Coordination and communication inside game engine
Abstract

This study examines the communication and coordination among different parts of game engine. As games contain multiple co-operating processes, each of which handles a small part of whole game experience, game engines must handle the management of co-operation efficiently. The issue is that while the competitive and demanding game industry has hastened the development of game engines, the academic research has been lagging behind.

This study aims to bridge the gap between industrial know-how and academic research by studying how game engines handle managing the processes and data that they require. The study uses coordination paradigm as the research viewpoint and focuses on finding the coordination model used in game engines. The target of the study is to understand how the coordination model is implemented in game engines.

The study followed design science research, whereby both industrial know-how and academic literature were used in finding the coordination model and implementation method of it. The coordination model was first searched among the existing models, but as those were proven inefficient in describing the overall theory, a new architectural theory was built. This new theory, called “Communication-oriented game engine architecture”, was then analysed against communication mechanisms used in open source game engines for verification.

The result of the analysis indicates that the theory proposed in this thesis explains the common features among analysed game engines. Since the proposed architecture is based on existing and well-known event-based communication mechanism, the theory behind proposed architecture aids in understanding the design of game engines in respect to the design of other software.

Keywords
game engine, software architecture, domain-specific architecture, control structures, gaming
Foreword

Games, be it board games or computer games, have always interested me. As I have witnessed the evolution of gaming from simple toys to complex social platforms, I have learnt to appreciate both technical and artistic merits of games. Especially now, during the later part of my studies, I have noticed the meaning and value of those things that makes games unique among software. They offer new insight to software engineering and I hope that this thesis will encourage others to dive in to explore games as software.

I would like to thank following people for supporting me during the development of this thesis: University Lecturer Jouni Lappalainen who guided me as my supervisor; my friends who were always ready to talk about games and game engines; my family for supporting and encouraging me forward; and of course my fiancée for taking care of me when I studied long hours among articles, books, and source code repositories.

Tuomas Kukkamaa

Oulu, Thursday, May 29, 2014.
Contents

Abstract ................................................................................................................................. 2
Foreword .............................................................................................................................. 3
Contents ............................................................................................................................... 4
1. Introduction ..................................................................................................................... 5
   1.1 Research topic and questions .............................................................................. 6
   1.2 Research methodology ....................................................................................... 7
   1.3 Structure of the thesis ......................................................................................... 9
2. Games as software ......................................................................................................... 10
   2.1 Games as data-driven software .......................................................................... 11
      2.1.1 Processes inside game ................................................................................. 12
      2.1.2 Data inside game ....................................................................................... 15
   2.2 Game engines as managers .................................................................................. 17
      2.2.1 Process management .................................................................................... 18
      2.2.2 Data management ....................................................................................... 20
3. Communication inside game .......................................................................................... 22
   3.1 Communication mechanisms ................................................................................ 23
   3.2 Architectural solutions ......................................................................................... 25
4. Communication architecture of game engine ................................................................ 35
   4.1 Events as communication mechanism .................................................................. 36
   4.2 Depiction of the architectural solution .................................................................. 37
5. Analysis of existing game engines ............................................................................... 41
   5.1 Blender .................................................................................................................. 43
   5.2 Delta 3D ................................................................................................................ 45
   5.3 Gameplay3D .......................................................................................................... 48
   5.4 Polyclode .............................................................................................................. 49
   5.5 Comparison between game engines and the proposed architectural solution ......... 51
6. Discussion ....................................................................................................................... 55
   6.1 On the verification of the architectural solution and theory .................................. 55
   6.2 Implications for game engine theory .................................................................... 58
   6.3 Implications for game engine industry ................................................................. 61
7. Conclusion ...................................................................................................................... 63
References .......................................................................................................................... 65
Appendix A. Structural and functional diagrams of analysed game engines .......... 70
1. Introduction

Computer games (hereinafter referred as games) present a great challenge for both developers and scholars: While they can be treated as usual software and designed under common software requirements, they are often required to handle many computationally-challenging tasks that are not present in other software. These tasks, such as simulating independent parts of the game world and rendering that game world efficiently to user, must be often designed and programmed under strict schedule limits that are imposed by the very competitive and demanding video game industry. This has led to situation where the best practises of industry have been developed a lot while scientific research has been lacking.

This does not, however, mean that there hasn't been any scientific research on games. Many game related topics have been studied from variety of viewpoints during last decades. Some of the most common research topics have been usage of games in different purposes (for example, as a simulation platform (Zyda, 2005)), improving a specific technological or mechanical section of game (for example, improving usage of database in games (Demers, Gehrke, Koch, Sowell & White, 2009) and improving the artificial intelligence of a game (Spronck, Ponsen, Sprinkhuizen-Kuyper & Postma, 2006)). The problem is that while research on games has increased (Ampatzoglou & Stamelos, 2010), most of the research focuses on studying how games, as an abstract entity, work in different situations (“usage”) or how certain low-level functions can be implemented (“implementation”).

This imbalance can be especially noted in the research done on game engines. Game engines form the “heart” of modern games and since early 1990s, when the first game engines were constructed, have developed into large and complex pieces of software. Game engines function as a link between different major parts of the games, driving their actions and facilitating their co-operation and thus form an important base architecture for the games. These roles and the related complexity would make them prime targets for studies on architecture and communication but currently, the academic research is mostly focusing on how to perform simulations and mathematically complex issues efficiently (Anderson, Engel, McLoughlin & Comninos, 2008). This research focus has also shown up in the industry-based literature as, for example, the books by industry veterans such as McShaffry (2009), Rabin (2009) and Zerbst and Dueval (2004) all focus on telling how the specific parts of the game engine can be implemented without discussing the overall architecture and design reasoning behind it.

These circumstances have led to a situation where the modern game engines have seen little focus from the academic research and thus their developments have not been analysed as a part of larger software engineering literature. While there has been some studies about the theory behind and implementation of game engines, the theory of game engines is currently seen as lacking behind the development (Anderson et al, 2008). This is something that this study is aimed to answer.
1.1 Research topic and questions

One particularly interesting topic in game engines is the way how they coordinate all the processes inside game to provide efficient game experience for the user. Efficient coordination requires also communication and this is required not just between the game engine and the processes that simulate the gameplay but also between the processes and the data they display. In software engineering literature, one viewpoint to this topic is provided by “coordination paradigm” (Papadopoulos & Arbab, 1998) which separates the processes inside software into two groups: computing processes which manipulate the data and coordination processes which handle the communication and cooperation between computing processes. From this point of view, the topic of coordinating the processes inside game can be researched by studying how the coordination processes inside game engine are handled. In the coordination paradigm, the specific way how management of co-operating processes is called coordination model (Ciancarini, 1996; Papadopoulus, Arbab, 1998).

The importance of studying the coordination occurring inside game engine derive from the current trends in game industry: As the consumers request larger and more dynamic game worlds with better graphics, the core functionality which allows these improvements, i.e. the coordination and communication functionality provided by the game engine, has to be improved all the time. This has caused the game engines to leap forward while their improvements have not been studied in the software engineering research. By studying the modern game engines, the know-how from the game industry can be brought up, analysed and hopefully further developed by the research while at the same time, the improvements of game engines could be potentially brought to other software too.

In this study, the issue of coordination and communication inside game engines is studied from the viewpoint provided by coordination paradigm. The aim is to study how game engines function as a communication channel between game data and processes. By using terminology of coordination paradigm, the aim of this study can be summarized as finding the coordination model used in games. The focus of this study is especially on studying how the coordination model in game engines is implemented at architectural level. This research aim of this study can be summarized as main research question and its two supporting research questions which are shown below:

- **RQ1.** How does the game engine function as a communication channel between data and processes?
- **RQ1.1.** What kind of functionality and architecture game engines contain to provide necessary coordination?
- **RQ1.2.** What kind of functionality and architecture game engines contain to provide necessary communication?

While the research questions shown above direct the research, they are broad topics especially when considering the modern game engines and the functionalities that they offer. For example, taking into account networking of games would make the research too large in scope. For this reason, networking is not considered at all in this study and each game engine is studied as independent software running in single computer. Also, some game engines support multiple players playing the same game in same machine.
As this support for multiplayer features may have an effect on the communication and coordination occurring in game engines, that feature is not taken into account in this study.

1.2 Research methodology

The research method used in this study was design science research, where the aim was to improve the existing theory of game engine's role in communication by combining existing architectural theories with the knowledge from the industry. The architectural theories used in this study were different proposed models of internal structures of game engines. These models provided a subset of models that showcased the academic knowledge of game engines. To keep the focus of the study in practical level, knowledge from game industry was also used in the study to provide practical background for the study.

Design science research has its origins in research of information systems. The focus of the design science research is on building novel or innovative artifacts, be it software, algorithms or coding languages, etc., to extend the knowledge by analysing the performance of new artifact (Vaishnavi & Kuechler, 2012). The conceptual framework of the method (Figure 1) is based on the feedback and information cycles between two knowledge repositories, knowledge base and environment, and the actual research process (Hevner, March, Park & Ram, 2004). The core idea behind the framework is to design and build a new artifact using the knowledge from both of the two repositories. By analysing the design process and the new artifact against the two knowledge repositories, one can gain new valuable information to add to both repositories.

In this study, both academic knowledge of game engines and practical know-how from game industry were used to provide both the knowledge base and the environment for this study. In practise, this meant that the analysis of academic knowledge and industrial know-how provided the research questions which were then answered by information sought from both knowledge repositories. The answers to the research questions were then searched by reviewing current knowledge and creating a new artifact, an architectural model in this case, to fill the deficiency found in the academic knowledge. The new artifact created in this study was a new communication-oriented architectural model of game engines and by comparing it to other similar artefacts, in this case the architectures of existing game engines, new insight to both academic and industrial knowledge of game engines was gained.

The study was conducted in three phases. The first phase was a literature review where the knowledge from both academic studies and game industry was gathered and presented. The aim of the literature review was to find sound theoretical background which could be used to understand key terms and concepts behind game engines. In second part of the research, analysis on different communication models was conducted. This analysis was based on the research question and the knowledge sought from literature in first part. Last part of the second phase of the study was using the results of analysis in forming of the new, communication-oriented architectural model. Lastly, in the third phase, the new model was evaluated by examining how well it fits with existing game engines. The results of this evaluation were analysed to give a base for future studies.
Narrative literature review was used in the first phase, where the knowledge from both academic studies and game industry was gathered and presented. Narrative literature review was chosen as it can be used to generate a broader understanding of certain topic by together linking findings from different topics (Baumeister & Leary, 1997). The main difference between narrative literature review and other literature review approaches, such as integrated and systematic reviews, is that while other methods utilize quantitative methods to prove the strength of hypothesis, in narrative literature reviews the strength is proven by how strongly the results of different studies correlate with each other (Baumeister & Leary, 1997). As the study of game engines, as well as other technologies behind games, is still taking it first steps, it is suitable that the literature review method allows us to take in account the findings from other relevant disciplines.

In second part of the research, analysis of existing architectural models of game engines from the viewpoints of research questions and the established knowledge was performed. The results of analysis were then used in forming the new, communication-oriented architectural model. This was the main develop/build task of the research (see Figure 1). Lastly, in the third phase, the new model was evaluated by examining how similar functionalities presented in the model were accomplished in existing game engines. The results of this evaluation was then analysed to give a base for future studies. The design evaluation method was architecture analysis where the constructed model was compared to existing game engines, and their similarities and differences are further analysed. This concluded the justify/evaluate task of the research (see Figure 1).
1.3 Structure of the thesis

The structure of the thesis follows the guidelines of scientific articles while also following the narrative created by the research plan.

Chapter 2 contains the result of the literature review. The results provide introduction to games as software and then delve deeper into describing the components and inner working of game engines. In later part of the chapter the multiple communication related roles that game engine have are especially discussed.

In Chapter 3, the analysis of existing communication solutions ranging from specific mechanisms inside game to architectural models of games is performed.

In Chapter 4, the new architectural model is introduced and described in conceptual level. Also, an example of component deployment is shown and described to showcase how the model works.

In Chapter 5, the new model is evaluated against the architectures of existing game engines. The aims of this evaluation are to see how much the model follows the architecture of modern game engines and to analyse what are the causes and effects of similarities and differences found in the evaluation.

In Chapter 6, selected findings from the analysis done in the previous chapter are discussed further and the implications of this study to academic research and game industry are proposed.

And lastly, Chapter 7 concludes the thesis by providing summary and conclusions of this study.
2. Games as software

When following classical glossary, computer game can be defined as “game that is carried out with the help of a computer program” (Smed & Hakonen, 2003). This means that all the components of the game, e.g. player(s), rules and goals, are incorporated to computer software which then simulates the game to the players by using electronic devices (Esposito, 2005). When following terminology of the computer game industry, this simulation can be explained in following terms: computer game consists of simulated world, e.g. game world, which is inhabited by game objects that follow game logic to let the user, e.g. player, play the game. (Gregory, 2009; Rabin, 2010). Some of the game objects may be controlled by the player either directly or indirectly.

While description above defines computer games by their use, it does not delve deeper to describe games from technical point of view. One technical description of games was given by Gregory (2009), who defined computer games in theoretical terms as soft real-time interactive agent-based computer simulation. Each term of this definition can be further examined in following way:

*Soft real-time* indicates that game, as interactive multimedia application, has time constrains to execute necessary processes, and to showcase the results to user before certain “deadline” (Joselli, Zamith, Clua, Montenegro, Leal-Toledo, Valente & Feijó, 2010). These deadlines are most strict for processes dealing with user interaction, but in some games actual simulation of game world, i.e, updating the states of the objects inside game world, also has to perform under the given time limit (Valente, Conci & Feijó, 2005). If simulation would lag behind the commands made by user, such as throwing a ball, moving forward or opening a door, the immersion would suffer. However, if the lag caused by the simulation is under acceptable limit and output of simulation and user interaction can be showcased to user in completed way, deadlines can be sometimes lessened to allow simulation to occur. This indicates that games follow “soft deadlines” and thus have soft real-time requirements.

*Interactivity* comes from the fact that games need the user inputs to work. Since the player is a necessary element of a game, user is also a necessary element of a computer game (Smed & Hakonen, 2003). This marks the importance of user, and separates games from model simulations, videos, or other purely watchable programs.

