Service Oriented Interactive Media (SOIM) Engines Enabled by Optimized Resource Sharing

Aly, Mahy; Franke, Michael;
Kretz, Moritz; Schamel, Folker
Spinor GmbH
Munich, Germany
{ mahy.aly, michael.franke, moritz.kretz, fms }
@spinor.com

Simoens, Pieter
Ghent University - iMinds
Ghent, Belgium
pieter.simoens@intec.ugent.be

***Abstract*—In the same way as cloud computing, Software as a Service (SaaS) and Content Centric Networking (CCN) triggered a new class of software architectures fundamentally different from traditional desktop software, service oriented networking (SON) suggests a new class of media engine technologies, which we call Service Oriented Interactive Media (SOIM) engines. This includes a new approach for game engines and more generally interactive media engines for entertainment, training, educational and dashboard applications. Porting traditional game engines and interactive media engines to the cloud without fundamentally changing the architecture, as done frequently, can enable already various advantages of cloud computing for such kinds of applications, for example simple and transparent upgrading of content and unified user experience on all end-user devices. This paper discusses a new architecture for game engines and interactive media engines fundamentally designed for cloud and SON. Main advantages of SOIM engines are significantly higher resource efficiency, leading to a fraction of cloud hosting costs. SOIM engines achieve these benefits by multi-layered data sharing, efficiently handling many input and output channels for video, audio, and 3D world synchronization, and smart user session and session slot management. Architecture and results of a prototype implementation of a SOIM engine are discussed.**

Keywords—cloud gaming; service oriented interactive media engines; SOIM; resource sharing; resource optimization; service oriented networking; remote rendering; 3D engines

I. INTRODUCTION

The advent of cloud computing and ubiquitous broadband network access has created new prospects for delivering interactive software. For reasons of piracy and cost reduction, applications are nowadays being delivered following the Software-as-a-Service model. Instead of installing the entire application at the client, the output is streamed from the cloud-hosted application to the client [1].

The transition from single-user locally running applications to internet based delivery models has basically triggered new service-oriented architectures for software development [2]. Cloud-based applications must handle simultaneously multiple application tenants. Instead of instantiating the entire application per user, the pay-as-you-go billing models in cloud computing mandates the efficient usage of compute and memory storage by sharing resources where possible.

Interactive media and entertainment software face the need to adapt to the architectural implications that are introduced alongside the new infrastructure in order to optimize its use. Currently, it can be admitted that in the process of this architectural transition, the interactive media industry, in large part of it, is still lagging behind. Since the launch of that industry and mostly throughout its history, games, or in case of server-based multi-user games, thick game clients have been single-user applications running completely on end-user devices. These needs had a significant impact on the architecture of the game and media engines being the software which provides core functionalities like input handling, game simulation, AI, scripting, sound and rendering. Typical engine architectures were normally built around these single-user assumptions. For example, a typical game engine architecture assumes that there is one single active 3D world and therefore the logic handling assets like geometry, textures and shaders is not adequately prepared for these resources to be shared by different logical users.

The emerging of cloud gaming drew new requirements. Many users should be handled by fewer instances of engines, thus requiring a suitable architecture of 1:n relationships of the engine software components. However, game engines have often been developed over a longer period of time and therefore could not be easily redesigned without having to re-implement a lot of features differently. Accordingly, the industry was caused to slow down in terms of adapting to the new demands of cloud gaming. As a result, most games ported to cloud platforms are simply running as one software instance per user, usually in a virtualized environment. In this respect, a new architecture for game and media engines supporting the required aforementioned 1:n relationships of components is missing.

The transition from single-user to service oriented architectures is getting even more complicated because today's media engines are typically not only using classical object-oriented or component case software architectures, but also have multiple layers of abstraction to provide user-friendly tools, up to pure graphical based instruments, to media designers. These tools abstract a lot of underlying technical details, and subsequently have to undergo major changes for cloud-optimized paradigms.

In this paper, we share experiences of the Development Team at Spinor as regards the redesign of our service-oriented media engine (SOIM) Shark 3D. Shark 3D is a proprietary media engine developed by Spinor [3], utilized for the creation of several award-winning game titles and used daily in virtual studio applications of major broadcasting companies. The software provides an extended production-grade framework for creating interactive media applications like games and other real-time applications of movie and TV industry.

