SMASH: a Distributed Game Engine Architecture

Dario Maggiorini*, Laura Anna Ripamonti*, Eraldo Zanon*, Claudio Enrico Palazzi† and Armir Bujari†

*Department of Computer Science, University of Milano, Italy
†Department of Mathematics, University of Padua, Italy

***Abstract*—In these last few years we are witnessing a change in the way video games are implemented. Starting from an early age, where a single developer was sometimes in charge of the whole creative process, we have moved now toward extremely large groups with a multi-layered organisation. This increasing complexity of team organisation, together with a tremendous growth of projects size, calls for the adoptions of development approaches leveraging on scalability and distributed computing environments. Unfortunately, today's development and execution environments for games (usually called *game engines*) are suffering from a number of architectural constraints. As a result, we strongly believe current engines will not be able to provide the flexibility and scalability required by game developers of the next generation. To overcome the above limitations, in this paper we propose SMASH (*Stackless Microkernel Architecture for SHared environments*): an architecture where a game engine is decomposed in several dynamic and independent software modules interacting via a microkernel-like message bus. Game modules can just be inserted, debugged, and removed from a running engine once its internal messaging protocol is clearly defined. Moreover, following this approach, game modules can be dynamically dislocated on multiple machines to obtain a truly distributed, scalable, and fault-resilient system where adaptation can be achieved mostly without downtime.**

I. 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 every aspect of software production. As a matter of fact, we can see that many block-buster games for home entertainment in the '80s such as Pitfall! [1], Tetris [2], and Prince of Persia [3] 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. Video game creation is now a collaborative effort of tens – or even thousands – of programmers organized in groups and assigned to specific tasks. Of course, the increasing complexity of team organisation and the tremendous growth of projects size force gaming companies to seek development approaches leveraging on scalability and distributed computing environments. In order to better allocate competencies and effort and enforce code and resources reusability, video games are currently implemented by means of software environments called *game engines*.

A game engines, as largely discussed in [4], is usually organized as a software stack rooted in the operating system with an increasing level of abstraction layer-by-layer, up to a point where game mechanics are described. Adopting this kind of architecture as a development and execution environment for large scale games may lead to a number of constraints, as already presented in [5]. To summarize, a video game may suffer from the following limitations: it may be monolithic, it may be centralized and difficult to scale upward, and it may be platform dependent. Despite all the aforementioned limitations, modern game engines stands to be very good tools for the trade. Nevertheless, it is legitimate to call into question how long current engines (re-baptized software stacks) are likely to provide the flexibility and functionalities 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. For this reason we are proposing here a distributed game engine: *SMASH* (Stackless Microkernel Architecture for SHared environments). With SMASH, a game engine is decomposed in several dynamic and independent software modules interacting with each other via a microkernel-like message bus. This way, game modules can just be inserted, debugged, and removed from a running engine once its internal messaging protocol is clearly defined. Moreover, modules can also be dynamically dislocated on multiple machines in order to achieve a truly distributed, scalable, and fault-resilient system where adaptation can be achieved mostly without downtime. By following the SMASH approach, the development efforts shift from coordinating code production to coordinating functionalities. With *coordination* we mean the global organisation of game modules and features belonging to the hosting game engine.

The remainder of this paper is organized as follows. In Sec. II related work on the same topic we address is presented while in Sec. III background information about game engines is provided. Section IV provides all details about the SMASH internal architecture while Sec. V offers a discussion about its implementation and a demo application. Sec. VI concludes the paper and proposes future work.

II. 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., [6]) or physics

(e.g., [7], [8]). Issues related to portability and development have been addressed, among the others, by [9], [10], and [11]. Authors of [9] 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 [10] focus on development complexity and propose a solution based on modern model-driven engineering while in [11] an analysis of the open source version of the 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 [12]–[14]. 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 simulation pipeline or 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 mention the Inform interpreter [15] for the Z-Machine [16]. With Inform, the algorithmic description of a text adventure is compiled into a binary package. This binary package is, in turn, executed by a Z-Machine, which is a software available for many platforms. Unfortunately, Inform is limited to text-based games (such as Zork [17]) and has never evolved toward modern interfaces technology. Nevertheless, we believe that modern game engines should reconsider Inform and Z-Machine as a viable approach.

