Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 75 (2015) 257 – 266

Procedia Computer Science

2015 International Conference on Virtual and Augmented Reality in Education

Adopting a Game Engine for Large, High-Resolution Displays

Anton Sigitov[a]*, David Scherfgen[a], André Hinkenjann[a], Oliver Staadt[b]

[a]*Bonn-Rhein-Sieg University of Applied Sciences, Grantham-Allee 20, 53757 Sankt Augustin, Germany*
[b]*University of Rostock, Institute of Computer Science, Albert-Einstein-Str. 22, 18051 Rostock, Germany*

Abstract

The steadily decreasing prices of display technologies and computer graphics hardware contribute to the increasing popularity of multiple-display environments, like large, high-resolution displays. It is therefore necessary that educational organizations give the new generation of computer scientists an opportunity to become familiar with this kind of technology. However, there is a lack of tools that allow for getting started easily. Existing frameworks and libraries that provide support for multi-display rendering are often complex in understanding, configuration and extension. This is critical especially in educational context where the time that students have for their projects is limited and quite short. These tools are also rather known and used in research communities only, thus providing less benefit for future non-scientists. In this work we present an extension for the Unity game engine. The extension allows – with a small overhead – for implementation of applications that are apt to run on both single-display and multi-display systems. It takes care of the most common issues in the context of distributed and multi-display rendering like frame, camera and animation synchronization, thus reducing and simplifying the first steps into the topic. In conjunction with Unity, which significantly simplifies the creation of different kinds of virtual environments, the extension affords students to build mock-up virtual reality applications for large, high-resolution displays, and to implement and evaluate new interaction techniques and metaphors and visualization concepts. Unity itself, in our experience, is very popular among computer graphics students and therefore familiar to most of them. It is also often employed in projects of both research institutions and commercial organizations; so learning it will provide students with qualification in high demand.

Peer-review under responsibility of organizing committee of the 2015 International Conference on Virtual and Augmented Reality in Education (VARE 2015)

Keywords: large-high-resolution displays; rapid prototyping tool; Unity; tools for education

* Anton Sigitov. Tel.: +49-2241-865-266; fax: +49-2241-865-8266.
E-mail address: Anton.Sigitov@h-brs.de

Peer-review under responsibility of organizing committee of the 2015 International Conference on Virtual and Augmented Reality in Education (VARE 2015)
doi:10.1016/j.procs.2015.12.246

1. Introduction

The significant progress in the areas of display and graphics hardware technologies are reflected in a price decrease for visual output devices and graphics cards as well as in an enormous efficiency increase of the latter. Thus it became possible to build affordable large, high-resolution displays (LHRD). In general, LHRDs differ from mainstream desktop displays in two aspects: physical size and resolution. They can be defined as a combined visual output perceived as a single, continuous visual space that provides significantly more pixels and is distinctly larger by comparison with a normal display. LHRDs are usually built from an array of projectors or LCD displays. Both technologies have their advantages and disadvantages. For example, the individual tiles of an LCD-based display are disjoint because of bezels. This results in a discontinuous or distorted image output. Opposed to LCD-based displays, projector-based displays don't suffer from this problem. Though, they have to struggle with a colour and brightness inconsistency as well as permanently changing alignment, which requires complex and frequent calibration. The main properties of LHRDs are[1]: size, pixel density, resolution, brightness, contrast, viewing angle, bezels, display technology, and form factor. These properties are also applicable for normal desktop displays. In case of LHRDs they vary stronger, though. The application field of LHRDs is very broad. They are used for visualization of common office and multimedia applications, thus foster collaborative work. They enable the visualization of complex datasets, affording a better insight into complex relationships of data entities. Moreover, they enable a full size visualization of industrial constructions and machines, therefore ensure better spatial impression. The typical applications of LHRDs are: command centrals, vehicle design, geospatial imagery, scientific visualization collaboration and tele-immersion, education and training, immersive applications, and public information displays. The survey by Ni et al.[2] gives an extensive overview on LHRD technologies, applications, and challenges.

