A game engine plugin for ubigames development

Luciano H. O. Santos Fabricio N. Buzeto Lucas N. Carvalho Carla D. Castanho

Department of Computer Science
University of Brasília
Brasília – Brazil

Abstract

Game engines are an integral part of game development, greatly easing the overall effort. Even though, ubigames still lack the availability of such advanced tools as those found for other platforms like consoles and mobile phones. In order to fill this gap, we have developed a Unity 3D plugin for building games compatible with the uOS ubiquitous middleware. The results of this implementation are evidentiated through two use case games, showing its compliance and stability.

Keywords: Pervasive Gaming, Ubiquitous Computing Games, Game Engine, Unity

Author's Contact:

lucianohenriquesantos@gmail.com
fabricio@aluno.unb.br
lucasnvcarvalho@gmail.com
carlacastanho@cic.unb.br

1 Introduction

The concept of ubiquitous computing or pervasive computing [Weiser 1991] stimulated the emergence of several new fields of research. The idea was to take advantage of the increasing presence of computing capable devices in people's lives, and create environments which employ technology to ease daily tasks in a minimally invasive way, without requiring attention or direct interaction from the user [Weiser and Brown 1996], the so called *smart spaces*.

Eletronic games, at first sight, could be viewed as incompatible with these goals, since they focus on exactly the opposite idea, i.e., they try to entertain through engagement and immersive gameplay, requiring the user's constant and deliberate interaction. For this reason, proposals of ubiquitous games (ubigames) tried to apply pervasive computing's concepts and principles in new ways, by exploring different attributes of environment and several aspects of the player's reality to create inovative mechanics and integrate the real world and the game world, improving gameplay and player imersion.

Ubigames, as ubiquitous applications themselves, brought along all the technological challenges faced by ubiquitous computing in general, such as heterogeneity of devices and communication interfaces, scalability, fault tolerance, security and privacy issues, context adaptability and user interfaces [da Costa et al. 2008]. Several models and proposed frameworks aim at creating an architectural foundation to facilitate the construction of ubiquitous software [Sousa and Garlan 2002][Modahl et al. 2004][Aitenbichler et al. 2007][Sacramento et al. 2004] or, in some cases, ubigames [Stach 2012], but they stumble uppon a variety of limitations on their design.

Among the many platforms that propose to ease this task, the uOS middleware can be highlighted. Based on a architecture model – DSOA [Buzeto et al. 2010] –, a communication protocol – uP [Buzeto et al. 2011] –, and a *middleware* – uOS [Buzeto 2010] –, it allows ubiquitous applications to publish and list devices, resources and services present in the smart space and access them over a transparent, platform-independent layer. The middleware also provides means to create an hierarchy of equivalence between resources, and ontologies, making it possible to create heuristics and choose the best available resource for a given task in different contexts.

Even though uOS is supported by multiple platforms, and has already been used for game development [Buzeto et al. 2013], it does not provide any range of tools focused on this specific task. Integration with a game engine, for instance, would make it much more easier to handle common game development problems, like graphic and physics components and user interaction. An engine largely reduces the overral development effort, by implementing common game logic abstractions and entity representation, like scenes, maps and game objects [Rabin 2010].

Game engines have been used for decades now, featuring as essential tools for game developers. The techniques employed by modern engines to solve the several technical challenges faced in game development were tested and improved over the years, creating a set of traditional architecture models to efficiently deal with them [Rabin 2010] [Gregory 2009].

For this reason, the integration of ubiquitous computing capabilities to a well established engine would greatly benefit ubigame development and research, allowing developers to focus on mechanics and user interaction, instead of infrastructure issues and technical details.

This work presents a plugin for the Unity 3D[1] game engine that enables communication between games created with the engine and uOS-based smart spaces. Unity is a well established game engine with support for a wide range of platforms. It provides tools for rapid creation of game scenes and game objects' behaviour, as well as an advanced API to handle user inputs, graphics, audio, physics simulation and AI. The solution here presented extends the capabilities of such engine in order to use the uOS middleware for communication and device integration.

This paper is organized as follows. Section 2 makes a more detailed description of DSOA and uOS main concepts and elements. Section 3 describes our implementation, and the specific issues faced for the target environment. Section 4 describes two ubiquitous games we have developed to evaluate the solution. Finally, Section 5 covers our conclusions and possible topics for future research.

