CONCEPTS FOR DISTRIBUTED INPUT INDEPENDENT ARCHITECTURE FOR SERIOUS GAMES

Stefan Stavrev[1], Todorka Terzieva[2], Angel Golev[3]

Abstract: Serious games (SG) allow us to learn even when we are relaxing. These games are called “serious” because they allow us to be trained at domain-specific knowledge level. That is the main reason SG are gathering an even increasing research interest in recent years. In contrast with traditional, purely entertainment games, SGs architectures and design principles are under active investigation by researchers. Recent work in that field attempts to define how SG are structured, built, used and extended. However, there is still a lot of debate which design techniques are adequate or which techniques can be borrowed from other fields – such as computer science or mainstream entertainment games. The main objective of our research is three-fold: investigate and analyze current architectural approaches; summarize the top characteristics of a modern serious game; and propose an architecture that is coherent with current approaches. Following these principles, we determine that the prevailing views in the SGs area are that they should be distributed and modular, service-based and easily extendible. Building on top of that, we come up with a novel concept for creating serous games that are independent of their input devices and propose two ways that independence can be achieved. We briefly discuss the possible integration of 3-rd party services by using message queue brokers in a publish / subscribe manner. Finally, we summarize and propose different methods for extending our proposed approach.

UDC Classification: 004.94; **DOI:** http://dx.doi.org/10.12955/cbup.v6.1310

Keywords: serious games, architecture, design principles, input independence

Introduction

Serious games (SG) are a kind of video games that focus on the educational part of play rather than the entertainment one. That doesn’t mean they cannot be entertaining, as well. Since SG have a lot in common with mainstream video games, it is only natural that SGs will need to borrow or follow the software design principles and architectures used for creating mainstream video games, according to BinSubaih, Maddock & Romano (2006) and Van Nuland (2010). However, the purpose of SGs is different: to train, educate and simulate real-life experiences. In order to achieve their goal, SGs employ different kinds of content - assets and resources that often needed to be shared between games, as is the case with the work of Der Vegt, Westera, Nyamsuren, Georgiev & Ortiz. (2016). In addition, it is often the case that SGs need to run on low-end hardware systems. Those constraints make their architecture and development extremely challenging.

In this paper we strive to propose an architecture for SG that is coherent with major tendencies in the field, which are presented and summarized in later sections. We achieve this while keeping in mind the above-mentioned hardware restrictions. It is our goal to also make that architecture independent of hardware input, in order to be easily extensible and playable with various input devices. Last but not least, – we would like to incorporate 3rd party data into our SG architecture so that we can make use of real-time services such as news feeds, stock prices, outside temperature, etc. Those 3rd party services may or may not influence the game’s dynamic and logic, depending on concrete requirements and implementation.

Adaptability in training

A lot of recent work has been done in the architecture and design of serious games development. Adaptive content delivery has been proposed by Bontchev & Vassileva (2009). The researchers Sobke & Streicher (2015) propose a classification of the various types of SG according to the view of their design, purpose and implementation. They suggest a 3-way decomposition:

- the use of typical components for building SGs
- distributed architectures
- data models and interoperability.

[1] Department of Software Technologies, Faculty of Mathematics and Informatics, Plovdiv University “Paisii Hilendarski“, Bulgaria, e-mail: stavrev@fmi-plovdiv.org
[2] Department of Software Technologies, Faculty of Mathematics and Informatics, Plovdiv University “Paisii Hilendarski“, Bulgaria, e-mail: dora@uni-plovdiv.bg
[3] Department of Software Technologies, Faculty of Mathematics and Informatics, Plovdiv University “Paisii Hilendarski“, Bulgaria, e-mail: angel.golev@fmi-plovdiv.org

Their first point is on monolithic approaches in terms of the target platform the game is going to be played on: -mobile, Internet games, desktop and console. The authors also include physical sensors (Kinect, Oculus rift) as typical input components. The second view is that of decentralized systems where the game is distributed over different domains. Certain aspects of a game should be available in real-time as a service, hence the name Service Oriented Architecture, or SOA (Carvalho et al., 2015). Another aspect of the distributed SGs development is their modular nature. Design of SGs modules in a way that facilitates asset reuse has been researched by several authors (BinSubaih, Maddock, & Romano, 2006), (Der Vegt et al, 2016). The third part of the decomposition is that of data models and interoperability. That view focuses on the data flow and the use of data structures. The main idea is that common data formats, data structures and protocols should be used for creating SGs architecture and assets (content).