With regard to steadily growing popularity of LHRDs in different domains, we believe it to be of great importance to provide an opportunity for upcoming generations of computer scientists to become familiar with that type of displays, so they can gain an essential competence for their future career. Developing applications for LHRDs, students will recognize that common, desktop-oriented interaction devices and techniques are not suitable for this kind of system. Thus, they will be forced to experiment with design, and research for better solutions. However, in order to create this opportunity, specific tools have to emerge which will allow a quick and easy start on application development for LHRDs. These tools are necessary in the first place because of the limited time students have at their disposal during semester projects. Most of currently available solutions are too complex in configuration and extension, and offer scarce support. Thus, students have to spend a lot of time learning how to utilize them, before being able to proceed with their ideas. Moreover, none of those tools provide any modules for rapid prototyping, like a visual scene editor, animation libraries, input device libraries, graphical user interface libraries, etc. Therefore, students are often forced to combine multiple libraries which stem from different developers, and which do not necessarily work together flawlessly. Consequently, many projects remain unfinished, and interest in the technology decreases due to negative experience.

Establishing a common ground on the basis of a well-known framework, which provides better support for rapid prototyping, better support on the part of the software developer, and a large developer community will contribute to fluent project progress. As a result, the students' experience may become improved in a positive way, better outcomes may be achieved, and the technology may be promoted. Moreover, having a unified basis will result in less heterogeneous source code throughout the projects, making them more appropriate for communication and advancement.

In this paper we present an extension for the Unity game engine[3] that allows to run Unity applications on LHRDs. Our experience shows that many media informatics students are intimate with Unity and have utilized it for their hobby projects beforehand. The game engine itself provides a lot of useful tools for swift content creation. Thus, we think it could be the common ground we are looking for. Although Unity's primary focus lies on digital games, it is often used for the development of applications in different other domains, like architecture, engineering and construction, simulation in the medical and security fields, serious gaming, and so on. Currently, we intend to utilize Unity for rapid prototyping only, in order to evaluate ideas on interaction and visualization concepts. We hope to produce completed software products for LHRDs later on, though.

The paper is organized as follows: First, we present related work, which contains information on distributed rendering frameworks. Then, in section 3, we describe our large, high-resolution display, called HORNET, which

we use for evaluation purposes. In section 4, we present the extension for Unity alongside with some aspects of adopting an application for an LHRD and some examples. Section 5 concludes the results and presents possible future work.

2. Related Work

In this section, we give a brief overview of frameworks and tools whose goal is to support a developer in porting his application to an LHRD. The following overview is not meant to be a complete list of all available frameworks, but rather to introduce the different approaches pursued by the frameworks. For an extended introduction and comparison of the most current distributed rendering frameworks and toolkits, refer to the survey by Haeyong et al.[4]

WireGL[5] was one of the first frameworks to support the creation of scalable systems for interactive rasterization rendering on a cluster of workstations. The framework is OpenGL-oriented and overrides its API in order to distribute rendering jobs over a network. The OpenGL commands applied by the user are converted into so-called stream filters. These streams are then conveyed to workstations which are in charge of image rendering. In total, there are three types of stream filters: sorting streams into tiles, dispatching streams to the workstations, and reading back a frame buffer from the workstations. For render job distribution, WireGL uses a sort-first rendering approach[6]. The framework supports different display configurations from common single desktop displays up to tiled multi-projector or multi-LCD displays. The project is currently inactive.

The Chromium framework[7] was developed with the goal of improving WireGL in terms of flexibility and more comprehensive functionality. The new features include, but are not limited to: a sort-last approach for image rendering, integration of custom filters, and integration of custom parallel algorithms. Similarly to WireGL, Chromium is bound to OpenGL, thus limited to rasterization rendering techniques only.

The DRONE framework[8] addresses different distributed rendering scenarios: single-screen rendering, multi-screen rendering, remote rendering, and collaborative rendering. The framework is built upon the Network-Integrated Multimedia Middleware library[9] that provides to DRONE such properties like scalability and flexibility. For configuration purposes, a set of different node types is available to the user. These are: manager, render and assembly node. Each node type provides its own unique functionality. For instance, a manager node is in charge of generating render jobs. These are then conveyed to render nodes over a network in a round-robin manner. Render nodes process the jobs and submit generated sub-images to an assembly node. Thanks to its clear interface, DRONE may be combined with any type of render engine, thus supporting ray-based renderers as well as rasterization renderers. However, the fixed pipeline of the framework makes it difficult to integrate custom render job scheduling approaches. This may be crucial for some scenarios, though.