III. 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 such as rendering, physics, and input management, so that developers can focus on implementing game-specific features. A summary of a standard architecture for modern game engines can be observer in Fig. 1. As it is easy to see in the picture, the adopted architecture is not really different from other solutions embraced in non-gaming environments where a layered abstraction is required. The lowest layer is an interface to the (sometimes proprietary) hardware. Going up, the next layers provide first access to undisclosed APIs, then a degree of platform independency, and finally game-related services. In game-related services we can find core services such as memory management and IPC and, further raising the level of abstraction, libraries whose function is actually perceivable by the used such as the graphical interface, the physic simulation, or the audio subsystem.

To achieve their goals, 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 provides services for game-specific functionalities. A portion of the runtime is usually linked inside the game or get distributed along with the executable. The tool suite is usually a collection of external programs that can be used to manage all the data feed to the runtime and to manipulate the runtime itself.

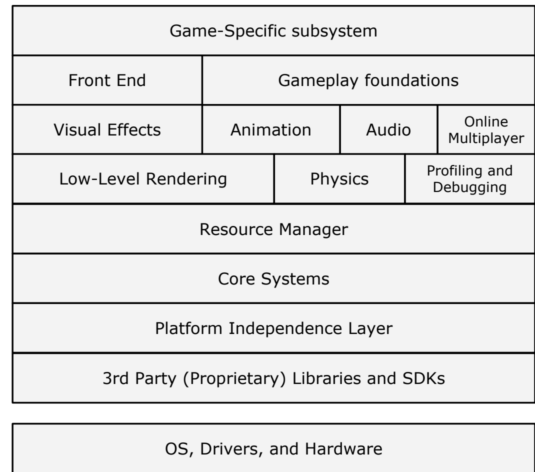

Fig. 1. Summary of a general game engine architecture.

A. Potential Shortcomings of Current Game Engines

As already mentioned in Sec. I, the architectural model employed by modern game engines may suffer from three shortcomings. Let's summarize them briefly.

First, a modern game is often a monolithic piece of software. Being monolithic means that developers must rebuild/relink the whole project at every change. For huge repositories a global rebuild might become a significant bottleneck. Anyway, even in small projects, global rebuilding requires a clean and well synchronised source tree together with good coordination among developers to avoid build breaks.

Second, game functionalities are usually centralized: all software performing computational activity sits on the same machine. In particular, online games clients connect to and depend on a centralized authority (the game server) in charge for data consistency and to update the system status. Considering the recent trend in online gaming (see e.g., [18]), especially with MMOs (Massively Multiplayer Online games) and MOBAs (Multiplayer Online Battle Arena) ruling the market, the capability to distribute workload over multiple nodes is paramount for the stability of a gaming network infrastructure. Centralized services are usually hard to scale upward and perform poorly with high workload. 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.

Third, every game exposes some degree of platform dependency. Even if an engine claims to be cross-platform and uses its lower layers to adapt to vendor-specific hardware, seamless deployment across multiple platforms is not always possible. This behaviour depends on a number of factors: from undocumented/proprietary APIs to loss of performances due

to optimisation for a specific hardware. Today, developers are required to write code that, inside the same engine, behave – or compile – differently, based on the underlying platform. As a result, an engine may be technically cross-platform but developers' skills and code are likely to be diversified between different platforms.

IV. The SMASH Architecture

In this section we are going to present the SMASH architecture in detail.

To devise our architecture we took inspiration from the word "*engine*" itself. An engine, as describe by Merriam-Webster is "*a machine for converting any of various forms of energy into mechanical force and motion*". Then, a game engine should also set a game in motion rather than being part of it. Game developers should put energy (data assets) in an engine and describe the way the machinery should move (game rules) rather than recombining (relinking) every time the runtime component with the game rules to obtain a standalone software. This kind of approach sees a game engine much closer to a runtime environment (see, e.g., Java [19] and CLI [20]) rather than a library stack. A SMASH engine is basically an execution environment providing three basic functionalities: (i) a soft real-time scheduler, (ii) a dynamic game modules manager, and (iii) a messaging system between modules. A diagram of the SMASH architecture is reported in Fig. 2. The proposed solution is designed taking inspiration from microkernel architectures (such as Amoeba [21] or QNX [22]) and is conceived to offer a solution to the issues described in the previous section.