*Agent-based* means that the game world is inhabited by individual agents, which work autonomously and, if needed, interact with other agents to generate simulated, living game world experience (Macal & North, 2010). The term 'agent' comes historically from simulation literature and other terms like game entities and objects have been used in the game industry to mean conceptually similar things (Gregory, 2009; Rabin, 2010). However, the agents used in games should not be mixed up with the agents from multi-agent system research as there are conceptual and technical differences between the terms (Gemrot, Brom & Pich, 2011).
Computer simulation implies that video games are inherently programs which simulate and thus model some part of existing or imaginary world (Narayanasamy, Wong, Fung & Rai, 2006). The simulated world can also be very abstract, say chess board in chess game but still, it can be defined as a game world (Smed & Hakonen, 2003).

While the mentioned definitions describe games themselves, they leave some uncertainty about games' function and concept. For that reason, it is good to shortly go through few other closely related application types to see how they differ from games in respect of these features. One of these related application types is simulation applications, which seem to follow same basic functionality, as in, they also consist of simulated world where different independent actors follow the set of rules to simulate the situation. One critical difference that some researchers have pointed out between simulation applications and games is the lack of goal-orientation and the following lack of gameplay in simulations. While this topic is still under debate (Esposito, 2005), the consensus seems to be that these two application types are different (Narayanasamy et al, 2006; Smed & Hakonen, 2003).

Another closely similar application type that resembles games is virtual reality applications. Virtual reality applications are a subset of simulation applications and they have been described as “...simulation in which computer graphics are used to create realistic looking world” (Burdea & Coiffet, 1994). While this description is similar to description of simulation application, in virtual reality applications interaction channels between user and simulated world are taken to the next level as specific software and hardware are used to engulf the user to the experience. Often this usage of highly tuned interaction channels and purpose-built hardware is one of the main interest areas of virtual reality research. While the link between simulation and games can be seen as conceptual level similarity (as in game world vs simulated world, game logic vs simulation rules and game objects vs actors), the similarities between games and virtual reality applications are mostly technical. In both application types, interaction and immersion are key elements of the application (Burdea & Coiffet, 1994; Steuer, 1992; Zyda, 2005) and similar technologies have been used in both application types to strengthen those elements. For example, graphical and haptic interfaces as well as realistic real-time 3D graphics are all widely used technologies in both applications. Thus, there is a clear technical synergy between these two application types although, like with simulation and games, the division between the two exists as the virtual reality applications do not focus on the gameplay aspect (Narayanasamy et al, 2006).

2.1 Games as data-driven software

While there exists different, complete definitions of what games are, the question of how games work lacks definitive answers. One reason for this is the ever-changing nature of the game industry: games have changed internally a lot since the early pioneering entertainment applications (Bishop, Eberly, Whitted, Finc & Shantz, 1998; Lewis & Jacobson, 2002). One of these changes is the movement towards data-driven architecture (Gregory, 2009), where logic of the game and content of the game are separated further and further from their original hard-coded roots (BinSubaih, Maddock & Romano, 2007). This movement has brought forward a new view of the structure of modern game and how it is compared to other applications. Data-driven architecture has been widely adopted in the game industry and as it is currently considered as one of the
key features of modern games, this section of the study focuses on determining what kind of effect it has had on the architecture and communication inside game.

Since there exists few different definitions of data-driven architecture, the usage of the term is not always consistent. This is because the term data-driven has its root in engineering theories (see, e.g., Treleaven, Brownbridge & Hopkins, 1982), but the term is used to describe larger approach, not a single paradigm as it is the case in engineering and in information processing science. In order to avoid the confusion, in this paper the term is understood using the definition given by literature from game industry: in data-driven architecture, the behaviour of the system is defined by data that is provided to it (Gregory, 2009; Rabin, 2000). This can be seen as opposite from hard-coded architecture where the behaviour stays always same as defined in the code.

The key idea behind data-driven architecture is the clearer separation between the data and the process using the data: processes inside a program represent the methods which can be used by the data to adapt the behaviour of the system and therefore change the output of the system. As the data can be modified by processes and in turn, be used again to adapt the behaviour of the system, data-driven architecture can be seen as similar to dataflow architecture, where each actor can consume information and then produces new information for other actors to consume (Lee & Parks, 1995). This complex interaction between different processes and sets of data makes it important to understand the relationships between them. Thus, to understand more how the concept of data-driven architecture works in games, one must delve deeper to see what the key processes of the games are and what kind of data they use.

In next two sub-sections, common processes of the games and the corresponding data will be detailed. Also, short description of the ways to implement both processes and data will be discussed.

2.1.1 Processes inside game

Like in other applications, in games, different calculations can be structured into processes. Processes deal with data to provide meaningful result allowing user to actually use the application. As is usual in software, games have multiple different processes happening simultaneously, but in the context of this study, it is not meaningful to understand all of them. To briefly describe inner workings of games, following list from Gregory (2009) provides some idea of what kind of processes occur in games:

- Asset management: Games often require extensive management of game assets, externally saved elements that are loaded at run-time to provide content of the game (Andersson, 2011). These assets can be, for example, graphical elements, map layouts and sound clips, which are used at least once but often many times during the usage of the game application. Often to save memory, single asset is used by instancing copies of it when necessary. Both loading assets and instancing them are often happening all the time during gameplay among other asset management related processes.
• Input gathering: Interactivity requires games to collect and analyse the input given by the player or players nearly all the time. The processes handling these are either responsible for gathering input from local users via devices like keyboards, joysticks and mouses or from remote players via network connections.

• Simulation: The game world inside games requires some form of simulation, that is, applying different sets of rules imposed on the game world and its game objects. This means that both physics simulation and artificial intelligence simulation can be considered simulation processes (Joselli et al, 2010).

• Presentation: As games are visual media, rendering is the main presentation process in any game. However, it is not the only way to present feedback to the user. Playing sounds and music and giving force feedback via game controllers are also commonly used techniques to present the state of the game world to the player and to increase the immersiveness of the gaming experience. All of these presentation techniques require complex processes to handle them.

Most vital process to game experience is the combination of input handling, simulation, and presentation processes which is often called as *game loop* in the industry (Valente, Conci & Feijó, 2005; Gregory, 2009). This loop acts as main controlling logic of the whole game, orchestrating other processes to work in unison (see Figure 2). The concept of game loop comes from virtual reality research where a very similar concept of *simulation loop* has existed for a long time (Burdea & Coiffet, 1994; Joselli et al, 2010). The main difference between these two concepts is that while simulation loop takes in consideration sensory data, like haptic devices surrounding the user or sensors measuring the real live data like a gas flow of a pipe, game loop is more focused on the internal game world and its actions.

![Figure 2. An example of simple game loop (Joselli et al, 2010)]
Figure 2 gives us an example of simple game loop where each process performs tasks in a linear fashion. This kind of structure of game loop was more prevalent in earlier games as it was simple to implement and to work with (Rollings & Morris, 2003; McShaffy, 2009). However, because the computer architecture have evolved and games have become more complex, new ways to implement game loop has surfaced. For example, due to wide-spread usage of multi-core computers, modern games have increased separation between rendering, simulation, and input gathering processes to allow them run more independently. Example of this can be seen in Figure 3. This kind of structure allows games to use available resources, like separate, specialized processing units such as GPUs and PPUs, more efficiently.

In modern games, managing the game loop and other processes of the game is the responsibility of game engine. A game engine can be described as a collection of modules which are controlled by main managing process to operate in conjunction to create wanted game experience (Lewis & Jacobson, 2002). These modules are often called as subsystems and usually each of them handle designated kind of processes. For example, a game engine always has a rendering engine to handle the outputs of the visual data to user and also often an audio engine to handle audio-related processes and in some games, also a physics engine to take care of objects physical movement in the game world (Gregory, 2009). The role of the subsystem is similar to the role of middleware, given both allow the developers of the larger system to manage the overall complexity by using a premade component specialized on doing that task (Campbell, Coulson & Kounavis, 1999).
Above described modular architecture inside game engine allows game engine to specialize in handling high-level management of the game. This is necessary since the dataflow between processes in usual game is complex (see Figure 4) and managing the whole dataflow, especially handling concurrency and timeliness of processes, requires care and balance. Therefore, when individual processes are handled by their own, specialized modules and game engine is built around the co-operation among these specialized modules, developers can concentrate more on implementing the features that make the game more unique without worrying implementation details of each process (Demers et al, 2009).

2.1.2 Data inside game

All processes mentioned in previous sub-section require data. However, as there are various processes inside a game engine, data those processes require varies. In case of games, most data can be categorized either as game content or as game logic (Wünsche, Kot, Gits, Amor, Hoskin & Grundy, 2005). The term game content refers the data that is used as a part of output and which is often saved as its own media, like how a certain audio file can be played as background music or how image can be used as a texture of 3D model. The other category, game logic, is more dynamic: data belonging to game logic category is used as lightweight and modifiable “ruleset” that allows developers and designers to craft and fine-tune the dynamic aspects of games. Good example of this kind of data is an artificial intelligent script, which can be used to dictate how certain computer-controlled character acts in a game.

The game content has been developing almost as fast as the technology behind games has evolved. Although at the start there was need to use game- and platform-specific methods to save the content, games use a number of standardized methods and formats nowadays. This means that the content is portable, i.e. usable in other games and softwares too (BinSubaih, Maddock & Romano, 2007). However, while the way how the content has been saved has changed, the way the content is used has stayed mostly
same. This is due to its static nature: any kind of content a game uses is just meant to save static data, i.e. work as a resource or as an asset, which can be used when needed (Gregory, 2009).

While game content is largely meant to represent static, representative resources, game logic is meant to facilitate more dynamic aspects of the game. In short, game logic is used in many situations inside the game where the saved information cannot be saved, for whatever reason, as static media. At first it may seem like this would not occur that often in games but in practise it is rather typical. For example when games have configuration settings, they have characters that respond to the player's action or when the attributes of game objects must be dynamically balanced, the algorithms behind these gameplay features are often saved as own separate game asset (White, Koch, Gehrke & Demers, 2009). The actual algorithm is usually saved by writing its logic in a script language (Andersson, 2011).

Isolation of game logic as an individual asset is a comparatively recent event in the evolution of games. While the earliest known attempts were relatively successful, for example, popular SCUMM game engine, which was developed for graphical adventure games, used extensively scripts (Rodman, 1999, as cited in Moreno-Ger, Sierra, Martínez-Ortiz & Fernández-Manjón, 2007, p. 4), separating game logic became more common later on. Study made in 2007 by BinSubaih, Maddock and Romano revealed that close to 75 % of the studied game engines supported scripting, and this trend was believed to continue.

<table>
<thead>
<tr>
<th>Game Scripting Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initialisation Scripts (ST1)</strong></td>
</tr>
<tr>
<td>ST2a</td>
</tr>
<tr>
<td>Event Handler Scripts</td>
</tr>
</tbody>
</table>

**Figure 5.** Classification of scripting systems for games (Anderson, 2011)

As can be seen from Figure 5, scripting can be used in many places in a game. The figure represents a classification of scripting systems which can be used to perform certain tasks and as one can notice, the variety between these systems is large. The three large groups (marked as ST1, ST2 and ST3 in Figure 5) display the main types of the scripting systems: initialization scripts, which are used in configuring and setting up the software; trigger-only induced scripts, that are used in during actual gameplay to implement dynamic gameplay features; traditional-program-like scripts, which function as separately built and ran mini-programs that are focused on simple, well-defined tasks (Anderson, 2011). It should be noted that this classification is based on
the usage of the system, not on the actual domain of use. Still, this classification can help us in understanding the multiple roles which script systems can fulfill in games.

As the descriptions of the major types hint, game logic should not be treated just as a ruleset of the game, but rather as a representation of individual mechanics. Of course, in this case care must be taken when considering individual logic assets and their place in the concept. For example, if a script that controls presentation process (i.e., a specialized effect that is created in runtime), it can be questioned if it actually can be considered as a part of single game since it does not have a clear effect on the gameplay itself. Although it is important to analyse this kind of questions when discussing further about the role of game logic, this is not a topic in this study. Instead, in the later parts of the study, all scripts are considered as game logic of the game, be it part of any above mentioned three types, as long as it can be clearly separated as an asset, i.e. reusable and modifiable part of the whole game.

2.2 Game engines as managers

In earlier parts of the chapter the function and structure of games were discussed from static point of view, as in, how games are constructed by following the data-driven architecture. While this discussion aids in understanding the static view of the software, it is also important to study how software works during runtime. These different ways to understand the structure of software are called “architecture views”, and they are used to represent different kinds of aspects of the system (ISO, 2011). When the terms “static view” and “runtime view” are used to describe a system, these terms refer to logical and process views of a system. The significant difference between these two views is that while logical view is a tool for designers to plan the overall functionality and structure of system (the data-driven architecture of game in this case), process view is meant for aiding building a sophisticated and efficiently running system (Kruchten, 1995).

In process view, the focus is on individual processes, their overall management, their communication methods and, more broadly speaking, how those processes work together to address the main logical and conceptual abstractions designed in the logical view (Kruchten, 1995). As previously described, the overarching management task inside games is handled by the game engine while some major processes are given to different subsystems to handle. Following the concepts in data-driven architecture, game engine therefore manages both its own process and the execution of all the other processes. Since game engine also often manages the data access and delivery (Gregory, 2009), game engine can be seen as managing component that does not itself simulate the actual gameplay but provides support to all the other processes and components which handle the simulation.

When the number of different processes rises, the management of their corresponding components becomes increasingly complex task. As Figure 6 shows, games usually consist of large number of processes which are very interconnected with each other. Thus, the implementation of a game engine is important to study, as it plays a big role in the playability of the game.
Game engine has two managing tasks in a typical game: process management and data management. Process management deals with starting, running and stopping processes of the game in correct times while data management handles loading, storing and distributing of data processes require to function. Since both of these tasks are themselves complex and they need to synchronized effectively, game engine handles these major tasks. As these tasks are important to understand for the sake of overall performance, in next sub-sections they are described in more detail, and their common implementation methods and corresponding issues are discussed.