Equalizer[10] is a toolkit for scalable distributed and parallel rendering. It can be employed on both shared-memory systems and on clusters of workstations with a network layer. In the early stages, the toolkit focused on rasterization-based OpenGL rendering only. Nowadays, it is apt to work with all types of rendering engines from rasterization to volume rendering. It offers a set of ready-to-use load-balancing approaches and allows for the implementation and integration of custom techniques for task scheduling.

The Scalable Adoptive Graphics Environment[11,12] (SAGE) was developed with collaborative work in mind. Thus, its focus lies on the integration of multiple visualizations into an LHRD. The main idea is to use users' personal workstations, e.g. laptops, to run different applications and, at the same time, stream the visual output of those applications in form of pixel data to a large display. The data is conveyed then to appropriate SAGE Receivers that drive the individual displays. Coordination of the whole procedure takes place in the Free Space Manager (*FSManager*), which is another SAGE component. It handles requests, windows placement, and synchronization of receivers, to name a few. The SAGE application interface library (SAIL) enables developers to adapt their applications for the environment.

More an approach than an off-the-shelf solution was presented by Rehfeld et al.[13] It utilizes the Hewitt actor model[14] for real-time interactive, distributed rendering systems. Applications developed using this model are expressed in a set of actors. Each actor represents a small module which undertakes some specific job, e.g. physics simulation or image rendering. The communication of actors takes place asynchronously via a messaging system. The approach allows to build efficient distributed rendering systems. However, it requires fundamental changes in many existing toolkits and frameworks, thus making the integration long-winded.

So far, we introduced only freely available frameworks, which emerged from the research and development community. There are also some commercial solutions worth mentioning. For instance, Unity itself offers different license types that include tools for distributed rendering. These license types are not public and have to be requested by the software developer. The price of a license depends on the display configuration and tends to be rather expensive, even for educational organizations. Also, the support of Linux-based systems could not be clarified in correspondence at the time of writing.

Another commercial toolkit is MiddleVR[15]. It provides a powerful graphical configuration tool that allows the user to adapt his Unity application to his hardware environment. MiddleVR supports different types of display environments and input devices and provides distributed rendering capability. There is a free version of the software, but it has stringent limitations on multiple application properties, e.g. the maximum resolution, so it is not really usable on LHRDs. The commercial license types, including the academic license, amount to many thousands Euros, though. The main disadvantage of MiddleVR however is that it does not support Linux-based systems, which are virtually a standard in educational organizations.

Even though the frameworks and toolkits presented in this section provide a good aid in terms of bringing applications to work on LHRDs, there is a set of drawbacks to mention: adapting an application using them is not always straightforward and can take a lot of time; support for free tools is rather poor; configuration is usually laborious; extending a framework might be an issue, since developers have to grapple with an unfamiliar API. In addition, none of the freely available frameworks or toolkits provides a solution for rapid prototyping, which would be of significant importance to educational organizations.