Concerning this kind of software, the most resource-consuming factor with regard to computation power and memory are the world simulation (often combined with physics simulation) as well as the handling of assets like 3D models and the rendering of the output image which must be performed on specific, highly parallelized hardware, the GPUs. We provide an overview of the challenges that occurred during our software transition to support cloud-based hosting. We further elaborate on our approaches to optimize the HW/SW resource consumption of multi-user applications through sharing memory between instances where possible.

The paper is organized as follows: Related work is described in section II, whereas in section III, the workflow for creating scenes using graphs used by modern 3D media engines is illustrated. Section IV and V go more into implementation details to build the foundation of how sessions and session slots can be used to optimize resource usage implemented in Shark 3D. Section VI provides a set of standard use cases, that can be entirely improved by employing the optimized techniques explained above. Section VII shows an overview of setups that implement these use cases in different ways. Some experiments measuring differences in memory space used were also carried out, the results of which are shown in section VIII. The last section puts forward an outlook for future development in this connection.

II. Related Work

Related work on developing cloud-native media applications is mostly situated in the domain of cloud gaming. Our discussion is classified into two categories: alternative developer frameworks and strategies for cloud resource sharing. The best known alternative development frameworks to Shark 3D in the gaming and media market are Unity 3D [4] and Unreal [5]. Unity has included cloud support for running multi-player servers, as well as for management and organization tasks, but not for rendering in the cloud.

Other related work is found in the domain of cloud resource sharing. PS Now is a cloud gaming service covering technologies of Gaikai and OnLive which were both acquired by Sony. [6] hints at a separation between user sessions at the hardware level. A full hardware stack (CPU, GPU, memory) is assigned to each player, although realized on the same motherboard. NVIDIA GRID allows for efficient capturing and encoding of GPU output and GPU sharing. It offers two approaches [7]. The first approach applies a 1:1 mapping between GPU and OS. Although this is more efficient than offering a full hardware stack per user session, there is still the overhead of running a separate OS instance per session. The second possibility is called shimming and provides isolation through light sandboxes on a single OS.

Hou et al. [8] have integrated NVIDIA GRID GPU in the open source Gaming Anywhere cloud gaming platform. The main focus of the authors is to assign each VM its own GPU. Although the goal of the authors is identical to ours, our approach would allow higher optimization of resource consumption, especially texture memory because of application-level support. A similar approach is followed in [9], however by means of a new framework developed by the authors themselves. In [10], GPU memory is shared amongst different application instances by detecting already loaded content via hashing. This is performed on the driver level, whereas our approach implements this logic on the developer framework itself. This enables us to exploit not only more detailed information (e.g. resource comparison by file name), but also leverage on additional information unavailable at the driver level. For example, we can assign players having an almost identical set of resources to the same server even before starting the application, thus increasing the number of sharable resources. Another advantage of the Shark 3D approach is that the separation occurs in the application rather than by VM, resulting in the avoidance of duplication of main memory content for multiple instances.

III. Hierarchical node-based application definitions using templates

Assets, like 3D models, textures and sounds, play a key role with respect to resource usage optimization as they partly describe a virtual world. These assets commonly have a relatively large memory footprint compared to the program logic. Hence, today's software can possibly consume some hundred megabytes up to several gigabytes of hard disk space. On loading, such data as sounds and scripts has to be moved to main memory, while 3D models, textures and shaders to GPU memory. It might also be the case that some processing is required, for example textures often have to be decompressed and converted to GPU compatible format before moving it to the GPU memory.

In this section, we describe how Shark 3D organizes these assets in a tree structure and how this organization is adapted to cloud deployment.

A. *Graph based editing*

Typically today's professional engines provide graphical tools for the designer to define the virtual world's structure as graphs. In case of Shark 3D we use a tree for defining the scene augmented by (run time) references between nodes of different branches so that at runtime a directed graph is formed. These graphs often include nodes for physics behaviour, artificial intelligence, animations, visual elements and sounds. Moreover, like most other game engines, Shark 3D employs the concept of templates, which allows pre-defining parts of the graph such as complete characters and cars, and reusing it several times in the virtual world, often with the possibility of parameterizing these templates, e.g. by different logic scripts or models.

B. *Runtime and editing*

The runtime environment of the Shark 3D software consists of a very thin framework application which more or less loads configuration resources as well as instantiates and configures objects accordingly. This ensures flexibility since it implies that even the "basic" functionality of a game engine like instantiating a renderer is not performed automatically but is rather aligned with these configuration resources which are generated using the editing tools.