2.2.1 Process management

Because all processes inside game are governed by some specific component (either by game engine or some subsystem), process management in games mostly refers to the management of individual components. These components come in various sizes: some can be own subsystem with many intricate tasks to be performed, and some may be a part of subsystem that only do certain specific processes only once during whole gameplay session. Also, the location of these components is not always set: although majority of components are inside game, some processes can be ran as external services or as middleware software (Gemrot, Brom & Plch, 2011). Eventhought the variety of the components is very high, at their base level all components are similar; they offer similar functionalities for the game engine in order to allow the game engine to execute the processes handled by the component while they all require similar services from game engine to function. In essence, it is this similarity of their interface which allows game engine to orchestrate the whole complex task of running a game.

Figure 6. Major areas of functionality inside typical modern game (Blow, 2004)
Inside a typical game, there are two distinct manageable sides of processes. These are management of set-up and shut-down of components and execution management (Gregory, 2009). These sides encompass the whole life cycle of a game, and provide the highest-most control over whole game. However, neither of these manageable sides of process takes into account what the processes themselves are doing, they only affect managing the overall execution of the said processes. This kind of structure where game engine sees processes as similar executable computations supports the data-driven architecture that games commonly use.

Management of set-up and shut-down of components means that the game engine controls initialization, configuration and shutting down of different processes, i.e., the game engine controls the components and processes overall lifecycle. Especially initialization and configuration of components requires attention as each of the components are interconnected throught the data they require (Gregory, 2009). This is due to the data-driven architecture in which processes use data processed throught other processes: for each of the component to work, they need to linked by some means to the processes that generate required data. The actual tasks performed during initialization and configuration phases can be either hard-coded or saved as external asset (Andersson, 2011).

Managing of the game loop means that the game engine, which contains the game loop, handles the execution of the loop. As mentioned previously, during single cycle of game loop, all necessary processes are instructed to take one step forward in simulation. In the process view, this can be seen as game loop process owned by game engine, ordering all the other processes to advance a step forward (Gregory, 2009). While this may seem as like simple piece of code, in reality there are many things that can make the execution and management of the game loop challenging. Main issue that complicates the design is the time.

One way time can be seen as a challenge is to examine how it affects executing of a “step” inside a process. Since games are real-time software, which manages own simulated world inside itself, there exists different kinds of temporal frames inside games. Processes which take inputs and provide outputs often are required to work, in real-time in order to provide good interactive feed-back to the user while processes which work only on simulated data, i.e., physics simulation process, only focus on the simulated time (Zagal & Matteus, 2010). Thus, executing a single “step” is not as straight-forward as just ordering the corresponding component to take a single step but rather it is an intricate operation where the time itself has to be managed (Gregory, 2009).

Another set of challenges stem from synchronization between different processes. As game has to read input and provide output at relatively static intervals, the processes have to be managed to perform in cohesion but still to be able to adapt for subtle (and some times radical) changes in the performance. These soft real-time requirements are caused by demand to provide good interactive experience to the user all the time (Joselli et al, 2010). The challenge is that not all processes function at the same time but the output has to be delivered and input gathered in stable rate. Answer to this challenge are frame rate control algorithms which allow game engine to synchronize individual processes (Li & He, 2005).
2.2.2 Data management

As described in Chapter 2.1.2, games contain both game content and game logic, both of which are saved as reusable assets. These assets are not, however, directly usable by the subsystems which require them: assets need to be processed from raw assets into complete game objects before use. Single game object may be composed out of multiple different assets which also need to be loaded (for example, a torch on the wall has structure, i.e. mesh, texture, and flame effect with accompanying scripts) and multiple game objects may share same assets (all torches in a house should share the same individual mesh, texture and effect asset to improve the memory usage). This complexity of game objects and the assets that are used to compose the game object causes the overall management of game objects to be increasingly complex task.

![Diagram showing the position of runtime resource manager (shortened as resource manager in this diagram) between assets and rest of the game engine.](Anderson, 2011)

The management of game objects can be implemented in few different ways but no matter how the implementation method, the main task of the management system is same: the system has to load assets to runtime memory and then compile them into complete usable game objects. The system performing this task is usually called runtime resource manager or object system, and it can be seen either as individual system (see Figure 7) or, as is the case in this study, as a part of the game engine itself (Gregory, 2009; Doherty, 2003). In addition to handling the loading of assets, it often also handles the access to individual game objects, basically functioning as a game object database that can be queried by subsystems (Duran, 2003). In this way it works as unified interface through which systems can change the game world made up by different game objects (Gregory, 2009).

Game objects themselves serve an important role in the data management. They combine different assets into complete, usable “units” that are easier to handle in both conceptual and practical levels. Using the previously described “Torch” object as example, single game object can combine different assets into a coherent single “torch”, which can be placed, moved and interacted with. Sometimes single game objects can be
used as “elements” of larger game object. For example, a game object “Turret” is integral part of game object “Tank” in a war simulation game.

Constructing these objects is not straight forward, however. Due to the resource constrains and vast number of assets and individual game objects required in implementing modern games, the implementation method of game objects becomes important. However, while there are different ways to implement game objects, as can be seen in next paragraphs, game objects themselves show up to their users, i.e. to the subsystems which required them, as normal objects with only minor differences (Doherty, 2003; Gregory, 2009). Thus the following discussion on implementation methods mostly affects the structure of game objects, not the function. Still, to fully understand the challenges of designing the communication among the objects, this is vital information.

Two most prevalent game object implementation methods are object-centric or component-centric architectures (Church, 2002; Doherty, 2003). The difference between these two is the way the data is stored to the system and how it is retrieved. In object-centric architecture, game objects follow the usual object oriented programming paradigm where single game object can inherit attributes and interfaces from ancestors but still have some unique attributes or interfaces (Duran, 2003). This architecture is also known as inheritance-based architecture (Doherty, 2003). An example of this kind of architecture would be Animal_DalmatianDog class which can inherit the attribute “walking_speed” and its default value “5” from its parent class Animal_Dog. The instances of Animal_DalmatianDog class could then change this default value later on during gameplay if necessary. While this architecture is familiar to many developers and provides easy-to-grasp conceptual background, it is not so often used in games. This is because of the static nature of the architecture: once it has been implemented, it is hard to change things or add more unusual functions to the game (Duran, 2003).

Other way to implement game objects is component-centric architecture, also known as entity system architecture (Gestwicki, 2012), where game objects are implemented using database-like system: Single game object is just collection of data, that are linked together with similar unique identification number (Church, 2002; Duran, 2003). In this case, data is not collected with the game object but in the components that make up the various game objects, thus making the architecture follow delegation-based design (Doherty, 2003). Common example of this kind of architecture is Renderer-component that allows game object to be rendered through rendering process. This kind of component would save each renderable game objects texture and mesh in such a way that they can be easily referenced by unique identification number. Using this component-centric the previously described Animal_DalmatianDog example could be implemented by storing identification number and data pairs to specific component, Movement-component for example, where each instance of Animal_DalmatianDog object would have their own movement speeds saved. The strength of this system is the flexibility that allows different components to be integrated in to single game object easily, creating sometimes unique objects, with little overhead. The negative side of this architecture is that the overall component hierarchy may become very extensive which could cause problems in both efficiency of the system and understandability of the design (Duran, 2003).
3. Communication inside game

Based on the chapters above, game can be seen as having two major components, processes and data, that work together in order to provide required game experience. Processes process and manipulate the data that is given to them and after process is finished, the data is returned for other processes to use; this is the basis of data-driven architecture that many games and similar software use.

However, this simple concept described above does not address one important issue: How do all components across the whole system communicate in order to collaborate on fulfilling their required tasks? While at first glance this may seem as simple question, when one remembers the vast number of game objects and myriad of parallel processes that occur inside single game, the significance of this “communication issue” is clear.

The communication issue itself is not a new topic in software engineering. There are many other systems which, like games, require a large amount of data to be distributed and processed throughout large systems and in these systems, similar issue has risen. While the advent of software engineering solutions such as object orientation and messaging protocols have aided in resolving this issue, the evergrowing processing speed, distributed and parallel processes and growing amounts of data have always made the issue a challenging problem to solve.

One way to understand this communication issue is “coordination paradigm” which offers own viewpoint to the structure of software. The key idea behind this paradigm is to see the software as being encompassed out of two independent parts: the actual computing part that handles processing the data and coordination part that handles the overall coordination of multiple components (Papadopoulos & Arbab, 1998). While this paradigm has been around for a long time (for example, Gelernter's & Carriero's article from 1992 already talked about the significance of coordination between computing activities), it still provides a fresh view to the communication issue in games. Thus, the coordination paradigm will be used as a basis to study the communication issue in this part of the study as it emphasizes clear separation of the act of processing and the act of coordination. More specifically, it allows us to explore the act of coordination without focusing the act of processing.

In coordination paradigm, the coordination part of software is responsible of managing communication and cooperation between processes (Papadopoulos & Arbab, 1998) and the specific method how this is accomplished is called coordination model (Ciancarini, 1996). While this terminology is more aimed for describing specific purpose-built languages and similar high-level structures, the concepts also apply in other mechanisms and architectures that try to solve the communication issue. Thus, the different solutions to solve the communication issue between different components that have been offered by both game industry and academic researchers can also been seen as coordination models.
In this study, the coordination models which are meant to be solutions for communication issues in game engines are divided into two groups: solutions are either specialized communication mechanisms to be used inside games or overall architectural solutions that describe the general structure of the game. While many of these solutions do solve the issue at least in smaller scale, none of these solutions seem to offer overarching solution to the communication issue in the game. What's more, they do not provide a solid theory on the communication inside games that could be studied as own, overarching solution and possibly be either used in other software or be improved on systematic ways.

In this chapter, a number of solutions from both academic research and game industry will be overviewed to have an understanding of the current solutions. The solutions selected to the overview do not represent whole scope of solutions but rather the most common solutions offered. The coordination paradigm will be used to provide a clear perspective for this overview.

### 3.1 Communication mechanisms

Communication mechanisms are defined in this study as solutions that provide specific techniques that allow certain components to communicate with each other. While architectural solutions described in following sub-chapter focus on how the whole system should be structured and managed, communication mechanisms provide lower level solution for the communication. While some of the mechanisms can be extended to be used as communication methodology in all parts of the game, most of them try to solve the issue in smaller scale.

But what is this scale? Different mechanisms try to help coordination between components but these components are not always at the same scale. For example, trying to solve how a rendering process finds correct 3D model to display is vastly different from trying to get two seemingly independent game objects, like “Cat_A” and “Cat_B” to interact each other. This difference in scale will be henceforth identified as difference in perspective. This means that the communication needs between different components inside games can be separated as being either in “game world perspective” or in “raw data perspective” (see Figure 8).

![Figure 8. Different perspectives for communication needs inside game](image)
To understand the difference between these perspectives, consider what is required from the game to provide the player the game experience: From the perspective of game world, game experience relies on the cooperation of multiple different game objects inside the game world that is just periodically shown to the user to see while from the perspective of the real data, the game experience is built upon different processes that process same set of data to provide effect of simulated, lively game world. Thus, in “game world perspective”, data that is structured as game object performs actions as independent agent and it is the interplay of these actions that cause the simulation of game world to occur while in the “raw data perspective”, it is the role of processes to actually simulate the game world while game objects are just storage units of related data.

As both perspectives are focused on vastly different components (in game world perspective, major component is single game object; in real data, the major component is process), communication needs are very different. In game world perspective, the communication mechanisms are built to let individual game objects to communication with other game objects. Some common examples of such mechanisms are direct inter-object communication methods like “function-call”, “slot and signal” or “state listeners” (Haller, Hartmann & Zauner, 2002) and independent event-based messaging systems (McShaffry, 2003; Gregory, 2009). These kinds of mechanisms can be used to transform simple game objects into agents, which can sense their surroundings dynamically and make proactive and reactive actions according to the information they sense (Gemrot, Brom & Plch, 2011). As multi-agent systems have been studied widely lately, the literature from that field have also offered new solutions for the communication issue. One such a solution is the use of semantics to enrich the interaction between agents and their environment (Tutenel, Bidarra, Smelik & Kraker, 2008).

As real data perspective uses processes as major components, the communication mechanisms that are based on it are concerned on how to transport data between processes and data storage efficiently; the interaction based on individual game objects does not really exists in this perspective as the information transported from certain game object to another is presented as certain process changing appropriate values in the game objects based on the data it reads. A good metaphor for this perspective is the database: processes request the current state of database and then change and/or display the state. This metaphor is so strong, in fact, that recently there has been rising interest to bring the knowledge from database research to the game industry as well as inviting the researchers to tackle on the unique issues in games (White et al, 2007). This has been fruitful transfer of knowledge as new communication mechanisms have been proposed. One such example is state-effect pattern which defines the structure of game loop and game object to ease processing the interacting game objects (White, Sowell, Gehrke, & Demers, 2008).

The difference between the two perspectives can be also seen in the coordination paradigm literature. By using terms from the article by Papadopoulos and Arbab (1998), game world perspective can be seen to encompass data-driven coordination models, which rely on the processes, in this case game objects, manipulating the data and coordinating itself and/or other game objects. Real data perspective, in contrast, is encompassed from control-driven coordination model where the coordination processes
that coordinate the efforts of multiple data processing processes have been clearly separated.

While the communication mechanisms belonging to each “group” use very different perspective, they all share common attribute: they are mechanisms that are built upon lower level functions that are already provided by another lower level component. These lower level functions are getting and affecting the state of component. In previous Chapter 2.2.2, these functions were mentioned as being the responsibilities of game engine as it handles the management of data and data tells the state of the component. Thus, communication mechanisms can be seen as extensions for game engine as they extend the data management functionalities to provide higher level communication than what game engine originally would offer.

The two game object implementation architectures that were mentioned in the previous Chapter 2.2.2 can be seen as linked with the communication need perspectives. This linking is due to the similarities on how the game objects are formed: In object-centric architecture, as well as in game object perspective, mechanisms need to able to handle independent game objects efficiently thus requiring a communication mechanisms thought which they can send messages to other independent game objects. Same linking happens with component-centric architecture and real data perspective as in both, the game objects themselves are seen as a composition of different data. Thus, the communication mechanisms are affected on the choice of game engine architecture.

As a general note, in the game industry the direction for game object implementation has been towards component-centric architecture (Church, 2003) which would imply that real data perspective will be more commonly used in modern games in future. However, there is still need to understand the game in terms of game objects as they are conceptually higher constructures and thus easier to work with when designing a game. This may bring database researchers closer to game industry (see for example White et al, 2007).