The soft real-time scheduler is in charge to timely call scheduled functions and make sure the system is evolving at the right pace. The scheduler will just invoke methods in scheduled modules but will not take part to the evolution.

The game modules are independent entities providing gaming functionalities. Modules are not required to be game-specific but may implement general purpose engine services (such as graphic rendering or physic simulation). The possibility to swap in and out modules at runtime will allow developers to modify and extend games with a plug-in approach, down to a very fine granularity. The resulting games will not be monolithic but a composition of independent and replaceable modules. New and changed functionalities can be compiled as standalone entities and then loaded for testing and debugging in a running environment. Compilation errors in a module are not going to produce a build break and delay the work of uninvolved developers. Technology 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 use the class loader to pull the code inside to be immediately available inside the game.

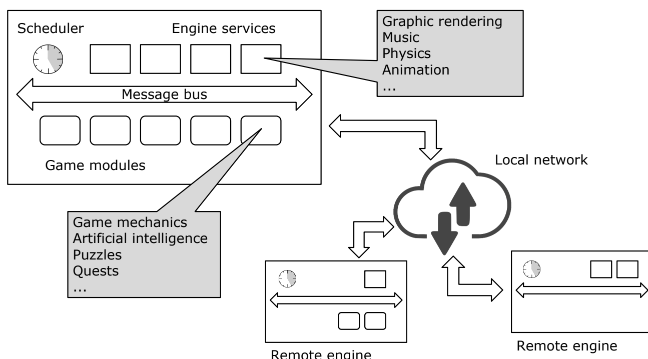

Fig. 2. Diagram of the SMASH architecture.

The message bus is actually the heart of the our architecture as it is in charge to implement communication between modules. Communication between modules is implemented by means of function calls. This approach is feasible since all modules will be sharing the same execution environment and allows to achieve good performances since parameters marshalling is not required. Each software module will uses the message bus to call service functions in other modules. As a matter or fact, these function calls will not be direct: a generic proxy function will be called on a shared object belonging to the message bus library. When calling the proxy function, the caller will specify a target module, a target function name, and all parameters. The proxy will then build a specific function call starting from its parameters. In modern languages, reflection can be used to achieve this goal easily and in an architecture-independent manner. Using a proxy also allows to perform additional sanity checks and to verify the existence of target modules and functions. In the case there is no target available, the proxy will just discard the call.

Of course, the message bus may not be limited to the local machine. In fact, a message bus can open a network connection and link to other buses belonging to remote game engines. This way, it is quite easy to build a real distributed system. In order to actually distribute our engine over a network all we have to do is to extend the proxy function of the message bus. In this extended proxy function, the destination of the call will be a target module together with a target engine. If the target module is not connected to the local bus, parameters marshalling will be performed and the call is converted in a message (a MessagePack [23] array) to be sent over the network. The destination of the network message will be a remote message bus in charge to decode the request, perform the (now local) call, and return the result to the caller using a reply message.

An interesting feature of our architecture is that we can start multiple instances of a module on different engines. By replicating the same module around the network we can implement high availability through fault tolerance. If a node hosting a module experiences a fault, the other copies will still be available. Each message bus can keep a list of remotely available modules and switch to another entry when the target is unresponsive or round-trip-time increases too much.

The architecture we described so far can also be very

dynamic. The overlay topology can be changed just by adjusting message buses' configuration. Whenever a system is overloaded and more scalability is required, it is very easy to start another engine on a new machine, relocate (or duplicate) some game modules, and link the new engine to the existing infrastructure. This way, the computational load will shed over one more node. In particular, this operation can be performed live, without any downtime.

Finally, we must also point out a couple of issues related to the infrastructure management. It is very important, to achieve optimal load balancing and reactivity to faults, to have a correct topology information (including game modules) on every node. To keep the addressing system in sync, each message bus should advertise immediately every change in its available modules to all connected peers. It is also reasonable to send periodic refresh messages, which can also work as keep-alive messages for the hosting nodes. With respect to peer discovery, nodes' address can be hard coded in the configurations (suitable for tunnels over the Internet) or unassisted (especially on local networks) using, e.g., SSDP (*Simple Service Discovery Protocol*) [24].

