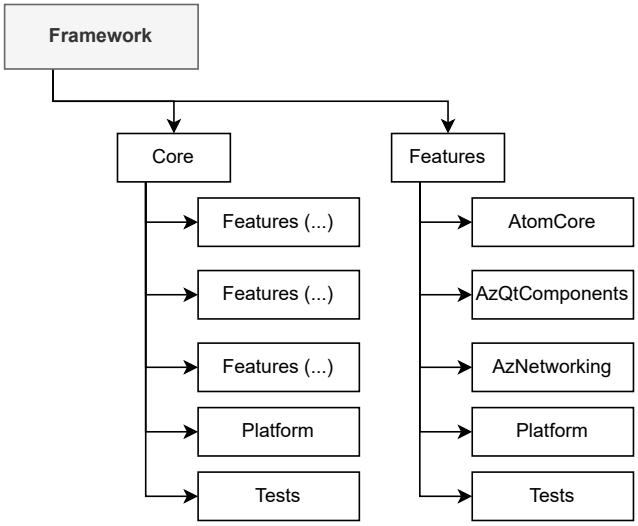


Figure 4.12: Alternative folder organisation for O3DE

In Figure 4.12, we show an alternative organisation which has two folders in its top level, "Core" and "Features", which are then subdivided according to their purpose: containing code from O3DE's

core only or from other subsystems. Each of these top-level folders keeps its own *Platform* and *Test* folders. This way, we avoid duplicated naming and create a more semantic folder hierarchy, which separates the subsystems and their features from the higher-level concept of "Core" vs. "Features".

FlaxEngine

- **Subsystems Detected:** 15 out of 16 (94%)
- **Main Source Code Folder:** ./Source
- **Largest Subsystems:** *Third-Party SDKs (SDK)* with 774 files, *Platform Independence Layer (PLA)* with 295 files and *Core (COR)* with 254 files
- **Reference Third-Party Libraries:** *DirectX*, *OpenGL* (*glslang* in FlaxEngine) and *PhysX*

We show FlaxEngine's folder organisation in Table 4.13 along with descriptions we adapted from the official documentation[36]. In contrast to other game engines, FlaxEngine favours file cohesion by centralising all files related to a subsystem in a single folder. For example, while Unreal and Godot distribute their *Third-Party SDKs (SDK)* and *World Editor (EDI)* functionality across multiple folders, FlaxEngine centralizes it within its *ThirdParty* and *Editor* folders.

Table 4.13: FlaxEngine *./Source* folder

Folder	**Files**	**Description**
Engine	1,280	FlaxEngine core and features
ThirdParty	774	*Third-Party SDKs (SDK)*
Editor	72	*World Editor (EDI)*

We detected 15 out of 16 subsystems in FlaxEngine, which shows its architecture mostly matches the reference architecture. However, we observe that, while being as conforming to the reference architecture as other game engines, it has a much lower file count. For example, while O3DE, in which we detected 14 out of 16 subsystems, contains 1,892 files in its *Low-Level Renderer (LLR)* subsystem only, FlaxEngine's entire *Engine* folder contains 1,280 files implementing the majority of its subsystems. This shows that the subsystem count in a game engine is not necessarily proportional to the amount of source code files it contains.

[36] https://docs.flaxengine.com/manual/editor/advanced/game-engine-flow.html

While FlaxEngine presents a high subsystem count and a low file count, we believe the subsystems within this game engine may also implement fewer functionalities compared to their counterparts in other game engines. In future work, we intend to further investigate the correlation between subsystem size and functionality, as we explain in Section 6.1.

GamePlay3d

- **Subsystems Detected:** 13 out of 16 (81%)
- **Main Source Code Folder:** ./gameplay
- **Largest Subsystems:** *Gameplay Foundations (GMP)* with 286 files, *Resources (RES)* with 97 files and *Low-Level Renderer (LLR)* with 44 files
- **Reference Third-Party Libraries:** not analysed because they are in a separate repository[37]

In Table 4.14, we show GamePlay3d's folder organisation along with descriptions we inferred from file/folder naming and source code comments. The root folder contains three subfolders: *gameplay*, which contains the source code for all subsystems; *samples*, which contains example games, and *tools*, which contains two developer command-line utilities. The first, named *luagen*, enables the generation of Lua script bindings for C++[38]. The second, named *encoder*, enables "encoding games assets like true-type fonts and 3D scene files into a simple binary-based bundle file format" which is specific to GamePlay3d[39].

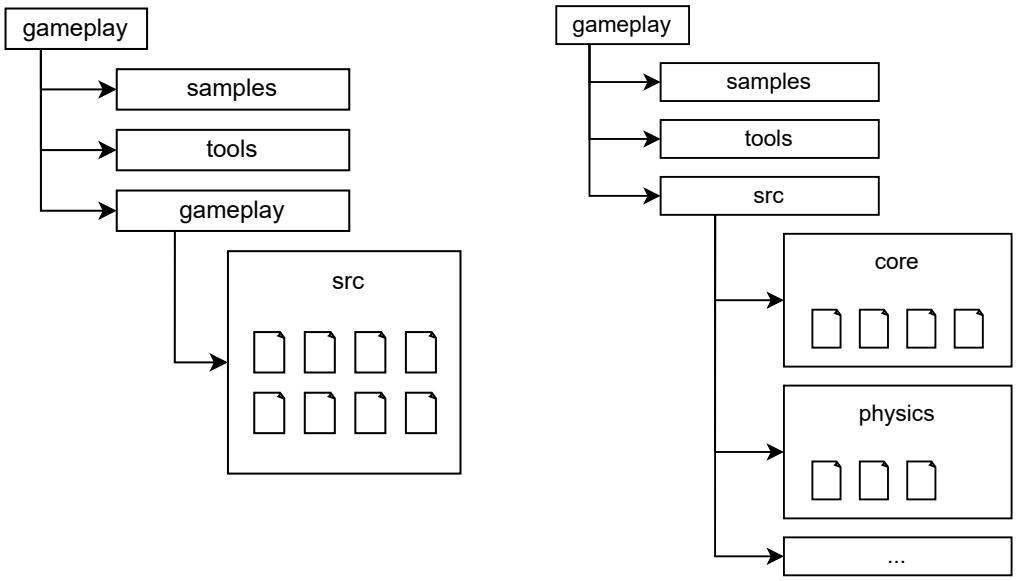
Table 4.14: GamePlay3d ./gameplay folder

Folder	Files	Description
gameplay	1,280	GamePlay3d core and features
samples	774	Example games and demos
tools	72	Tools for generating Lua script bindings/asset encoding

Differently from other game engines, GamePlay3d does not distribute its files in several folders named by subsystems. Instead, it centralizes all its files in the *gameplay* folder and its *src* subfolder, as we show in Figure 4.13a. In Figure 4.13b, we show an alternative folder organisation, where we

[38] https://github.com/gameplay3d/gameplay/tree/master/tools/luagen
[39] https://github.com/gameplay3d/gameplay/blob/master/tools/encoder/README.md

create folders for each of the 13 subsystems we detected in GamePlay3d. We also remove the *gameplay* folder to avoid ambiguity with the repository's name. We argue this alternative organisation would be more cohesive and easier to understand.



(a) Current (b) Alternative

Figure 4.13: GamePlay3d folder organisation

Urho3d

- **Subsystems Detected:** 12 out of 16 (75%)
- **Main Source Code Folder:** ./Source
- **Largest Subsystems:** *Third-Party SDKs (SDK)* with 3,489 files, *Low-Level Renderer (LLR)* with 182 files and *Core (COR)* with 143 files
- **Reference Third-Party Libraries:** *OpenGL*

In Table 4.15 we show Urho3d's folder organisation along with descriptions we inferred from file/folder naming and source code comments. We observe this organisation is very similar to GamePlay3d's, which also contains folders for *Samples* and *Tools*. They also have one tool in common: a script-binding generator. However, while GamePlay3d's script-binding generator is for Lua scripts, Urho3d works with AngelScript, a "flexible cross-platform scripting library"[40] used by several C++

[40] https://www.angelcode.com/angelscript/

applications. Also, differently from GamePlay3d, Urho3d distributes the files in its *Urho3d* folder into several subfolders, each corresponding to one subsystem, favouring file cohesion.

Table 4.15: Urho3d's *./Source* folder

Folder	Files	Description
ThirdParty	3,489	Third-Party SDKs
Urho3D	638	Urho3d's core and features
Samples	156	Example games and demos
Tools	28	Tools for asset importing, script binding

Cocos2d-x

- **Subsystems Detected:** 12 out of 16 (75%)
- **Main Source Code Folder:** the repository root
- **Largest Subsystems:** *Visual Effects (VFX)* with 415 files, *World Editor (EDI)* with 144 files and *Skeletal Animation (SKA)* with 107 files
- **Reference Third-Party Libraries:** not analysed because they are in a separate repository[41]

In Table 4.16, we show Cocos2d-x's folder organisation along with descriptions we inferred from file/folder naming and the official documentation[42].

The root folder contains four subfolders: *cocos*, which contains the source code for all subsystems; *extensions*, which are plugins which enable extending the functionality of Cocos2d-x's editor[43]; *tests*, which contains test projects generated for each of the five platforms Cocos2d-x supports (Android, iOS, Linux, Mac and Windows), and *templates*, which contains "template code used on automated project creation"[44]. Most files are in folder *cocos*, and each of its subfolders corresponds to one subsystem.

[42] https://docs.cocos2d-x.org/cocos2d-x/v3/en/basic_concepts/
[43] https://docs.cocos.com/creator/manual/en/editor/extension/define.html
[44] https://subscription.packtpub.com/book/game-development/9781785283833/1/ch01lvl1sec15/template-code-walk-through

Table 4.16: Cocos2d-x's root folder

Folder	Files	Description
cocos	923	Cocos2d-x's core and features
extensions	348	Plugins to extend editor features
tests	324	Test projects for Windows, Mac, Linux, etc.
templates	13	Template code for automated project creation

Piccolo

- **Subsystems Detected:** 12 out of 16 (75%)

- **Main Source Code Folder:** the repository root

- **Largest Subsystems:** *Third-Party SDKs (SDK)* with 1,301 files, *Core (COR)* with 64 files and *Low-Level Renderer (LLR)* with 45 files

- **Reference Third-Party Libraries:** *OpenGL* (*glad*, *glfw* and *glm* in Piccolo)

In Table 4.17, we show Piccolo's folder organisation along with descriptions we inferred from source code comments and Piccolo's GitHub wiki[45]. Most files are in folder *./engine/3rdparty*. Files that implement subsystem features are in the *./engine/source/runtime* folder, which is further subdivided into *core*, *function*, *platform* and *resource* folders. The files in *core*, *platform* and *resource* belong respectively to the *Core (COR)*, *Platform Independence Layer (PLA)* and *Resources (RES)* subsystems. The folder *function* contains files for six other subsystems: *Front-End (FES)*, *Gameplay Foundations (GMP)*, *Human Interface Devices (HID)*, *Low-Level Renderer (LLR)*, *Physics (PHY)* and *Skeletal Animation (SKA)*.

Table 4.17: Piccolo's root folder

Folder	Files	Description
./engine/3rdparty	1,301	Third-Party SDKs
./engine/source/runtime	132	Piccolo's core and features
./engine/source/meta_parser	37	C++ reflection pipeline
./engine/source/editor	13	World Editor

[45] https://github.com/BoomingTech/Piccolo/wiki

OlcPixelGameEngine

- **Subsystems Detected:** 5 out of 16 (31%)
- **Main Source Code Folder:** ./Extensions
- **Largest Subsystems:** *Low-Level Renderer (LLR)* with 3 files, while all other subsystems are implemented by a single file
- **Reference Third-Party Libraries:** not analysed because they are in a separate repository[46]

In Table 4.18, we show OlcPixelGameEngine's folder organisation along with descriptions we inferred from source code comments. The root folder contains two subfolders: *Videos*, which contains game examples implemented with OlcPixelGameEngine and *Extensions*, where the source code for all subsystems is located. We observe this game engine has only five subsystems, the lowest subsystem count of all analysed game engines. This happens because OlcPixelGameEngine was created for educational purposes and was never intended to be a production-ready game engine: "Fundamentally the olcPixelGameEngine is designed to support the output of the OneLoneCoder YouTube channel[47]. (...) For example, it does not provide tools to handle asset loading, collision detection, vector mathematics. As it is an educational tool, it is expected the user will provide this functionality."[48].

The largest subsystem in OlcPixelGameEngine is *Low-Level Renderer (LLR)*, implemented by three files in the *Extensions* folder: *olcPGEX_ Graphics2D.h*, *olcPGEX_Graphics3D.h* and *olcPGEX_TransformedView.h*. Files implementing basic *Audio (AUD)*, *Front-End (FES)*, *Online Multiplayer (OMP)* and *Physics (PHY)* features are also present.

Given the entire game engine is composed of only seven files, the absence of a more distributed folder organisation is not detrimental to cohesion. However, if OlcPixelGameEngine were to grow, we argue that the *Extensions* folder should be divided into subfolders, one for each subsystem.

[47] https://www.youtube.com/c/javidx9
[48] https://github.com/OneLoneCoder/olcPixelGameEngine/wiki#what-doesnt-it-do

Table 4.18: OlcPixelGameEngine's *./Extensions* folder

Folder	Description
olcPGEX_Graphics2D.h	Provides 2D sprite manipulation and drawing
olcPGEX_Graphics3D.h	Provides 3D software rendering
olcPGEX_Network.h	Provides networking based on the ASIO library[49]
olcPGEX_PopUpMenu.h	Provides a pop-up menu system
olcPGEX_RayCastWorld.h	Provides 3D ray-casting
olcPGEX_Sound.h	Provides sound generation
olcPGEX_TransformedView.h	Provides 2D drawing and conversion from local to global coordinates

Summary

We summarize our findings to answer RQ1 and discuss folder organisation in terms of naming, hierarchy and division. We also show the largest and smallest subsystems by file count and discuss the reasons behind their size.

Folder Organisation

- **Naming**: Unreal Engine and O3DE give codenames to their subsystems, such as Niagara and Atom Renderer, and use them as a way to delimit their own architectural divisions. However, most game engines follow a more utilitarian approach to subsystem folder naming (e.g., in Urho3d, the Physics subsystem is in the "physics" folder).

- **Hierarchy**: The depth of the folder hierarchy varies significantly between game engines. While Godot, Cocos2d-x, and Piccolo lay out their largest and most important folders in one or two hierarchy levels, Unreal Engine and O3DE use up to five hierarchy levels.

- **Division**: Most game engines have one folder that contains most of the source code that implements subsystems, along with folders dedicated to the *Third-Party SDKs (SDK)*, *World Editor (EDI)*, a plugin subsystem, sample code and supporting tools (e.g., debug tools in Panda3D, game asset encoding in GamePlay3d).

- **Documentation**: Unreal Engine and O3DE have articles in their documentation dedicated to discussing folder organisation. However, most game engines do not document the rationale behind their folder structure, rather focusing on implementation details.

