About Game Engines and Their Future

Dario Maggiorini[1(✉)], Laura Anna Ripamonti[1], and Giacomo Cappellini[2]

[1] University of Milan, Milan, Italy
{dario,ripamonti}@di.unimi.it
[2] Italian National Research Council, Milan, Italy
giacomo.cappellini@idpa.cnr.it

Abstract. In these last few years we are witnessing an increasing adoption of video games in learning and teaching environments. This change is coming thanks to the fact that video games allow students to take a more active role in learning as they develop skills needed to succeed in their professional careers. At the same time, we are also observing a change in the way video games are implemented. Today, the existence of very large teams with a multi-layered organisation calls for the adoption of structured development approaches with associated environments. These environments have been baptised *game engines*. Availability and usability of game engines, in the near future, will positively influence educational activities for the next generations. In this paper, we discuss the general structure of modern game engines and put into question their current architectural approach. Our goal is to raise the attention of the scientific community on the fact that re-baptised software stacks are unlikely, on the long shot, to provide the flexibility and functionalities required by game developers in the coming years. After a detailed discussion of the possible problems on the horizon, an alternative approach for a modular and scalable game engine architecture will also be presented.

Keywords: Game engines architecture · Game development · Scalability · Distributed systems

1 Introduction

In these last years, the way developers implement video games is undergoing a tremendous change. At the beginning of video games history, a very small group – or even a single person – was usually in charge of software production. As a matter of fact, we can see that many block-buster games in the '80s such as *Pitfall!*, *Tetris*, and *Prince of Persia* carry the name of a single developer. Today, with the evolution of the entertainment market and the rise of projects with seven (or eight) figures budget, this situation is changing. In order to (*a*) better allocate competencies and effort, and (*b*) enforce code and resources reusability, video games are now developed on top of software environments defined as *game engines*.

Game engines, as largely discussed in [1], are usually organized as software stacks rooted in the operating system with an increasing level of abstraction

B. Mandler et al. (Eds.) IoT 360° 2015, Part I, LNICST 169, pp. 276–283, 2016.
DOI: 10.1007/978-3-319-47063-4_28

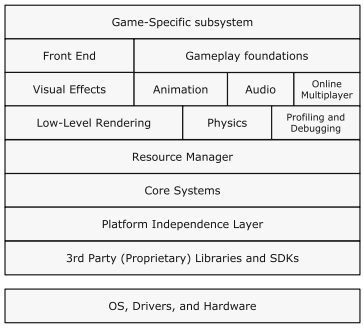

Fig. 1. Summary of a general game engine architecture.

layer-by-layer, up to describing game mechanics. See, as an example, Fig. 1. As it is easy to see from the picture, the adopted architecture is not really different from other solutions adopted in non-gaming environments. As such, given the different application field and performance requirements, a number of limitations may arise for large-scale game development. As we will see, three problems can arise with respect to the produced software: it may be monolithic, it is centralised and may not scale upward, and it may be platform dependent. Despite all the problems we just outlined, the final goal of this paper is not to prove that modern game engines are not fit to the task. Actually, they have demonstrated to be very good tools for the trade. Our aim here is to actually raise the attention of the scientific community that such tools (re-baptized software stacks) are unlikely, on the long shot, to provide the flexibility and functionalities that will be required by game developers in the next generations. We strongly believe that, in order to adapt to future evolutions, game engines should not just target better performances and advanced functionalities, but also provide more adaptable and serviceable internal structures.

2 Related Work

In the past, a fair number of scientific contributions has been devoted to the internal data structures of game engines. Nevertheless, at the time of this writing and to the best of our knowledge, only a very limited number of papers are specifically addressing the engine architecture. The majority of the literature seems to be focused on optimising specific aspects or services, such as 3D graphics (e.g., [2]) or physics (e.g., [3]). Issues related to portability and development have been addressed, among the others, by [4–6]. Authors of [4] propose to improve portability by providing a unifying layer on top of other existing engines. In fact, they extend each architecture with an additional platform-independent layer. Authors of [5] focus on development complexity and propose a solution based on modern model-driven engineering while in [6] an analysis of the open source Quake engine is performed with the purpose to help independent developers

contribute to the project. Other contributions try to improve performances by creating distributed implementations of existing engines [7–9]. Unfortunately, all of them aim to increase performances only for specific case studies by applying a distributed system approach to a specific internal service, such as the simulation pipeline or the shared memory.