In the Shark 3D engine, the editing of scenes is done on a running instance of the engine by connecting it to the editing tools and delivering new configurations to the runtime in the event of changing the graph or node properties. From the runtime perspective there is no technical difference between editing and running the final application. How the graph is translated into runtime configurations is described later.

Any extensions and changes in behavior necessary for editing (e.g. different behavior or editor camera and input) is defined on a higher level using either different nodes or a generic socket-based communication protocol. Any changes made in the tree or to node attributes are immediately translated into configuration data chunks and dynamically loaded by the running engine.

IV. TRANSLATION OF DEFINITION GRAPHS INTO DESCRIPTIONS OF THE RUN-TIME ENGINE

The Shark 3D node tree offers the possibility of defining templates which can be used anywhere in the tree. Each usage of a template is "inlined" during translation of the tree thus leading to an outcome equivalent to a repeated insertion by the user of all nodes inside the template. Therefore, each node in the tree may be available multiple times, however each time the context may differ depending on where the template is used. This results in a 1:n relationship that has to be managed.

With this growing number of nodes, especially those used in (often nested) templates, it can be concluded that the translation time will be longer when done brute force. In order to shorten the duration of compiling the tree into a runtime description, sub-trees resulting in identical outcome are translated for once and reused. Nevertheless, since the outcome also depends on the context in which the node resides (in the sense that templates can be parameterized and there may be implicit references from nodes to direct or indirect logical ancestors in the tree), it can be noted that combining the 1:n relationship caused by multiple usage of a template is a complex task.

Nodes may also be instantiated at runtime, for example by using the so-called producer node, where the whole subtree defined underneath it is instantiated upon sending a produce message to the runtime representation of the given node. This adds a possible 1:m relationship for each of the above mentioned 1:n relationships. For editing and inspection purposes, the editor logic has to track all runtime instantiations of a node. This yields a two-step process of translating the node tree into run-time instances of engine components, as shown in Figure 1.

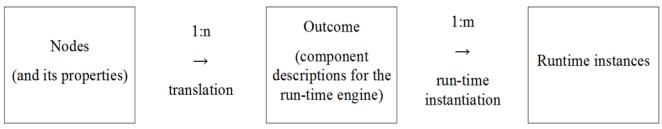

Figure 1. Steps for translating definition graphs into runtime instances.

A. *Translation of the tree into a runtime description*

The translation step includes the following operations:

- Resolving templates, which may be nested. This means that during tree traversal, the templates are "inlined" when they are used and the parameters are resolved.
- Resolving implicit references of a child to a direct and indirect logical ancestor.
- A parent node collecting information from direct children. An example is a node responsible for instantiating a renderer instance demanding a reference to all viewports defined as child nodes.
- Creating the outcome descriptions of individual nodes or group of nodes based on this information.

Towards additional optimization, it is upon primary compilation of the whole tree that only incremental changes are likely to be made. This constitutes a rather challenging task in view of the fact that all the relations between the nodes (parent to child, child to parent, etc.) have to be taken into account when recompiling only parts of the tree.

B. *Loading and instantiating the runtime description in the runtime environment*

The run-time instantiation process involves the following operations:

- Loading dynamic libraries containing the code for different components.
- Instantiating and initializing of objects based on outcome descriptions. For each node in the tree normally a number of runtime objects is created. The initialization parameters are derived from node attributes defined in the editor and passed to the configuration files.
- Resolving references to other objects defined by name in the outcome into efficient C/C++ pointers for direct access.
- Instantiating engine components.

This part of the run-time engine is implemented mainly in C++. Changes made in the editor are updated in the runtime by reloading the relevant configurations and resources.

V. RESOURCE SHARING USING SESSIONS AND SESSION SLOTS

A. *Concept*

As already mentioned above, a naive approach of enabling the Shark 3D software for cloud gaming would have been to package the software into a VM or a container (e.g. Docker

image) and to instantiate one image for each session. A session is defined in this case as a logical representation of the group of users who are involved in the same virtual environment. Normally, each running instance would only accommodate one session due to the fact that games were previously developed and ported to cloud servers as single-session applications. An instance started upon a user's request, simulates the game, renders the resulting 3D image and sends the rendered output as a video stream to be displayed on the user's end device. The user's input is sent from the client's device to the game instance to provide a feedback channel.