Subsystem File Counts

- **Largest Subsystems**: The *Third-Party SDKs (SDK)* subsystem has the largest file count in all game engines where it is present, frequently followed by *Core (COR)*. As we show in Table 4.2, these subsystems have a long list of responsibilities, and we believe this may be the reason why they frequently have a large file count.
- **Frequently Large Subsystems**: Aside from *Third-Party SDKs (SDK)* and *Core (COR)*, the *Low-Level Renderer (LLR)*, *World Editor (EDI)*, *Gameplay Foundations (GMP)* and *Resources (RES)* subsystems have a large file count in most game engines.
- **Smallest Subsystems**: The OlcPixelGameEngine implements all its features in 7 files, and some of its subsystems are implemented entirely in one file. While architecturally simple, it is also the game engine with the least features of all.
- **Third-Party SDKs in Unreal Engine**: Unreal Engine's *Third-Party SDKs (SDK)* subsystem has over 21,000 files, making it as large as all Godot, O3DE and Urho3d subsystem files together.

4.8.2 RQ2: Do Game Engines Share Subsystem Coupling Patterns?

In the following subsections, we show the Architectural Map visualisation we generated for each of the game engines we analysed. We also show examples which explain why certain subsystems are more coupled in certain game engines, whether these coupling patterns repeat in different game engines and what they can teach us about the architecture they are part of. Subsystems mentioned in the examples are highlighted in the Architectural Map for ease of observation. Additionally, we show coupling information for each game engine in a template as follows:

- The *include* graph density, as computed by Gephi.
- The count of frequent subsystem coupling patterns from Table 4.8 the game engine contains.

We chose to show these two items in the template because they support the understanding of the Architectural Map visualisations. While game engines with a similar degree of coupling also have very similar visualisations, the numerical representation of density is still different and enables more accurate comparison. Moreover, the count of frequent subsystem coupling patterns serves as an indicator of whether a game engine shares coupling patterns with its counterparts or if it exhibits unique coupling patterns distinct from the majority.

For brevity, we only mention the names of files and folders in the examples, not their full path. However, we provide the absolute paths for all mentioned files in Appendix B. We provide this information because it may be useful for researchers who wish to further explore the files we mention to understand their functionalities and whether their coupling could be reduced.

Unreal Engine

- **Include Graph Density:** 0,831
- **Frequent Coupling Patterns:** 11 out of 11 (100%)

Unreal Engine has the highest *include* graph density of all game engines we analysed, with its *Core (COR)* serving as the most substantial contributor to this density. It provides base classes, such as *UObject* and *UClass*, which are inherited by classes in several subsystems. They provide functionality such as garbage collection, reflection and automatic integration with the editor[50].

The *World Editor (EDI)* subsystem also has high in-degree and centrality, and it can be understood by observing its interface classes, for example, *IPluginManager*, which is used by other subsystems to retrieve the list of plugins known to the local installation of Unreal Engine. It is included, for example, by file *SkeletalRenderGPUSkin.h* from the *Core (COR)* subsystem, which checks for whether a hair simulation plugin is installed. If the plugin is found, Unreal Engine will disable deferred updates to skeletal data cache, because this feature needs continuous cache updating to work correctly[51].

[50] https://docs.unrealengine.com/4.26/en-US/ProgrammingAndScripting/ProgrammingWithCPP/UnrealArchitecture/Objects/

[51] https://github.com/EpicGames/UnrealEngine/blob/f1b664d974/Engine/Source/Runtime/Engine/Private/SkeletalRenderGPUSkin.cpp

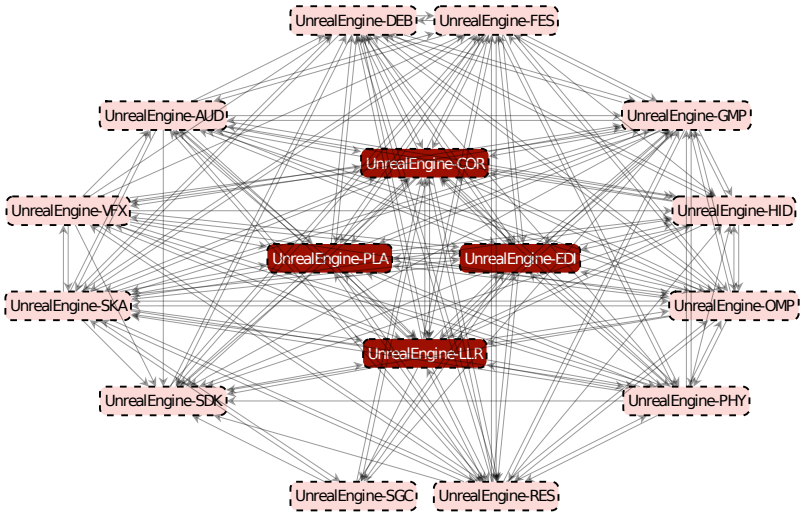


Figure 4.14: Unreal Engine's architectural model.

Godot

- **Include Graph Density:** 0,482
- **Frequent Coupling Patterns:** 11 out of 11 (100%)

In Godot, the *Core (COR)* subsystem has a high in-degree because its files provide project configuration data and implementations of mathematical functions used by several subsystems, such as the files *math_funcs.h* and *project_settings.h*. The *Scene Graph / Culling Optimizations (SGC)* subsystem has a high in-degree because it contains class definitions for game entities such as *Window*, *Camera2D* and *Camera3D* which are used by all graphics-related subsystems. Similarly, the *Resources (RES)* subsystem also provides class definitions for game graphical resources such as the class *Mesh*, which is "a type of Resource that contains vertex array-based geometry, divided in surfaces"[52].

The high in-degree and centrality of the *World Editor (EDI)* subsystem can be understood by observing the *editor_plugin.h* file. It defines a class, *EditorPlugin*, which is inherited by classes in other subsystems which integrate with Godot's editor. For example, the file *audio_stream_editor_plugin.h*

[52] https://docs.godotengine.org/en/stable/classes/class_mesh.html

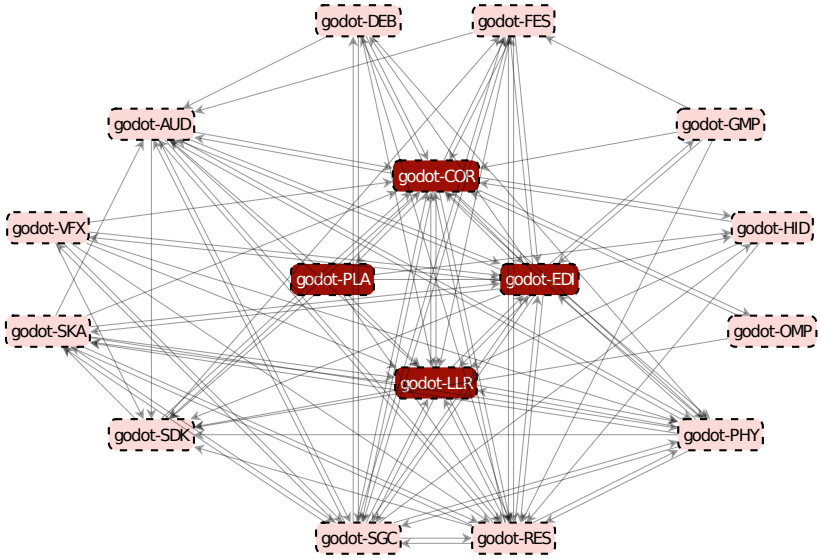


Figure 4.15: Godot's architectural model.

contains the *AudioStreamEditorPlugin* class, which inherits from *EditorPlugin*. It also includes several files from the *Audio (AUD)* subsystem, such as *audio_stream_player.h*. O3DE's *World Editor (EDI)* subsystem employs a similar architecture, as we show in Section 4.8.2.

Panda3D

- **Include Graph Density:** 0,438
- **Frequent Coupling Patterns:** 10 out of 11 (90%)

In Panda3D, the *Gameplay Foundations (GMP)* subsystem has a high in-degree because it contains event management and task management features which are used by several subsystems. For example, the file *nonlinearImager.h* from the *Visual Effects (VFX)* subsystem includes the file *asyncTaskManager.h*, which contains a class to "manage a loose queue of isolated tasks, which can be performed either synchronously (in the foreground thread) or asynchronously (by a background thread)"[53].

The *Low-Level Renderer (LLR)* subsystem has a high in-degree because of its *display* folder, which contains files implementing graphics-related functionality used by the *Scene Graph / Culling*

[53] https://docs.panda3d.org/1.10/python/reference/panda3d.core.AsyncTaskManager

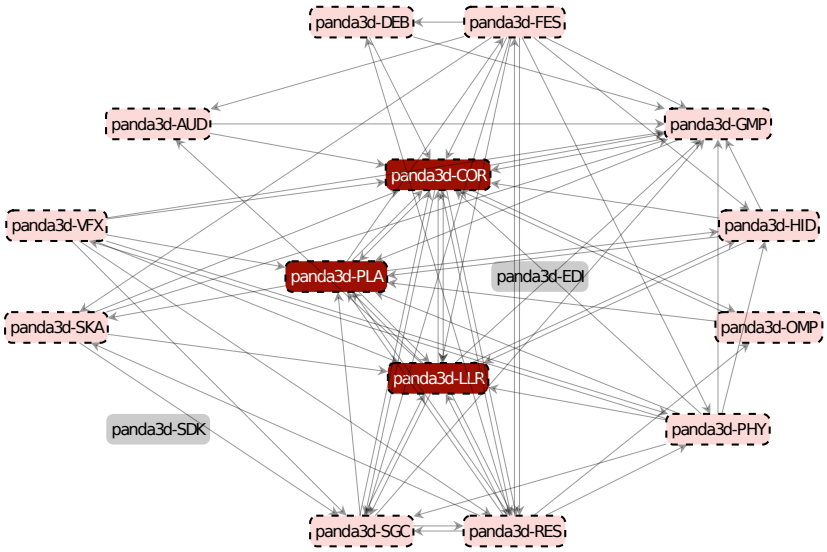


Figure 4.16: Panda3D's architectural model.

Optimizations (SGC) and *Visual Effects (VFX)* subsystems. One of these files, *graphicsStateGuardian.h*, implements a graphics state guardian (GSG), which receives high-level rendering instructions (e.g., drawing a character present in the scene graph) and, based on that, handling low-level rendering instructions in a format the operating system and graphics hardware can understand. As explained by Goslin and Mine [55, p. 112]:

> "All code specific to rendering on a particular platform is contained within a well-defined class called a graphics state guardian. After the system transforms and culls the scene graph, it hands off the graphics entities to the GSG for rendering. A game or application only needs to interact with the scene graph, which means the only part of the code that the system must port and optimize for a particular hardware platform is the local version of the GSG class itself."

O3DE

- **Include Graph Density:** 0,757

- **Frequent Coupling Patterns:** 11 out of 11 (100%)

In O3DE, both the *Platform Independence Layer (PLA)* and *Core (COR)* have high in-degree because they centralize utilities and data structures used by several subsystems, such as *Vector4.h*, a 4-dimensional vector data structure. Similarly, O3DE's *World Editor (EDI)* subsystem provides a common editor API which is included by files that wish to access editor features. For example, the file *AudioControlsEditorWindow.cpp*, which implements an audio system control panel, includes *ToolsApplicationAPI.h*, the file which implements several classes for the editor API[54].

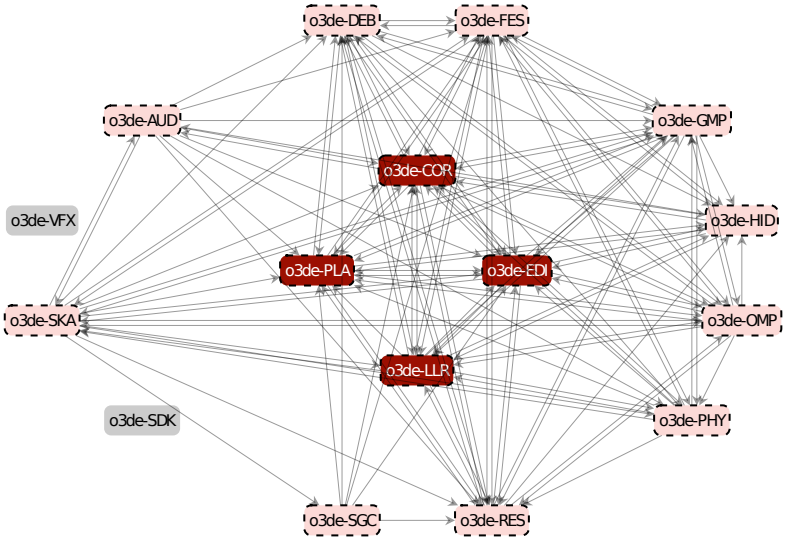


Figure 4.17: O3DE's architectural model.

The *Skeletal Animation (SKA)* subsystem has high centrality and its main component, the *EMotionFX* animation gem, contains features that include and are included by several subsystems. For example, the file *MorphTargetEditWindow.h* includes the file *SpinBox.h* from the *Front-End (FES)* subsystem. It is part of the animation editor which uses the *SpinBox* text input component.

[54] https://docs.o3de.org/docs/user-guide/interactivity/audio/audio-controls-editor/

Conversely, the *Skeletal Animation (SKA)* subsystem provides abstractions which are used by *Front-End (FES)*, such as *AnimKey.h*, a class which represents "a setting for a[n] [animation] property at a specific time"[55].

FlaxEngine

- **Include Graph Density:** 0,442
- **Frequent Coupling Patterns:** 11 out of 11 (100%)

In FlaxEngine, the *World Editor (EDI)* subsystem has high centrality and contains features that both include and are included by other subsystems. For example, the file *VisjectGraph.h*, which is responsible for FlaxEngine's visual scripting system, is included by *AnimGraph.h* from the *Skeletal Animation (SKA)* subsystem. This inclusion exists because animation playback logic is also represented as a graph in the editor, and therefore it uses the same data structure implementation.

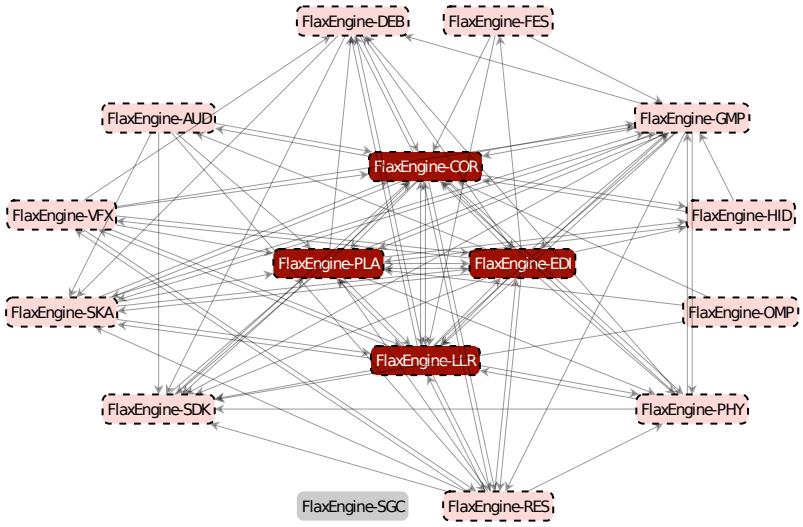


Figure 4.18: FlaxEngine's architectural model.