2 DSOA and the uOS Middleware

DSOA is an extension of SOA [Schulte and Natis 1996], and describes the smartspace as a collection of *devices*, i.e., elements of the environment with computational power, such as PCs, smartphones and embbeded hardware. Each device has *resources*, which are groups of logically related funcionalities. Usually this means the accessible hardware components like keyboards, mice, screens, cameras, sensors and other peripherals or embedded elements, but may also refer to more abstract concepts, such as device user information or data storage. Each resources is accessed as a group of *services*. A service is the actual implementation of a functionality, and provides a known public interface. For example, a smartphone may have a camera resource with a "take picture" service, receiving no parameters and returning the bytes representing an image.

The communication protocol proposed by Buzeto et al., uP, establishes how these concepts interact, defining the following entities:

- a *service* is the interface for a resource's functionaliy, and must have a *name* and zero or more *parameters*, which may be *optional* or *mandatory*, and may return response data when called;
- a *driver* models a resource, having a *name* and a collection of services, and optionally a collection of *events*, which are a special case of service, that allows the registration for asynchronous notifications;
- finally, the top level entity is the *device*, which has a *name*, one or more *networks* (a description of a communication channel, such as ethernet, bluetooth, etc) and one or more drivers.

[1]http://unity3d.com/

All uP messages are formatted in JSON[2], a lightweight text-based data-interchange format. A service call is a JSON message consisting of driver and service name, an optional driver instance id and the service's expected parameters. The response will contain any data returned by the call or an error message in case of failure. Special calls *registerListener* and *unregisterListener* on compatible drivers may be used to enable asynchronous notifications for specific events.

The uOS middleware implements uP and provides a basic infrastructure for smart space communication. One or more *channel managers* may be present, to allow communication on different interfaces, such as Ethernet and Bluetooth. Also, one or more *radars* may be instantiated, with different strategies to locate nearby devices: ping, multicasts, centralized registries, etc... With regards to resources, all uOS devices have at least a *DeviceDriver* driver, responsible for responding to device searches in the environment, with services to *handshake*, and list the drivers of the current device, among others. Internally, a *device manager* keeps track of all known devices and a *driver manager* registers all known drivers and their equivalence tree, as well as the ontology rules to look up for resources.

All these features are accessible through a centralized class, a *Gateway*, that allows applications to access the components of the middleware and aggregates the common functionalities, such as listing devices and calling a service on a device.

More details about the uOS implementation may be found at the project's website[3].

3 Integration with Unity

The Unity uOS plugin[4] replicates the behaviour of the reference middleware implementation, following the same software architecture and structure. The scope of the plugin is to enable the communication between games developed using it and other devices and services available on a uOS smart space. Since the middleware provides a wide range of features, complimentary to the comunication among devices, the set of features implemenented were constrained to the following:

- full set of uP entities and protocol translation;
- socket communication capabilities;
- device discovery and management;
- resource discovery and management;
- synchronous and asynchronous service communication;
- application integration.

Using these features, a game becomes aware of its surroundings and able to identify which devices are available and which resources they have, in order to access them. It also allows the game device to publish its own resources for use by other applications at the smart space. This enables the contruction of games that dynamically adapt to the capabilities of the environment as they change.

The main challenges faced in the implementation of the middleware features inside the Unity environment reside in the *threading model*, i.e., the concurrent code execution system. C# and .NET[5], the framework used by Unity for its API implementation, offer a threading environment very similar to the Java Virtual Machine's, the original platform of the middleware. However, Unity itself has a very restricted model for its threads. All the engine components and resources must be accessed exclusively from the game's main execution thread, i.e., from the callback methods defined by the engine for game routines. However, it's imperative for the middleware to run multiple execution threads, since most of the communication channels require blocking operations to be performed regularly, what causes the whole thread to stop. For this reason, a simple and efficient mechanism was needed to exchange information between the middleware's multiple threads and the game main thread created by Unity.