Some other authors have worked on a particular part of the 3-way decomposition mentioned above. For instance, recent research is conducted on architecture that is oriented entirely towards services (Carvalho et al., 2014), (Carvalho et al., 2015). Service-Oriented Architecture (SOA) is a set of practices for architectural design of software that exploits services as loosely coupled components orchestrated to deliver various functionalities. The SOA paradigm is not well established in the Serious Games (SG) domain, yet. The components provide independent services to other components of the serious game or application. The key principles in this particular design are modularization and re-use of functionalities. That concept in not new in the field of computer science but is relatively rarely applied, yet, in the field of games and serious games in particular. (Sprott & Wilkes, 2004), (Van Der Aalst, 2007). Additional benefits of using services is the lack of compile-time dependencies. Moreover, it is entirely possible to have the core gaming as a service in a centralized server. But the biggest advantage remains the re-use of components - shared user profiles, knowledge databases on learning topics, natural language processing dialog services, gestures (Stavrev, 2016). Of course, the SOA approach is not without shortcomings. Some of the challenges are that the quality assurance and testing module integration tends to be more difficult when developing SOA applications. In addition, sometimes the lack of documentation on the usage of interfaces makes integration with a certain service difficult. Furthermore, extra attention to services description needs to be kept in mind. Another limitation of the SOA approach is that the game needs to be constantly online, i.e. connected to a certain service or services. That last restriction makes the architecture less flexible. Finally, there is the additional performance cost due to network calls.

Other authors prefer to focus on the re-use of components and game assets. A notable example is the collaborative work between the “Complutense University of Madrid”, the Sofia University “St. Kliment Ohridski” and the Open University of the Netherlands. The project is called RAGE (Der Vegt et al, 2016) and is an attempt to make interoperable game components (assets) that can be shared between different games. An asset, as the authors stated, is not limited to a software class or interface but is more abstract “intellectual content, independent of their physical embodiments” (Dekkers, 2013). A component is usually a game object with some functionality attached to it. An example would be a vehicle asset that has its driving and steering logic implemented inside a component. The main idea of RAGE is that such components can be shred and re-used by different game engines, programming languages and architectures. Assets can be acquired or purchased from a central repository, such as the Unity asset store (Unity 3D asset store, 2018) or the Unreal Marketplace (Unreal Engine 4 marketplace, 2018). From software architectural point of view, the RAGE project adopts the component-based development approach (CBD) (Bachmann et al., 2000), (Mahmood, Lai & Kim, 2007).

A typical serious game usually combines the above-mentioned principles – a SG is self-contained – runs on a certain platform, can be distributed (use services) and can support various interoperable data models (Thillainathanl, 2014). One example of such a game is the Virtual-platform simulator for safety crossing (Stavrev & Terzieva, 2015) – it runs on a concrete PC platform (Windows), it employs the use of a Kinect sensor as an input (Stavrev, 2016), and it uses decentralized components that can be re-used. Another example is the well-known serious game Second Life (Second Life, 2018). It runs on a PC in a distributed manner; it supports the creation of in-game assets that can be transferred to other projects through the use of scripts (Second Life Wiki, 2018). Having reviewed recent work in the field of SGs architecture and design, we can summarize some of the key architecture principles:

- Distributed modular design plays an important role in scalability and content management
- Services and SOA help integrate various real-time information into games.
- Sensors and other haptic devices are important since they provide a unique and natural interaction, especially for educational purposes.

Combining the above principles, we propose the following: Distributed input-independent Architecture for Serious games, that we call DiAS for short.