Besides being included, we observe the *World Editor (EDI)* subsystem also includes files from many subsystems. The game cooking process, which "compiles the game scripts and processes all used assets to output standalone game files for the destination platform"[56], is an example. The

[55] https://docs.o3de.org/docs/user-guide/visualization/cinematics
[56] https://docs.flaxengine.com/manual/editor/game-cooker/index.html

file *GameCooker.cpp* from the *World Editor (EDI)* subsystem is responsible for game cooking and includes 37 files across several subsystems. This inclusion exists because the game compilation depends on OS-specific settings (provided by classes from the *Platform Independence Layer*) and high-level utilities such as JSON processing (provided by the *Core* subsystem).

GamePlay3d

- **Include Graph Density:** 0,489
- **Frequent Coupling Patterns:** 11 out of 11 (100%)

In GamePlay3d, the *Profiling & Debugging (DEB)* subsystem has a high in-degree because its *DebugNew.h* file is included by several subsystems to replace global *new* and *delete* C++ operators for "memory tracking"[57]. We observe a similar implementation in Urho3d, as we show in Section 4.8.2. We observe the *Logger.h* file is also frequently included for debugging purposes. Even though debugging code would normally be removed upon pushing to the *master* branch, we observe four files in GamePlay3d's repository still include either *DebugNew.h* or *Logger.h*.

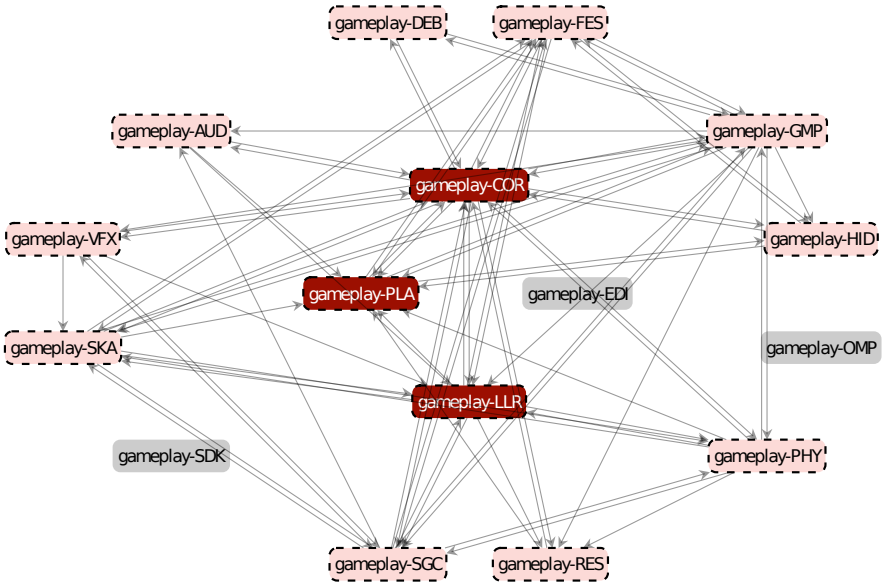


Figure 4.19: GamePlay3d's architectural model.

[57] https://github.com/gameplay3d/gameplay/blob/4de92c4c6/gameplay/src/DebugNew.h

The *Gameplay Foundations (GMP)* subsystem has high centrality, which indicates it has a gatekeeper role. For example, the file *lua_Rectangle.h*, which is part of the Lua bindings to GamePlay3d's scripting system, includes the file *Rectangle.h* from the *Low-Level Renderer (LLR)* subsystem. This way, developers write Lua code to draw a rectangle, which is then processed by GamePlay3d's scripting system and mapped into a C++ call to the renderer.

Urho3d

- **Include Graph Density:** 0,505
- **Frequent Coupling Patterns:** 11 out of 11 (100%)

In Urho3d, *Core (COR)* and *Third-Party SDKs (SDK)* are both frequently included and central because they contain data structures and utilities used by several subsystems, such as 2D and 3D matrices (*Matrix2.h*, *Matrix3.h*), threads (*Thread.cpp*) and mutual exclusion flags (*Mutex.cpp*).

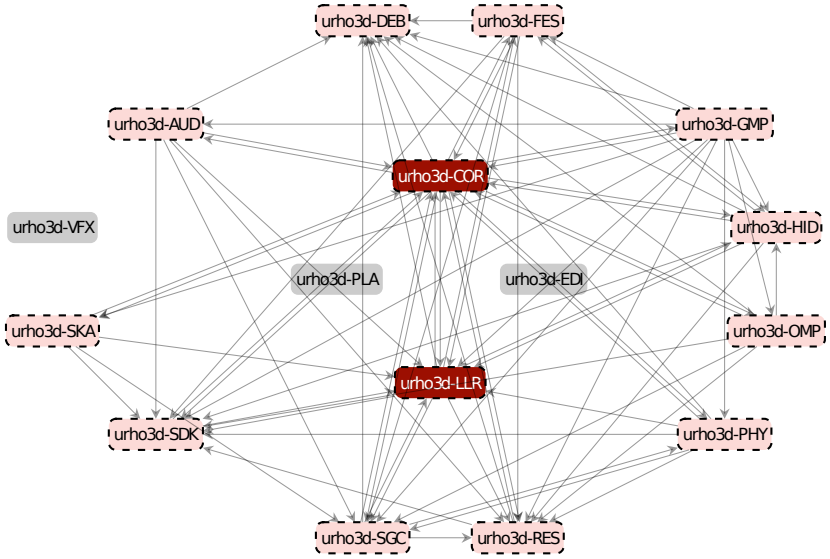


Figure 4.20: Urho3d's architectural model.

Similarly to what we observe in GamePlay3d, Urho3d's *Profiling & Debugging (DEB)* subsystem is one of the most frequently included by other subsystems and its features are centralized in a single file, *DebugNew.h*. It contains code that replaces the global *new* C++ operator to allow for

"easier memory leak detection on MSVC compilers"[58]. Even though debugging code would normally be removed upon pushing to the *master* branch, we observe 235 files in Urho3d's repository include it.

Cocos2d-x

- **Include Graph Density:** 0,363
- **Frequent Coupling Patterns:** 9 out of 11 (82%)

In Cocos2d-x, the *Core (COR)* subsystem is both frequently included and central to Cocos2d-x because it contains data structures and utilities used by several subsystems, such as 2D and 3D vectors (*Vec2.cpp*, *Vec3.cpp*) and a random number generator (*ccRandom.cpp*).

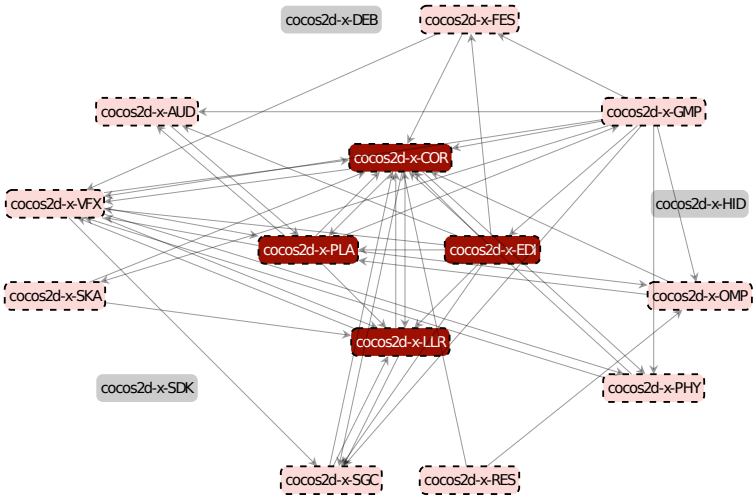


Figure 4.21: Cocos2d-x's architectural model.

The *Platform Independence Layer (PLA)* and *Visual Effects (VFX)* subsystems are among the most central, which indicates they have a gatekeeper role. For example, the *CCImage.h* file, which defines a class to represent a game 2D image, includes *CCTexture2D.h* from *Low-Level Renderer (LLR)*, and is included by *CCAutoPolygon.h* from the *Visual Effects (VFX)* subsystem. The latter

[58] https://github.com/urho3d/urho3d/blob/e0ce107356b255bf2e24d94a41d00b512b9ce633/Source/Urho3D/DebugNew.h

converts images into polygon sprites for renderer optimisation purposes[59]. Therefore, the *Platform Independence Layer (PLA)* provides a link between fine-grained features and the more coarse-grained domain classes they use.

Piccolo

- **Include Graph Density:** 0,273
- **Frequent Coupling Patterns:** 7 out of 11 (64%)

In Piccolo, the *Resources (RES)* subsystem is both frequently included and central because it contains definitions of game entities, such as *mesh.h* used by *Low-Level Renderer (LLR)* and *skeleton_data.h* used by *Skeletal Animation (SKA)*. It also contains a configuration and asset manager that is used by the *World Editor (EDI)* and *Gameplay Foundations (GMP)* subsystems.

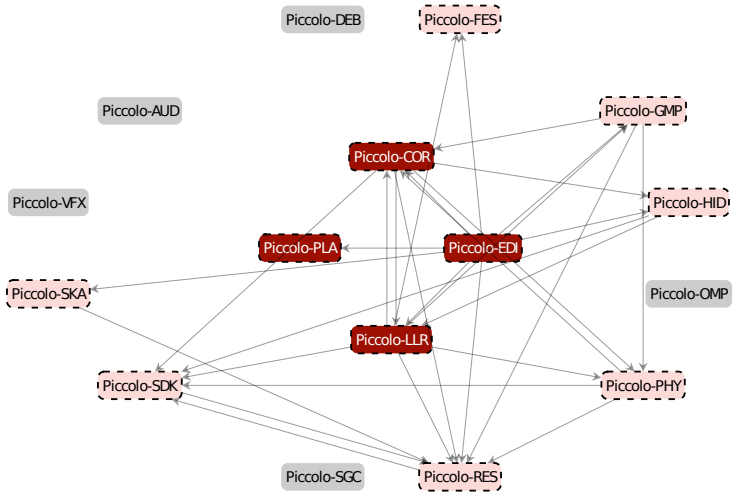


Figure 4.22: Piccolo's architectural model.

The *Low-Level Renderer (LLR)* subsystem has high centrality because contains files that depend on *Third-Party SDKs (SDK)*, *Gameplay Foundations (GMP)*, *Physics (PHY)* and *Front-End (FES)*. For example, the file *ui_pass.cpp*, which is responsible for the UI rendering step, includes the file *window_ui.h* from the *Front-End (FES)* subsystem, which defines a class which represents a UI window in a platform-agnostic way. At the same time, the file *render_camera.h* from the *Low-Level*

[59] https://docs.cocos2d-x.org/cocos2d-x/v3/en/sprites/polygon.html

Renderer (LLR) subsystem implements a *RenderCamera* class which is included by files in the *Human Interface Devices (HID)* subsystem to adjust camera position according to mouse movement.

OlcPixelGameEngine

- **Include Graph Density:** 0,000
- **Frequent Coupling Patterns:** 0 out of 11 (0%)

As we explain in Section 4.8.1, most of OlcPixelGameEngine's subsystems are small and contained in a single .h file. These subsystem files do not include each other, making this game engine fully decoupled. As an educational game engine, this is a design choice made by the game engine developers, which delegates to OlcPixelGameEngine's users the responsibility of coupling subsystems as they see fit. For this reason, in-degree, centrality and density measurements equal zero for all OlcPixelGameEngine subsystems, and therefore no coupling patterns exist as we show in Figure 4.23.

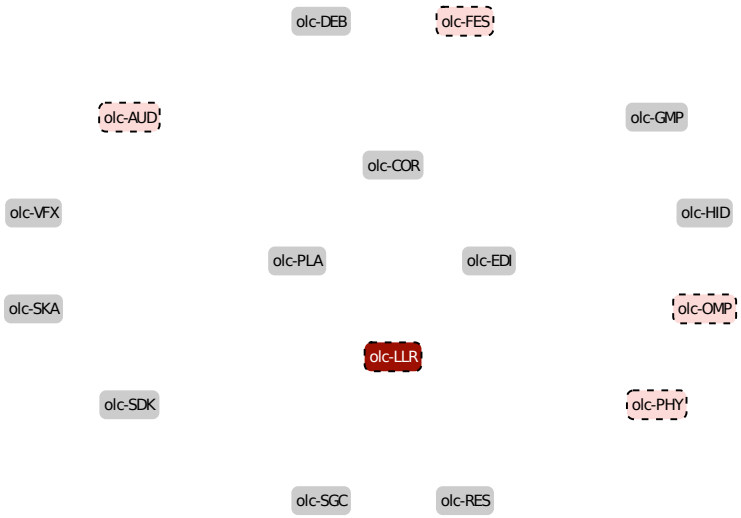


Figure 4.23: OlcPixelGameEngine's architectural model.

While a fully decoupled game engine may facilitate users' comprehension of its architecture, this design may not align with the needs of professional game development. Video game developers expect a game engine to offer a range of ready-to-use features, allowing them to initiate video game

development without the need to add or modify parts of the game engine's architectural structure. Therefore, OlcPixelGameEngine is better suited for educational rather than professional use cases.

Summary

We summarize our findings to answer RQ2 and discuss patterns we found in subsystem coupling, as well as their individual and shared responsibilities within the game engine. The responsibilities we found by analysing the source code of the selected game engines widely match the descriptions from the reference architecture as we show in Figure 2.7. However, different from the reference architecture, our results can demonstrate practically how these subsystems work together and which role each of them plays.

Subsystem Coupling Patterns

- **Frequent Patterns**: Out of the ten game engines we analysed, eight share the same frequent coupling patterns. This is evidence that game engines share architectural commonalities despite their differences in file count, subsystem count and features.
- **Foundations**: Subsystems with high in-degree play the role of foundations for their game engines by providing features used by several other subsystems, most frequently *Core (COR)* and *Platform Independence Layer (PLA)*. We show examples of foundation subsystems in Unreal Engine (Section 4.8.2) and O3DE (Section 4.8.2).
- **Gatekeepers**: Subsystems with high centrality play the role of gatekeepers, both using and providing features for several subsystems, most frequently *Platform Independence Layer (PLA)* and *World Editor (EDI)*. We show examples of gatekeeper subsystems in Cocos2d-x (Section 4.8.2) and GamePlay3d (Section 4.8.2).
- **Peripherals**: *Audio (AUD)*, *Physics (PHY)* and *Human Interface Devices (HID)* are among the least central subsystems overall. This shows these subsystems either include many others or are included by many others, but not both at the same time.

Other Findings

- **Highest and Lowest Graph Density**: Unreal Engine and O3DE have the highest *include* graph density of all game engines we analysed; Piccolo and OlcPixelGameEngine have the lowest. While we observe a positive correlation between density and file count, it does not imply causation. We discuss how this aspect will be explored in future work in Section 6.1.