### 3.2 Architectural solutions

In this study, different theoretical architectures offered as an answer to the communication issue are defined as architectural solutions. As opposed to communication methods which described the communication between components of the game, the architectural solutions try to structure the game in a way that makes the communication process easier to understand and to implement. In short the architectural solutions could be described as high-level solutions to the communication issue.

Many architectural solutions are based on researcher's studies rather than on the actual games or game engines. This makes the architectural solutions much more theory based than the communication mechanisms. While the reasons for this are not clear, it could be assumed that the reasons are two-fold: firstly, researchers have had growing interest in the games and game engines, which has resulted in rising number of studies related to them (see Amapatzoglou & Stamelos, 2010, Table 3, p. 892); secondly, when game engines are used in making a game, developers often customize them so far that setting common architectural solution would be very hard (White et al, 2007).
Another large difference between communication mechanisms and architectural solutions is that while the mechanisms can be divided to using either “game world” or “real data perspective”, architectural solutions are all based on the “real data perspective”. Although it is possible to propose architectural solution that would work from “game world perspective”, all the solutions showcased here look upon the games as whole systems, and thus they follow the “real data perspective”.

In following part, five proposed architectures from the existing literature will be analysed using the coordination paradigm. These solutions are not the only architectural solutions offered, but they represent most relevant and commonly referenced game architectures. It should be noted that the architectures were selected on the basis of how much their author promoted as being game specific, rather than common communication architecture. Also most of these architectures were presented as general architecture, not communication-oriented architectures. Still, as communication is a vital issue that must be handled accordingly, each of these presented architecture does contain some form of answer to the issue and as such, can be also seen as full coordination models.

The focus of the analysis is the communication, i.e., what kind of structures in the architectures handle coordination and communication, and how the authors of the architectures describe these communication mechanisms.

Rollings' and Morris' game architecture is one of the earliest architectural descriptions of games (Doherty, 2003; Plummer, 2004). It was part of their book Game Architecture and Design, published in 2000, and while the book was targeted to general audience, the book quickly became a part of academic research often being refered. The authors base this architecture on their own experience in the game industry, and they specifically point out that this is their interpretation of typical architecture of game.

According to Rollings and Morris, the architecture seen in Figure 9 represents all different systems used to build single game. While at first architecture may seem to be
built out of vast number of components, these systems can be grouped to form more familiar view on game architecture:

- Interface components: Components close to the hardware and system-wide services (Input, Graphics, and Audio) work as low-level interface between the system-wide services and the rest of the components. These components are User Interface, Graphics Engine, Sound Engine, and Music System and they form the platform which allows other components to use hardware and its services.

- Game data handling components: Physics Engine, Logic Engine, and Event Handler were also earlier noted as major processes but these components only handle game data, not direct input or output as the components listed in above group do.

- Game data storing component: Game data is the component handling game specific data, from the current state of the game world to the graphical and sound assets.

- Supportive components: The rest of the components handle specific tasks required by the graphical user interface. Exception to this is the configuration system, which is curiously put between the menuing system and game data. Eventhought Morris and Rollings do not tell why this is, it can be assumed that similar functionality can be done by the Game Data component, for it also handles the management of the game-related configuration. Another possibility is that authors feel the configuration only handles the input and output tasks of the game and thus should not be part of the specific game but rather the specific configuration of the surrounding environment.

As can be seen from the list above, while the architecture described Morris and Rollings does not specifically contain game engine, many of the systems mentioned in it are often set as subsystems of the game engine. For this reason, this architecture seems to follow the structure mentioned in the earlier chapters.

Eventhought the Morris' and Rollings' game architecture is clearly set as a structural diagram, one can notice the role of the communication in it by observing how Event Handler system is set between Game Data, Physics Engine and Logic Engine. In their book, the authors describe the game world being inhabited by tokens, which are supervised and managed by the computer and which communicate by using events (Rolling & Morris, 2003). These tokens can be seen as being conceptually equivalent to the game objects mentioned earlier in the study. Thus, since tokens are part of the Game Data component in this architecture, the Event Handler needs to be between the Game Data and the components interacting with tokens, Physics Engine and Logic Engine.

However, while the authors clearly show their game architecture uses events as communication mechanisms, they do not really describe how the Event Handler deals the events. The communication is somehow based on the interfaces in the game objects, but exact description is not given. Thus, this architectural solution to communication issue is clearly lacking.
Another, widely different perspective of game architecture was presented by Jeff Plummer in 2004. In his master's thesis titled 'A Flexible and Extensible Architecture for Computer Games', author constructed the game out of highly independent systems which all communicated through central “data repository” to generate simulated game experience (see Figure 10). This structure follows two main architectural styles: “system of systems” architecture in which complex systems are built out of domain-specific components and “data-centered” architecture in which the centralized data storage houses the data for the processes (Plummer, 2004).

Based on the earlier chapters, the architecture proposed by Plummer can be seen as the internal structure of the game engine, given each different domain-specific process is kept as independent subsystem. The architecture shown in Figure 10 do not contain higher “system level” component but Plummer does mention it and its role as overall “manager” and handler of timed game loop (Plummer, 2004, p. 54). Another detail about the proposed architecture is that the game objects in it are built with component-centric method, in which one main “game object” in the data repository is linked to the domain-specific data used by specific system via same identification number.

From the communication point of view, Plummer's architecture is lacking. He mentions how this system works efficiently from the viewpoint of single system and showcases examples of that but he has not mentioned how game state can be affected through interaction of different systems or game objects. The author himself points this out as he mentions that “...architecture takes a step back from looking at games as a system of game objects, and looks at them more as a data centered System of Systems” (Plummer, 2004, p. 50). It could be argued that the system can be extended to have functionality which would allow game objects themselves to affect other game objects, such as event management system, but as the author does not state this clearly, the architecture only shows static view of the structure of the game.
The third architecture analysed in this study is by Michael Doherty and it published in the article 'A Software Architecture for Games' in 2003. This architecture is based on the data-centered view, where the game engine is given purely rendering and input transporting role, while simulation is handled by separated process which manipulates the game world through object system (see Figure 11). The data manager handles the upkeep, storing and loading of the data to object system. As seen from the Figure 11, the interaction among the top-level components is not always synchronized (marked as double-headed arrow between components) meaning that not all components are required to interact with other systems. The author explains the interactions in following way:

- Game engine can only read the game world state from the object system, and render it to the player. As such, it has no means to affect the game world and any input from the player must go through simulation component which will transform the input into some kind of action inside game world (see Figure 12).

- Simulation and object system components need to communicate frequently since their interaction actually causes the game world to update. Their roles can be seen as same as between different simulation processes and their corresponding data: simulation component, which is actually composed of multiple subsystems (see Figure 12), changes and updates the data saved in object system.

- Data manager handles only the tasks required to happen between the game and the environment, such as saving the object system data to file system or other similar persistent storage.

Doherty continues to elaborate on the interaction between different components further by showing more low-level architecture supported by this kind of design. This elaborated architecture is shown in Figure 12 below. In this figure, two top-level components, ”game engine” and ”simulation”, are broken into smaller modules; modules belonging to ”game engine” are graphics, audio, OS interface, and Input and while ”simulation” modules are game logic, physics, bot artificial intelligence (AI), player AI, and Sim control. The data manager component is not shown in the figure for clarity.
As shown in the detailed architecture, Doherty's proposal also follows the structure of game engine and subsystems described in earlier chapters. Doherty also separates the unique simulation modules from the general game engine modules even as far as grouping them to different components. However, Doherty's description of game engine is more comparable to rendering and input gathering processes, and thus it does not follow the idea of game engine being the overall manager of the whole game. Still, he specifies that simulation modules are all managed through simulation control module (shortened as 'sim control' in Figure 12) and simulation modules all share same interface to the game data through centralized object system. These specialized data access and simulation management modules imply that Doherty notes that there is a need to have centralized system level mechanisms for such tasks. While Doherty uses the term "game engine" only for rendering and input gather processes, based on the game engine literature the term could be extended to refer also the component that handles overall management of different system-level tasks.

Although the architectural diagram presented by Doherty is clearly focused on the overall structure of the game, he does mention possible ways to implement the interaction among simulation modules. As can be seen from the Figure 12, simulation modules do not interact with each other directly. Doherty proposes that kind of interaction could occur by saving 'impulses' of certain action to the central game data storage which would be then read and acted by different simulation components. For example, if the AI of certain game object would want, based on the internal state of the game object, move towards nearly water, AI script ran by 'bot AI module' would save movement impulse to the object system. When a physics module would read the state of same game object and notice the impulse, it would try to use movement method linked with game object to move according to the impulse. This kind of 'impulse' concept seems very similar to event-based communication system.
Fourth architecture analysed is provided by Mike McShaffy, who published it in his book *Game Coding Complete* in 2003. Like Rollings and Morris, his knowledge comes from his experiences on working on the topic rather than academic studies and thus his architectural solution can be seen as having industry background. It should be noted that author mentioned that this architecture only describes the game architecture as he understands it, and as how he has implemented the architecture in the projects he has worked on.

The architecture, shown in Figure 13, is based on dividing all the subsystems inside game into three categories. The first category is *Game Application Layer* and the subsystems in this category represent the environment where game runs, as they function between the system and the actual game, supporting other components with their functionalities. The second category, *Game Logic*, contain subsystems that handle executing game logic and simulation and managing the state of the game. It also contains support systems like communication subsystem. The third category, *Game View*, contains systems responsible for presenting the game world and for reading and transporting the given input to the Game Logic component. McShaffry represents Game View as multiple components since there are different views of the game which need to built for other systems, not just with players: for example, an artificial intelligent system does not require the game state shown in as graphical terms as player would require but it must know the map layout in order to calculate the movement of computer-controlled game objects.

The interactions among these systems can be divided to two different interactions. Firstly, Game Application Layer provides basic functionalities on which other system components rely upon. These functionalities are used by calling them directly, making the Game Application Layer serve as an extensive library. Secondly, interaction between Game View and Game Logic is handled by event-based communication between them as single action in either component can cause myriad of changes in both components.

This event-based communication is described in detailed way in the book. In McShaffry's architecture, event-based communication is structured out of three elements, *events*, *event manager* and *event listeners*. Any system can use the global event manager to construct an event which then sent to all the event listeners registered for that kind of events. Event listener is an interface and thus any class inside game can implement it to register itself as an event receiver in the event manager. In the book, each of the three elements are explained down to code level. Overall, this structure is
very similar to publish/subscribe interaction scheme where interaction is handled by event publishers creating events which are directed by event service to event subscribers (Eugster, Felber, Guerraoui & Kermarrec, 2003).

McShaffry's architecture proposes interesting and throughout way of thinking of game engine which has more practical take on the issue. This is understandable because of the game industry background of the book. This has an effect on how the architecture is explained as the author does not explain the reasons why the event system has been built this way or to what extend the events are used in the system. Some parts of the books indicate that events would be used extensively in the games but since the book does not contain many examples of this, the utilization of events cannot be demonstrated. One key question that author does not answer is how event-based communication would differ when it occurs between major processes and when it occurs between game objects. Author's usage of processes as major computational units of the game engine indicates that the architecture is built around “game data perspective” in which communication always occurs between processes.

Compared to other architectures presented above, Smed's & Hakonen's article (2003) took different approach. It describes the game architecture using Model-View-Controller architecture pattern (see Krasner & Pope, 1988) as base. While the article itself about the definition of the game, authors do spend time on describing the architecture. The diagram of the architecture (Figure 14) shows the dataflow of the system: as in standard MVC architecture, Model contains the state of the data, which means the state of game in this case. This is shown to player and other components requiring it by different Views. Controller captures the input given by player and other components to run change the state of the game in Model.

![Figure 14. Game architecture modeled after Model-View-Controller model by Smed & Hakonen (2003)]
As Figure 14 reveals, authors extended the standard MVC architecture by defining more concrete components like rendering, output device and script. The components defined in the Smed's and Hakonen's architecture can be seen as mixture of computing processes, devices, data storage and user which are required in order for the game to function. This architecture is clearly player-centric as human player is identified as individual object that has clear ways to get input and send output to the game.

The authors of this architectural solution do not clearly define how these different components communicate but they have indicated the dataflow of the program in the diagram by red arrow. This arrow leaves the readers without too much knowledge on the internal communication occurring in games but still this arrow and the architectural diagram around it provide at least rudimentary idea on how the communication should occur. Model-View-Controller model itself is one of the standard ways to structure a software, and thus there are many articles and tutorials on how to implement it. That is the main reason why Smed's and Hakonen's architecture is interesting: it clearly links the game architecture with well known model.

In general, these five architectural solutions provide different ways to approach communication inside game engines but in the overall conceptual level they are remarkably similar. There are two major similarities: their general architecture resembles Model-View-Controller pattern and the most common communication mechanism used is event-based communication. To elaborate on the first point, while some models, like Smed's and Hakonen's architecture, followed the MVC pattern more carefully, all the other architectures could be also seen as presentations of the same common theme. In Table 1, major components of each architecture are compared to similar component in MVC model. Note that Smed's and Hakonen's architecture is left out for simplicity and to avoid redundancy.

<table>
<thead>
<tr>
<th>MVC</th>
<th>Rollings &amp; Morris</th>
<th>Plummer</th>
<th>Doherty</th>
<th>McShaffry</th>
</tr>
</thead>
<tbody>
<tr>
<td>View</td>
<td>Interface components</td>
<td>Graphic, UI, and Audio systems</td>
<td>Game Engine</td>
<td>Game View</td>
</tr>
<tr>
<td>Controller</td>
<td>Game data handler components</td>
<td>Artificial Intelligence system</td>
<td>Simulation</td>
<td>Game View</td>
</tr>
<tr>
<td>Model</td>
<td>Game data storing component</td>
<td>Object Management System</td>
<td>Object system</td>
<td>Game Logic</td>
</tr>
</tbody>
</table>

Each component in second row is responsible for acting as “View” component of the whole system: They contain logic to display the state of the model (game world in this case) and to transport the user input to the “Controller” component. In third row, components that can be seen as “Controller” are shown: These contain all the business logic required to change the state of the game world. This state is kept in “Model” which are marked in fourth row: All of these components handle managing the state of the game data and provide means to access and manipulate it to other components. While only Smed and Hakonen clearly show how the MVC model is used in their architecture, it is evident that other architectures follow the same structure.
The second similarity is not as prevalent as the usage of MVC pattern but it can still be observed from the analysed architectural solutions: three of the five solutions name events as an integrated communication mechanism. These three, Rollings' & Morris' architecture, Doherty's architecture and McShaffry's architecture, all explain how event-based communication is done between the game data and simulation processes. However, the perspective used in each case is different: in Rollings' and Morris' architecture, events are used to communicate mainly between game objects and thus their architecture follows “game world perspective” but in Doherty's model, events are used to allow interaction between different simulation processes, making his architecture follow “real data perspective”. McShaffry does not explain in detail how the event-based communication occurs between the components but rather just tells that events can be used in both perspectives, i.e. both between major components and between individual game objects.
4. Communication architecture of game engine

The analysis conducted in Chapter 3 shows that while there are multiple different communication mechanisms and architectural solutions to the communication issue, there is still a lack of synergy among these provided solutions. Whilst there are capable solutions in both categories, the lack of overall theory is evident since each solution is built around different environment and uses mostly different components. This lack of overall theory for the communication occurring inside game engine makes it challenging to understand the communication aspect of modern game engines, especially in relationship to the theories which illustrate the communication occurring inside other software.