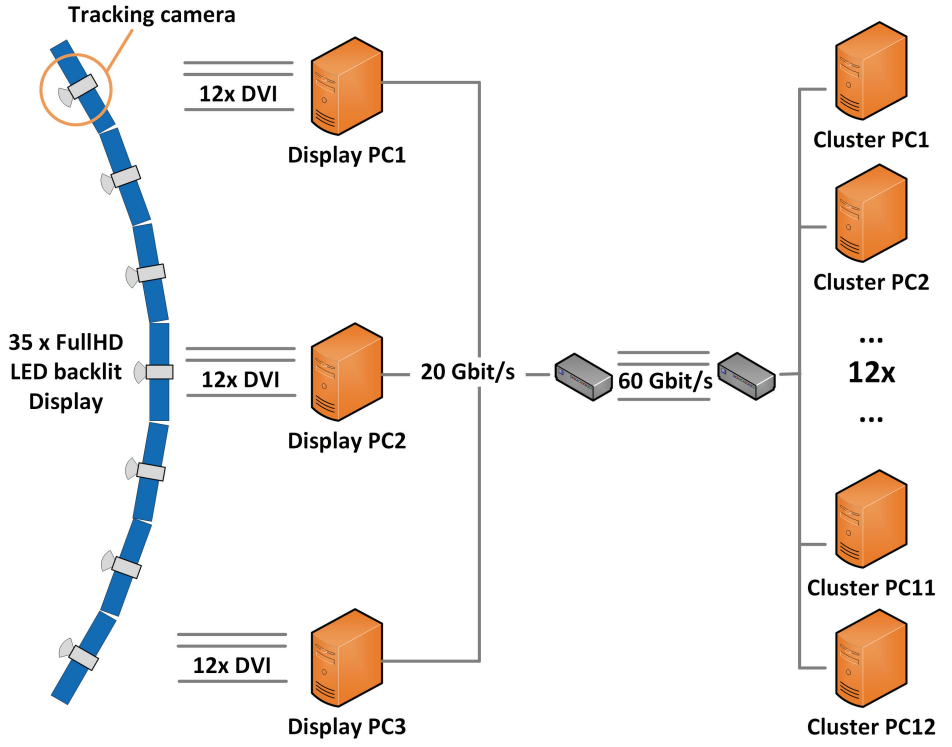

Fig. 1. HORNET component diagram: The 35 displays are driven by a small cluster of three display PCs, which is connected to a second, larger cluster of twelve nodes.

3. The HORNET System

For evaluation purposes, we used an LHRD system called HORNET, which is installed at Bonn-Rhein-Sieg University of Applied Sciences. Fig. 1 depicts a component diagram of the system. HORNET consists of 35 LCD displays with a resolution of 1920×1080 pixels and bezels of less than three millimeters each. The displays build a matrix of seven columns and five rows, providing a total surface area of 7×3 square meters. The cumulative resolution of HORNET adds up to 72 megapixels. The displays provide a pixel density high enough to exceed the human eye's resolution when standing more than two meters away from them. A computer cluster of three nodes drives the displays. Each computer is equipped with three NVIDIA GeForce GTX 780 Ti graphics cards, thus providing in total twelve DVI outputs per node. This small cluster offers enough computing power to run interactive, OpenGL-based applications with a moderate scene size of one million triangles

in full resolution. A second computer cluster of twelve nodes can be addressed over a network link. Each node in that cluster has three NVIDIA GeForce GTX Titan graphics cards. Thus, the cluster may be utilized for tasks of high complexity, like high quality physically based global illumination rendering. Each display node provides two 10 Gbit/s Ethernet links, which are virtually bundled into a single logical link of 20 Gbit/s. Both clusters are interconnected via six optical Ethernet links of 10 Gbit/s each. For comprehensive interaction scenarios with HORNET, an ARTTRACK optical tracking system is available. It comprises seven infrared tracking cameras that provide a tracking area of roughly 5×4 square meters. In our university we use HORNET for experiments in the following research areas: visualization of large data sets, one-to-one high-quality rendering, computer-aided collaborative work, and human-computer interaction.

4. Unity Extension

Based on our previous experience[16,17] we developed an extension for the Unity game engine that allows for running Unity applications on different types of large, high-resolution displays. The extension pursues three goals:

1. Ease the process of application configuration in terms of distributing application instances among compute units and displays.
2. Ease the process of camera configuration in terms of distributing view frustum fragments among application instances.
3. Solve or at least ease the task of synchronizing application instances.

The extension consists of two general components: a software module in form of Unity scripts, and a guidance component that provides hints for adapting applications for LHRDs. The software module itself can be divided into editor and run-time modules. An overview of both software module types is given in sections 4.1 and 4.2.