- **Relevance of Third-Party Libraries**: While playing a foundational role in several game engines, the *Third-Party SDKs (SDK)* subsystem is among the lowest in-degree overall. However, its relevance may have been diminished by the fact we detected this subsystem in only 4 out of 10 game engines.

- **Debugging Code**: GamePlay3d, Urho3d and Unreal Engine have highly coupled *Profiling & Debugging (DEB)* subsystems because they have method calls for error logging and memory profiling in several files.

4.8.3 Unclustered Files/Folders

As we explain in Section 4.7, we have a number of files/folders in each game engine which were not clustered into any subsystem because their responsibilities were not described in the reference architecture. As we show in Section 2.3, we expanded our analysis to other game engine architectures but observed they also do not describe the responsibilities of these files. Therefore, we discuss two possible reasons why these responsibilities are not considered to be subsystems in any of the analysed software architectures:

- **New Technologies:** The use of AR/VR technology in video games is relatively new and game engines have implemented it only recently. For example, the commit which adds the folder to Godot is from 2017[60], and we observe the same for Unreal Engine's folder[61]. This can be explained by the fact the game engine architectures we analysed were all created in the early 2000s, including Gregory [6], who published the first edition of their book on game engine

[60] https://github.com/godotengine/godot/commit/ca4f055db0a4e6f9ea7b38cde14dc8
[61] https://github.com/EpicGames/UnrealEngine/commit/14d4bfaf3d10a67e7d7d8b984391d4

architecture in 2009. Therefore, these architectures do not represent all currently popular game engine technologies.

- **Low Generalisability:** There is no consensus among game engine designers as to whether some features are game-specific or generalisable. For example, while Gregory [6, p. 33] describes Artificial Intelligence as a "game-specific" feature, these are also provided by Godot and Unreal Engine, as we show in Table 4.7.

However, the extent to which a feature is generalisable is open to debate. For example, Juan Linietski, Godot's creator, wrote the following about terrain modelling: "I insist [the] terrain [feature] has to be an add-on. There are just *so many different ways* to do it and all are great but incompatible with each other"[62]. Therefore, he argues that, if terrain modelling cannot be generalised, it would be better implemented as an add-on, which is external to the game engine, and not as a subsystem, which is within the game engine. This point of view is supported by Thorn [5, p.9]:

> "The idea the game engine is the heart or core containing almost all the generalizable components that can be found in a game implies that there are other parts of a game and of game development that do not belong to the engine component on account of their specific, nongeneralizable nature. These parts include at least game content and game tools."

There is also debate as to whether test and build tools should be considered game engine subsystems. For example, while studies argue that there is "a clear need for open-source, general tools" for video game testing [56], we did not find this kind of feature in the game engines we analysed. The unit test files we found, for example, in O3DE, are for O3DE itself. Therefore, by acting as a support for the game engine's own development, we argue test and build functionalities should not be considered subsystems.

In summary, we observe that what differentiates a subsystem from a feature is its generalisation and game-making capabilities. The task of the game engine architect is defining what they want the game engine to do and with which level of generalisation. This way, they can give developers a starting point for building functionality in an understandable and consistent way.

[62] https://twitter.com/reduzio/status/1712403709785710598

Chapter 5

Evaluation

In this section, we evaluate the effectiveness SyDRA in helping developers better understand and maintain game engines. We also use architectural descriptions in literature as a reference for comparison with our findings. We perform a qualitative evaluation and a user study. By performing the evaluations, we answer the following research questions:

RQ1: (Compliance, Qualitative) To what extent do the game engines we selected for architecture recovery with SyDRA match the architectural descriptions provided in the literature? We compare the reference architecture used for subsystem detection in SyDRA to architectures proposed by four other authors. We discuss how our subsystem counts and coupling patterns converge and diverge with each of these architectures, and how a more complete architecture could be created by combining information from these different sources.

RQ2: (Understanding/Maintainability, User Study) Does SyDRA help developers understand and maintain the architecture of game engines? We conduct a user study with 16 software developers where they use SyDRA to perform architectural understanding and impact analysis tasks in Godot. We discuss how the use of SyDRA influences task completion time and correctness.

5.1 RQ1 - To what extent do the game engines we selected for architecture recovery with SyDRA match the architectural descriptions provided in the literature?

We chose the "Runtime Game Engine Architecture" by Gregory [6, p. 33] as our reference architecture for subsystem detection with SyDRA, as we explain in detail in Section 2.3. However, before applying SyDRA to open-source game engines, we were not sure whether we would find a match between what the reference architecture states a "usual" game engine looks like and what open-source game engine developers actually build. After studying the folder structure and *include* relationships of 10 open-source game engines, we observed that the reference architecture and actual game engine architectures share many similarities, which we discuss in detail in Section 4.7.1 and Section 4.7.2.

However, we are aware these matches might be biased because we used a single reference architecture, and therefore we did subsystem detection with the concepts of this architecture in mind only. In this section, we compare the "Runtime Game Engine Architecture" to architectures proposed by four other authors: Bishop et al. [2], Rollings and Morris [4], Sherrod [3] and Thorn [5], which we presented in Section 2.3. We discuss how our subsystem counts and coupling patterns converge and diverge with each of these architectures, and how combining information from these different sources enables us to have a more holistic view of game engine architecture.

In Figure 5.1, we show how the reference architecture and the four other architectures diverge in naming but converge in subsystem functionality. Three subsystems are unanimous: Audio, Graphics and Input. Given all modern video games draw things on screen, play sounds and receive commands from the player via an input device (e.g., a joystick), this is in line with our intuition about video games. We observed these subsystems are also frequently present in open-source game engines. We found a *Low-Level Renderer (LLR)* subsystem in 9 out of 10 game engines, and *Audio (AUD)* and *Human Interface Devices (HID)* in 8 out of 10 game engines.

The only subsystems described solely by Gregory [6] are *World Editor (EDI)* and *Third-Party SDKs (SDK)*. In tandem, these subsystems are among the less common in open-source game engines. We found a *World Editor (EDI)* subsystem in 6 out of 10 game engines, and *Third-Party*

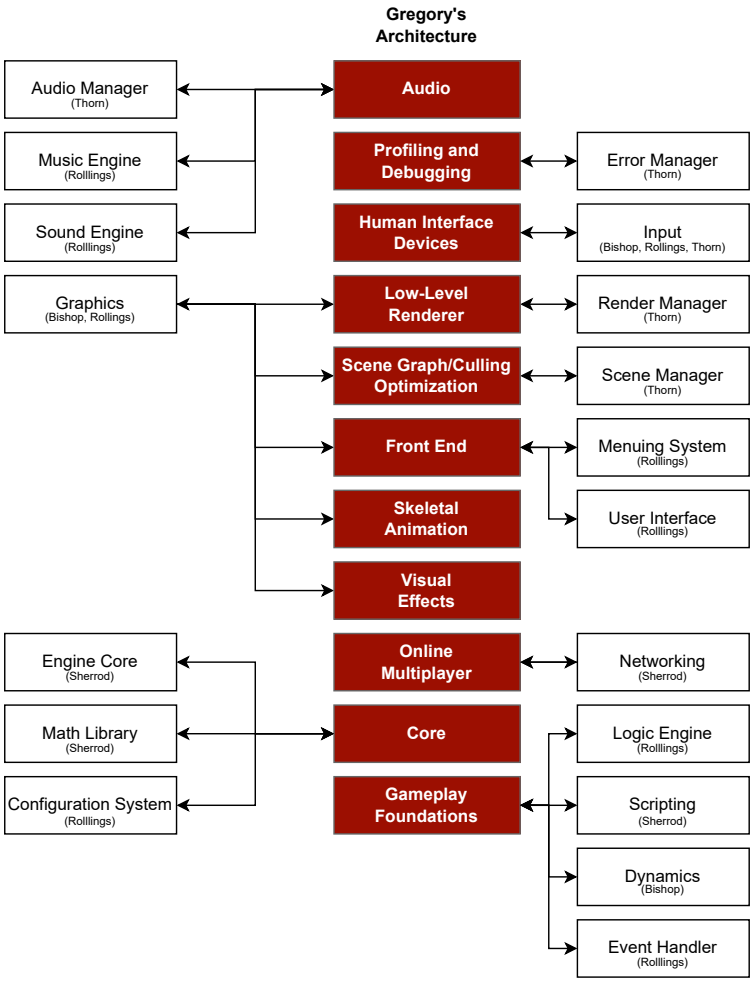


Figure 5.1: Subsystem naming differences among architectures

SDKs (SDK) in 5 out of 10 game engines. This is evidence that some architects and developers might not think of these subsystems as essentials, or rather as subdivisions of other subsystems. For example, one could argue that instead of having one editor that supports all subsystems, each subsystem could encapsulate its own editor (e.g. audio editor, animation editor, etc.), leaving the editor abstraction for another subsystem such as *Core (COR)*.

Moreover, the definition of the game engine core diverges between architectures. On one hand, Bishop et al. [2] and Thorn [5] do not acknowledge the existence of a subsystem named "core". On the other hand, Gregory [6] and Sherrod [3] not only represent the game engine core as a subsystem in their architecture but also describe its function in detail. Gregory [6, p.39] describes core as "a grab bag of useful software utilities" that includes, for example, math libraries, data structures

and game engine configuration files. Rollings and Morris [4, p. 626] also describe a "configuration system" as a distinct subsystem, but does not refer to it as the core of the engine.

Sherrod [3] provides a detailed description of the game engine core and divides it into two parts: math library and engine core. The engine core contains utilities that are "parts of the game engine framework that are not specific to any one system and are used to aid in the completion of a task". We observed these definitions of core match what we found in files and folders of open-source game engines, which we describe and discuss in detail in Section 4.7.1.

Also as discussed in Section 4.7.1, we found several features in open-source game engines which are not described by any architectures, such as AR/VR, AI character behaviour, AI navigation and terrain creation. We argue that an extended version of the "Runtime Game Engine Architecture" could be created by considering these newly found subsystems, as well as a wider range of naming variants and subdivisions proposed by Gregory [6] and other authors. In future work, we intend to re-apply SyDRA while using this extended "Runtime Game Engine Architecture" as a reference for the subsystem detection step. We believe that making the reference architecture more representative of the actual structure of most game engines may enable us to observe more fine-grained coupling patterns and a more accurate emergent architecture.

5.2 RQ2: Does SyDRA help developers understand and maintain the architecture of game engines?

We conducted an observational comparative user study with 16 developers to determine the qualitative success of SyDRA in supporting developer understanding and maintenance of game engines. We based our study design on another similar study by Briand et al. [8, p. 518], hereby called "original study". We provide the replication package for the study on Zenodo[1].

In the original study, the authors observed and measured the impact of "good" and "bad" object-oriented design on system understandability and maintainability. In our study, we observed and measure the impact of the use of SyDRA on game engine understandability and maintainability. We asked 16 developers to perform nine tasks of architectural pattern identification and impact analysis

[1] https://zenodo.org/records/10210702

for the Godot game engine. While the control group performs these tasks aided only by Visual Studio Code, the treatment group is also aided by a Moose model, which is the result of applying SyDRA to Godot.

In this section, we describe the user study. In Section 5.2.1, we describe the rationale of the study and its null and alternative hypotheses. In Section 5.2.2, we explain the participant selection process. In Section 5.2.3 we explain our choices of game engines and tools, in Section 5.2.5 our process of task elaboration and in Section 5.2.7 how the division between control and treatment groups was made. In Section 5.2.8, we explain the dependent variables of the study and how we measured them. In Section 5.2.6, we explain the steps performed by participants and their interaction with us. In Section 5.2.9 we explain the statistical analysis of the study data concerning normality and statistical significance. Finally, we show the results of the study in Section 5.2.10, discuss them in Section 5.2.11, and discuss possible threats to validity in Section 5.2.12.

5.2.1 Hypotheses

The null hypothesis is stated as:

- H_0 The use of SyDRA provides no significant difference in the understandability and maintainability of game engine architecture.

The alternative hypotheses, i.e., what is expected to occur, are stated as:

- H_1 Game engine architecture is significantly easier to understand with the use of SyDRA.
- H_2 It is easier to perform impact analysis (locate changes) on game engines with the use of SyDRA.

5.2.2 Participants

We selected 16 participants for the study, all of them over 18 years of age and with prior experience in object-oriented programming. The number of participants was determined before the beginning of the study by using a two-sample T-test, which is explained in detail in Section 5.2.9. We recruited participants via email or by asking them in person. Most participants based in Canada

chose to participate in person at the Griottes Lab at Concordia University in Montreal, Canada. However, most participants participated remotely by connecting to a Windows 10 virtual machine we set up on the Microsoft Azure cloud service.

In Table 5.1 and sub-tables we show study participant demographics. Most participants are men under 30, currently based in either Brazil or Canada. They are mostly students, researchers or software developers outside the video game industry. They have mostly 2 to 5 years of software development experience and have used Unity for student or hobby projects. However, we observed participants' familiarity with game engine usage and development varies greatly. For example, while 57% of the participants reported having no experience with game engines, two participants reported coding their own game engines. This diversity of experience levels is important to our study because it allowed us to observe how the tools we selected for the study were used differently by each kind of developer and the challenges faced by each of them.

While the selected participants were very diverse with regard to development experience, they were less diverse with regard to age, gender and country of residence. However, these demographics match those found in well-known yearly surveys about the video game industry such as GDC's 2023 State of the Game Industry Report[2] and IGDA's Diversity in the Game Industry Report 2021[3]. We discuss the influence of diversity in our study results and how we addressed related threats to validity in Section 5.2.12.

5.2.3 Experimental Materials

In this study, participants analysed Godot[4], a cross-platform, free and open-source game engine released by Juan Linietsky and Ariel Manzur in 2014. We chose Godot due to its relevance to the open-source developer community on GitHub, as explained in Section 4.1. While control group participants used exclusively Visual Studio Code to analyse Godot's source code, treatment group participants used both Moose + Visual Studio Code.

We created instructional documents to teach participants to use both tools. In these documents, we provided step-by-step instructions on how to use tool features pertinent to the study, such as

[2] https://reg.gdconf.com/state-of-game-industry-2023
[3] https://igda.org/dss
[4] https://github.com/godotengine/godot

Table 5.1: Demographics of the user study participants

(a) Current job position

Job position	**Count**
Student/Researcher	8
Non-Video Game Developer	5
Video Game Developer	2
Other: Software Eng. Teacher	1

(b) Years of software development experience

Years of experience	**Count**
Between 2 and 5 years	7
Between 5 and 10 years	5
More than 10 years	2
Between 0 and 2 years	2

(c) Gender

Gender	**Count**
Male	13
Female	3
Other	0
I prefer not to say	0

(d) Country of residence

Country of residence	**Count**
Canada	8
Brazil	6
France	1
United Kingdom	1
I prefer not to say	0

(e) Age group

Age group	**Count**
23 to 27 years	7
33 to 37 years	5
28 to 32 years	2
18 to 22 years	1
38 to 42 years	1
43 to 47 years	0
48 years or older	0
I prefer not to say	0

(f) Type of game engine experience

Game engine experience	**Count**
I never used a game engine	7
I have used game engines as a student/hobbyist	6
I have used game engines as a video game developer	2
I have used game engines as a game engine developer	1

(g) Game engines used (multiple choices allowed)

Game engine	**Count**
I never used a game engine	8
Unity	8
Godot	3
Unreal Engine	3
Other: “I coded my own game engine”	2
CryEngine or any of its derivatives (e.g. Amazon Lumberyard, O3DE)	0

searching for files. We provided screenshots to illustrate the instructions and, at the end of the documents, optional exercises to help participants further familiarize themselves with the tools. We explain how these documents were introduced and used by participants in Section 5.2.6.