V. Implementing SMASH

After defining an architecture, we have to start taking decisions about which technologies and practical approaches should be adopted to implement SMASH.

A. Selecting an Language

The first phase is to select a reference language. This is an important step since the language will potentially decide the level of adoption of the game engine. Today, the most adopted object-oriented languages in game engines are C++, followed by Java, and then C#.

Our final decision was to select C# for a number of reasons. Java is a good option for a multi-platform standpoint, nevertheless it has been discarded mainly because it provides low performances, especially when dealing with garbage collection. C++ is definitely the most used language in the gaming industry due to very good performances and its capability to access low level hardware. Unreal Engine [25] version 4 is regarded today as the most significant commercial product using C++. Unfortunately, C++ does not provide reflection natively; this can make it difficult to provide an efficient proxy functionality in the message bus. Reflection can be added at user level, but with a significant performance degradation. Moreover, C++ is also not providing a runtime class loader. There are third-party implementations of middlewares to load classes at runtime, like the one provided by DCOM [26]. Unluckily, DCOM is a proprietary Microsoft technology and does not bode well for cross-platform implementation. C#, on the other hand, is capable of reflection and dynamic class loading. Moreover, it is quite well established thanks to the Unity [27] engine, which is currently one of the big players on the market. As added values, C# has also a gradual learning curve and, thanks to the CLI [20] intermediate language, can inter-operate with a number of other object-oriented programming languages.

B. Runtime Performances Analysis

Once we selected a reference language, the second step is to make sure the underlying execution environment can provide enough performances to satisfy the requirements of a gaming application even when calls are mediated by the message bus proxy. The use of a proxy function will introduce an overhead on both local and remote calls.

To estimate the proxy overhead, we performed extensive tests by defining a dummy function accepting two floats as parameters and returning their sum. The dummy function is called 10 millions times and the average execution time is calculated. The host machine is a PC with Intel i5 2410 processor running at 2.3 GHz and 8GB DDR2 Dual Channel RAM at 667 MHz. The host operating system is a 64 bit Windows 7 SP1 with .NET version 4.5.1 installed. No other applications are running during the tests.

The benchmark for our tests has been calculated using a direct call without any proxy. In such case, the average execution time is 7.8 nanoseconds.

If we add the proxy to the picture, things get more complicated. The experienced overhead is not just the added time to perform an additional call but also a management time introduced by the execution environment. When a software module is loaded at runtime, the execution environment places it in a separate container (called *domain*) sharing the same address space but with dedicated access privileges. Cross-domain calls require additional management form the operating system and the .NET middleware. As a matter of fact, the average execution time ramps up to 211.4 nanoseconds (27 times the benchmark). Despite the fact that overhead proved to be considerable, the system is still more than able to provide a good user interaction.

Execution time is obviously going further up when accessing a game module on a remote engine. To add networking to the picture, we use both TCP and UPD to perform a call between two modules on the same host. This way, we can estimate the overhead generated by execution middleware and operating system. Excluding transmission delay is reasonable in the evaluation, since the network behaviour will not depend on the software implementation. With this setup, observed execution times are 63.1 microseconds using TCP and 56.4 microseconds using UDP. As it can be observed, performances are around 10,000 times the benchmark and 370 times a cross-domain call. Nevertheless, we can still run our system with 15,000 synchronous interactions per second. As a result, performance loss is sustainable and our architecture seems to be up to the task.