Analysis also indicates that there are some possibilities of integration between low-level and system level communication. For example, the architectural solutions provided by Doherty (2003) and Rollings and Morris (2003) do mention the usage of events as communication mechanism, but they do not provide exact information on how the event system works in their architecture. This is problematic since communication mechanisms, such as events, play an important role in defining the coordination model the software adheres. By not having clear understanding of which coordination model suits games and how choice of coordination model affects games, both academic research and industry knowhow suffer, as these issues can not be further analysed.

In order to bring the focus of the research to the coordination and communication occurring in games, in this study a new architectural solution is proposed, as a counter to the architectural solutions shown in Chapter 3.2. This new solution is named “Communication-oriented game engine architecture” as it focuses only on the coordination and communication among the major components of game engine, and as thus does not delve deeper on how the components have been constructed or how gameplay itself is handled by games using the game engine built based on this theory. The architecture itself showcases how the communication dataflow among major components of the game can be structured, and how this structure allows the coordination of overall efforts. In short, because the focus of the new architectural solution is in coordinating the efforts of running the game, the solution can be defined as a coordination method, as per Papadopoulos & Arbab's definition (1998). This is vastly different approach compared to other architectural solutions, as those have tried to show how games are structured but not how they are managed or how they run at detailed level.

In this chapter the new architectural solution is described. Before that the main communication mechanism, event-based mechanism, is first described and discussed upon, as it plays an important role in this solution. After that, the architectural solution itself is presented as a conceptual diagram, and an exampled component diagram is presented in order to introduce how the principles of the new solution work in action.
4.1 Events as communication mechanism

The event-based communication mechanism is an integral part of this architecture and thus more discussion about this mechanism is in order. The concept of events is not a foreign one in software engineering or in game industry. In software engineering literature events have been the topic for a long time and they have been used to allow communication not only among software in same computer but also among multiple systems (see Meier, 2000). In game industry, event-based communication has been used but it is not clear how frequently used solution it is the details of commercial game engines understandably have not been published. Books which are based on industrial know-how (i.e., books by McShaffry (2003) and Gregory (2009)) do implicate that event-based communication is a viable and common communication mechanism.

In the software engineering literature, an event represents a change in the state of entity, which is signaled outwards to other, potentially numerous, entities. The entity whose state was changed is often called either a producer or an event source, and the entity responding in some way to this event is called either a consumer or an event listener. The event can contain related meta-data, which can be read by the consumer of the event. Events can be either sent from one entity to another via direct links or, in more evolved form, can be sent to an event manager which directs the events forward to consumers. Direction can occur in many ways, but filtering according to the interest of consumer is the most common one. The way how producers, consumers, and event managers interact and how they perform event routing, filtering, storing and other common tasks is called event model (Tarkoma & Raatikainen, 2006; Meier, 2000.) In software, events are mainly used in communication, i.e. sending information from producer to consumer, and in coordination, i.e. directing the flow of control around different component (Mehta, Medvidovic & Phadke, 2000).

In game related literature, this event-based communication has been described as happening between game objects and between different processes. In game object related literature, events are often linked with the artificial intelligence of the game objects as a fundamental way to let game objects interact with each other and with the environment (Macal & North, 2010; Ocio & Brugos, 2008); the previously described “game world perspective” can be used to described this kind of game object centric view. However, other literature points out that events can be used to implement message-style communication among major processes, which follows the “real data perspective”: events can be seen as messages which currently running process leaves to temporary storage for the following processes, possibly itself, to consume. Depending on the game and game engine in question, the number of different kinds of events vary a lot but some suggest that everything from user using interface elements to player's character kicking a ball, could be set as its own event (McShaffry, 2003).

Key element to note when considering event-based communication from “real data perspective” is in this view, events themselves are only small storage units containing some dynamic data temporarily. While they are linked with game world, they exist outside this game world: events do not themselves have an impact on the state of the game world and its content, they simply hold the data that informs the processes how to act upon the change. This is different from the “game world perspective”, where messages are considered more as a whole, semantical language used to allow communication.
It is important to clarify again that two perspectives on data do not contradict as even in “real data” based event system, individual game objects can communicate with other individual game objects. However, the difference is that in “real data” perspective, the event is handled as any other kind of event, not as a special “game object to game object” message.

4.2 Depiction of the architectural solution

The name of this architectural solution is “Communication-oriented game engine architecture”. As the name tells, there is one major difference between it and those architectural solutions shown in Chapter 3.2. Because the architectural solution is focused on the communication aspect, it does not affect how other parts of the game engine are implemented as long as they share certain common interface that can be used when enabling communication and coordination among the parts.

The architecture highlights the key role game engine should have on the communication occurring in the game: game engine should be the fundamental foundation for different components by providing all the necessary functions they require. In other architectural solutions, the role of game engine was mostly to handle the input and output of the game, but in this one, the game engine handles the main communication functions by controlling it through game loop and providing a common interface that other components can use.

At conceptual level, the communication-oriented game engine architecture is built upon three major components and the dataflow between them (see Figure 15). Different components are processes, events, and game data: processes are the major processes described in Chapter 2.1.1, they are the method how user can see and interact with the game world and the method used to simulate the game world. Events are temporary messages processes can leave for other processes to act upon. Lastly, game data encompasses both the state of the game world and the state of individual game objects. The components and their interactions can be seen from Figure 15.

![Diagram](image)

**Figure 15.** Conceptual level diagram of main dataflow in games

As seen from the figure above, each component interacts with other components in specific ways. The interactions can be characterized as follows:

- Processes and game data – Processes read the state of the game world from game data. They only read the data they need, and thus this communication
follows component-oriented game architecture. If necessary, process can change the state of the world by writing new state to the game data.

- Processes and events – Processes can generate events temporarily saved for later use. These events are characterized as something with an effect on the game world or the software, but not needing immediate action. After the process has produced an event, next process can consume events which can affect the execution of the process.

- Events and game data – While these two components do not have any direct interaction, it is worth noting they can affect each other via processes: a state of game world at certain moment can cause a process to produce events, and an event consumed by a process can cause the process to change the state of the game world residing in game data component.

There are three noteworthy details regarding the architecture. Firstly, the relationship between processes and game data is similar to the relationship between processes and events. Both events and game data represent information storages which store new information given by processes and which release stored information to processes requiring it. It is not the act of storing information which influences the information itself, but the act of processing which done by the processes. Since the process acts upon both game data and events it has gained at the start of its execution, and both game data and events are only influenced by other processes, this architecture follows the data-centered architecture prevalent in modern game engines.

Secondly, because the architecture follows data-centered architecture by illustrating game engine as consisting of processes and data, the architecture is built based on the “real data perspective”. This means that the game world is seen as data that is manipulated through processes which communicate among each other via events. While the relationships among data, e.g. which data is connected to which game object, is relevant to processes, they are not relevant to the game engine as a whole. In case process requires information about the relationships, they can be requested from game data but still, for the larger architecture, that information is data among other data.

Thirdly, the architecture proposed is related to the whole game engine. In principle, the conceptual level description shows what the main components necessary in games are. All components are part of larger environment provided by the game engine, and it is through the game engine that all components manage the interactions shown in concept level description. Therefore, while game engine is not illustrated in the diagram, it is still there as the environment where components are located in.

To showcase how the conceptual level diagram could work in reality the ensuing component level architecture of game engine is shown below in Figure 16. Note that this is just one example on how the conceptual level diagram could be implemented. The ensuing architecture has four major components: game loop, subsystem, event manager and game data manager. Three components, subsystem, event manager and game data manager have, in principle, same functionalities and interactions as their counterparts in conceptual level diagram: subsystem contains single process, event manager provides both event creation and event querying services and game data manager provides services to change and query game data. Each of these components are managed by the game engine which handles the lifecycle of components and
provides the connections between them. Event manager and game data manager also handle longtime storage of said elements of game. These storages are, at least in componental level, separated from each other.

![Diagram of main components of game engine](image)

**Figure 16. Interaction between main components of game engine**

The only addition in the component level diagram is the game loop. It handles execution of subsystems, i.e., it commands a subsystem to run at certain time. The reason why there is no interaction among it, the event and game data managers is that both managers handle storing and managing their data independently without requiring game loop to command them.

Runtime interaction and collaboration among these components follows the interactions mentioned in the conceptual level architecture. The interaction is controlled by game loop component, which executes all the subsystems by allocating them runtime and by ensuring to halt them when they start taking too much time. When each subsystem is executed, they first query the data they are interested in from game data manager. This data varies according to on which subsystem is in execution phase, and for what purpose the subsystem is querying data. As an example, while rendering subsystem is interested only in visual presentation of a game object that is currently showing to the player, a physics subsystem is interested in physical properties of all game objects.

After querying the data from game data manager, subsystem queries for the events that are related to its domain from the event manager. As there can be various events, not all events affect every subsystem and even if they affect, the effects can differ a lot. Notable commonality between different events is that they all cause the related subsystem to affect their related data somehow. For example, rendering subsystem might start a new animation on a 3D model when it gets a certain event X, whereas input gathering subsystem might respond to the same event by disabling the mouse input. Another important aspect is that while an event has an influence on the processing done by a subsystem, that effect does not always causes change in game data. An example of this would be user clicking on empty area of game screen: the input-gathering subsystem generates a mouse click event on that area and subsystem responsible for mouse click events consumes that event. As there is nothing in the screen that would react to mouse click event, the event is consumed but does not have impact on the game data.

After querying the events, the subsystem can then generate new events based on the events and the state of the game world it has gathered. These new events are saved in the game event manager, possibly using its interface to generate events. The events can be addressed by the event sending subsystem to other subsystems or even to subsystem
from which the event is originated from so that any of these subsystems can act upon the event later on. Each event may also have some kind of timestamp to add another dimension on the event-based communication. These timestamps could then allow game data manager to order the events in more fitting and efficient ways. After generating the events, the last action subsystem can do is saving the related data over the old data and thus updating the state of the game world. This is done by using the interface game data manager provides. After the updating the game world, subsystem notifies the game loop that it has completed and then halts to wait for next time it is executed.

The description above illustrates the interaction between different components of game, but it is just a single example how the system could work. The provided example of interaction among the components of game engine assumes that the game is run in an optimal environment where memory and runtime are limitless. This optimal environment does not take in account overhead of resource management of both memory and allocated runtime which complicate the communication issue further when certain limits are present in the environment. The principles of the communication shown in the above description do not, however, lose meaning in multi-processor environment, where more computing power can be given not only to the game data and event managers but also to the game loop and subsystems. The same principles can apply even if every subsystem and game loop would run in dedicated processors, game loop just must have a mechanism and authority to control when every subsystem is run and when it needs to be halted to synchronize the efforts of the whole game.
5. Analysis of existing OSS game engines

The theory of the communication solution shown in Chapter 4.2 is rooted on the academic research and industry knowhow, which were summarized in previous chapters. Thus, the architectural solution and the theory surrounding it should represent how the communication occurs in modern game engines. However, due to the lack of existing literature on this specific topic and the lack of information from the industry knowhow, the similarities between the proposed theory and current game engines cannot be verified in a straight-forward manner. This verification is needed since it would determine how well or loosely the proposed theory explains the current state-of-art in game industry.

In order to verify the extent that the proposed theory of the communication solution accounts for the communication occurring in modern game engines, an analysis of existing game engine’s communication mechanisms was performed. This analysis had two goals:

1. To analyse how much the theory of the communication solution resembles the communication that occurs in modern game engine, and therefore verify how correct the theory is in relation to the current engineering knowhow.

2. To see how the architectural solution differs from the existing game engines, and to understand the reasons and the effects of these differences.

Since the architectural solution is built as a high level construct explaining both common structure and high level logic of message management, the analysed game engines had to be also analysed at the same level. In order to perform the analysis, both structure and logic of the game engines had to be modeled. As a common modeling language, UML diagrams were chosen as the modeling tool; more specifically class diagram was used to represent the common structure, i.e., architecture, while sequence diagram was chosen to represent how the message management logic works.

To use UML diagrams in the analysis, a collection of said diagrams had to collected. This meant either gaining access to diagrams used in development of game engines or drawing the diagrams from existing material, i.e., source code. Majority of game engines are commercial tools and neither the diagrams used in their creation nor their source code are publicly available, thus using commercial game engines for this analysis was not viable. However, there are open source game engines and while UML diagrams are not commonly utilized in their development, their source code is available. This meant that for this analysis, open source game engines were used as target group for analysis. It should be noted, tho, that as it is uncertain how much open source game engines differ functionally and structurally from commercial game engines, the result of this analysis cannot be directly used on commercial game engines. This topic is further discussed in the chapter 6.1.

Following steps were taken to perform the analysis:
1. Select target group of open source game engines for analysis

2. Gather the documentation and source code of each game engine

3. Analyze communication methods offered by each game engine for...
   3.1 Subsystems that require data from game world and game objects
   3.2 Communication among game objects

4. Produce following diagrams:
   4.1 Diagram which explains the main concept behind communication in target game engine.
   4.2 Class diagram(s) of the architecture for handling the messaging
   4.3 Sequence diagrams for main communication functionalities
   4.4 Other diagrams if necessary

5. Compare the results of the analysis and produced diagrams to the theory and architecture shown in the previous chapter

6. Document the results

As shown in above list, first step in analysis process is selecting the target engines out of all possible open source engines. This filtering was to seek out most potential game engines for analysis, for there are many differences between different game engines. The following criteria were used in filtering:

- True game engine: the analysed game engine should act as a game engine, not just as a graphical engine. This means that game engine should contain enough functionalities for a complete game experience.