In this section, we provide an overview of Moose's and Visual Studio Code's main features, how they were used by participants and our rationale for choosing these tools for the study.

5.2.4 Visual Studio Code

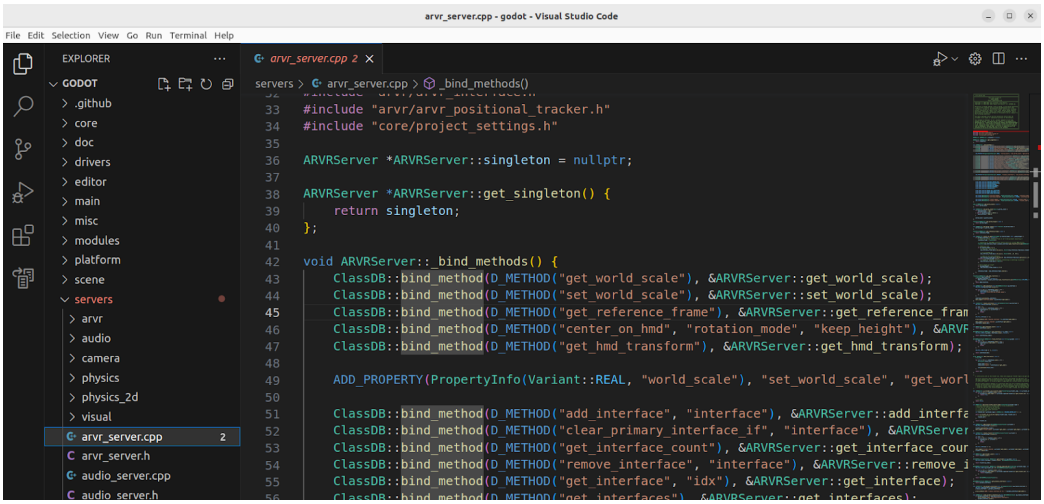


Figure 5.2: Visual Studio Code set up for the study, showing a C++ file from Godot Engine

Visual Studio Code[5] is a source code editor released by Microsoft in 2015. It supports multiple programming languages and allows users to write, execute and debug code. It also provides utilities to aid code editing, such as code folding, syntax highlighting and word search both for single or multiple files. During the study, control group participants located files and folders inside Godot's repository, read the source code and searched for words and file names as directed by the task statements, as we show in Figure 5.2.

We chose Visual Studio Code due to its popularity among professional developers. According to the Stack Overflow Developer Survey 2023, 74.09% of professional developers and 78.39% of developers learning to code use Visual Studio Code[6]. By asking our participants to use a tool they were likely to be familiar with, we attempted to decrease learning effects on the study, as we show

[5] https://code.visualstudio.com/
[6] https://survey.stackoverflow.co/2023/#technology-most-popular-technologies

in Section 5.2.12. Moreover, Visual Studio Code supports C++, the language in which Godot is written, is cross-platform and can be installed and distributed freely without the need for a licence. This platform and license flexibility allowed us to promptly set up a development environment for the participants locally and on the cloud, as explained in Section 5.2.2.

Moose

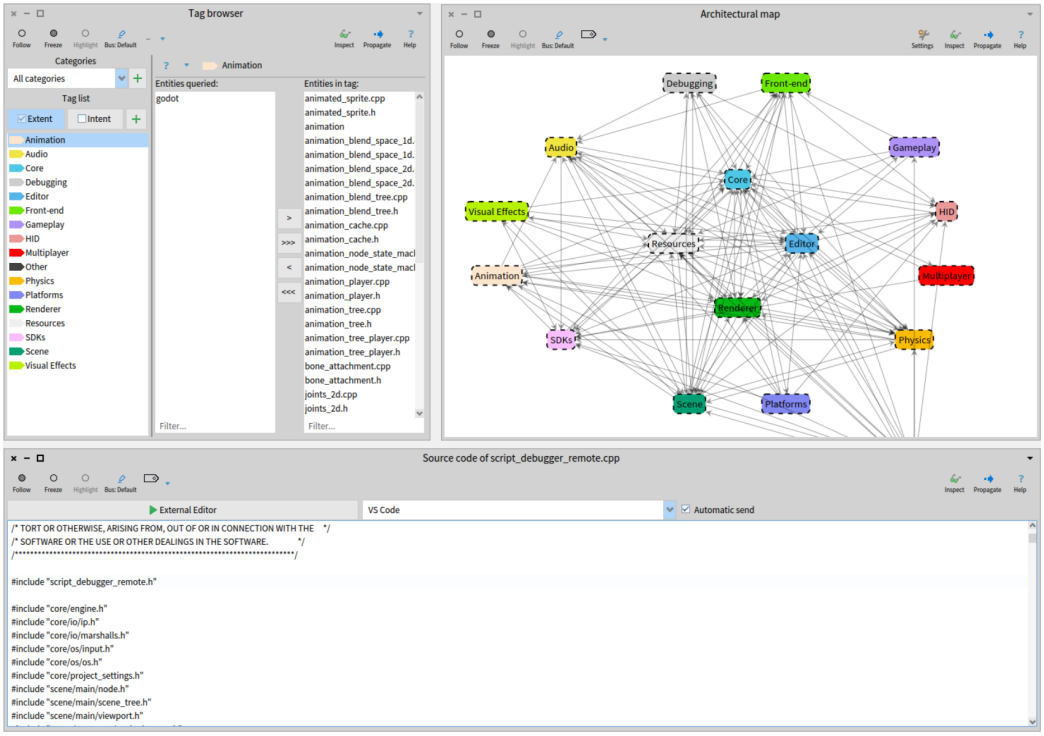


Figure 5.3: Moose set up for the study, with Architectural Map visible on the top right

Moose 10 is a platform for software analysis we used in the implementation of SyDRA, as explained in Section 4.5 and Section 4.6. During the study, treatment group participants located files and folders using Moose's Architectural Map visualisation (Figure 5.2, top right) and propagated them to a built-in source code browser to inspect the source code (Figure 5.2, bottom). Participants could also launch Visual Studio Code from Moose's built-in editor to use features such as code folding, syntax highlighting and word search, not available on Moose. They could also use Moose's tag browser to see the list of files clustered into each subsystem, represented as coloured tags (Figure 5.2, top left).

We chose Moose because we used it in the implementation of SyDRA, and the objective of the user study is to evaluate whether this implementation helps developers understand and maintain game engines. We also chose Moose because it is cross-platform, free and open-source, and therefore allowed us to promptly set up a development environment for the participants locally and on the cloud, as explained in Section 5.2.2.

5.2.5 Experimental Tasks

Participants are expected to answer nine tasks during the study, which are divided into two parts: architectural understanding and impact analysis. In architectural understanding tasks, participants are asked to explain game engine subsystems and dependencies between files. In impact analysis tasks, participants are asked to point out which files should be changed as a result of a change/removal of functionality in another part of the system. Participants performed seven architectural understanding tasks and two impact analysis tasks.

We wrote our task statements based on those provided in Appendix A of the original study [8, p. 527]. We changed the statements slightly to make them easier for novice developers to understand and also to conform them to the scope of analysis of our Moose model. For example, while the original study asked participants to identify classes and operations, we could not ask participants to do the same with a Moose model because it is more coarse-grained, containing only files. The changes we did are described in Table 5.2.

Table 5.2: Original words used by Briand et al. [8, p. 527] in their task statements and how we changed them

Original study	**Our study**
Class	File
Component	File
Externally visible system operations	Functionalities
Services	Functionalities
System	Subsystem

Task statements were also slightly adapted to reflect the steps participants in different groups had to perform to find files in the tools they were using. For example, while in Task 2 we asked treatment group participants to "Expand the Audio subsystem" and then "propagate" a given file, we asked control group participants to search the file by name and then open it (see Appendix A, Table A.1 and Table A.2 for reference). This way we made sure both participants were directed to the same file, even though they followed different steps to find and open it.

We have no reason to believe that the changes we made to the task statements changed their meaning or their level of difficulty compared to the original study. As for the choice of files and subsystems for the tasks, we also have no reason to believe any given group of files or subsystems is significantly easier for participants to understand than another. However, some subsystems contain fewer files than others, and we favoured subsystems with fewer files when elaborating the tasks. We did so because there are tasks where participants are expected to list subsystem files, and writing the names of hundreds of files is unfeasible. Besides file count, we did not follow any other subsystem selection criteria, the same as done in the original study.

5.2.6 Procedures

Before the actual study took place, we performed a trial study with three participants to ensure the task statements were understandable and the tasks feasible. The trial study participants were selected following the same criteria as the actual study and also performed the same tasks. The feedback they provided helped us develop a time estimation for task completion, and improve task statements and instructional documents. The answers they provided were not considered in our results and they were not invited to participate in the actual study.

The actual study session was divided into four parts. First, participants were asked to read and follow the instructional document described in Section 5.2.3. As soon as participants were ready, they could choose to move on to the second stage, which was performing the tasks. Finally, participants completed a debriefing questionnaire and exit interview, where we asked them for background information and also for a workload assessment of the tasks they performed. For the second and third parts, participants submitted a form with their answers, so we could record both the answers and the time elapsed between them.

The debriefing questionnaire was divided into two parts (see Appendix A, Table A.3, for reference). In questions 1 to 7, we asked participants about their professional backgrounds and demographics. In questions 8 to 13 we asked participants to make a workload assessment of the tasks they performed, based on the NASA TLX (Task Load Index) questionnaire. We chose the NASA TLX questionnaire because it has been used for over 20 years by several studies that evaluate software development [57, p.668], interface design and decision-making activities [58, p.906].

We used the information provided by participants in the debriefing questionnaire to qualitatively measure their perception of effort and stress concerning the tools they used and the tasks they performed. We also correlated their performance with their number of years of development experience and familiarity with game engines to understand how each of these variables influences performance. We discuss these comparisons in more detail in Section 5.2.10.

After the debriefing questionnaire, we conducted a semi-structured exit interview. We asked participants whether they would like to share their thoughts about the tasks they completed and the tools they used in the study. In the case they accepted, we listened to their comments, wrote them down and later analysed them using open card sorting, which we discuss in Section 5.2.11. In case the participants did not want to share their thoughts, we told them their participation in the study was finished.

We remained available by chat, email or in person throughout the actual study session to provide support to participants. Our support was limited to clarifying task descriptions when asked and resolving technical issues with the computer and tools related to the study. When the study was administered in person, we remained sitting a few meters away from the participants in a position where we could not see their screens. We chose to physically distance to mitigate the observer effect, as we discuss in Section 5.2.12. For the same reason, we did not enforce a time limit for task completion. When participants asked us about the expected completion time, we told them the tasks could be completed in 40 to 60 minutes, but they could take as long as they considered necessary.

5.2.7 Design

We employed a between-group 2 x 1 design, as described in Table 5.3. The independent variables are the experimental runs (X) and the tools used to analyse the game engine (Y). Under the

Table 5.3: Experimental Design

Variable X	**Variable Y - Tool**	
Run	**Visual Studio Code**	**Moose + Visual Studio Code**
1	A	B

names of the tools, the letters denote control (A) and treatment (B) groups. The assignment of participants to the groups was done randomly, as a way to control learning and fatigue effects. Tasks were shown in random order to participants, except for Tasks 3 and 4 which depend on each other and therefore cannot be understood if shown in inverse order.

5.2.8 Dependent Variables and Their Collection Procedures

We measured the level of game engine architectural understanding by the participants by measuring the time they spent on tasks and how correctly they completed tasks. From this data, we derived six dependent variables, described as follows in the original study [8, p. 518]:

- *UndTime*: Time spent on architectural understanding tasks, in seconds.
- *UndCorr*: Correctness of architectural understanding tasks (e.g. the number of tasks correctly answered).
- *ModTime*: Time spent on impact analysis tasks, in seconds.
- *ModComp*: Completeness of the impact analysis, obtained by dividing the number of correct files informed by the participant by the actual number of correct files. The maximum value is two because there are two impact analysis tasks.
- *ModCorr*: Correctness of the impact analysis, obtained by dividing the number of correct files informed by the participant by the number of files informed by the participant. The maximum value is two because there are two impact analysis tasks.
- *ModRate*: Modification rate, obtained by dividing the number of correct files informed by the participant by *ModTime*.

5.2.9 Data Analysis Procedure

Data was collected for all participants during a single experimental run which lasted for about one month. Therefore, eight data points were available for the control group and eight for the treatment group. All participants answered all tasks and debriefing questions. In this section, we describe the statistical techniques we employed before the study to determine the number of participants, as well as to determine data normality, a measurement that will be later used to determine statistical significance.

Number of participants

Our first data analysis procedure was done to determine the number of study participants. We used a two-sample T-test to determine the number of participants we would need to detect a statistically significant difference in the dependent variables. The two-sample T-test takes four inputs:

- **Significance level (α)**: we considered $\alpha = 0.05$, which is generally considered adequate for quantitative studies. [59, p.150]
- **Difference (delta)**: minimum difference in the value of a variable we wish to detect in the study. We considered variables pertinent to each dependent variable.
- **Statistical power (power)**: the probability the study will find a statistically significant difference. We considered power=0.9.
- **Standard deviation (s)**: We considered the values provided in the original study for each dependent variable.

We ran the two-sample T-test considering all the dependent variables of the study as we show in Table 5.4. For the *UndTime* and *ModTime* variables we chose delta=60.00 because any difference lower than 60 seconds would hardly be of practical use in game engine development work situations. For example, considering the study's architectural understanding tasks take up to 60 minutes to complete, if we showed that SyDRA allows developers to save less than 60 seconds of work time that would not be significant, given it represents less than 2% of the total work time.

Table 5.4: Two-sample T-test results for study dependent variables

Variable	Difference (delta)	Std deviation (s)	Total participants
UndTime	60.00	9.61	6
UndCorr	3.00	1.38	12
ModTime	60.00	9.74	5
ModComp	0.66	0.22	14
ModCorr	1.00	0.12	6
ModRate	0.65	0.31	11

Concerning task correctness, we wanted to detect a minimum 30% difference. Given there are seven architectural understanding tasks and two impact analysis tasks, this corresponds to delta=2 for *UndCorr* and delta=1 for *ModCorr*. The delta values for *ModComp* and *ModRate* were based on the average values provided in the original study for the "Good OO Design" scenario [8, p.520].