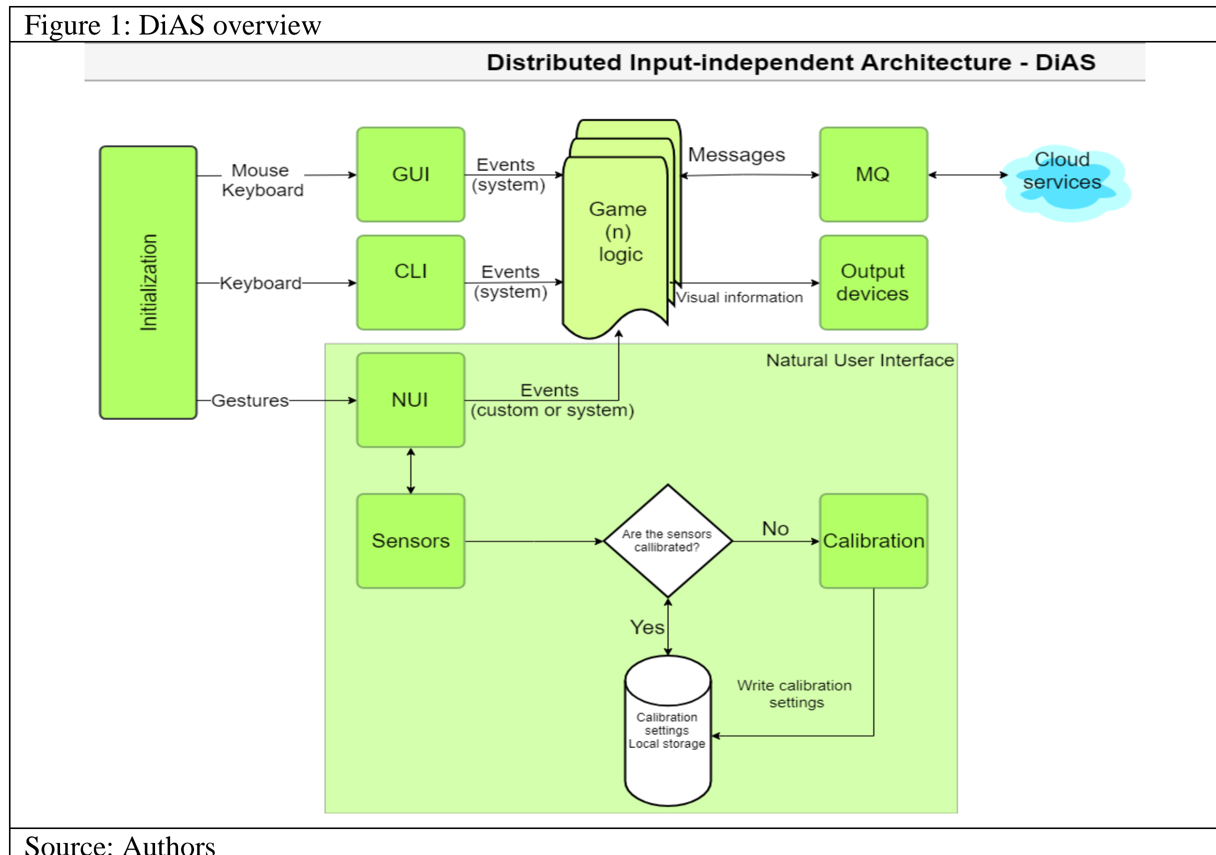
DiAS architecture overview.

Let's take a look at the DiAS architecture (Figure 1). There are several phases in this approach. In the initialization phase, we setup a number of different components – the presence or absence of sensors, the number (and type) of the output devices – monitors, projectors, etc. This is one of the important processes since it will decide how the system will respond from now on to the user's interactions. We have designed 3 generic interfaces of communicating with the system.

- The graphical user interface (GUI)
- The command line interface (CLI)
- The natural user interface (NUI)

Each of these interfaces is a way of the user or player to interact with the system (game logic). By default, only the GUI is turned on. If the initializer detects the presence of sensors, connected to the system, the NUI will be turned on, as well. The CLI is a special interface and will only be accessible if allowed through a configuration file.