- Existance of communication mechanism: not all game engines contain built-in method for communication among processes or between game objects. Some game engines function only as an interface to hardware and using them requires the game developers to develop their own communication system. Since built-in communication mechanism is the target of analysis, analysable game engine has to have it.

- Availability of source code: this is prerequisite requirement for the models cannot be constructed without access to the source code.

- Availability of documentation: in order to understand how developers have meant their game engine to be used, documentation has to be available. Without proper documentation, there is a risk that models built out of source code may be misunderstood.

- Single-player-focused architecture: this study does not take online or multiplayer experience in to account and as communication of those game
modes is often built upon game engines, we would need to study the communication between different machines as well. This is not the aim of this study so it is left for other researchers to tackle on.

This filtering was used on the base group of open source game engines which were gathered from online sources. Three online sources were DevMaster (2013), Wikipedia (2013) and IndieDB (2013) and they were chosen as they all provided an open platform for game engine developers to promote their work. The reasoning behind this was if game engine developers would be promoting their project, it should be somehow available online and thus could be further analyzed by reading the project's documentation and browsing the source code repository.

After filtering the base group, four game engines were selected for the analysis: Blender, Delta 3D, Gameplay and Polycode. Each of these game engines offer same basic functionalities but there are many key differences in the architectural and functional levels. Key information of each game engine is listed in the Table 2 below.

<table>
<thead>
<tr>
<th>Table 2. Key information of selected game engines</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Analysed version</td>
</tr>
<tr>
<td>Release date of version</td>
</tr>
<tr>
<td>Organization</td>
</tr>
<tr>
<td>Development starting year</td>
</tr>
<tr>
<td>Game object impl.</td>
</tr>
</tbody>
</table>

The last row in the table above shows the game object implementation used in each game engine. As explained in Chapter 2.2.2, game objects can be implemented by using either component-centric or object-centric view. All the analysed game engines implemented game engines using component-centric architecture where each game object was built around domain-specific components storing own domain-specific data. While this feature of the game engines was not explicitly stated, it was evident from the architecture and the source code of each engine.

In the following part of this chapter, each engine is analysed and their communication mechanisms are explained in both conceptual and architectural level. After these individual analysis, game engines are analysed against the architectural solution and the theory behind it.
5.1 Blender

Blender is a multi-purpose toolset for 2D and 3D content creation which features range from 3D modelling and animation to video editing and scripting (Blender 2.6 manual – Introduction, 2013). Among its features is a built-in game engine called Blender Game Engine (hereinafter BGE) which provides basic functionalities that can be used with the Blender rendering software to produce fully playable games.

Major part of BGE is the Logic Editor, a part of Blender toolset that combines the 3D models and other objects of the game world and their related logic in to objects that closely resemble common game object (Blender 2.6 manual – Game engine logic, 2013). The logic of the objects can be either scripted or modeled via the Logic Editor tool that is part of the Blender editor. In either case, the basic concept of the logic is same:

An object is built out of logic brick, properties and state. Logic brick is reusable part of logic and can be either sensor, controller or actuator. The difference between these three is that the sensor can sense the object itself and the surrounding environment, controller can combine different impulses from different sensors and after using assigned logic with the impulses, can forward a new impulse to actuator, that can finally influence the object (by changing properties or state) and the environment. Actuators and sensors can communicate with each other either via direct messages or indirectly by actuator changing a part of the object or game world that a sensor is currently set to observe. This concept is showcased below in Figure 17. Note that in the figure, the changes that actuators cause and that are sensed by sensors are labeled as “Events” in order to simplify the figure.

![Figure 17. Communication logic of Blender](image)

The implementation of this logic inside the BGE follows quite closely to the concept shown above. In Figure 18 below, the architecture that implements the logic is shown. In the architecture, KetsjiEngine can be considered as the main game engine as it contains the actual game loop that runs the game while Scene is an organisational object that contains references to all the data that is currently shown and loaded and that also contains references to the high level managers such as LogicManager. On the bottom layer, ISensor, IController and IActuator are interfaces which handle the logic under the management of LogicManager and different EventManagers. In the figure below, only those parts of the architecture which take care of coordination and handling of logic are
shown and thus the figure does not include all the details. This also explains why in the explanation of figure written below refers to the ISensor, IController and IActuator interfaces: Those are the interfaces that describe how system works and thus they are the part of architecture that is required to be understood. While actual logic handling the actual processing can be implemented in multiple different ways, ISensor, IController and IActuator interfaces are used to manage the workflow.

During the execution of single loop of the game loop, the logic management is done in two sections. The first section occurs during the start of the loop when LogicManager calls every connected EventManager to scan the game world for changes. These changes are sensed by sensors connected to EventManager and if sensor senses the change it has been targeted for, it activates the controllers that are connected to it. Section concludes with LogicManager going through all the activated sensors and setting their corresponding controller as “triggered” to mark it for later activation.

Second section is during the middle of the loop. This time LogicManager traverses through all controllers which were set as “triggered” and calls them to evaluate their internal logic. If this evaluation returns true, controller sets its associated actuators as active. These recently activated actuators are then finally ordered by LogicManager to do the changes that they are set to do. It should be noted that while this description referred to changes in the game world, i.e., events, same logic can be used to messages. In this architecture, messages are just special kind of change in the game world.

For full details of communication logic execution in Blender, refer to Figures 25 and 26 in Appendix A.
5.2 Delta 3D

Delta 3D is a simulation and game engine that originated from military training requirements but ended up being a mature open source game engine (McDowell, Darken, Sullivan & Johnson, 2006). It can be considered as a collection of different open source projects as the Delta 3D relies heavily upon different open source modules and subsystems that mostly make the game engine. For example, Delta 3D uses currently OpenSceneGraph as a rendering subsystem, while GUI can be implemented as, for example, in Qt (Delta 3D – Features, 2005). Delta 3D software itself can be thought as implementation of various systems, which each handle own specific task, to form full-fledged game engine. A good example of this implementation strategy is the system called “Game Manager” which handles the interactions and connections between game objects (referred as actors in Delta 3D) and logic processes (referred as components in Delta 3D) (BMH Associates, Inc, 2005): Game Manager provides necessary functions to create game objects, provides communication method between them and provides a method to insert more abstract “game rules” without touching other subsystems such as rendering.
The communication logic incorporated in Delta 3D is shown in Figure 19. The basis of the system are messages that are stored by the Game Manager. These messages are the basic communicational unit not only between game objects but also between multi game manager instances and their processes. A subset of these messages are messages contain a specific event. In Delta 3D, events mean a specific action that has an effect on the game logic but not on other parts of a game. Game Manager stores the messages and forwards them to all components, to selected group of gameactors and to all global invokables. Each of these recipient group serves different purpose: Components contain some logical rules of the game world and can react to the received messages by their own internal logic, possibly sending more messages. GameActors, which represent single game objects and which are filtered on the basis of their set interest to specific type of messages, can activate Invokable function contained within them to respond to given message. Some of these invokable functions may also be set as global invokables which will receive all the messages but can only react to the messages that contain certain message type. It should be noted that messages can be either targeted to specific GameActor when they are created or the messages can be just sent for everyone. By targeting to specific GameActor, message can be sent to just a single GameActor but it will still be processed by components and global invokable functions.

Figure 20. Communication architecture of Delta 3D
The communication architecture that provides the above described functionality is shown in Figure 20 above. The architecture can be thought to work upon in two layers: First is the supporting layer which include GameManager, GMImplementation, GameEventManager and MessageFactory. These components handle overall management of the classes in second layer and provide the game loop functionality. Second layer consist of the Message, its supporting objects like Event and GameEventMessage and the objects that finally process it like GMComponent, GameActorProxy, GameActor, Invokable and InvokableFunction.

When GameActor or GMComponent wish to send a message, they can build the Message by requesting it from MessageFactory. If necessary, they can also register new game event to GameEventManager to extend the list of useful messages. After the message is fully prepared, either GameActor or GMComponent can send it to the global message list through GameManager.SendMessage() function.GameManager is instructed during the game loop to send the messages it has stored via GameManager.PrefFrame() and GameManager.PostFrame() functions. Both of these functions traverse the list of messages, albeit at different times of single game loop, by sending messages to components, interested game actors and global invokable functions. After sending specific message to all interested recipients, the message is terminated.

If message recipient is GameActor, the message is sent to the GameActorProxy that represent the actual GameActor. Reason for this is to create a common interface for all the different GameActors (BMH Associates, Inc., 2005). This interface has to roles: first it works with Delta 3D's editor, STAGE, to provide user friendly way to access and edit properties and second, it provides a common interface that the game engine understands and can use in creating and deleting GameActors and in relaying messages. Upon creation of GameActor, GameActorProxy can be used to register its interest in a certain kind of messages to GameManager and via this mechanism, GameManager knows which GameActorProxies and therefore which GameActors are interested in certain messages. Similar structure is used in GMComponent, although GMComponent does not need a proxy or message filtering mechanism as it gets automatically each message broadcasted.

The received messages are handled by GameActor and/or by GMComponent using the ProcessMessage function. This function can be used to implement more filtering by for example using switch statement. Finally, it should be noted that ProcessMessage function is not the only function that can be used when dealing with received messages as other functions inside GameActor can be registered to receive messages. These functions are structured as InvokableFunctions which are connected through Invokable interfaces and finally to GameActorProxy of GameActor.

For full details of Delta 3D's messaging sequence, see Figures 27 and 28 in Appendix A.

5.3 Gameplay3D

Gameplay3D is a game engine produced by Research in Motion (Research in Motion Inc, 2013). Unlike previous Blender and Delta 3D, it is far smaller and newer open
source project as it was started on late 2011. This has a significant effect on the amount of available documentation and support material on the design of the project as there are hardly any design documents available. However, Gameplay3D is still a full game engine with rendering, physics, audio, and other similar main functional components.

Another big difference between Gameplay3D and previously shown game engines is the overall aim of the engine as the Gameplay3D was designed to be small and easily understood software for smaller projects. This simplicity can be also seen in the implementation of the communication logic in the game engine as there are currently no overall communication system implemented in the engine. Closest to this kind of component is the communication system between non-player controlled entities which are called AIAgents according to the terminology of Gameplay3D engine. While this system is mainly built for allowing AIAgents, i.e., game objects to communicate between each other, the system is powerful enough to change the properties of game objects and influence other components of game objects.

The communicating logic behind AIAgent communication is shown in Figure 21. This logic is follows two very well-known patterns: publish/subscribe and state pattern (for description of patterns, see Eugster et. al., 2003 and Gamma, Helm & Vlissides, 1993). The publish/subscribe pattern is used with AgentListeners listening to AIMessages: each AgentListener listens to the messages that are addressed to the AIAgent it has been linked to. State pattern is used in controlling the logic behind each AIAgent: Each AIAgent has a number of different AIStates that it can be and each AIState can have multiple StateListeners that actually perform the actions when the active AIState changes.
It should be noted that while the names of each object refer to AI, these communication mechanisms can be used to perform actions other than AI related actions, such as influencing rendering and audio related properties of given game object. This is because AIAgent is a part of larger hierarchical structure of game object called Node and the StateListeners and AgentListeners connected to the AIAgent can get the reference to the Node which contains the AIAgent in question. This relationship as well as the architecture that allows above described functionalities are shown in Figure 22 below.

Compared to previously analysed game engines, the simplicity of the communication architecture in Gameplay is evident. While this simplicity can be also seen in the communication sequence during the message sending and in the main logic loop of the engine (see Figures 29 and 30 in Appendix A.), those functionalities are handled in very similar manner as in Blender and Delta 3D engines.

5.4 Polycode

Polycode is a small game engine which development started on December of 2010. Unlike other game engines analysed in this study, its development is not linked to any particular organisation as it has been developed as a hobby project. Like Gameplay, it lacks the similar level of documentation that Delta 3D and Blender have, especially since lower number of developers working on it. Still, it is a full-fledged game engine with rendering, audio, and physics components.

Polycode uses event-based communication methodology which is modelled after similar methodology used in Adobe Flash ActionScript (New York University, Game Innovation Lab, 2012). At both conceptual level (shown in Figure 23) and architectural
level (shown in Figure 24) it resembles the publish/subscribe pattern (described in Eugster et al, 2003): Events can be generated by EventDispatcher that sends these Events to the EventHandlers that are listening to it, i.e., EventHandles subscribe to different EventDispatchers that will publish the messages to the subscribed EventHandlers.

In Polycode, this event-based communication system is used in many places in the system, for example in input handling, thread managing and in timers. It can be also integrated to game objects (entities as per the terminology used in Polycode) handle messaging between them. However, while the event-based communication is heavily integrated to the system as there are many EventHandlers in the Polycode's core library and also many implemented as game objects, the event handling is done in rather straightforward way: Each time a EventDispatcher publishes a new Event, it directly sends the message to all the listening EventHandlers.

Figure 24. Communication architecture of Polycode
5.5 Comparison between game engines and the proposed architectural solution

While each analysed game engine had very different background, they had very similar structures and mechanisms incorporated within. These similarities help in understanding the role and functionality of events in modern game engines and thought this knowledge it is easier to analyse the major differences and similarities between the new communication-oriented theory showcased in this study and the general structure of modern game engine.

In this chapter, the comparison between the game engines and the new theory is performed by first showing the general findings of analysis of game engines, then summarizing these findings and lastly comparing these summarized findings to the proposed architectural solution. These findings were done by comparing the analysed game engine's communication concept, architecture and execution and the findings are shown in Table 3 below.