[2]http://json.org/
[3]http://www.unbiquitous.org/
[4]https://github.com/lhsantos/unity_uos_plugin
[5]http://www.microsoft.com/net

We designed a basic message engine that uses a queue associated with a lock to ensure thread synchronization. Anytime the middleware needs to deliver a message, either a data exchange, a log entry or an exception report, it puts a message in the queue, which is guarateed for consistency because of the lock object. A specialized *MonoBehaviour*, i.e., a game object's component inside the game, uses the Unity frame update callback to regulary empty this queue and handle all the pending messages. When an object running at the main thread needs to request a service from the middleware, it may use the asynchronous version of the call, in which a new thread is automatically created and reports via callback when the response is ready. This mechanism permits other game objects to interact with the middleware without performing blocking operations or causing delays at runtime.

For uP message serialization, a free open source library was choosen, MiniJSON[6]. This is a lighweight implementation of JSON for C#, best suited for small and simple message formatting.

All network communications were performed with the standard .NET networking API, as provided by Unity.

4 Use Cases

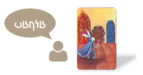
To evaluate this solution, two ubigames were proposed and developed. In each case, the games were used to test the system functionalities or their integration with the reference implementation of the middleware. The next two sections describe these games.

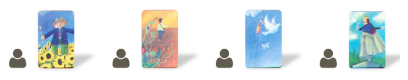
4.1 Vidi

Vidi[7] is collectible card game based on the popular board game Dixit[8]. Each player has a deck of cards, and his/her goal is to expand this collection by interacting with other players.

When a player enters a smartspace where other devices are running the game, he or she may propose a challenge. The challenger must pick a card and describe it with one word (Figure 1 (a)). The other players then must pick a card they think matches the description (Figure 1 (b)). Then, all the players must guess which of the cards was the chosen one (Figure 1 (c)). If there's an equilibrium between right and wrong guesses, the challenger wins all the proposed cards; otherwise, the cards are distributed amongst the other players.

Players may also discover new cards by directly trading with other players in the smartspace.


(a) challenge


(b) players pick their cards

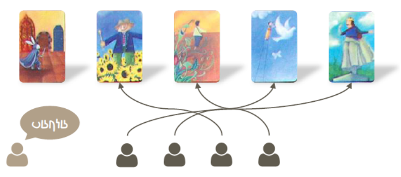

(c) players make guesses

Figure 1: Vidi challenge mechanics (the cards' artwork is copyrighted to Libellud – http://en.libellud.com/).

Using the Unity uOS plugin, Vidi makes available for the smart space a resource driver that represents the player. This driver contains synchronous services to implement the basic elements of the game mechanics like "start a challenge", "find challenge" and "register a challenge guess". It also uses events to notify the end of a challenge or a card trade.

[6]https://gist.github.com/darktable/1411710
[7]https://bitbucket.org/lucasncv/vidi
[8]http://en.libellud.com/games/dixit

Because of the spontaneous nature of the game, it's best suited for mobile devices where the test efforts where focused.

4.2 Ubimon

Ubimon[9] is inspired in the game Pokémon™[10], by Nintendo, and is designed to explore ubiquitous games concepts. The player takes the role of a Ubimon Trainer, travelling around the (real) world and encountering monsters in his way, which he must face and may capture.

The battle system is very similar to the original Pokémon™ game, the ubiquity aspects explored for our test purposes reside in the concept of device capacity and stations. Each device the player may use has a limited capacity, defined by the resolution of the device, in comparison to a reference resolution adopted by the game. If a device is full and no longer able to carry any more ubimon, all newly captured monsters will be immediately released, forcing the player to choose carefully which and how many ubimons he will carry at one time. To store unused ubimon, trading stations are available around the world. These stations are desktops with a public display, and may be used by any player.

The player uses the radar (Figure 2 (a)) to find out nearby players and stations. When a station is within reach, the player may access it by clicking on the button with its name at the screen. If nobody else is already using the station, the player is granted access to it and his device switches to the station control mode (Figure 2 (b)), while the station itself enter the player mode (Figure 2 (e)). In this setting, the player may use his device to get (Figure 2 (c)) or send (Figure 2 (d)) ubimon between his current device and the station.

To implement this mechanic, two ubiquitous applications were developed. A client application, which is the game running on mobile devices, implemented in Unity with the proposed plugin; and a server application, running on desktops, implemented in Java using the reference implementation of the middleware.