This approach is far from being optimal with respect to the use of the underlying hardware resources: If multiple users are running the same application, chances are high that they at least partly access the same data, so sharing this data between users would optimize the overall memory footprint. There are also situations where computation results can be shared amongst multiple users, for example physics simulations in a multi-player game.

The solution for this problem resides in sharing a single running engine instance amongst several users and handling the separation of different users inside the engine rather than using the environment like the separate VMs. The media engine maintains better knowledge about resource usage and how to optimize resource sharing compared to the VM manager. The game engine is also cognizant about which users are using which 3D states and therefore which models and other assets are required. In turn, this can influence the distribution of users over the running instances.

B. *Introducing Session Slots*

To this end, Shark 3D as a service oriented interactive media engine allows one running instance of a service to handle multiple independent sessions to be used by different groups of users.

The maximum number of possible sessions that can be run in one engine instance is limited by the amount of (virtualized) resources allocated to the engine. This introduces the concept of session slots, which explains the idea that each session started in the running engine occupies one of the available slots, which is then released when the session has been terminated. When all available slots are occupied, no further sessions can be started until another slot is released. Besides depending largely on the specific application, the number of available session slots is volatile in a sense that it relies on the resources used by the running sessions. It cannot therefore be predefined but must be reported to the cloud management software on a regular basis.

On the other hand, this indicates that the application setup defines which components must be instantiated separately for each user. This can be implemented by using different nodes in the tree. This guarantees flexibility that for each application, it can be determined which parts are to be instantiated per user and which parts are to be shared.

The component-based architecture of Shark 3D engine described in section IV has been modified to accommodate the creation of multiple-session services. Accordingly, it can be used by the developer to define factories for instantiating session-specific node graphs, which access and use shared data (e.g. assets). Such shared data can then be used by independent 3D states that define the virtual worlds of the services.

C. *Implementation*

Resources possibly shared between different sessions of the same application can be divided into several categories as follows:

- On-disk resources: these include the program files and assets. Shark 3D offers a number of object types to store different kinds of assets such as 3D models, textures and sounds.
- GPU resources: such as vertex buffers which contain the vertices of the models and shaders.
- CPU resources: excessive calculations especially for simulating physics need to be performed by the CPU. Additional computation power is also consumed to run the application logic.

In Shark 3D, the management components for these resources are provided as nodes so that the resources can be shared by all the 3D states which are created as children of these nodes.

Furthermore, it is possible to split multi-user services into two components: the world simulation shared amongst all users of the same session and rendering services which are user specific.

D. *Sample Node Setup for Network interconnection*

As mentioned earlier, Shark 3D provides the possibility to create a factory for session slots necessary to allow resource sharing. In the editor, these factories, known as “producers”, can be used to define which functional components (e.g. unique 3D state) must be created for each session. This mostly relates to relatively lightweight C++ objects in contrast to shared data such as textures, sounds or other assets, which can be created and loaded only once for use by the different sessions.

If both the world simulation and rendering services are running as one service on the same machine, then only one state would be sufficient for a session. Otherwise for each rendering service performed on another machine, a new 3D state needs to be created and synchronized with that of the server. A new view must be created for each client once connected to a state. Figure 2 shows a graph example designed in the Shark 3D editor to define session slots using state and network nodes.

VI. USE CASES

In order to set examples of session-based applications that can be implemented via Shark 3D, the following use cases are explored:

- Single and multi-user dashboards.
- Thin client single and multi-player games.

A. *Single and multi-user dashboards*

A dashboard is an interactive 3D environment combining a number of different services dynamically, based on the users’ requests. Therefore, it allows the user to move around and choose one of the services offered such as video streaming of

real time data from various sources or chatting. The video streams are rendered on textures within the 3D world, which could be the same or different for each user even within the same session in case of the multi-user dashboard. To implement this, each user must have a unique 3D state and a view, where the 3D states of all the users joining the same session are synchronized via Shark 3D networking protocol.

B. *Thin client single and multi-player games*

This is a single or multi-player game where the rendering of the 3D scenes may be done on a separate server instance deployed on the network from where it will be sent as a video stream to the end user's device connected to the server. The end users only launch a viewer application (referred to as thin client), which is responsible for decompressing the video stream, displaying it, as well as capturing the user's input and passing it to the server.