As it can be observed, none of the papers cited above is pointing to a completely new architecture. Anyway, we must also mention that not all existing game engines have been designed as a library stack. For this, we can address the Inform design system [10] for the Z-Machine [11]. With Inform, the algorithmic description of a text adventure is compiled into a binary package. The binary package is, in turn, executed by a Z-Machine, which is a software available for many hosting platforms. Unfortunately, Inform is limited to text-based adventure games (such as *Zork*) and has never evolved toward modern interface technologies. Nevertheless, we believe that modern game engines should reconsider Inform and Z-Machine as a viable approach.

3 Background on Game Engines

Although game engines have been studied and perfected since mid-'80s, a formal and globally accepted definition is yet to be found for them. Despite this lack of definition, the function of a game engine is fairly clear: it exists to abstract the (sometime platform-dependent) details of doing common game-related tasks, like rendering, physics, and input, so that developers can focus on implementing game-specific aspects. To achieve this goal, game engines are usually divided into two parts: a tool suite and a runtime component. The runtime component assembles together all the internal libraries required for hardware abstraction and to provide services for game-specific functionalities. A variable portion of the runtime is usually linked inside the game or get distributed along with the executable. The tool suite is, on the other hand, a collection of external programs that can be used to manage all the data we feed to the runtime and to manipulate the runtime itself.

Since the tool suit is not part of the internal architecture, in this paper we are going to tackle only on runtime-related issues.

3.1 A Brief History of Game Engines

The first example of game engine dates back to 1984 with the game *Doom*. Doom was not intended to feature a game engine. Nevertheless, it has been designed with a number of software engineering best practices in mind. In particular, we could find a strict separation between software modules and clear distinction from software and data assets. Moreover, a development approach strongly oriented to code reusability was enforced all over the project. To actually see the concept of game engine popping out, we had to wait until mid-'90s for the release of Quake Engine (by IDSoftware) and Unreal Engine (by Epic Games). Starting from this point, games opened up to user customisation and, most important

of all, their engines have been regarded by software companies as a separate product to be sold to game developers. As of today, we have companies making games and selling their internal technology (like Epic Games) competing with companies focused only on distributing and supporting a game engine platform (Such as Unity Technologies).

Of course, game engines have not been immune from the *open source movement*. Many open source projects exist; they can provide specialised functionalities (e.g., Ogre3D) or a full application stack (e.g., Cocos2D). On the other side of the spectrum there are proprietary game engines, which are kept secret by gaming companies and are usually born around a specific project. Their architecture can be only guessed. Nevertheless, it is known that engine-focused companies usually fish their architects from gaming companies. Since they are also proposing us a library stack, one may presume that proprietary/private engines are following the same philosophy.

4 Potential Shortcomings of Current Game Engines

As already briefly presented in Sect. 1, creating a software by deploying code on top of a library stack may lead to potential problems when trying to create a video game. In this section we are going to look into these problems in detail and discuss what should be done to overcome them.

4.1 Monolithic Software

Being monolithic means that a gaming software is completely self-contained and developers must rebuild/relink the while project at every change. This global rebuild is required also because, as already mentioned, a portion of the engine is usually distributed together with the game. Since, in modern productions, the size of source code and assets is growing exponentially, a global rebuild might become a significant bottleneck for large projects. Even in small projects, being able to perform global rebuild means to have a clean and well synchronised source tree and requires good coordination among developers to avoid build breaks.

We should provide a way for developers to modify and extend games by means of a plug-in approach with a very fine granularity, even at runtime. This way, only new and changed functionalities will require compilation as standalone components. Moreover, a malfunctioning component is not going to compromise the entire project for other developers.

Technology exists and is already available for a program to load binary code on demand at runtime. All modern languages feature dynamic class loading; moreover, dynamic libraries management facilities are already included in mainstream operating systems. It is technically possible to compile the object code for a game item (only), have the engine recognise it, and finally use the class loader to draw the code inside to be immediately available in the game.