<table>
<thead>
<tr>
<th>#</th>
<th>Description of finding</th>
<th>Categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Different major processes were clearly separated and all were controlled by main manager</td>
<td>I General Structure</td>
</tr>
<tr>
<td>2</td>
<td>Each game engine used event-like communication</td>
<td>I General Structure</td>
</tr>
<tr>
<td>3</td>
<td>Three-quarter of game engines had their event management controlled by main loop</td>
<td>II Message Management</td>
</tr>
<tr>
<td>4</td>
<td>Three-quarter of game engines had a manager that managed event handling</td>
<td>II Message Management</td>
</tr>
<tr>
<td>5</td>
<td>Each game engine had event filtering system incorporated</td>
<td>II Message Management</td>
</tr>
<tr>
<td>6</td>
<td>Three-quarter of game engines had messaging only between game objects and components of game objects</td>
<td>III Game Objects</td>
</tr>
<tr>
<td>7</td>
<td>Each game engine used “Sender → Event → Handler” structure for event passing</td>
<td>III Game Objects</td>
</tr>
<tr>
<td>8</td>
<td>Each game engine had event handling as own component that could be added to different game objects</td>
<td>III Game Objects</td>
</tr>
<tr>
<td>9</td>
<td>In Blender and Delta 3D, messages and events were separated while Gameplay and Polycode only used events</td>
<td>IV Difference between messages and events</td>
</tr>
</tbody>
</table>

Findings can be categorized into four categories: General Structure, Message Management, Game Objects and Difference between messages and events. First three of these categories summarize the major similarities and differences between each game engine at different levels while the fourth category brings a new topic to the discussion about the event-based communication. Next, each category of finding is described and its ramifications are analysed against the new architectural solution.
First category, General Structure, contains findings that have to do with overall architecture of game engines. These findings show that all the analysed game engines shared the same common structure with one main managing process that contained the game loop that controlled the rest of the major processes and the components that contained those major processes. Another similarity was that all of them used event-like communication: while the structure that provided messaging service was different (discussed more deeply later on), the base concept and high level structure were very similar. This shows that the previously discussed game engine structure is actually used and based on this small scale study, it is common among game engines. This also means that there is strong evidence that the basic theory behind the proposed architectural solution and suppositions used in creating it are grounded in current knowhow of the industry.

Second category, Event Management, groups together findings about the methods how event creation, storage and distribution is handled. In this category, there was more differences among the game engines as Polycode engine does not contain own management component for event management, i.e., event manager while all the others do. While this does not mean that more advanced features like event filtering could not be built to the engine, this does have direct consequence on overall handling of events and the performance of the engine: While in Polycode each event can be created and forwarded to all the EventHandlers that are listening to the sending EventDispatcher at run-time as soon as event is created, in other engines the events are listed to event manager which then dispatches them during the execution of game loop. This means that Polycode engine can potential halt if too many events are created at the same time meaning that the gameplay would not be smooth anymore. In the other engines, likelihood of this happening can be more easily adjusted by changing how the event manager dispatches the messages during single rotation of game loop. It is possible to do the same in Polycode but that would entail that some kind of event manager would have to incorporated to the engine.

As three of the four analysed engines contain own event manager component that handles the events, the proposed communication-oriented theory seems to be true as it also contains separate event storage and own event handling component that takes care of event creation, event storage and eventual dispatching of events. However, the example component level architecture built upon the new theory (see Figure 16) proposed that event manager would not be linked directly with game loop as event manager could manage the events on its own. As evident from the analysed game engines that do have own event manager, each of these event managers were controlled by game loop as during each rotation of game loop, their “event dispatching” function was called. As the component level architecture shown in Figure 16 is an example, this difference in the game engine architectures can be explained as different architectures having different ways to implement same functionality.

The third category is Game Objects and the findings in this category have to do with how events were used by game objects. All the analysed game engines shared the same structure for event handling done by game objects: each of them implemented event sender and event receiver components that could be connected to and later executed by game objects. The difference was in the exact details of how these event sending and receiving functionalities were implemented as Blender used basically own components for these exact purposes (lActuator for sending, lSensor for receiving), Delta 3D
implemented them inside GameActor class, Gameplay relied on separate interface AIAgent::Listener and Polycode used fairly straight-forward publish/subscribe pattern with game objects and other core classes using the same event sending and receiving interfaces. These findings demonstrate that while the event-based communication is a common feature in modern game engines, the exact way how it is implemented is not standardized in the practise. That means that while proposed architecture theory with its separation of game objects, processes and events is similar to the current knowhow in the industry, the topic of implementing the actual architecture has not reached conclusion in the industry.

Last category of findings is *Difference between messages and events* which notifies the conceptual difference between said two messaging types that exists in both Blender and Delta 3D. The main finding in this category that both Blender and Delta 3D used messages and events for different purposes but which are conceptually still similar (see Figures 17 and 19 respectively). In Blender, there does not exist a class or classes that encapsulate “event” but rather in this study, the term event was used to define a change in game world that was caused by an actuator and sensed by a sensor. This is different to message as Blender contains one specific actuator and sensor pair (classes KX_NetworkMessageActuator and KX_NetworkMessageSensor) which specifically handle sending messages between game objects. In Delta 3D, the difference between messages and events is more clear-cut as there exist own classes for events and messages (GameEvent and Message respectively) and they have different uses in the engine: GameEvent are meant to represent simple changes in the state of the game world like when certain door is open or when a player has picked certain item (Alion Science and Technology Corporation, 2006) while messages are method to transport different data across the simulated world. More specifically, messages do not contain only information that is related to the logic of the game world, like events, but also information that is needed for simulation to occur. Some examples of these messages that simulation requires are different ticks, which inform when single frame has been passed or 'ticked', game object management related messages (when game object is created, deleted, updated etc.) and map loading messages (BMH Associates, Inc, 2005).

As can be seen how messages and events are treated by both Blender and Delta 3D, there are clear conceptual difference between them: While both terms are strongly linked to different parts of game engine and game world communicating each other, one of the terms refers to the container that holds the transported data while another one refers to some important change or happening in the game world. Interestingly, the terms messages and events have bit different roles in Blender and Delta 3D: In Delta 3D, message is a container of data that is broadcasted and then received by wide variety of game objects and other parts of game world while an GameEvent, i.e., event, refers to meaningful happening or change in the game world. In Blender, messaging is directed method for game object to send data across game world while event is, as defined in this study, change in game world that can be sensed by interested game object via specialized sensors.

This variety of terminology and small but important difference in terms does not show up in the proposed architectural solution and its accompanying theory as in there, events are defined as “temporary messages that processes can leave for other processes to act upon”. This leaves theoretical inconsistency as analysed game engines do show that there is difference between messages and events. However, the analysed game engines
indicate that at least in them, the connection between messages and events is not clear-cut: In Blender, messages are own structure that uses the same actuator-sensor mechanism, i.e. event-based communication mechanism for transporting data whereas in Delta 3D, messages provide the communication mechanism which can be used to transport events. This leaves the question remaining: what is the difference between messages and events and how this difference should be reflected on the communication theory of game engines and in the game engine architectures? This question is further discussed in following Chapter 6.1.
6. Discussion

The main goal of this study was reducing the gap between the game industry and academic research in game engine related literature. This goal was approached by studying game engine and its fundamental functionalities from the architectural viewpoint which led to development of new architectural model of the game engine. This communication-oriented architecture was developed from both academic and industry sources and validated against the real game engine in order to see the trustworthiness of the new architecture and the theory behind it.

During the initial research on the topic of communication in games and the development and evaluation of the proposed architecture many interesting topics rose. These topics reached from the details of the verification process to the usefulness of the new architectural theory for both industry and academic research. Many of these interesting topics need to be discussed further as they provide insight both to the developed architecture and to the general game engine research. For this reason, in this chapter, three main topics are discussed. These are the validity of the architectural solution and its theory as analysed in the Chapter 5.5, the implications of the architectural solution and its theory to the academic research and lastly the implications to the game industry. All three of these main topics consists of more detailed subtopics.

6.1 On the verification of the architectural solution and theory

As explained in the start of the Chapter 5, the analysis of game engines was done to verify how well the proposed communication-oriented game engine architecture and its accompanying theory accounts for communication inside modern game engine. While the results of this analysis were shown in Chapter 5.5, there was one topic that required to be discussed further and this topic was the difference between messages and events. In this section, this topic is discussed more deeply not only from the viewpoint of the analysis but also through the viewpoint of the academic research and game industry. In addition to this topic, also the critique against the analysis is presented in this section.

The topic of difference between messages and events was brought up in the analysis when the findings of the analysis indicated that two of the four game engines analysed, Blender and Delta 3D, had both conceptual and pragmatic difference between messages and events. In Delta 3D, messages were containers that allowed different data to be transported from the sender to any recipient that was interested in the type of messages while events (more specifically GameEvents as per Delta 3D terminology) were meant to be used by game developers to indicate special happenings in the game world that had some meaning to the overall game logic of the game (Alion Science and Technology Corporation, 2006). In Blender, messages were a communication method to send data to from game object to any game object that was interested in that kind of messages while events, as defined in this study, represented any change in the game world. It should be noted that the difference between messages and events ran deeper
than just among game objects as in Delta 3D, messages were part of the core game engine which were used not only in communication between game objects but also in performing key processes include the simulation, rendering and networking.

While events were discussed in Chapter 4.2, the difference between messages and events was not mentioned there. This is not to say that there is no difference, on the contrary, there is a distinction between these two concepts present in the software engineering literature: an event refers to any change in the status of the entity which can be signaled to other any number entities (Tarkoma & Raatikainen, 2006) while a message is certain method to allow this signalling to occur. The linkage between messages and events does not have to be one-to-one meaning that each change in the entity does not have to signaled outwards; single signal may be caused by certain pattern of changes, i.e. events (Metha, Medvidovic & Phadke, 2000).

In literature of games and game engines, the above mentioned distinction is not so clear. This is due to the different perspectives on games and their technology that the authors take upon the subject. These perspectives were mentioned and described in Chapter 3.1 and they were game world and real data perspectives. When the difference between messages and events is studied from these two perspectives, different results show up. From the game world perspective, where game objects are considered as the major components of the game, the concepts from the software engineering literature are used: message system is one kind of service that the game engine provides for game objects to allow communication among other game objects (Ocio & Brugos, 2008) and this communication is required in order to let game objects to interact, i.e. change their status according to the messages that they get from the game world and other game objects (Macal & Noth, 2010). Those messages which orientate from a change in game world, for example as from changing status of single game object, can be considered as messages that signals an event or events.

However, when messages and events are looked upon from the real data perspective where the major components of game are processes of the game, messages and events are very similar: they are temporary data that is written and read by any process. If the definition given by software engineering literature is considered from real data perspective, the distinction between messages and events can be considered as thus: messages are the data that is transmitted from process to process when the data accompanying the process has been changed in certain way. Thus, events would not be temporary data, they would only be the change in data that would cause the process to send messages.

Taken the above descriptions to account, the difference between messages and events does not have large ramifications in the communication-oriented game engine architecture but it does provide clarifications for the theory behind it. The major implication of this topic is that events and messages should be separated as two concepts: events refer to changes in the data, i.e. game world, that have an effect on process whereas messages refer to specific communication method that is used to transmit and transport the data linked to the event to any interested process. For the proposed architecture and its theory, this implication does not change the actual architecture but it provides a clarification on the theory and clearer distinction on what basis this theory and architecture works upon. The lack of change in the architecture is explained by the fact that as the architectural solution is based on events, i.e. changes in the game world, the exact communication method used to transport these events does
not have to be specified. Since the proposed architecture provides only high-level structure for the game engine, a message-based communication tool could be used to handle the transmitting the events and data.

After discussing the difference between messages and events, next the critique for the analysis is presented and discussed. There are two issues that stand against the validity of the analysis and they are *using open source games as source* and *breadth of selected source game engines*.

The issue of *using open source games as source*, has to do with how well open source games in general represent the current knowhow of the game industry. As the study used design science research as the research method, the analysed material should represent current knowledge of both industry and the academic research. Especially in this kind of topic where the major argument is that there is lack of knowledge in academic literature, the evidence shown from the industry needs to actually represent the knowledge, otherwise the comparisons between the evidence and literature are not valid.

To display the current knowhow of the industry, it was important the knowhow could be presented in unified manner that allowed comparison between architectures. Open source game engines were chosen, as discussed in Chapter 5, since the source code of commercial game engines is not commonly available and access to source code is necessary to construct the UML diagrams which were used in the analysis. As it was necessary to use open source games as the source for the analysis, the question is then that do open source game engines actually represent the knowhow of the industry?

During last decades open source game engines were commonly either made by amateurs or that they were old commercial game engines that were re-released with open source licenses (Wünche et al., 2005). However, nowadays many larger open source projects, including game engines, are run by organisations that consists either of people developing the project or of other organisations that support development of the project. This is also the case in the analysed game engines where three of the four engines were maintained mostly by large organisation (see Table 2 for more details). Thus, while the analysed game engines were not as commonly used as commercial game engines, their development was not lacking. There are differences between commercial and open source game engines due to different development environments but there is no evidence currently which would indicate that environment would have an effect on the core functionality and structure of the engine.

The second issue against the validity of the analysis is *breadth of selected source game engines*. One of the conclusions done in the analysis phase is that all four analysed game engines were very similar; they each followed the same structure, all used similar event management methods and overall architecture for events was similar. This homogeneity among game engines has been assumed in both academic and industrial game engine literature and while the analysis done in this study supports the assumption, the number of analysed game engines is too small to make a strong case to support or hinder the assumption. Increasing the number of analysed game engines would be optimal solution but due to the limited resources, only few game engines were analysed in this study. In future studies, it would be helpful to have larger selection of game engines as the target of analysis since that would help in creating stronger case of the game engine architecture.
6.2 Implications for game engine theory

The implications of the proposed architecture of game engine for the game engine theory are twofold. Firstly, as an architecture that explains one of the core functions of modern game engines and the structure that can be used to explain this functionality, the proposed architecture helps to ground the understanding of game engine in many levels. Secondly, it opens possibilities for future studies which can extend the knowledge of game engines further. In this section, both of these implications are discussed further and in more details.

The first implication, the grounding of understanding of game engines, has been called for by the academic researchers. In their article “The Case for Research in Game Engine Architecture” (2008), Anderson, Engel, McLoughlin and Comminos presented a list of five broad research topics that outlined some of the possible future research directions for the game engine literature. These five topics were terminology, boundary between game and game engine, effect of game genre on game engine, design dependencies between low-level issues and top-level design and lastly best practises in game engine design. When studying these broad research topics, common theme can be seen: there is a lack of knowledge regarding the general design, architecture, and role of game engine. By using this common theme, the research topics listed by Anderson et al. can be compressed to four topics: terminology, identification of game engine, genre-independent game engine and best practises. These four topics each focus on different, distinct topic but still support each other: Terminology provides academic theories and concepts that can be used in identification of game engine and lastly, by studying best practises currently utilized by game industry, the academic theories and concepts can be connected to the real-world applications. Each of these three topics can be used to study the implications that this study had on game engine research.