4.1. Editor Module

The editor module provides a tool for creating a camera system that reflects the geometrical arrangement of displays in an LHRD. In order to configure it, the user has to fill in the displays' dimensions as well as their positions and orientations in space relative to a common coordinate system. The configuration process is aided by visual feedback that provides the user with a graphical representation of the camera system. The visual components of the editor extension are depicted in Fig. 2. Moreover, the editor module allows to configure how the application instances will be distributed among display units. For that purpose, the so-called MPI ranks have to be appropriately assigned to the display units. Based on that information, an MPI configuration file will be generated and then evaluated by the MPI environment when the application starts. Additionally to instance distribution, a manager instance should be defined. The manager instance is a process that receives user input, runs simulations, e.g. animation or physics, and conveys the results to the other instances, which we refer to as worker instances.

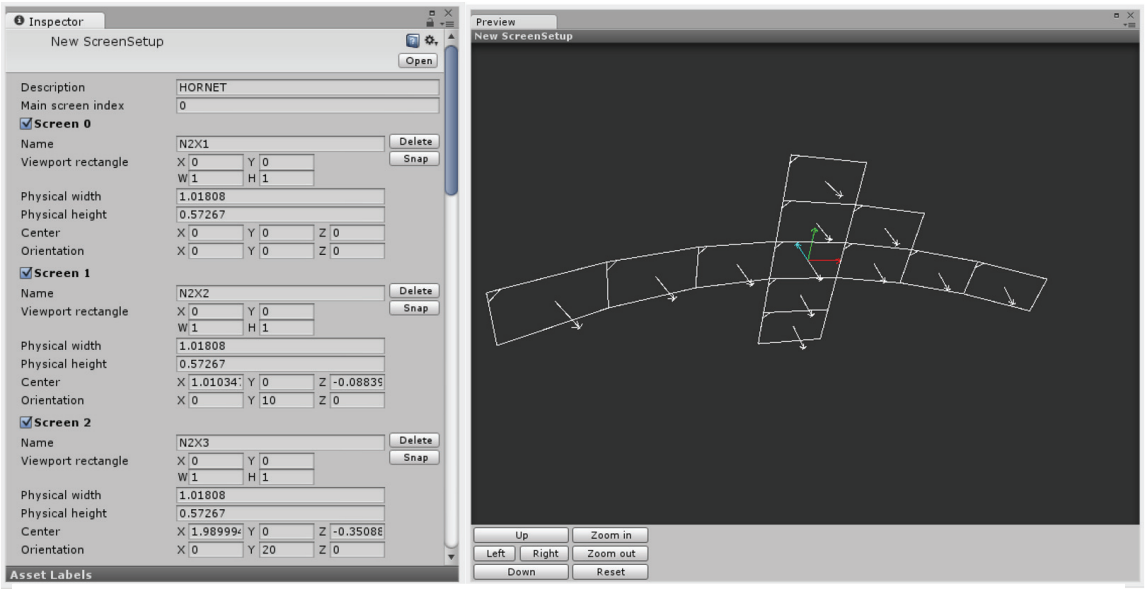

Fig. 2. Unity Camera Configuration Tool – Configuring a curved LHRD: (left) custom editor for displays, (right) graphical representation of the camera system

4.2. Run-Time Module

The run-time software module consists of multiple Unity scripts. The primary purpose of that module is the synchronization of application instances. In the given context, we distinguish between four synchronization types: frame synchronization, camera state synchronization, transform state synchronization, and event synchronization.

We call application instances frame-synchronized if, at every point in time, the sequence numbers of frames they process are equal. However, this ideal case is hard to ensure. Usually, there is a relative short time span where the sequence numbers differ by one. In order to achieve visual output synchronization between the processes, a locking mechanism has to be added right before the function call that swaps the back and front buffers, which makes the previously rendered image visible. There are two fundamental types of such locks: hardware-based and software-based. Hardware-based locks like NVIDIA's Swap Sync[18] require very expensive professional graphics cards. It is the best way of synchronizing buffer swapping, though. Opposed to hardware-based locks, software-based locks are less precise, but totally free. They are usually implemented on the application layer. Different strategies might be incorporated for that purpose. We use the MPI barrier mechanism from the OpenMPI library[19].