VII. Typical Setups

These use cases can be set up in different ways according to both the hardware requirements of the services to be deployed on the network and the users' locations. Examples of the former are represented in the world simulation that requires high CPU power and also in the rendering services which require access to the GPU. Such requirements put limitations on where the service instances should be deployed and whether or not they can run on the same machine. The users' locations on the other hand, determine where the rendering instances need to be deployed as these should be as close to the users as possible. The following reflects the possible scenarios whether or not the services are split for each of the use cases of the dashboard and the game.

A. *World simulation and rendering running on the same machine*

1) Single-user dashboard / game

In this case, an instance of Shark 3D engine is instantiated, offering a number of session slots where each slot is equivalent to a new dashboard or game session comprised of a 3D state and a view. The rendered output is then streamed to the thin client running on the end user's device.

2) Multi-user dashboard

A 3D state and a view are required for each user, hence is the consumption of one session slot per player to allow users to have different output streams as explained earlier. For each group of users sharing the same session, the 3D states must be synchronized via Shark 3D networking protocol.

3) Multi-player game

As for this use case, only one 3D state is required per game session to which all players of the same session are connected. A new view is created for each player.

B. *World simulation and rendering services running on different machines*

1) Single-user dashboard / game

Considering a single-user dashboard or game where a new slot is accorded to each user and no synchronization between states is required, assigning a separate instance for a server and another for a client consumes two session slots instead of one. Hence, the setup mentioned in section VII.A.1) is more efficient.

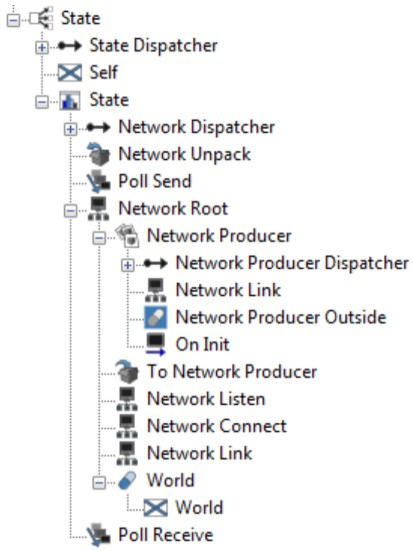

Figure 2. State and network nodes.

2) Multi-user dashboard / game

Similar to the multi-user dashboard requirements, once the rendering services are running on different machines, a new 3D state will be required for each user in the multi-player game. Therefore, it can be safely admitted that the same setup is applicable to both, the dashboard and the game.

In this scenario, a session slot of a server instance containing the decisive 3D state is requested per session. Additionally a session slot of a client instance containing a state replication and a view are also created (consuming one session slot) for each of the players within the session. All of the clients' states should be synchronized with that of the server using Shark 3D networking protocol.

3) Multi-user dashboard / game without centralized world simulation

The last scenario is a combination of the previous two categories where the rendering services are instantiated on different machines depending on the users' distribution, while the world simulation is not separated from the rendering services. In this case, one of the 3D states created for one of the users will be acting as a client as well as a server by opening a listening port for the rest of the clients who would like to join the same session to connect to. The benefit of this approach is that it uses less number of session slots against the previous scenario. It also reduces the number of states that need to be synchronized with each other.

VIII. Results

To test improvements achieved by our approach, a few experiments were conducted. The single-user game scenario was employed for testing, being the one, in which normally separate application instances are launched, e.g. using separate

VMs. This accounts for an ideal scenario that deems resource-sharing as considerably beneficial. The 3D world used to make the measurements is the one depicted in Figure 3. The setup was as follows: One PC acting as server, instantiating a state and a view for each incoming client.

Figure 3. The 3D world used for the measurements.

There was an additional management service written in Python which executed the logic normally handled by a portal application, i.e. processing incoming requests for new services from the clients and distributing them over the servers by returning the respective IP. At the level of our experiments, this service routed all client requests to the server where the measurements were performed.

Two series of experiments were carried out: In the first, for each client, a separate instance of the Shark 3D engine was started to measure the base line; in the second the optimization was used and each client was assigned his own private session but within a common instance of Shark 3D. Both graphics memory and main memory consumption were measured for one to eight clients connected to the server. The results are shown in Figure 4. As the memory of the graphics card used was about 4000 MB, it was able to handle up to five individual instances of the Shark 3D engine. It is noticeable from the graph that for the fifth instance, the limit of the GPU memory was being approached. Therefore, the consumption of memory for this specific instance was less than the previous ones, with respect to GPU memory but higher in terms of that of the CPU, resulting in the same total amount of memory used. Launching a sixth instance of the application simply failed. As for the shared instance, it was able to handle at least eight sessions, with relatively much smaller increases in both CPU and GPU memory usage.