4.2 Centralised Solution

Centralised software is usually difficult to scale upward. Scaling up is required every the computational power required by a software component is exceeding the capacity of the hosting system. This is usually achieved by shedding the offered load between multiple machines. In particular, it is a desired behaviour for online games where many users connect simultaneously to the same (server) engine. In current implementations, the game itself must be aware of the location of a service (on the same machine or elsewhere) and implement a distributed computation. Considering the recent trend in online gaming [12], especially with MMOs (Massively Multiplayer Online games) and MOBAs (Multiplayer Online Battle Arena) ruling the market, the possibility to transparently relocate a component without restarting the server – and interrupting the service – will be an incredible benefit for next generation games. Just consider that the top-tier MOBA *League of Legends* reported, in 2014, 7.5 millions concurrent players at peak time.

Inside a library stack, modules communicate by means of function/method calls. While remote function call is possible in many ways, it is usually bound to a language (e.g., Apache River, only for Java) or a platform (e.g., Microsoft DCOM). An high performance messaging system between modules provided by the engine itself should be considered instead. The engine can simply remap an address from local to remote and vice versa when a component is relocated.

Technologies implementing lightweight and high-performance messages exchange are already available (see, e.g., mbus [13]). Of course, performances may be an issue when relocating a module over the network. Anyway, we already have many time-critical services provided over local area networks with good results (e.g., iSCSI [14]). Moreover, we must remember that relocation is usually performed across a dedicated infrastructures and not over the Internet; cross-traffic is definitely not an issue here.

4.3 Platform Dependency

Even if the engine claims to be multi-platform and uses its lower layers to manage adaptation for vendor-specific hardware, seamless deployment across platforms is not always guaranteed. This behaviour may depend on a number of causes: from undocumented/proprietary APIs preventing actual porting to loss of performances due to a library optimised only for specific hardware. Today, to address this issues, developers are required to write code that, inside the same engine, behave – or compile – differently, based on the underlying platform. As a result, the engine is technically cross-platform, but developer skills and code are likely to be diversified between deployment platforms. Needles to say, this is going to negatively impact team management and production time in many ways.

Once again, a modular architecture may be a viable solution to address the problem we just outlined. Using messages for inter-modular communication can also be beneficial since they can be platform independent and are easier to standardise. Nevertheless, this issue is not purely technical and involves marketing

policies from hardware vendors. As a matter of fact, in many cases only vendors can write device drivers, as hardware interface are undisclosed for technical and opportunity reasons. At least, an approach based on software plug-in will help developers to easily modify, upgrade, or deactivate problematic and unsupported software modules.

5 An Alternative Approach

As already discussed in the previous section, potential solutions for each of the envisioned problems are already available. Putting all these solutions together can lead us to propose an alternative architecture for game engines.

We already know that this new architecture should be modular and provide a messaging service between modules. These modules must not be only game-specific but may also implement engine internal services. To achieve this, the engine needs just to implement some sort of sandbox where modules can be efficiently swapped in and out at runtime, plus the messaging system and a basic soft real-time scheduler. Moreover, the messaging system can also be distributed and link engines on different machines to perform transparent load shedding. An existing library stack could be easily transformed in modules inside one or more engines. If we need to preserve the library hierarchy, a message policy subsystem can be easily added.

What we just exposed lies in the middle between a runtime environment (like Java or CLI) and a micro-kernel operating system (such as Amoeba or QNX) with added distributed functionalities. See Fig. 2.

As a matter of fact, since our solution can be obtained from the combination/adaptation of existing technologies, a microkernel game engine seems to be actually implementable. Nevertheless, questions remain open about performances loss and willingness of companies to adopt it. While it is difficult to argue about performances without a working prototype, some considerations can be drawn on a perspective adoption. From an industrial perspective, there is only

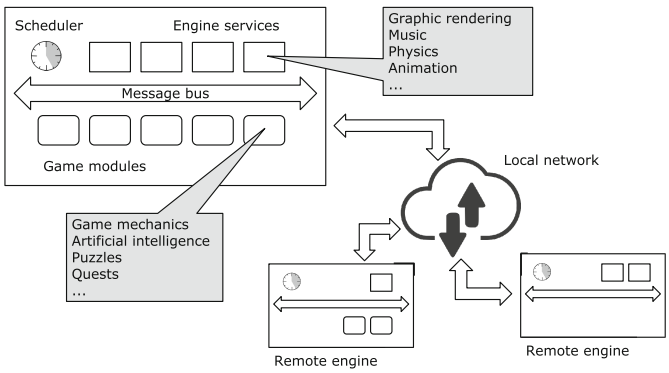

Fig. 2. Possible architecture for a microkernel game engine.