A barrier is usually placed right before a buffer swapping function call. Since no process can pass through the barrier before all other processes have reached it, it serves as a synchronization point, thus ensuring that all processes display their frames roughly at the same time and run at equal speeds. In order to introduce such a barrier, the programmer needs access to the source code of the framework to manipulate it. However, there is no such possibility in the case of Unity, since its source code is closed. While it would be possible to use techniques such as DLL hooking, we implemented a different workaround based on the concept of coroutines[20] available in Unity. A coroutine is a type of function with the unique ability of being able to actively interrupt its own execution and return control to Unity. All local variables and the instruction pointer are saved when this occurs and restored later on when the coroutine resumes, similarly to a thread when it is interrupted by the operating system. By default, active coroutines are called every frame. There are some functions which allow to alter that behavior, though. They are: *WaitForSeconds*, *WaitForFixedUpdate*, and *WaitForEndOfFrame*. The latter will stop the execution of a coroutine and activate it again just before displaying the frame on the screen, according to the Unity manual. Thus, a barrier

placed right after this function yields feasible results. In our extension, we provide a script with a coroutine which implements this approach.

Another script, called *CameraSyncer*, provides functionality for camera synchronization. Camera state synchronization takes place in two stages: First, the camera-specific values are collected by the manager instance and conveyed to all worker instances. These values could be: position, rotation, frustum parameters, projection mode, culling mode, and background color, to name a few. Currently, not all values of the Unity camera class are supported, but we are working towards it. During the second stage, the received values are applied, and if head tracking is used, a new camera frustum is calculated in accordance with the created display configuration to match the user's perspective. Camera state synchronization proceeds in a broadcast manner. It means that all worker instances will receive the values, since they all need them to produce a correct output.

Transform state synchronization has its name from the Unity component called *Transform*. That component manages three game object properties: position, rotation and scale. Transform state synchronization is implemented in a script called *TransformSyncer* and is responsible for synchronizing the mentioned properties. Oppositely to camera state synchronization, it incorporates a publisher-subscriber communication pattern. This procedure may be described as follows:

1. All application instances share the same object.
2. One manager instance runs all simulations on the object, e.g. animation and physics, which usually affect the properties of the *Transform* component. Based on the camera system configuration, the manager instance determines which camera can see the object and submits the values to the respective worker instances.
3. The worker instances manage only the visual representation of the object. They do not simulate physics or animations, neither do they react to user input. Their main task is rendering. Right before a buffer swap synchronization barrier, they go into a listening mode and wait for updates from the manager instance.
4. After the manager instance has distributed all updates, it broadcasts the *EndOfFrame* message and then paces to the synchronization barrier. The *EndOfFrame* message forces the worker instances to leave the listening mode and approach the synchronization barrier as well.
5. All application instances pass through the barrier, starting a new frame.