The first research topic studied is terminology. Due to the relative small number of studies about games and especially game engines, many of the common features, functionalities and components of games and game engines had not been named consistently. As this study required understanding of overall role and structure of game engine, a literature study was conducted to collect information from different sources and combine this information into solid theory behind game engine. As the game engines have often been developed quickly by the game industry, the terminology was partly based also on the sources from game industry to generate overall terminology.

The result of the literature study is shown in Chapters 2 and 3 and while the literature study is mostly aimed at describing the basic theory behind game engine, many of its key features and components are also named there. While this terminology is not most complete one by far, it does explain most fundamental concepts and terms that relate to game engine. The most important distinction done in the literature study was the separation of game engine from other parts of games. This was called upon also by Anderson et al. in their article (2008) as they found that there was no definitive consensus on the identification of game engine. While in their paper, Anderson et al. noted that there were signs of emerging consensus, the literature study conducted in this study showed that that the consensus has been at least partly researched: The game engine is a managing component that handles management of both processes which handle necessary computing occurring in games, such as rendering and physics simulation, and data these processes require. While this description is made using
terminology from data-driven software and thus does not describe all the features of

game engine, it is a stepping stone for next studies that can work more on separation

and description of each component and element of game engine.

While the topic of identification of game engine can be taken as theoretical issue where

the terms and definitions need to set, the topic of genre-independent game engine has

more to do with practical software engineering. Anderson et al. (2008) wrote that

genres, i.e. type of gameplay, affected both the definition and the design of game engine

pointing out that game engines were usually, at least during the time of the article,
genre-specific with specialized optimizations. However, results from neither of

literature study or the study of modern game engines supported this theory. On the

contrary, all the results pointed to strong core mechanisms that every game engine has
to offer. Some of these were provided by subsystems like rendering and audio but some
core ones, like the communication mechanism which was the topic of this study, were

provided by game engine. This is not to say that genres could not do affect game

engines at all but their effect may not be as strong as Anderson et al. Hypothesized in

their article.

The last of the four research topic studied is best practises. In their paper, Anderson et

al. (2008) described the research topic of best practises as being interested in the design

tools or architectural models that are common in game engines. As explained by

authors, the available literature during the time their article was written tended to

overlook these issues and focus mostly on implementing game components and

algorithms related to them. This trend on focusing on the implementation over design,
or as Anderson et al. refer it, “micro over macro architecture”, was also observed by

Ampatzoglou & Stamelos (2010). While understanding of implementation of certain

feature is naturally important, as a young research field, game engine research also

requires more focusing on larger overarching topics.

In this study, the topic of best practises was approached as a mean to bring established

industry knowhow to the academic research. The industry knowhow presented an

environment which could be used to understand, evaluate and evolve the theories from

the academic side. This work brought upon a new communication-oriented theory of

game engine which has three important ramifications on the academic research. Firstly,
it helps to explain the coordination and communication aspect of games and the role of

game engine in it on the theoretical side. Secondly, as the theory suggest that

management of communication and coordination are vital tasks of game engine, this

information can be used to link game engine to other similar software. By finding

similarities between game engines and other software, knowledge from both software

can be used to benefit each other. Thirdly, the theory provides a way to understand the

core feature of game engine and thus a way to analyse and develop game engines

further.

After going through the first implication of the proposed architecture and its theory, the

grounding of understanding of game engines, next implication is discussed. This

implication is called future research and it, as the name suggest, focuses on how the

proposed architecture and theory could be further studied. There are four future

research topics presented in this section: networking, multi-threading, implementation

and further comparative analysis.
One of the limitations set for this study was that this study focused only on single player game experience, not multiplayer or networked games (see Chapter 1.1). This limitation meant that when game engines and their theory were studied, only their core functions that they provide for single player games were studied. This limitation was put in place in order to direct the study to focus on core features of general game engine and as multiplayer and/or networking features are not considered as common features as, for example, rendering or physics, they were left out.

Interestingly, this limitation did not affect very much the available literature for the literature or the available selection of game engines for the analysis. Given the current trend of connectiveness between applications and services, game engines were broadly speaking rather independent and while they might offer networking services, those were implemented usually as own components. This was implied by both literature review and the analysis of open source game engines. This leads to interesting question on the effect that networking has on game engine architecture and especially the event-based communication architecture proposed by this study. As a well-known and commonly used architectural model for distributed computing, event-based mechanisms have been studied in networking both from the theory and from the implementation side (see for example Starovic, Cahill & Tangney, 1996; Meier, 2000; Tarkoma & Raatikainen, 2006). As game engines seem to use event-based communication as a core mechanism to handle communication and coordination, there is vast existing literature that could be studied and extended from the viewpoint of game engines. As an example of this, van Houten and Jacobs (2004) presented an architecture for distributed simulation games and this architecture used messages and events extensively. Architectural proposals such as van Houten's and Jacob's and more theoretical studies could help a lot in understanding modern game engines as part of larger network of games as well as the role of events in networked games.

Next future research topic is multi-threading. In the academic game engine literature, coordination of the major processes done by game loop is often discussed in terms of most computationally complex and resource consuming processes such as rendering, physics or AI (see for example Valente et al, 2005; Joselli et al, 2010). Results of this discussion often propose clearer separation of those processes by, for example, separating those processes to own threads and/or to use own specialized hardware and software solutions for handling the complexity. However, in this discussion the core mechanisms that game manager has to offer like communication between processes and management of game data are not brought up.

As discussed in this study, the core mechanisms of game engines like communication and coordination play a vital role in the overall management of game. While they may not be as complex and consuming as other major processes, they clearly take their own toll on the performance of a game. For this reason, it would be interesting to study what kind of effects multi-threading of other major processes would have on the core mechanisms of game engines. Another topic would be to study if the core mechanisms themselves could be implemented using multiple threads and what kind of effects that would have on the core mechanisms.

The third topic brought up by this study for future research is implementation of the event-based communication architecture proposed in this study. As much studied topic in software engineering, there has been many implementation methods described for
event-based systems (see for example Meier, 2000; Tarkoma & Raatikainen, 2006). These existing theories and systems could provide interesting background for implementing the proposed communication-oriented game engine architecture and work as a counter for the knowhow from existing game engines. Using both of these sources, the academic research and industrial knowhow, implementing event-based communication feature to game engine or possible modifying and extending existing one would provide new data for both sources.

Fourth and last future research topic is performing further comparative analysis. As discussed in the Chapter 5.5, while many of the analysed open source game engines did feature similar general structure and functionality as the proposed theory for game engine architecture suggested, none of them were implemented exactly as the theory would suggest. As the analysed engines only presented a small subset of all modern game engines, it would be worthwhile to continue doing the analysis on other game engines to find out more about the current state of the industry. This analysis could be potentially extended to other software that use event-based communication as core coordination and communication mechanism in order to gain wider understanding of the state of game engines against other event-based systems.

6.3 Implications for game engine industry

While this study originates from the research topics brought up by academic researchers and thus is more built around the academic theory of game engines, this is does not mean that the proposed theory would not have some implications on the industrial know-how. These implications revolve around the event-based communication systems and the developments surrounding them that could potentially help in alleviating their development work and enhancing their performance. There are three major implications discussed in this chapter and they are separating event management from game engine, tool support for events and distributed event management. Common theme on both of these implications is that they rely heavily on bringing established academic knowledge to the game industry.

First implication discussed is separating event management from game engine. As shown in this study, event-based communication is one of the core mechanisms of the game engine and thus event management is an important part of game engine. However, it is still often treated as small internal service offered by game engine to other larger processes. As mentioned in the Chapter 6.2 when speaking about the multi-threading as future research topic, these larger processes have often own devoted complex components that handle them. Some of these processes are extended to own subsystems and some are so advanced in both complexity and independency that the major processes are served as middleware to the game engine (Funge, 2008). It is worthwhile to ask why event management could not also be considered as a service offered by a specialized middleware linked to the game engine?

While event handling is very different from other processes that are usually handled by a middleware such as rendering, physics, and audio subsystems, to certain extent they are similar. For example, like other processes, not all game objects may require event receiving and sending services. A game object that is only meant to represent a static renderable image does not need to send or receive events, at least if game object loading...
and activation is done without events. As only certain game objects require event services, the whole event system could be set as a component of game object in game engines that use component-centric object systems. This is, to extend, similar to the architecture used in Blender game engine where event sending and receiving was handled by common interfaces available as components to game objects (see Chapter 5.1).

On the other hand, event-based communication can be seen as such an intrinsic part of game engine and as such cannot be externalized to middleware. In that case, it would be even more vital to understand how event management can be arranged to be part of game engine but still available for easy development, analysis, and improvement. This is linked to helping in the development of games: As events are core communication mechanism between game objects, event handling has to be visible, understandable and easy to use when developing games. In short, this topic has to do with tool support for events and it can be seen as larger topic of tools and game developing environments which was brought up to the academic discussion by Lewis & Whitehead (2011). They see this larger topic as one of the points of information exchange between software engineering and game industry. For game industry, they argue, there is a lot of potential gain in studying already established tools for event handling. For example, event systems such as OMG, CORBA and JEDI (Meier, 2000) provide mature examples which game engine developers could study, and possibly use, to improve the event management system of the game engines.

Last implication discussed in this study is distributed event management. This topic revolves closely around the implications discussed above but takes into account usage of events in distributed gaming. This is a broad topic and as such is not discussed at length here, partly also because there is currently lack of academic research on networking in games (see discussion on networking on Chapter 6.2). In software engineering literature, however, it is a widely discussed topic with a number of established theories and systems (see Meier, 2000; Tarkoma & Raatikainen, 2006). This academic knowledge could be helpful when new event systems are built for networked games.

Common task for distributed event systems is to manage event-based communication between different parts of large, networked system. In principle, this can be seen as being similar to networked games where each individual game run on own environment but need to inform other game instances on the actions occurring locally and need to gather information from other game instances. Due to this similarity, game industry could investigate on the current theories and distributed systems to see what new knowledge could be inserted to existing game engine knowledge.
7. Conclusion

In this chapter, this study is concluded by giving short summary of the whole study and by providing answers to the research questions. The aim of this study was to study how major processes and data required by them are separated in game engines and what kind of functionalities do game engines provide in order to coordinate the work of all of these different processes. This research aim has been called for by the academic researchers (see for example, Anderson et al, 2008) who see that the quick development of game engines caused by the competitive and demanding game industry has not been caught up by academic literature. This causes a divergence between academic literature and industry knowhow, especially when considering that while games are studied in academic literature from both usage and implementation aspects, certain core functionalities of game engines have not been studied much. One of these core functionalities is handling the coordination between the processes and it was this gap in research that the study was aimed to fill.

In this study, the issue of coordination in game engines was studied using the viewpoint provided by “coordination paradigm” (Papadopoulos & Arbab, 1998). In the coordination paradigm, software is seen as group of connected processes which are divided in to two groups: computing processes and coordinating processes that handle co-operation of computing processes. The method on how coordinating processes handle accomplish the co-operation is called coordination model (Ciancarini, 1996). Since knowing which coordination model was used in game engines would be important to understand how coordination between different parts of game engines occurs, the aim of this study was to find the coordination model. This search of coordination model was specified in following main research question and its two supporting sub questions which were used to guide this study:

- **RQ1.** How does the game engine function as a communication channel between data and processes?

  - **RQ1.1.** What kind of functionality and architecture game engines contain to provide necessary coordination?

  - **RQ1.2.** What kind of functionality and architecture game engines contain to provide necessary communication?

Each of these questions presented were studied and answered during this study following the design science research as the research method. This method and its usage are described in detail in Chapter 1.2. In short, the study was conducted in three parts: first, a narrative literature review was used to define the concepts and terminology required by later sections of the study. Second, by analysing existing coordination models, i.e., communication mechanisms and architectural solutions of game engines, it was seen that there was a demand for communication-oriented game engine architecture theory. By using the results of the analysis, this new theory was proposed and architectural solution based on it was built. In third and last section of this study, the
architectural solution was evaluated against established open source game engines in order to see the differences between proposed theory and current industry knowhow.

The demand for new communication-oriented theory of game engine emerged from the literature study and was further enforced by the analysis of existing communication mechanisms and architectural solutions. Both literature study and the analysis of existing solutions indicated that there was currently no established theory on how game engine functioned as the communication channel between processes and data. Literature study and analysis also indicated that this functionality was one of the core functions of game engines and understanding it was important. For these reasons, a new theory was proposed and analysed in this study. As the theory explained how communication among different components of game engine could be arranged, it also answered the RQ1.

The two subquestions, RQ1.1 and RQ1.2, elaborated the main research question by focusing on the two key functionalities required for the coordination to occur in software. These functionalities were coordination and the communication mechanism used by communication. In this study, the event-based communication mechanisms was found to be the communication mechanism used by game engine while the game loop inside game engine handled the coordination process. However, while both event-based communication and game loop are established concepts and studied both in academic and game engine literature, their co-operation and especially the architecture in game engines that handles that co-operation has not been studied. The theory of communication-oriented game engine architecture shown in this study can be seen as a first step in understanding this co-operation.

The name of this proposed game engine architecture theory is “Communication-oriented game engine architecture”. The key feature of this architecture is separation of processes and data and enabling communication between these separated parts. Architecture also allows communication between processes by using events. Main difference between it and existing coordination models is that the proposed architecture focuses on communication aspect and does not therefore affect the implementation of other parts of the game engine.
References


Appendix A. Structural and functional diagrams of analysed game engines

Figure 25. First part of Blender's logic loop – Sensor and controller evaluation
Figure 26. Second part of main logic loop in Blender – Actuator management
Figure 27. First part of message handling sequence in Delta 3D – message forwarding to GMComponents and global invokables
Figure 28. Second part of message handling logic in Delta 3D – message forwarding to recipient Actor and to Invokables registered to listen it
Figure 29. Main logic loop for Gameplay
Figure 30. Messaging sequence of Gameplay