At the client's side, two uOS drivers were created:

- a *GlobalPositionDriver*, with a discrete service *getPos* returning a tuple <latitude, longitude, delta>, and a *POS_CHANGE* event, which notifies listeners any time the device's global position changes;

- a *GoogleMapsDriver* with services *updatePos*, receiving a tuple <latitude, longitude> to inform the driver of the current global position, and *render*, which schedules a *GoogleMaps*[11] texture capture and render at the world map, given the current global position.

At the server's side, a *PositionRegistryDriver* was created to model a global registry for entities in the world, with the services:

- *checkIn*, receiving parameters *clientName*, *latitude*, *longitude*, *delta* and *metadata*, to insert a client at the registry and retrieve a *clientId* for subsequent calls;

- *update*, receiving parameters *clientId*, *latitude*, *longitude*, *delta* and *metadata*, to update the current data for a client;

- *checkOut*, receiving a *clientId*, to remove a client from the registry;

- and *listNeighbours*, receiving *latitude*, *longitude*, *delta* and *range*, to list all registered clients near a given global position;

The server also implements a set of application services, all related to the station logic:

- *enter*, receiving a *playerId*, admits a player to the station, as long as no one else is already using it;

- *leave* releases the station from the current player, making it available for other players;

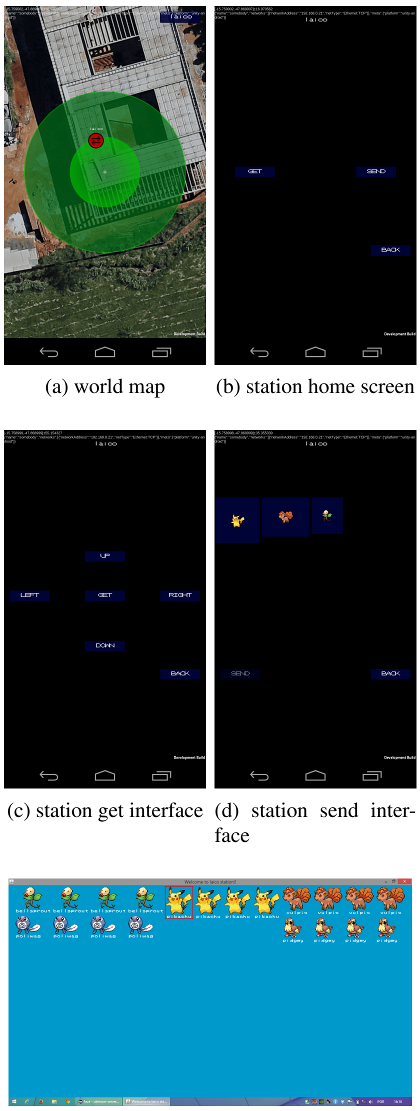
- *cursorToLeft*, *cursorToRight*, *cursorToUp* and *cursorToDown* all receive a *playerId* and are used to move the selection cursor inside the station;

- *peek* retrieves a representation of the currently selected ubimon;

- *removeSelected* removes the currently selected ubimon from the station;

- and *store* sends a ubimon to be saved at the station, as long as there's enough room.

Ubimon demonstrates how the Unity uOS plugin integrates seamlessly with other uOS devices using the reference implementation in the smart space, all of them consuming and providing resources and services.



(a) world map (b) station home screen

(c) station get interface (d) station send interface

(e) station server interface

Figure 2: Ubimon station interfaces (all Pokémon™ graphics in these images are copyrighted to Nintendo – http://www.nintendo.com/).

5 Conclusion

In this paper we presented a solution to integrate a game engine with an existing ubiquitous application framework and facilitate the development of new ubigames. Our solution creates a basic infrastructure for games based on the Unity game engine to communicate with uOS smart spaces, listing available devices and resources, and calling services, as well as making its own resources and services available for the smart space. This also improves the uOS ecosystem to a broader range of platforms supported by the Unity 3D Game Engine, like iOS and Windows Phone.

[9]https://github.com/lhsantos/ubimon
[10]http://www.pokemon.com/
[11]https://developers.google.com/maps/

The games Vidi and Ubimon demonstrate how the solution is a stable and powerful tool for developing games that dynamically integrate devices in the smart space.