IX. Conclusion

Moving existing interactive media engines to the cloud requires deep architectural adjustments. Key to building efficient software is sharing the underlying hardware resources. To achieve this goal, Shark 3D was adapted to accept multiple independent sessions running in parallel, separating different users and sessions inside the engine. This allows sharing the fundamental resources like memory and CPU power in an efficient manner. The tree based editing in Shark 3D enables software developers to edit this on a fine grained level, allowing adaptation to different possible architectural setups depending

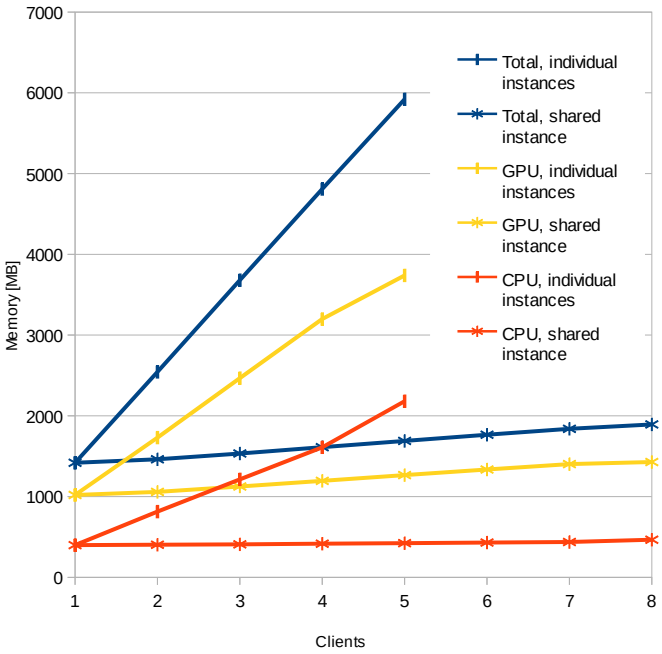

Figure 4. Comparison of memory usage between individual application instances and shared instance offering multiple session slots.

on the kind of software (multi user, single user, shared state, etc.) being developed.

Further challenges not covered in this work target the optimal distribution of sessions amongst the running instances since the more resources (e.g. CPU and GPU memory) can be shared, the more sessions can run on a single instance. On the other hand, assigning a session to a running instance also has impact on the network round trip time depending on the specific location of the running instance. Finding solutions to these issues and optimizing them is the goal of the FP7 funded project FUSION, where Spinor is also contributing as an industry partner. The implementations mentioned before are also used in that project as samples and test cases.

[1] P. Simoens, F. De Turck, B. Dhoedt, and P. Demeester. “Remote display solutions for mobile cloud computing”, Computer, vol. 8, 2011

[2] W. Cai, M. Chen, and V. Leung, “Toward gaming as a service”, Internet Computing, IEEE, vol. 18(3), 2014

[3] Shark 3D – Spinor GmbH, http://www.spinor.com

[4] Unity – Game Engine, https://unity3d.com

[5] Unreal Engine, https://www.unrealengine.com

[6] Extremetech - http://www.extremetech.com/gaming/175005-sonys-playstation-now-uses-custom-designed-hardware-with-eight-ps3s-on-a-single-motherboard

[7] F. Diard, “Cloud Gaming with Nvidia Grid Technologies”, GDC 2014

[8] Q. Huo, C. Qiu, K. Mu, et al. “A Cloud Gaming System Based on NVIDIA GRID GPU”. In Distributed Computing and Applications to Business,Engineering and Science (DCABES), 2014 13th Intl. Symposium on, IEEE, 2014

[9] R. Shea, D. Fu, J. Liu, “Rhizome: utilizing the public cloud to provide 3D gaming infrastructure”. Proceedings of the 6th ACM Multimedia Systems Conference, ACM 2015.

[10] X. Wu, Y. Xia, N. Jing, et al. “CGSharing: Efficient content sharing in GPU-based cloud gaming”. In Low Power Electronics and Design (ISLPED), 2015 IEEE/ACM International Symposium on, IEEE, 2015