Figure 1: DiAS overview



Source: Authors

The most common and well-known interface is the GUI. It consists of series of graphical elements that are drawn on top of the game world – buttons, menus, heads-up displays, progress bars, and texts. The player interacts with the game by pressing a visual button (rendered on the screen), dragging around with the mouse, or pressing a certain key from the keyboard. This interface is mainly intended for use by educators and occasionally, by the players. The communication with the game logic is event – driven. Each time the user interacts with a graphical component from the GUI – for instance, pressing a button or opening a game menu, an event is fired. We call this type of events system events since the operation system takes care of firing them and they are not a responsibility of the application level.

The second interaction interface is the CLI. As we mentioned, it is a special interface in the sense that it is intended for use by a system administrator or a QA tester. The interface allows the user to enter pre-defined commands through the in-game command line. Different commands trigger their respective functions or group of functions in the game logic.

The third and probably the most interesting interface is the NUI. It is a way of interacting with the game logic by capturing and processing gestures from users. The gestures themselves can be pretty basic (clapping hands, shaking one's head, jumping / walking on the spot, waving, etc.) or more complex – sensory inputs from multiple haptic devices. Once captured by a sensor, each gesture can be processed in two different ways – either as a custom event or as a generic system event.

Decoupling the game logic. Generality vs. speed.

The game logic module is the actual game. Each game receives input (in the form of events) through one of the user interfaces already mentioned. There are two approaches for firing input events – either firing a custom event or firing an emulated OS event. Each of these methods has its pros and cons. The advantage of translating a gesture to a system/OS event is that the architecture becomes more generic. In that way, the game logic can be decoupled from the user interface. The events themselves can be either COM events on Windows systems or D-bus messages on *nix systems. If the game logic is programmed to listen for system events, it will receive system events. For instance, the game listens for a mouseClick event. The user claps with his / her hands thus generating a gesture. The NUI captures the gesture (through a sensor), consumes it and fires a mouseClick event. The game logic has no information that it has actually received a gesture as an input, thus allowing for a backward compatibility with existing games and systems. Although this approach makes the architecture decoupled it also makes it slower. The process of gesture recognition, processing and mapping to a system event simply takes more time. The other approach of using the NUI is to fire a custom event for each gesture. The speed of firing and processing a custom event will be higher but the game logic needs to know in advance what kind of custom event it should listen for – thus adding a dependency to the NUI.

Sensor Calibration.

In order for the proposed architecture to be adaptable, scalable and independent, it was envisioned to be used with different setups including different input sensors. The current system design can be used with (but not limited to): Kinect, generic depth cameras, Orbbec Astra, Leap motion and Intel RealSense. Our proposed architecture allows decoupling of the user interface (which can be sensory) from the actual game logic (gameplay). As such, any game that can be played with a mouse and a keyboard can be played with a sensory input.

Sensor calibration is a usually a tiresome and time-taking process. In our proposed architecture, we strive to minimize the calibration time by employing the so called "calibration settings sharing" – sensor calibration settings can be shared among different games and game instances. The initialization module of the proposed architecture can be implemented and build in every game that will support a NUI. In that way, the sensors, attached to the system, can be calibrated once (in one game); their settings (depending on the sensor – range from the user, types of interactions, resolution, etc.) can be written on a local storage and be re-used for another game, without the need to go through the calibration process again, provided the system is not moved in any way (Figure 2).

MQ broker

The MQ broker is the system component middleware responsible for the game communication with the outside world. We added that additional layer of abstraction so that the game logic would not have to initiate requests to 3rd party services or to a database storage. Instead of using request / response design pattern (that is common for communication in client – server manner or requesting data from services) we decided to use a publish / subscribe paradigm. The game logic subscribes (becomes a listener) for updates on a certain topic on the MQ. When there is new information published on that topic, the game logic will receive it.