one undesirable constraint: in order to implement an efficient dynamic loader, a reflection/introspection enabled language is required. As of today, for performance reasons, the main development language is in many cases C++, which does not natively support dynamic class loading. It is possible to add dynamic class loading to C++, with a significative loss in performances. As a result, it will be difficult that the changes we envision will take hold in corporate production pipelines with the current generation of developers, unless the scientific community will be able to provide a solution for a dynamic management of classes inside a sandbox based on C++.

6 Conclusion and Future Work

In this paper we analyzed the current approach for the internal structure of modern game engines. Our intention was to raise the community attention on the fact that this approach is unlikely to provide the flexibility and functionalities required by game developers in the next generations. We believe that, by developing a game on top of a library stack, the resulting software may suffer from being monolithic, centralised, and platform-dependent. We discussed all these issues in detail and it seems that solutions are already available by switching toward a microkernel-like architecture.

In the future, we are planning to create a prototype game engine based on the envisioned architecture and verify if it can be valid substitute for current engines with respect to performance and functionalities. This will be useful to actually deploy widely distributed games such the ones envisioned in [15].

References

1. Gregory, J.: Game Engine Architecture. A. K. Peters/CRC Press, Boca Raton (2014). ISBN 978-1466560017
2. Cheah, T.C.S., Ng, K.-W.: A practical implementation of a 3D game engine. In: International Conference on Computer Graphics, Imaging and Vision: New Trends (2005)
3. Mulley, G.: The construction of a predictive collision 2D game engine. In: Proceedings of the 8th EUROSIM Congress on Modelling and Simulation (2013)
4. Darken, R., McDowell, P., Johnson, E.: Projects in VR: the Delta3D open source game engine. IEEE Comput. Graph. Appl. **25**(3), 10–12 (2005)
5. Guana, V., Stroulia, E., Nguyen, V.: building a game engine: a tale of modern model-driven engineering. In: Proceedings of the 4th International Workshop on Games and Software Engineering, GAS (2015)
6. Munro, J., Boldyreff, C., Capiluppi, A.: Architectural studies of games engines the quake series. In: International IEEE Consumer Electronics Society's Games Innovations Conference, ICE-GIC (2009)
7. Xun, W., Xizhi, L., Huamao, G.: A novel framework for distributed internet 3D game engine. In: Proceedings of the Third International Conference on Convergence and Hybrid Information Technology, ICCIT 2008 (2008)

8. Gajinov, V., Eric, I., Stojanovic, S., Milutinovic, V., Unsal, O., Ayguade, E., Cristal, A.: A case study of hybrid dataflow and shared-memory programming models: dependency-based parallel game engine. In: 26th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD (2014)
9. Huiqiang, L., Wang, Y., Ying, H.: Design and implementation of three-dimensional game engine. In: Proceedings of World Automation Congress, WAC (2012)
10. Nelson, G.: Natural Language, Semantic Analysis and Interactive Fiction. Whitepaper (2006)
11. The Z-Machine Standards Document version 1.1 (2014). http://inform-fiction.org/zmachine/standards/z1point1/index.html. Accessed 7 Aug 2015
12. Gerla, M., Maggiorini, D., Palazzi, C.E., Bujari, A.: A survey on interactive games over mobile networks. Wirel. Commun. Mob. Comput. **13**(3), 212 (2013)
13. Ott, J., Perkins, C., Kutscher, D.: A message bus for local coordination, IETF RFC 3259 (2002)
14. Satran, J., Meth, K., Sapuntzakis, C., Chadalapaka, M., Zeidner, E.: Internet small computer systems interface (iSCSI), IETF RFC 3720 (2004)
15. Maggiorini, D., Quadri, D., Ripamonti, L.A.: Opportunistic mobile games using public transportation systems: a deployability study. Multimedia Syst. J. **20**(5), 545 (2014)