Finally, the largest value we obtained by using the two-sample T-test was 14 participants, which means this is the minimum number of participants we would need to detect a statistically significant difference on all dependent variables. As described in Section 5.2.7, we exceeded this value by two, totalling 16 participants divided equally between control and treatment groups.

Normality

As in the original study, we ran normality tests for data collected for the dependent variables. We used both the Kolmogorov-Smirnov test and the Shapiro-Wilks' W test. As we show in Table 5.5, the variables' p-values are all below 0.05, which means their distributions are non-normal. Therefore just like in the original study, we used a non-parametric significance test that is adequate to non-normal data, the Wilcoxon Matched Pairs test, which we explain in Section 5.2.9.

Table 5.5: P-values obtained with normality tests of the *UndTime*, *UndCorr* and *ModTime* variables

Variable	Kolmogorov-Smirnov	Shapiro-Wilks' W
UndTime	$p < 2.2 \times 10^{-16}$	p = 0.00582
UndCorr	$p = 2.8 \times 10^{-14}$	p = 0.00048
ModTime	$p < 2.2 \times 10^{-16}$	p = 0.00916
ModComp	$p = 5.2 \times 10^{-8}$	p = 0.01114
ModCorr	$p = 1.1 \times 10^{-5}$	p = 0.03985
ModRate	$p = 6.7 \times 10^{-4}$	p = 0.03805

Table 5.6: Wilcoxon Matched Pairs test results for study dependent variables

Variable	Z	Crit. $Z_{0.95}$	p-value
UndTime	20.0	5	0.8438
UndCorr	10.5	5	0.6049
ModTime	5.0	5	0.0781
ModComp	5.0	5	0.0781
ModCorr	7.0	5	0.2719
ModRate	12.0	5	0.7768

Statistical Significance

Same as in the original study, we ran the Wilcoxon Matched Pairs test to determine whether a significant difference was detected for each of the variables, as we show in Table 5.6. Column one shows the Z value of the Wilcoxon Matched Pairs test, column two shows the critical value for $\alpha = 0.05$, one-tailed, which Z has to exceed to be significant, and column six provides the p-value.

The result of the Wilcoxon Matched Pairs test exceeded the critical value for the variables *UndTime*, *UndCorr*, *ModCorr* and *ModRate*. On the other hand, it did not exceed the critical value for *ModTime* and *ModComp*. We discuss how statistical significance supports the testing of our hypotheses in Section 5.2.10.

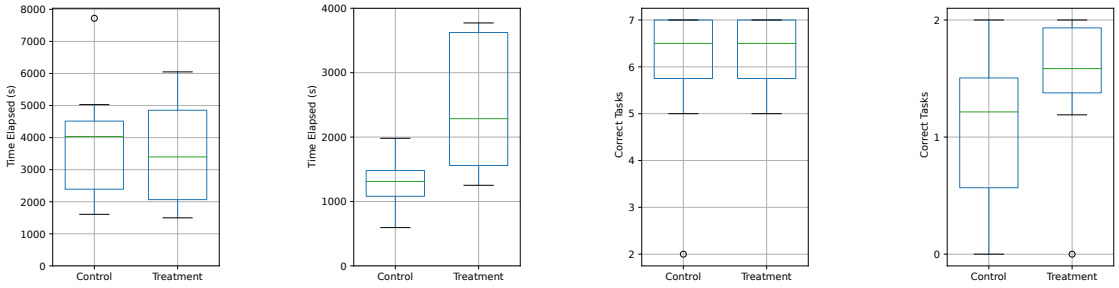
5.2.10 Results

In Table 5.7 we show a summary of the dependent variables collected from the 16 participants of the study. The columns represent the mean ($\overline{X}$), the median ($\tilde{m}$), minimum and maximum values and standard deviation (s). On average, participants took 62 minutes to complete understanding tasks and 31 minutes to complete impact analysis tasks, totalling 1 hour and 33 minutes. This completion time is higher than the 60 minutes we initially estimated based on three trial study participants, which demonstrates the importance of conducting the study with a statistically significant amount of participants.

The high variability in both completion time and correctness, as indicated by the standard deviation, reflects the participants' diverse levels of experience. By observing variables related to time

Table 5.7: Descriptive statistics for each dependent variable

Variable	$\overline{X}$	$\tilde{m}$	**min**	**max**	**s**
UndTime	3,717.73	4,035.36	1,498.74	7,721.69	1,802.58
UndCorr	6.06	6.50	2.00	7.00	1.34
ModTime	1,882.41	1,521.75	594.72	3,772.32	1,025.89
ModCorr	1.28	1.41	0.00	2.00	0.68
ModComp	1.53	1.15	0.64	3.61	0.93
ModRate	0.00	0.00	0.00	0.01	0.01



(a) *UndTime* (b) *ModTime* (c) *UndCorr* (d) *ModCorr*

Figure 5.4: Correctness and completion time distribution for both study groups

and correctness together in Figure 5.4, we can understand how the tool used by each group influenced the performance of participants. For example, the treatment group completed architectural understanding tasks faster, but both groups had the same level of task correctness. This shows that the use of Moose + Visual Studio Code decreases architectural understanding time but has no effect on architectural understanding correctness.

In impact analysis tasks, the control group completed tasks faster than the treatment group, but also less correctly. This happened because, by using exclusively Visual Studio Code, participants had more difficulty finding all outgoing and incoming *include* relationships between files and therefore ended their analysis prematurely. In contrast, participants using Moose + Visual Studio Code took longer searching for relationships in the Architectural Map, but that helped them complete impact analysis more correctly.

However, statistical significance and effect size should also be considered when interpreting these results. For example, while *UndTime*, *UndCorr* and *ModCorr* are statistically significant, *ModTime* is not, which means the average impact analysis time observed in this study cannot be

Table 5.8: Effect size for each dependent variable

Variable	$\overline{X}$ Control	$\overline{X}$ Treatment	Effect Size	Z
UndTime	4,377.78	3,545.19	0.368	20.0
UndCorr	5.87	6.25	0.303	10.5
ModTime	1,282.73	2,482.08	1.71	5.0
ModComp	1.02	2.04	1.48	5.0
ModCorr	1.09	1.46	0.55	7.0
ModRate	0.00	0.00	0.23	12.0

generalized. Moreover, we do not observe a large effect size for any of the statistically significant variables, as we show in table Table 5.8. Considering the scale defined by Kampenes et al. [60, p. 1077] for the studies in the Software Engineering domain, the largest effect size we observe for a statistically significant variable is "medium" for *ModCorr*.

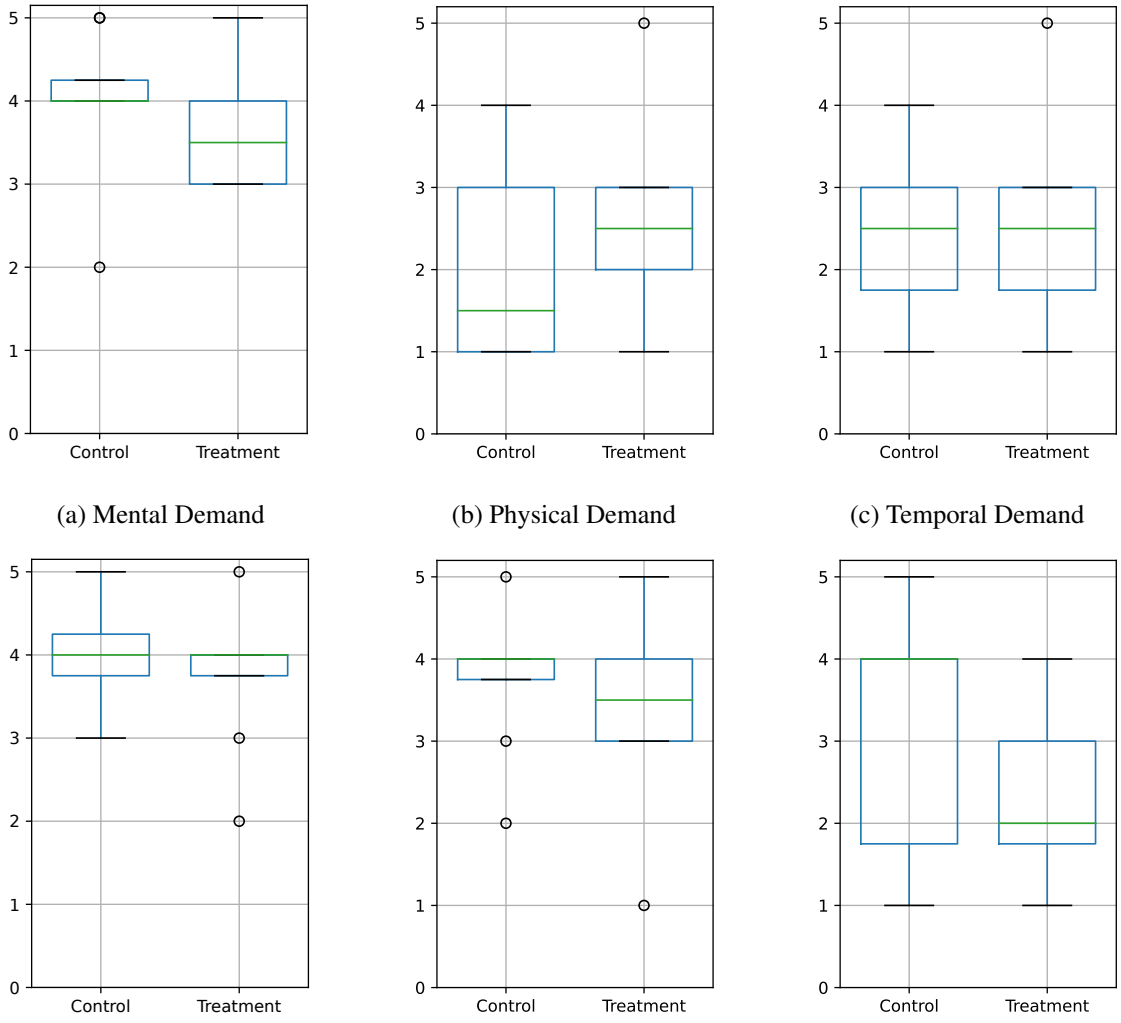
Finally, based on our observations, we accept both H_1 and H_2. The results show that the use of Moose + Visual Studio Code enables faster architectural understanding while not affecting its correctness. With respect to impact analysis, the use of Moose + Visual Studio Code enables slightly higher correctness, but no statistically significant difference in completion time.

5.2.11 Discussion

In Figure 5.5, we compare participant answers about six aspects of task load described in the NASA TLX questionnaire: mental, physical and temporal demand, perception of success, effort and frustration. As for the perception of success and temporal demand, there was no difference between groups, which is evidence that the tools the participants used did not make them feel overwhelmed.

We also observe that the participant's perception of success correlates with their professional experience. For example, video game developers reported a higher perception of success, lower mental demand and lower frustration compared to non-video game developers and student/researcher developers. We observe the same pattern when comparing novice (less than five years of professional experience) and experienced (five years of professional experience or more) developers.

The treatment group reported lower mental demand, perception of effort and frustration when compared to the control group. This is evidence that participants felt more comfortable using Moose + Visual Studio Code instead of Visual Studio Code only. In contrast, the treatment group reported



(a) Mental Demand (b) Physical Demand (c) Temporal Demand

(d) Success (e) Effort (f) Frustration

Figure 5.5: Participants' answers for NASA TLX questions

higher physical demand than the control group. However, physical demand is not a major contributor to workload in software development [57, p.671] and therefore we believe this is not evidence that using one tool demands more physical effort than using another.

Therefore, the results show that the participants perceive a lower task load when using Moose + Visual Studio Code, even though this perception is correlated to each participant's amount of professional experience and familiarity with the video game domain.

We also analysed participants' perceptions of the task, questionnaires and tools presented in the study by applying open card sorting to 17 participant comments collected via exit interview.

After the card sorting three major categories emerged: **1) unclear terminology**, **2) unclear task statements or debriefing questions** and **3) usability issues in the Architectural Map**. We show participants quotes and discuss them:

1) Unclear terminology

(a) Several participants asked us to explain in more detail the terms "functionality" and "subsystem" because they thought the meaning of these terms was "way too abstract or large". We instructed them to refer to instructional documents, where definitions were provided.

(b) Several participants admitted that they did not read the instructional documents in detail at the beginning of the study session, even though we instructed them to do so. However, they later referred to the documents when questions arose.

2) Unclear task statements or debriefing questions

(a) For Task 5, which instructed participants to open a file and list its "dependencies to other files of its subsystem", participants asked whether the term "dependencies" meant outgoing or incoming dependency. We instructed them to choose the one they believed to be the most correct. Given the original study does not provide any clarification about the direction of the dependencies, we considered both incoming and outgoing dependencies to be the correct answers for this task.

(b) A participant reported they could not answer Question 11 of the debriefing questionnaire ("How successful were you in accomplishing what you were asked to do?") because they did not know whether they actually completed the tasks correctly: "I don't know if my answers are correct".

(c) A participant reported they could not answer Question 13 of the debriefing questionnaire ("How insecure, discouraged, irritated, stressed, and annoyed were you?") because the feelings listed in the question are not mutually inclusive: "I was stressed and insecure but I wasn't discouraged and irritated".

3) Usability issues in the Architectural Map visualisation

(a) Several participants reported it was hard to find the files and folders they were looking for among thousands of others, even though hiding parts of the visualisation helped in this process: "I got really lost with the mapping at times, but understanding how to remove and add links helped a lot".

(b) A participant reported the lack of borders or contrasting colours in the rectangles representing folders made the visualisation "confusing".

(c) A participant reported being frustrated by not being able to propagate entities from the tag browser to the source code browser in the same way they did in the Architectural Map: "I know what the file [I am looking for] is, but as I can't propagate it via tags [browser]".

We reported issues concerning Architectural Map usability to the Moose development team. As of the 1st November 2023, issues described in 3b[7] and 3c[8] have been fixed. Currently, we are experimenting with drawing folders and tags in a different order to mitigate the issue described in 3a. However, we did not implement a fix as of this writing. Additionally, participant's comments on ambiguous terminology can help us to improve our task statements for future experiments, as well as other researchers conducting similar experiments.

Our study's most important contribution is showing there is a small yet statistically significant decrease in completion time for architectural understanding tasks when using Moose + Visual Studio Code. By using these tools together, developers can better understand game engine architecture while not increasing their perceived workload. Our study also provides insights into how developers use software analysis tools and how they can be improved to decrease the time and effort needed to complete architectural understanding and impact analysis tasks, especially for novice developers.

[7] https://github.com/moosetechnology/MooseIDE/issues/889
[8] https://github.com/moosetechnology/MooseIDE/issues/879

5.2.12 Threats to validity

First, understandability and maintainability are "difficult concepts to measure" [8, p. 524]. While we measured how swiftly and correctly participants completed understanding and impact analysis tasks, we did not measure whether they would be able to implement the changes in the source code swiftly and correctly as well. Also, while tasks allowed for several possible solutions, we did not verify whether the solution provided by the participant was the best or most optimized in practice, only whether it was architecturally sound.