MQs are reliable and well-known publish / subscribe messaging transport mechanisms that operate on top of the TCP. Most MQ (Apache Active MQ, 2018), (Apache Kafka, 2018), (Rabbit MQ, 2018), (Zero MQ, 2018) support two exchange formats – text messages and blob messages. Text messages are represented by strings, regardless of their text data format. The format itself can be JSON, XML,

plain text, etc. The blob message, on the other hand, is an equivalent of a binary message. The communication between the game logic module and the MQ broker can be done via one of several transport protocols, including STOMP (STOMP, 2018), Open Wire (Open wire protocol, 2018), MQTT (MQTT, 2018), and AMQP (AMQP, 2018). The messages are exchanged through different channels, called topics and queues. Topics are more suitable for publish/subscribe scenarios whereas queues are better suited for load balancing. The concrete implementation – which brokers, protocols or communication channels to use, depends on the use cases and business requirements.

Figure 2: Calibration screen

Source: Authors

Cloud services

Last but not the least is the cloud services component. It is not really a part of the system but rather a way of gathering, storing and serving content to the game module via the message queue. This component is outside of the game logic since we set a goal to target low-end hardware and not make unnecessary processing in the game itself. One use case is using a cloud database for keeping record of players' records and other important data. In another scenario, there could be small modules in the cloud that constantly gather info (sending requests) to 3rd party service providers. Imagine the following use-case: a DiAS service, running in the cloud makes a request to a 3rd – party weather service. From the cloud, a message with the current weather for a given location is passed to the MQ and from there – to the game logic. The game may decide to use that information and alter its inner state – if it is raining outside, it will be raining in the game, as well. If the game had to make all those requests, constantly pulling data from a 3rd party service, we would not have achieved the decentralization we strived for.

There are several options when it comes to cloud services. The most common ones are Azure (Microsoft Azure, 2018) by Microsoft™, amazon web services (Amazon Web Services, 2018) by Amazon™ or a virtual private cloud (VPC). There are several advantages of using a paid-subscription for cloud services (AWS or Azure) over VPC. First and foremost is the maintenance cost and availability. For serious games that involve a lot of players' real-time data synchronization, easy scalability and redundancy, that is the preferred option. On the other hand, for small-sized institutions, such as schools, kindergartens, or day centers, VPC is the preferred option.

Conclusion

In this paper we have presented an abstract architectural approach for serious games that combines well-known stand-alone design principles under a new light. Analyzing and summarizing those common principles in the field of SG, we suggested to focus on three points in our approach: device-

independent input interfaces, adaptability, and the use of services for in-game functionalities and features. Device independence in DiAS is achieved in two ways – by firing and processing custom events or by emulating native system events. Adaptability is achieved by sharing configuration options that different game instances can use. Connecting to 3-rd party services is achieved in a publish - subscribe manner by using a MQ broker as a middleware. In addition, the actual modules and services that request the data from those 3rd-party services are put on the cloud. That decentralization and modularization is essential in order to lower the workforce the core game logic has to do. Less work means the core gameplay can run on low – end hardware. In DiAS, not all of the mentioned components or principles are required but most of them are recommended in order for a game or games to be scalable and extendable.

Acknowledgements

Parts of this work are supported by a project СП17-FMI-011 of the Scientific Research Fund of Plovdiv University "Paisii Hilendarski", Bulgaria.

References

Amazon Web Services, Retrieved March 11, 2018, from https://aws.amazon.com/

AMQP, Retrieved March 11, 2018, from https://www.amqp.org/

Apache Active MQ, Retrieved March 11, 2018, from http://activemq.apache.org/

Apache kafka, Retrieved March 11, 2018, from https://kafka.apache.org/

Bachmann, F., Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R., & Wallnau, K. (2000). Volume II: Technical Concepts of Component-Based Software Engineering, 2nd Edition (CMU/SEI-2000-TR-008). Retrieved March 11, 2018, from the Software Engineering Institute, Carnegie Mellon University website: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=5203.

BinSubaih, A., Maddock, S., & Romano, D.M. (2006). An Architecture for Portable Serious Games, In: Doctoral Symposium hosted at the 20th European Conference on Object-Oriented Programming ECOOP 2006, Nantes, France. Serious Games: A New Paradigm for Education

Bontchev, B. & Vassileva, D. (2009). Adaptive courseware design based on learner character, Proc. of Int. Conf. on Interactive Computer Aided Learning (ICL2009), 23-25 Sept., 2009, Villach, Austria, pp. 724-731.