For future research topics, we propose the expansion of the plugin's feature set, to include security measures, resource rerouting and ontology and heuristics support, all already available for the original middleware.

We also propose the adaptation of the plugin's networking layer to remove all references to the .NET standard API on mobile devices. The use of the standard API imposes a restriction for the plugin, since Unity only gives access for this API at mobile devices in its professional version. This means that it's currently impossible to build applications for Android, iOS and Windows Phone using the plugin without a Unity Pro license. However, this limitation would be removed with the replacement of the networking layer with native networking libraries for these platforms, broadening the range of potential uses for this solution.

At last, our current efforts are also placed in the creation of other games using the platform. These games aim to stress not only how the use of a game engine can reduce the development effort, but also enable the evaluation of new genres and the collection of important metrics regarding the scalability of the solution.

References

AITENBICHLER, E., KANGASHARJU, J., AND MÜHLHÄUSER, M. 2007. MundoCore: A Light-weight Infrastructure for Pervasive Computing. *Pervasive Mob. Comput. 3*, 4, 332–361.

BUZETO, F. N., PAULA, C. B., CASTANHO, C. D., AND JACOBI, R. P. 2010. DSOA: A Service Oriented Architecture for Ubiquitous Applications. In *Advances in Grid and Pervasive Computing*, P. Bellavista, R.-S. Chang, H.-C. Chao, S.-F. Lin, and P. Sloot, Eds., vol. 6104 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 183–192.

BUZETO, F. N., CASTANHO, C. D., AND JACOBI, R. P. 2011. uP: A Lightweight Protocol for Services in Smart Spaces. In *Ubi-Media Computing (U-Media), 2011 4th International Conference on*, Ieee, 25–30.

BUZETO, F. N., CAPRETZ, M. A. M., CASTANHO, C. D., AND JACOBI, R. P. 2013. uOS: A Resource Rerouting Middleware for Ubiquitous Games. In *Ubiquitous Intelligence and Computing, 2013 IEEE 10th International Conference on and 10th International Conference on Autonomic and Trusted Computing (UIC/ATC)*, Ieee, 88–95.

BUZETO, F., 2010. Um conjunto de soluções para a construção de aplicativos de computação ubíqua.

DA COSTA, C. A., YAMIN, A. C., AND GEYER, C. F. R. 2008. Toward a General Software Infrastructure for Ubiquitous Computing. *IEEE Pervasive Computing 7*, 1, 64–73.

GREGORY, J. 2009. *Game Engine Architecture*. CRC Press, Boca Raton, FL, USA.

MODAHL, M., BAGRAK, I., WOLENETZ, M., HUTTO, P., AND RAMACHANDRAN, U. 2004. MediaBroker: an architecture for pervasive computing. In *Pervasive Computing and Communications, 2004. PerCom 2004. Proceedings of the Second IEEE Annual Conference on*, 253–262.

RABIN, S. 2010. *Introduction To Game Development, Second Edition*. Cengage Learning, Independence, KY, USA.

SACRAMENTO, V., ENDLER, M., RUBINSZTEJN, H. K., LIMA, L. S., GONCALVES, K., NASCIMENTO, F. N., AND BUENO, G. A. 2004. MoCA: A Middleware for Developing Collaborative Applications for Mobile Users. *Distributed Systems Online, IEEE 5*, 10, 2.

SCHULTE, W. R., AND NATIS, Y. V. 1996. *Service Oriented Architectures Parts 1 and 2*. Gartner.

SOUSA, J. A. P., AND GARLAN, D. 2002. Aura: An Architectural Framework for User Mobility in Ubiquitous Computing Environments. In *Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance*, Kluwer, B.V., Deventer, The Netherlands, The Netherlands, WICSA 3, 29–43.

STACH, C. 2012. GameworkA Framework Approach for Customizable Pervasive Applications. *mirlabs.org 4*, 66–75.

WEISER, M., AND BROWN, J. J. S. 1996. Designing Calm Technology. *POWERGRID JOURNAL 1*, 1–5.

WEISER, M. 1991. The computer for the 21st century. *Scientific american 3*, 3, 66–75.