As we explain in Section 5.2.11, we are aware that most task statements and debriefing questions allowed for multiple interpretations and that may have been the reason for the large variation in terms of task completion time and task correctness we observed. An observer effect may have also contributed to this variation, considering participants may have felt less stressed or behaved differently if they were not being observed. As we explained in Section 5.2.6, we tried to mitigate this effect by physically distancing from the participants and interacting with them only when necessary.

With regard to experimental design, while the original study used a 2x2 within-subject design, we used a 2x1 design, not within-subject. We chose this design due to limited participant availability, which made it hard or sometimes impossible to ensure all participants could participate twice in the study. As explained in the original study, by using the 2x2 within-subject design "the error variance due to differences among subjects is reduced" [8, p.518]. We are aware that by choosing the 2x1 design we risked obtaining higher error variance for all dependent variables. In future work, we intend to run another study with more participants and also use a 2x2 within-subject design.

During the study, a maturation effect may have occurred in the study due to participants learning how the tools work as the study proceeded. Some participants also had prior experience with the tools used in the study, which might have helped them complete tasks faster and more correctly than others. As stated in Section 5.2.3, we mitigated this effect by choosing a tool that is known by most developers (Visual Studio Code), as well as providing instructional documents for both tools.

An instrumentation effect may also have occurred during the study due to differences between control and treatment task descriptions. As explained in Section 5.2.5, we did our best to ensure the tasks could be completed with a similar amount of effort in both tools and that the task descriptions

were clear and stated in the same way to both groups. However, it is possible that these differences also influenced the participants' abilities to understand and complete tasks more correctly.

A confounding effect may result from our selection of game engines, analysis tools and participants. For example, Godot may not be representative of all open-source game engines in size and complexity. Also, most participants did not have prior experience with video game development and therefore do not accurately represent developers in these domains. Finally, we did not detect a large effect size for any of the statistically significant dependent variables, which is evidence that our results may not generalize to other game engines or more diverse participants.

Chapter 6

Conclusion

In this work, we proposed and implemented SyDRA, an approach for software architecture recovery. We then applied this approach to 10 open-source game engines and used the extracted architectural models to identify and understand game engine subsystem coupling patterns. Finally, we evaluated SyDRA by comparing the extracted architectural models to architectural descriptions from the game engine architecture literature. We also conducted a user study with 16 software developers to determine whether visualisations based on the extracted architectural models can help developers better perform architectural understanding and impact analysis tasks in a game engine.

In Chapter 3 we proposed SyDRA, an approach for software architecture recovery which extracts architectural models from software systems based on their source code, documentation and folder hierarchy, as well as human expertise about its subsystems.

In Chapter 4 we implemented SyDRA and applied it to 10 popular open-source game engines selected from GitHub, obtaining an architectural model of each game engine as a result. We used Moose and other tools to create visualisations of these extracted architectural models. We then used these visualisations to observe subsystem coupling patterns for each game engine, as well as emergent architectural characteristics common to all analysed game engines. We show that developers can use architectural models to understand game engine architecture in more depth.

In Chapter 5 we evaluated SyDRA by comparing the extracted architectural models to game engine architectural descriptions existing in the literature. Finally, we conducted a user study with 16 software developers to determine whether visualisations based on the extracted architectural

models can help developers better perform architectural understanding and impact analysis tasks in Godot. We show there is a small yet statistically significant decrease in completion time for architectural understanding tasks when using Moose + Visual Studio Code.

We conclude the architectural models and visualisations we produced with SyDRA help developers understand game engine architecture more swiftly and correctly than by only reading the source code. Moreover, the completeness and generalisability of the emergent architecture we created with SyDRA is corroborated by the literature and is a step towards the development of best practices of game engine design, understanding and maintenance.

6.1 Limitations and Future Work

In this section, we discuss threats to the validity of our results. First, while SyDRA applies to any software system, in this work we limited its application to game engines. In future work, we intend to apply SyDRA to diverse software families, such as databases and web browsers. In the context of video games, we intend to apply SyDRA to specific-purpose game engines and to diverse game genres (e.g., first-person shooters, point-and-click adventures, etc.). Our goal is to uncover subsystem coupling patterns and emergent architectures across various domains.

We acknowledge that the game engines we selected for analysis may not be entirely representative of all open-source game engines or the entire video game industry. We mitigated this issue by selecting game engines based on their popularity, as described in Section 4.1. We confined our analysis to C++ game engines, which may have led to the exclusion of pertinent game engines developed in other programming languages. Furthermore, we considered exclusively .h and .cpp files, omitting other types of files that could potentially provide valuable architectural insights, such as assets and scripts in languages such as Python. In future work, we plan to conduct a comprehensive study of these additional software artefacts.

Also, we acknowledge we employed the "Runtime Game Engine Architecture" [6, p. 33] in subsystem detection across all game engines, which potentially introduced a bias. As a mitigation strategy, we cross-referenced the "Runtime Game Engine Architecture" with existing literature, which corroborates its applicability within a broader context. However, the new subsystems we

found by applying SyDRA to 10 open-source game engines were not considered for subsystem detection, as described in Section 4.8.3. In future work, we intend to encompass a wider spectrum of subsystems, both obtained via SyDRA and from the literature.

The subsystem detection step of SyDRA was performed manually by the thesis' author, which may have introduced a bias in the process. To mitigate this issue, we intend to assign multiple people to work in this step and later combine their results by consensus. We also intend to explore quasi-automated approaches for subsystem detection to determine the most suitable method for game engines and other types of software.

The architectural models we generated with SyDRA currently offer insights into file dependencies, folder hierarchy and subsystems. In future work, we intend to add more architectural information to these models, enabling developers to explore software quality metrics such as cohesion and cyclomatic complexity, as well as more relationships between model entities, such as classic inheritance and method calls. We also plan to study the correlation between file/folder count and the range of functionality provided by a system, as well as how it affects its coupling.

Moreover, we are aware that SyDRA is dependent on the behaviour, metrics and visualisations provided by Moose and Gephi, and changing them could also change the results and therefore our perception of these game engine architectures. In future work, we intend to experiment with different software analysis and visualisation tools and measure to what extent they can help developers perform architectural understanding, impact analysis and testing activities.

Finally, all URLs in footnotes and references mentioned in this thesis were, to the best of our knowledge, available on the Web as of 1st November 2023. However, we are aware availability may change in the future.

Appendix A

Task Statements and Questionnaires

Table A.1: Architecture understanding tasks

#	**Control Group**	**Treatment Group**
1	Give a brief description of the Audio subsystem functionality. Write two sentences maximum.	Give a brief description of the Audio subsystem functionality. Write two sentences maximum.
2	Search for the file servers/audio/audio_effect .h and open it. Provide a short description of its functionality.	Expand the Audio subsystem, expand the "audio" folder and propagate the file servers/audio/audio_effect.h. Provide a short description of its functionality.
3	List a minimum of three functionalities provided by the Audio subsystem and provide a short description of each of them.	List a minimum of three functionalities provided by the Audio subsystem and provide a short description of each of them.
4	For each functionality you named in Audio, please name the file(s) that implement them.	For each functionality you named in Audio, please name the file(s) that implement them.
5	Open the file scene/2d/particles_2d.h and list its dependencies to other files of its subsystem.	Expand the Visual Effects subsystem and list the dependencies of the file scene/2d/particles_2d.h to other files of its subsystem.
6	Name the subsystem(s) of Godot which handle WebRTC functionality.	Name the subsystem(s) of Godot which handle WebRTC functionality.
7	Open the file scene/2d/particles_2d.h and give a short description of its functionality.	Expand the Visual Effects subsystem, propagate the file scene/2d/particles_2d.h and give a short description of its functionality.

Table A.2: Impact analysis tasks

#	Control Group	Treatment Group
8	Suppose the rich text functionality in the Front end subsystem of Godot was removed. Please mention all files which may have to be changed as a result of the removal of these functionalities.	Suppose the rich text functionality in the Front end subsystem of Godot was removed. Please mention all files which may have to be changed as a result of the removal of these functionalities.
9	The file scene/gui/video_player.cpp in the Front End subsystem provides video player functionality. Suppose a developer wants to implement the following feature: change the video playback speed (e.g. speed x2) for any video stream. Please mention all files which may have to be changed to implement this feature.	The file scene/gui/video_player.cpp in the Front End subsystem provides video player functionality. Suppose a developer wants to implement the following feature: change the video playback speed (e.g. speed x2) for any video stream. Please mention all files which may have to be changed to implement this feature.

Table A.3: Debriefing Questionnaire

#	Question	Answer Type	Required?
1	Which of these titles best describes your current job position?	List, single choice	Yes
2	How old are you?	List, single choice	No
3	What is your gender?	List, single choice	No
4	What is your country of residence?	List, single choice	No
5	How many years of software development experience do you have?	List, single choice	Yes
6	Which of the following best describes your experience with game engines?	List, single choice	Yes
7	What game engines have you used?	List, multiple choice	No
8	How mentally demanding were the tasks?	Likert (1-5)	Yes
9	How physically demanding were the tasks?	Likert (1-5)	Yes
10	How hurried or rushed was the pace of the tasks?	Likert (1-5)	Yes
11	How successful were you in accomplishing what you were asked to do?	Likert (1-5)	Yes
12	How hard did you have to work to accomplish your level of performance?	Likert (1-5)	Yes
13	How insecure, discouraged, irritated, stressed, and annoyed were you?	Likert (1-5)	Yes

Appendix B

Absolute Path to Files in Results

Repository	File or Class Name	Absolute Path
UnrealEngine	IPluginManager	./Engine/Source/Runtime/Projects/Public/Interfaces/IPluginManager.h
UnrealEngine	SkeletalRender GPUSkin	./Engine/Source/Runtime/Engine/Private/SkeletalRenderGPUSkin.h
UnrealEngine	UObject/UClass	./Engine/Source/Runtime/CoreUObject/Public/UObject/UObjectBase.h
Godot	math_funcs.h	./core/math/math_funcs.h
Godot	project_settings.h	./core/config/project_settings.h
Godot	Camera2D	./scene/2d/camera_2d.h
Godot	Camera3D	./scene/3d/camera_3d.h
Godot	Window	./scene/main/window.h
Godot	Mesh	./scene/resources/mesh_texture.h
Godot	EditorPlugin	./editor/editor_plugin.h
Godot	audio_stream_editor _plugin.h	./editor/plugins/audio_stream_editor_plugin.h
Godot	audio_stream _player.h	./scene/audio/audio_stream_player.h

Panda3d	nonlinearImager.h	./panda/src/distort/nonlinearImager.h
Panda3d	asyncTaskManager.h	./panda/src/event/asyncTaskManager.h
Panda3d	graphicsStateGuardian.h	./panda/src/display/graphicsStateGuardian.h
O3DE	Vector4.h	./Code/Framework/AzCore/AzCore/Math/Vector4.h
O3DE	AudioControlsEditorWindow.cpp	./Gems/AudioSystem/Code/Source/Editor/AudioControlsEditorWindow.cpp
O3DE	ToolsApplicationAPI.h	./Code/Framework/AzToolsFramework/AzToolsFramework/API/ToolsApplicationAPI.h
O3DE	MorphTargetEditWindow.h	./Gems/EMotionFX/Code/EMotionFX/Tools/EMotionStudio/Plugins/StandardPlugins/Source/MorphTargetsWindow/MorphTargetEditWindow.h
O3DE	SpinBox.h	./Code/Framework/AzQtComponents/AzQtComponents/Components/Widgets/SpinBox.h
O3DE	AnimKey.h	./Code/Legacy/CryCommon/AnimKey.h
FlaxEngine	VisjectGraph.h	./Source/Engine/Visject/VisjectGraph.h
FlaxEngine	AnimGraph.h	./Source/Engine/Animations/Graph/AnimGraph.h
FlaxEngine	GameCooker.cpp	./Source/Editor/Cooker/GameCooker.cpp

GamePlay3d	DebugNew.h	./gameplay/src/DebugNew.h
GamePlay3d	Logger.h	./gameplay/src/Logger.h
GamePlay3d	lua_Rectangle.h	./gameplay/src/lua/lua_Rectangle.h
GamePlay3d	Rectangle.h	./gameplay/src/Rectangle.h
Urho3d	Matrix2.h	./Source/Urho3D/Math/Matrix2.h
Urho3d	Matrix3.h	./Source/Urho3D/Math/Matrix3.h
Urho3d	Thread.cpp	./Source/Urho3D/Core/Thread.cpp
Urho3d	Mutex.cpp	./Source/Urho3D/Core/Mutex.cpp
Cocos2d-x	Vec2.cpp	./cocos/math/Vec2.cpp
Cocos2d-x	Vec3.cpp	./cocos/math/Vec3.cpp
Cocos2d-x	ccRandom.cpp	./cocos/base/ccRandom.cpp
Cocos2d-x	CCImage.h	./cocos/platform/CCImage.h
Cocos2d-x	CCTexture2D.h	./cocos/renderer/CCTexture2D.h
Cocos2d-x	CCAutoPolygon.h	./cocos/2d/CCAutoPolygon.h
Piccolo	mesh.h	./engine/source/runtime/resource/res_type/components/mesh.h
Piccolo	skeleton_data.h	./engine/source/runtime/resource/res_type/data/skeleton_data.h
Piccolo	ui_pass.cpp	./engine/source/runtime/function/render/passes/ui_pass.cpp
Piccolo	window_ui.h	./engine/source/runtime/function/ui/window_ui.h
Piccolo	render_camera.h	./engine/source/runtime/function/render/render_camera.h

Table B.1: Absolute path to files mentioned in the Section 4.7

Bibliography

[1] Jaeyong Park and Changhyeon Park, "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. 10 Years Networking the Pacific Rim, 1987-1997*, vol. 1. Victoria, BC, Canada: IEEE, 1997, pp. 101–104. [Online]. Available: http://ieeexplore.ieee.org/document/619911/

[2] L. Bishop, D. Eberly, T. Whitted, M. Finch, and M. Shantz, "Designing a PC game engine," *IEEE Computer Graphics and Applications*, vol. 18, no. 1, pp. 46–53, Feb. 1998. [Online]. Available: http://ieeexplore.ieee.org/document/637270/

[3] A. Sherrod, *Ultimate 3D game engine design & architecture*, 1st ed. Boston, Mass.: Charles River Media, 2007, oCLC: 72161915.

[4] A. Rollings and D. Morris, *Game architecture and design*, 2nd ed. Indianapolis, Ind: New Riders, 2004.

[5] A. Thorn, *Game engine design and implementation*. Sudbury, Mass: Jones & Bartlett Learning, 2010.

[6] J. Gregory, *Game engine architecture*, 3rd ed. Boca Raton: Taylor & Francis, CRC Press, 2018.