As already mentioned, the delay introduced by the network does not depend on the software infrastructure as long as we use protocols implemented in the operating system. In a well connected datacenter we may expect an additional millisecond for each traversed link. Of course this is going to have a perceivable impact on the overall performances. Anyway, in a datacenter this transmission delay is expected to be fairly constant and bounded to a few links. Moreover, we

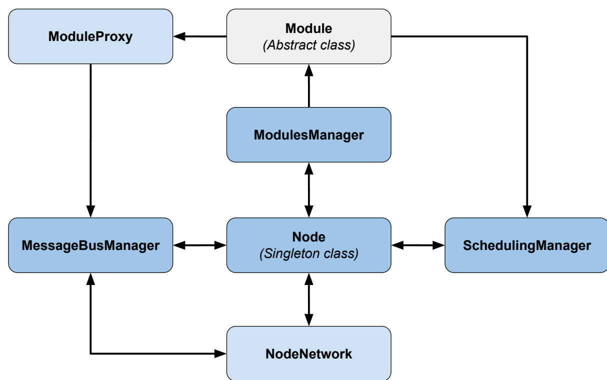

Fig. 3. SMASH prototype class diagram.

are expected to address this from an architectural standpoint: identifying sets of game services distringuished by similar time constraints and distributing them accordingly.

C. Prototype's Software Architecture

As a last step, a smash prototype has been implemented following the diagram reported in Fig. 3.

As we can see in the picture, there are a number or service classes: one for every basic function in the engine.

The two main classes are *Node* and *Module*. The Node class is responsible for managing the engine: it instantiates all other classes and takes care of configuration settings. Given its function, Node is implemented as a singleton class. The Module class, on the other hand, is an abstract class that developers must extend in order to create game custom modules. Inside this abstract class, all functionalities to register and de-register from the engine are already provided. Many instances of classes extending Module can be associated to the Node singleton. Association is performed through the *ModulesManager* class. The ModulesManager is in charge to load and unload modules in the engine. Modules loading and unloading has been implemented using the file system. ModulesManager monitors a specific directory; when a new package (a ".dll" or a ".exe" file) is added to the directory ModulesManager will inspect the file content and load all classes extending Module in a dedicated domain. Vice versa, when a package file is moved out of the directory, all modules associated to its domain are unloaded.

Connected to the Node class we also have classes to manage the other two basic functions: scheduler and message bus. The *SchedulingManager* class keeps an high resolution timer and accepts scheduling requests coming from modules. When a scheduling is due, the SchedulingManager will notify the Node singleton which will perform the call. Calls between modules, as already described, will take place through the message bus managed by the *MessageBusManager* class. When a class extending Module needs to call a function in another module, it will use the *ModuleProxy* class as an intermediary. The ModuleProxy class will relay the request to the MessageBusManager taking care to determine if the callee is in a local or remote Module. Based on directions from the ModuleProxy, MessageBusManager will perform the call locally or over the network.

All network operations must pass through the *NodeNetwork* class. NodeNetwork takes care of managing links to other engines, send and receive call messages, and receive notification for modules change from other engines. In particular, incoming call messages are forwarded to the MessageBusManager for local delivery while remote modules updates are relayed to the Node singleton in order to update its data structures. These data structures will be used by MessageBusManager to devise the address of a remote module. On the other hand, when the local modules list is changing, the ModulesManager will notify NodeNetwork through the Node singleton and an update will be sent to all linked engines.

D. Demo Application

In order to test our SMASH prototype, we implemented a distributed version of a puzzle game: a simulator for the Rubik's Cube.

In this game we are using three PCs; each one is running an independent copy of the SMASH prototype. The three engines are linked to each other in a full mesh. On the first PC there is a main engine, which is in charge to simulate the cube configuration using a dedicated module: (*RCSimulator*). On the second PC the engine holds three modules: an *InputManager* module to manage commands from the user, an *RCConsoleRenderer* module to show the status of the cube using ASCII art (see Fig. 4), and an *RCLogger* module for debugging. On the third PC there is a similar configuration as the second one, but for the fact that the RCConsoleRenderer is replaced with an *RC3DRenderer*, which is drawing the cube using 3D graphics (see Fig. 5).