Carvalho, M. B., Bellotti, F., Berta, R., De Gloria, A., Gazzarata, G., Hu, J., & Kickmeier-Rust, M. (2014). A case study on Service-Oriented Architecture for Serious Games, 2014, Entertainment Computing, https://doi.org/10.1016/J.ENTCOM.2014.11.001

Carvalho, M. B., Bellotti, F. , Hu, J., Baalsrud Hauge, J., Berta, R., De Gloria, A., & Rauterberg, M.(2015). Towards a service oriented architecture framework for educational serious games, 15th IEEE international conference on advanced learning technologies (ICALT2015)

Dekkers, M. (2013). Asset Description Metadata Schema (ADMS), W3C Working Group, 2013, Retrieved November, 10, 2017 from http://www.w3.org/TR/vocab-adms/

Der Vegt, W., Westera, W., Nyamsuren, E., Georgiev, A., & Ortiz, I.M., (2016). RAGE Architecture for Reusable Serious Gaming Technology Components, International Journal of Computer Games Technology, Volume 2016, Article ID 5680526, 10 pages, https://doi.org/10.1155/2016/5680526

Mahmood, S., Lai, R. & Kim, Y. S. (2007). Survey of component- based software development, IET Software, vol. 1, no. 2, pp. 57–66, 2007, https://doi.org/10.1049/iet-sen:20060045

Microsoft Azure, Retrieved March 11, 2018, from https://azure.microsoft.com/en-us/

MQTT, Retrieved March 11, 2018, from http://mqtt.org/

Open Wire protocol, Retrieved March 11, 2018, from http://activemq.apache.org/openwire.html

Rabbit MQ, Retrieved March 11, 2018, from https://www.rabbitmq.com/

Second Life, Retrieved January, 10, 2018 from http://secondlife.com/

Second Life Wiki, Retrieved January, 10, 2018 from http://wiki.secondlife.com/wiki/Object_to_Data_v1.4

Sobke H., & Streicher A. (2015) Serious Games Architectures and Engines. In: Dorner R., Gobel S., Kickmeier-Rust M., Masuch M., & Zweig K. (eds) Entertainment Computing and Serious Games. Lecture Notes in Computer Science, vol 9970. Springer, Cham, 15283, 2015, Dagstuhl Castle, Germany, https://doi.org/10.1007/978-3-319-46152-6_7

Sprott, D., & Wilkes, L.(2004). Understanding service-oriented architecture, The Architecture Journal 1 (1), pp. 10–17

Stavrev, S. (2016). Natural User Interface for Education in Virtual Environments, REPLAY, Polish Journal of Game Studies 03, pp.67-80, https://doi.org/10.18778/2391-8551.03.04

Stavrev, S., & Terzieva, T. (2015). Virtual environment simulator for educational safety crossing. Proceedings of the 11th Annual International Conference on Computer Science and Education in Computer Science (CSECS), June, 2015, Boston, MA, USA, pp. 92-98, ISSN 1313-8624.

STOMP, Retrieved March 11, 2018, from https://stomp.github.io/

Thillainathan, N. (2014). A Model Driven Development Framework for Serious Games. (Unpublished doctoral dissertation), Doctoral Consortium, Kassel University, Koblenz, Germany, https://dx.doi.org/10.2139/ssrn.2475410

Unity 3D asset store, Retrieved March 11, 2018, from https://assetstore.unity.com/

Unreal Engine 4 marketplace, Retrieved March 11, 2018, from https://www.unrealengine.com/marketplace

Van Der Aalst, W. M., Beisiegel, M., Hee, K. M. V., Konig, D., & Stahl, C. (2007). A soa-based architecture framework, International Journal of Business Process Integration and Management 2 (2), 2007, pp. 91 – 101, https://doi.org/10.1504/IJBPIM.2007.015132

Van Nuland, B. (2010). A Service Oriented Architecture Solution for Gaming Simulation Suites, Thesis report, Software Engineering Research Group, Department of Software Technology, Faculty EEMCS, Delft University of Technology, Delft, the Netherlands, 2010.

Zero MQ. Retrieved March 11, 2018, from http://zeromq.org/