[7] S. Ducasse and D. Pollet, "Software Architecture Reconstruction: A Process-Oriented Taxonomy," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 573–591, Jul. 2009. [Online]. Available: http://ieeexplore.ieee.org/document/4815276/

[8] L. Briand, C. Bunse, and J. Daly, "A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs," *IEEE Transactions on Software Engineering*, vol. 27, no. 6, pp. 513–530, Jun. 2001. [Online]. Available: https://ieeexplore.ieee.org/document/926174/

[9] C. Politowski, F. Petrillo, G. C. Ullmann, and Y.-G. Guéhéneuc, "Game industry problems: An extensive analysis of the gray literature," *Information and Software Technology*, vol. 134, p. 106538, Jun. 2021. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950584921000252

[10] E. F. Anderson, S. Engel, P. Comninos, and L. McLoughlin, "The Case for Research in Game Engine Architecture," in *Proceedings of the 2008 Conference on Future Play: Research, Play, Share*, ser. Future Play '08. New York, NY, USA: Association for Computing Machinery, 2008, pp. 228–231, event-place: Toronto, Ontario, Canada. [Online]. Available: https://doi.org/10.1145/1496984.1497031

[11] G. C. Ullmann, Y.-G. Guéhéneuc, F. Petrillo, N. Anquetil, and C. Politowski, "An Exploratory Approach for Game Engine Architecture Recovery," in *2023 IEEE/ACM 7th International Workshop on Games and Software Engineering (GAS)*. Melbourne, Australia: IEEE, May 2023, pp. 8–15. [Online]. Available: https://ieeexplore.ieee.org/document/10190470/

[12] ——, "Visualising Game Engine Subsystem Coupling Patterns," in *Entertainment Computing – ICEC 2023*, P. Ciancarini, A. Di Iorio, H. Hlavacs, and F. Poggi, Eds. Singapore: Springer Nature Singapore, 2023, vol. 14455, pp. 263–274, series Title: Lecture Notes in Computer Science. [Online]. Available: https://link.springer.com/10.1007/978-981-99-8248-6_22

[13] M. Fowler, *Patterns of enterprise application architecture*, ser. The Addison-Wesley signature series. Boston: Addison-Wesley, 2003.

[14] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*, 3rd ed., ser. SEI series in software engineering. Upper Saddle River, NJ: Addison-Wesley, 2013.

[15] E. W. Dijkstra, *Selected writings on computing: a personal perspective*, ser. Texts and monographs in computer science. New York: Springer-Verlag, 1982.

[16] I. T. Bowman, R. C. Holt, and N. V. Brewster, "Linux as a case study: its extracted software architecture," in *Proceedings of the 21st international conference on Software engineering - ICSE '99*. Los Angeles, California, United States: ACM Press, 1999, pp. 555–563. [Online]. Available: http://portal.acm.org/citation.cfm?doid=302405.302691

[17] K. Wong, S. Tilley, H. Muller, and M.-A. Storey, "Structural redocumentation: a case study," *IEEE Software*, vol. 12, no. 1, pp. 46–54, Jan. 1995. [Online]. Available: http://ieeexplore.ieee.org/document/363166/

[18] A. Hassan and R. Holt, "A reference architecture for Web servers," in *Proceedings Seventh Working Conference on Reverse Engineering*. Brisbane, Qld., Australia: IEEE Comput. Soc, 2000, pp. 150–159. [Online]. Available: http://ieeexplore.ieee.org/document/891462/

[19] 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)*. Pune India: ACM, Feb. 2019, pp. 1–10. [Online]. Available: https://dl.acm.org/doi/10.1145/3299771.3299787

[20] A. Baabad, H. B. Zulzalil, S. Hassan, and S. B. Baharom, "Software Architecture Degradation in Open Source Software: A Systematic Literature Review," *IEEE Access*, vol. 8, pp. 173 681–173 709, 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9200327/

[21] M. Shaw and D. Garlan, *Software architecture: perspectives on an emerging discipline*. Upper Saddle River, N.J: Prentice Hall, 1996.

[22] P. Kruchten, "The 4+1 View Model of architecture," *IEEE Software*, vol. 12, no. 6, pp. 42–50, Nov. 1995. [Online]. Available: http://ieeexplore.ieee.org/document/469759/

[23] J. Keim and A. Koziolek, "Towards Consistency Checking Between Software Architecture and Informal Documentation," in *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. Hamburg, Germany: IEEE, Mar. 2019, pp. 250–253. [Online]. Available: https://ieeexplore.ieee.org/document/8712160/

[24] H. Vangheluwe and J. De Lara, "Meta-Models are models too," in *Proceedings of the Winter Simulation Conference*, vol. 1. San Diego, CA, USA: IEEE, 2002, pp. 597–605. [Online]. Available: http://ieeexplore.ieee.org/document/1172936/

[25] G. Chartrand, *Introductory graph theory*, 1st ed. New York: Dover, 1985.

[26] A. Pektaş and T. Acarman, "Deep learning for effective Android malware detection using API call graph embeddings," *Soft Computing*, vol. 24, no. 2, pp. 1027–1043, Jan. 2020. [Online]. Available: http://link.springer.com/10.1007/s00500-019-03940-5

[27] 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*, vol. 94, no. 2, pp. 755–775, Feb. 2013. [Online]. Available: http://link.springer.com/10.1007/s11192-012-0764-z

[28] M. Fowler, "Reducing coupling," *IEEE Software*, vol. 18, no. 4, pp. 102–104, Jul. 2001. [Online]. Available: http://ieeexplore.ieee.org/document/936226/

[29] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, and R. Kroeger, "Comparing Software Architecture Recovery Techniques Using Accurate Dependencies," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Florence, Italy: IEEE, May 2015, pp. 69–78. [Online]. Available: http://ieeexplore.ieee.org/document/7202951/

[30] J. Garcia, I. Ivkovic, and N. Medvidovic, "A comparative analysis of software architecture recovery techniques," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Silicon Valley, CA, USA: IEEE, Nov. 2013, pp. 486–496. [Online]. Available: http://ieeexplore.ieee.org/document/6693106/

[31] Y. Wang, P. Liu, H. Guo, H. Li, and X. Chen, "Improved Hierarchical Clustering Algorithm for Software Architecture Recovery," in *2010 International Conference on Intelligent Computing and Cognitive Informatics*. Kuala Lumpur, Malaysia: IEEE, Jun. 2010, pp. 247–250. [Online]. Available: http://ieeexplore.ieee.org/document/5565989/

[32] W. Heijstek, T. Kuhne, and M. R. Chaudron, "Experimental Analysis of Textual and Graphical Representations for Software Architecture Design," in *2011 International Symposium on Empirical Software Engineering and Measurement*. Banff, AB, Canada: IEEE, Sep. 2011, pp. 167–176. [Online]. Available: https://ieeexplore.ieee.org/document/6092565/

[33] M. Abi-Antoun and J. Aldrich, "A field study in static extraction of runtime architectures," in *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. Atlanta Georgia: ACM, Nov. 2008, pp. 22–28. [Online]. Available: https://dl.acm.org/doi/10.1145/1512475.1512481

[34] M. Consalvo and D. Staines, "Reading Ren'Py: Game Engine Affordances and Design Possibilities," *Games and Culture*, vol. 16, no. 6, pp. 762–778, Sep. 2021. [Online]. Available: http://journals.sagepub.com/doi/10.1177/1555412020973823

[35] M. Abbadi, "Taxonomy of Game Development Approaches," in *Entertainment Computing and Serious Games*, R. Dörner, S. Göbel, M. Kickmeier-Rust, M. Masuch, and

K. Zweig, Eds. Cham: Springer International Publishing, 2016, vol. 9970, pp. 119–147, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-319-46152-6_6

[36] B. Cramblitt, "Start your engines," *Computer Graphics World*, vol. 22, no. 6, p. 43, 1999. [Online]. Available: https://go-gale-com.lib-ezproxy.concordia.ca/ps/i.do?p=CPI&u=concordi_main&id=GALE|0CGW&v=2.1&it=aboutJournal

[37] V. Agrahari and S. Chimalakonda, "What's Inside Unreal Engine? - A Curious Gaze!" in *14th Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference)*. Bhubaneswar, Odisha India: ACM, Feb. 2021, pp. 1–5. [Online]. Available: https://dl.acm.org/doi/10.1145/3452383.3452404

[38] D. Maggiorini, L. A. Ripamonti, E. Zanon, A. Bujari, and C. E. Palazzi, "SMASH: A distributed game engine architecture," in *2016 IEEE Symposium on Computers and Communication (ISCC)*. Messina, Italy: IEEE, Jun. 2016, pp. 196–201. [Online]. Available: http://ieeexplore.ieee.org/document/7543739/

[39] C. Marin, M. Chover, and J. M. Sotoca, "Prototyping a game engine architecture as a multi-agent system," in *Computer Science Research Notes*. Západočeská univerzita, 2019. [Online]. Available: http://wscg.zcu.cz/wscg2019/2019-papers/!!_CSRN-2802-4.pdf

[40] P. E. Dickson, J. E. Block, G. N. Echevarria, and K. C. Keenan, "An Experience-based Comparison of Unity and Unreal for a Stand-alone 3D Game Development Course," in *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*. Bologna Italy: ACM, Jun. 2017, pp. 70–75. [Online]. Available: https://dl.acm.org/doi/10.1145/3059009.3059013

[41] P. Mishra and U. Shrawankar, "Comparison between Famous Game Engines and Eminent Games," *International Journal of Interactive Multimedia and Artificial Intelligence*, vol. 4, no. 1, p. 69, 2016. [Online]. Available: http://www.ijimai.org/journal/node/1232

[42] S. Pavkov, I. Frankovic, and N. Hoic-Bozic, "Comparison of game engines for serious games," in *2017 40th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. Opatija, Croatia: IEEE, May 2017, pp. 728–733. [Online]. Available: http://ieeexplore.ieee.org/document/7973518/

[43] E. Christopoulou and S. Xinogalos, "Overview and Comparative Analysis of Game Engines for Desktop and Mobile Devices," *International Journal of Serious Games*, vol. 4, no. 4, Dec. 2017. [Online]. Available: http://journal.seriousgamessociety.org/index.php/IJSG/article/view/194

[44] D. H. Eberly, *3D game engine design: a practical approach to real-time computer graphics*, 2nd ed. Amsterdam ; Boston: Elsevier/Morgan Kaufmann, 2007.

[45] G. C. Ullmann, C. Politowski, Y.-G. Guéhéneuc, and F. Petrillo, "Game Engine Comparative Anatomy," in *Entertainment Computing – ICEC 2022*, B. Göbl, E. van der Spek, J. Baalsrud Hauge, and R. McCall, Eds. Cham: Springer International Publishing, 2022, vol. 13477, pp. 103–111, series Title: Lecture Notes in Computer Science. [Online]. Available: https://link.springer.com/10.1007/978-3-031-20212-4_8

[46] M. Srsen and T. Orehovacki, "Developing a Game Engine in C# Programming Language," in *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*. Opatija, Croatia: IEEE, Sep. 2021, pp. 1717–1722. [Online]. Available: https://ieeexplore.ieee.org/document/9596801/

[47] J. Munro, C. Boldyreff, and A. Capiluppi, "Architectural studies of games engines — The quake series," in *2009 International IEEE Consumer Electronics Society's Games Innovations Conference*. London, UK: IEEE, Aug. 2009, pp. 246–255. [Online]. Available: http://ieeexplore.ieee.org/document/5293600/

[48] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating GitHub for engineered software projects," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3219–3253, Dec. 2017. [Online]. Available: http://link.springer.com/10.1007/s10664-017-9512-6

[49] H. Borges, A. Hora, and M. T. Valente, "Understanding the Factors That Impact the Popularity of GitHub Repositories," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Raleigh, NC, USA: IEEE, Oct. 2016, pp. 334–344. [Online]. Available: http://ieeexplore.ieee.org/document/7816479/

[50] O. Dabic, E. Aghajani, and G. Bavota, "Sampling Projects in GitHub for MSR Studies," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. Madrid, Spain: IEEE, May 2021, pp. 560–564. [Online]. Available: https://ieeexplore.ieee.org/document/9463094/

[51] C. Politowski, F. Petrillo, J. E. Montandon, M. T. Valente, and Y.-G. Guéhéneuc, "Are game engines software frameworks? A three-perspective study," *Journal of Systems and Software*, vol. 171, p. 110846, Jan. 2021. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121220302363

[52] W. K. Mizutani, V. K. Daros, and F. Kon, "Software architecture for digital game mechanics: A systematic literature review," *Entertainment Computing*, vol. 38, p. 100421, May 2021. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1875952121000185

[53] P. Petridis, I. Dunwell, D. Panzoli, S. Arnab, A. Protopsaltis, M. Hendrix, and S. De Freitas, "Game Engines Selection Framework for High-Fidelity Serious Applications," *International Journal of Interactive Worlds*, pp. 1–19, Jun. 2012. [Online]. Available: http://www.ibimapublishing.com/journals/IJIW/2012/418638/418638.html

[54] N. Anquetil, A. Etien, M. H. Houekpetodji, B. Verhaeghe, S. Ducasse, C. Toullec, F. Djareddir, J. Sudich, and M. Derras, "Modular Moose: A New Generation of Software Reverse Engineering Platform," in *Reuse in Emerging Software Engineering Practices*, S. Ben Sassi, S. Ducasse, and H. Mili, Eds. Cham: Springer International Publishing, 2020, vol. 12541, pp. 119–134, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-030-64694-3_8

[55] M. Goslin and M. Mine, "The Panda3D graphics engine," *Computer*, vol. 37, no. 10, pp. 112–114, Oct. 2004. [Online]. Available: http://ieeexplore.ieee.org/document/1350741/

[56] C. Politowski, Y.-G. Guéhéneuc, and F. Petrillo, "Towards automated video game testing: still a long way to go," in *Proceedings of the 6th International ICSE*

Workshop on Games and Software Engineering: Engineering Fun, Inspiration, and Motivation. Pittsburgh Pennsylvania: ACM, May 2022, pp. 37–43. [Online]. Available: https://dl.acm.org/doi/10.1145/3524494.3527627

[57] N. Al Madi, S. Peng, and T. Rogers, "Assessing Workload Perception in Introductory Computer Science Projects using NASA-TLX," in *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education*. Providence RI USA: ACM, Feb. 2022, pp. 668–674. [Online]. Available: https://dl.acm.org/doi/10.1145/3478431.3499406

[58] S. G. Hart, "Nasa-Task Load Index (NASA-TLX); 20 Years Later," *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, vol. 50, no. 9, pp. 904–908, Oct. 2006. [Online]. Available: http://journals.sagepub.com/doi/10.1177/154193120605000909

[59] S. Diehl, *Software Visualization*, 1st ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. [Online]. Available: http://link.springer.com/10.1007/978-3-540-46505-8

[60] V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. Sjøberg, "A systematic review of effect size in software engineering experiments," *Information and Software Technology*, vol. 49, no. 11-12, pp. 1073–1086, Nov. 2007. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950584907000195