On each satellite PC, the InputManager module asks to be scheduled at a fixed rate by the engine and polls the keyboard for user input. When input is available, a message with the key pressed is sent to the RCSimulator module on the first PC. If the request is correct, the RCSimulator module changes the status of the Cube and notifies the new color configuration to all known renderers. As a result, players on the satellite PCs are playing on the same Rubik's Cube. Moreover, it is very easy to *upgrade* the visual from ASCII to 3D on the second PC just loading another instance of RC3DRenderer. This new

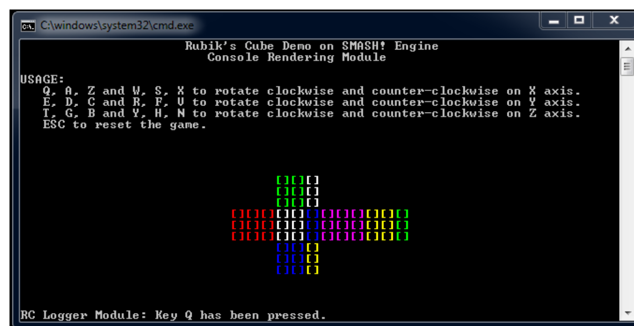

Fig. 4. Rubik's cube demo: 2D renderer using ASCII art.

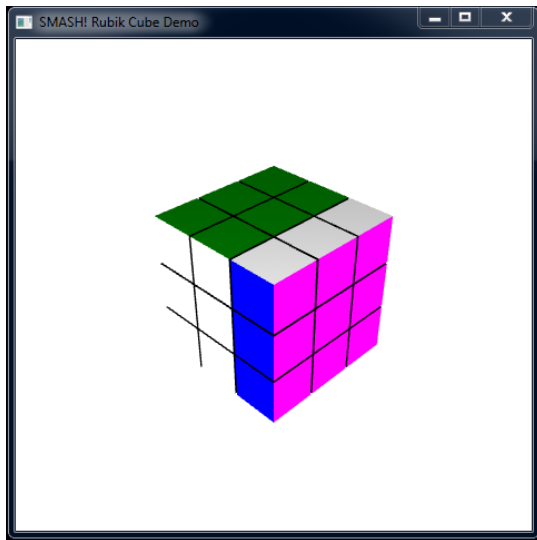

Fig. 5. Rubik's cube demo: 3D renderer.

instance registers in the local engine and the message bus will advertise the new module to the rest of the architecture. From this moment onward, the new 3D renderer will be also updated from the main engine.

VI. Conclusion and Future Work

In this paper we put into question the internal structure of modern game engines as we believe that it is unlikely to provide the flexibility and functionalities required by game developers in the next generations. As a possible solution we propose SMASH: *Stackless Microkernel Architecture for SHared environments*. SMASH features a completely distributed architecture taking inspiration from microkernel operating systems. In SMASH, game modules can be dynamically added and removed while the engine is running. A prototype implementation is presented using C# in windows environment. Performance tests hint that the resulting system is fit to host a soft real-time application such as a video game. Finally, we are also presenting a demo application: a distributed multiplayer version of a Rubik's Cube puzzle. This application is working nicely and multiple rendering modules can be applied and changed at runtime.

In the future, we are planning to extend our prototype and test it in a more complex environment in order to understand its behaviour over a large scale network. This will be useful to actually deploy widely distributed games such as the ones envisioned in [28] and [29].

References

[1] Activision, "Pitfall!" http://en.wikipedia.org/wiki/Pitfall!, Apr 1982.

[2] Infogrames, "Tetris," http://en.wikipedia.org/wiki/Tetris, Jun 1984.

[3] Bröderbund, "Prince of Persia," http://en.wikipedia.org/wiki/Prince_of_Persia_(1989_video_game), Oct 1989.

[4] J. Gregory, *Game Engine Architecture*, 2nd ed. A K Peters/CRC Press, 2014.

[5] D. Maggiorini, L. A. Ripamonti, and G. Cappellini, "About Game Engines and Their Future," in *Proceedings of EAI International Conference on Smart Objects and Technologies for Social Good (GOODTECHS 2015)*, Oct 2015, pp. 1–6.

[6] T. C. S. Cheah and K. W. Ng, "A practical implementation of a 3D game engine," in *Computer Graphics, Imaging and Vision: New Trends, 2005. International Conference on*, Jul 2005, pp. 351–358.

[7] G. Mulley, "The Construction of a Predictive Collision 2D Game Engine," in *Modelling and Simulation (EUROSIM), 2013 8th EUROSIM Congress on*, Sep 2013, pp. 68–72.

[8] D. Maggiorini, L. A. Ripamonti, and F. Sauro, "Unifying Rigid and Soft Bodies Representation: The Sulfur Physics Engine," *International Journal of Computer Games Technology*, pp. 1–12, May 2014.

[9] R. Darken, P. McDowell, and E. Johnson, "Projects in VR: the Delta3D open source game engine," *IEEE Computer Graphics and Applications*, vol. 25, no. 3, pp. 10–12, May 2005.

[10] V. Guana, E. Stroulia, and V. Nguyen, "Building a Game Engine: A Tale of Modern Model-Driven Engineering," in *Games and Software Engineering (GAS), 2015. IEEE/ACM 4th International Workshop on*, 2015, pp. 15–21.

[11] J. Munro, C. Boldyreff, and A. Capiluppi, "Architectural studies of games engines: The quake series," in *Games Innovations Conference, 2009. ICE-GIC 2009. International IEEE Consumer Electronics Society's*, Aug 2009, pp. 246–255.

[12] W. Xun, L. Xizhi, and G. Huamao, "A Novel Framework for Distributed Internet 3D Game Engine," in *Convergence and Hybrid Information Technology, 2008. ICCIT '08. Third International Conference on*, vol. 1, Nov 2008, pp. 289–294.

[13] V. Gajinov, I. Eric, S. Stojanovic, V. Milutinovic, O. Unsal, E. Ayguad, and A. Cristal, "A Case Study of Hybrid Dataflow and Shared-Memory Programming Models: Dependency-Based Parallel Game Engine," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on*, Oct 2014, pp. 1–8.

[14] H. Lu, W. Yijin, and Y. Hu, "Design and implementation of three-dimensional game engine," in *World Automation Congress (WAC), 2012*, Jun 2012, pp. 1–4.

[15] G. Nelson, "Natural Language, Semantic Analysis and Interactive Fiction," St Annes College, Oxford, Tech. Rep., Apr 2006, http://inform7.com/learn/documents/WhitePaper.pdf.

[16] D. Fillmore, "The Z-Machine Standards Document version 1.1," http://inform-fiction.org/zmachine/standards/z1point1/index.html, Feb 2014.

[17] Infocom, "Zork," http://en.wikipedia.org/wiki/Zork, 1979.

[18] M. Gerla, D. Maggiorini, C. E. Palazzi, and A. Bujari, "A survey on interactive games over mobile networks," *Wireless Communications and Mobile Computing*, vol. 13, no. 3, pp. 212–229, 2013.

[19] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, "The Java Virtual Machine Specification, Java SE 8 Edition," Oracle Corporation, Tech. Rep., Feb 2015.

[20] ECMA International, *Standard ECMA-335 - Common Language Infrastructure (CLI) Specification*, 6th ed., Jun 2012.

[21] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren, "Amoeba A Distributed Operating System for the 1990s," *Computer*, vol. 23, no. 5, pp. 44–53, May 1990.

[22] D. Hildebrand, "An Architectural Overview of QNX," in *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*. Berkeley, CA, USA: USENIX Association, 1992, pp. 113–126.

[23] S. Furuhashi, "MessagePack," http://en.wikipedia.org/wiki/MessagePack, Apr 2013.

[24] Y. Y. Golan, T. Cai, P. Leach, and Y. Gu, "Simple Service Discovery Protocol/1.0," IETF Internet Draft – work in progress, Oct 1999.

[25] Epic Games, "Unreal Engine," http://en.wikipedia.org/wiki/Unreal_Engine, 1998.

[26] "Distributed Component Object Model (DCOM) Remote Protocol Specification version 18," Microsoft Corporation, Tech. Rep., Oct 2015.

[27] Unity Technologies, "Unity3D," http://unity3d.com/, Jun 2005.

[28] D. Maggiorini, C. Quadri, and L. A. Ripamonti, "Opportunistic mobile games using public transportation systems: a deployability study," *Multimedia Systems*, vol. 20, no. 5, pp. 545–562, 2014.

[29] C. Prandi, P. Salomoni, M. Roccetti, V. Nisi, and N. J. Nunes, "Walking With Geo-Zombie: A Pervasive Game to Engage People in Urban Crowdsourcing," in *Proceedings of International Conference on Computing, Networking and Communications (ICNC 2016)*, Feb 2016, pp. 113–126.