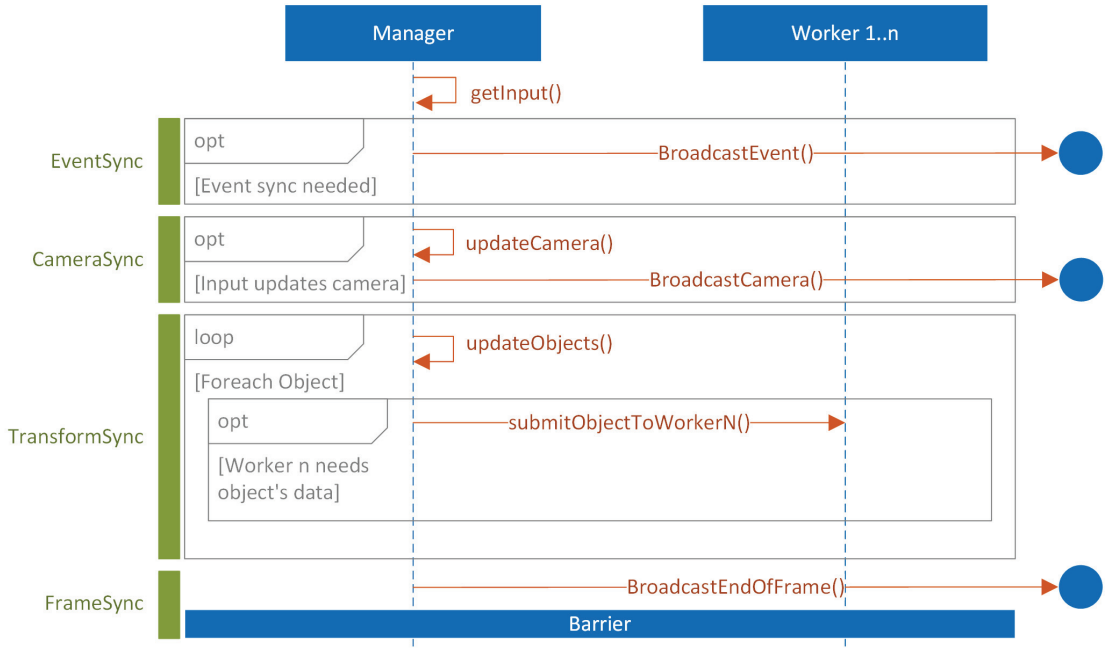

Fig. 3. Overview of the four different types of synchronization used in our approach

From the statements made in points 2 and 3, we can see that the same objects have to differ in terms of components they contain: only the manager instance is allowed to have components which are relevant to simulation. In order to make the transition of an application from a single-display to a multiple-display environment easier for the developer, the *TransformSyncer* script provides an array field where references to a number of arbitrary object components may be set. When running in the multi-display environment, the script will use these references to remove the specified components from the object within the worker instances.

Event synchronization usually has to take place if an object has to be created, removed or modified in response to user input or some other event. That is because only the manager instance may acquire user input and compute simulations. Currently, we do not provide any off-the-shelf solution for that type of synchronization, but rather give hints about how to handle such cases. Usually, the developer has to modify his script where the event fires.

An overview of all different types of synchronization is given in Fig. 3. Overview of the four different types of synchronization Fig. 3.

4.3. Proof of Concept

We evaluated our extension on the HORNET system described in section 3. We used scenes of different types and complexity. Two applications were taken from the Unity Asset store: one 2D application, called “2D Platformer”, and one 3D application, called “Unity Labs”. Another two applications were of our own implementation: one 2D arcade game, and one static 3D scene. All applications were instantiated 35 times. All instances ran in parallel and were distributed as follows: one manager instance and eleven worker instances on display node 1, twelve worker instances on display node 2, and eleven worker instances on display node 3. The synchronization of the instances took place using the OpenMPI library and the provided routines described in section 4.2. The visual outputs of the applications are presented in Fig. 4. The borders of the HORNET display were cropped in the images for better representation. The Unity quality settings were set to the default “Good” preset. Additionally, vertical synchronization (*VSync Count*) was set to *Every VBlank* and *Anti-Aliasing* to “*4× Multi Sampling*”. For interaction purposes, a common keyboard was utilized. All four applications ran at stable frame rates of least 30 frames per seconds. A few lags could be recognized occasionally. However, these could be tied to a naïve, unoptimized implementation.

Fig. 4. Sample applications: (top left) custom 2D arcade game; (top right) custom static 3D scene; (bottom left) modified “2D Platformer” application from the Unity Asset store; (bottom right) modified “Unity Labs” application from the Unity Asset store

5. Conclusion and Future Work

In this paper we emphasized the necessity for rapid prototyping tools for LHRDs in educational organizations. We gave a brief overview of existing frameworks and toolkits, which allow to adapt an application to work on LHRDs. They are often complex in configuration, learning and extension, though. Students in universities do not have enough time to learn these tools in order to success in their projects. Additionally, these frameworks do not provide any modules for rapid prototyping. At the same time, game engines like Unity and the Unreal Engine are well known among many students, provide good support, a large community and many off-the-shelf solutions. Utilizing these tools for LHRDs will allow for better outcomes, resulting in better experience and increased motivation. Based on these considerations, we developed an extension for Unity, which allows for developing and adapting existing applications for LHRDs. We described the extension's components and their interaction.

In the future, we like to investigate the possibility of developing a similar extension for the Unreal Engine. Its advantage over Unity is its open source code. Thus, it can be manipulated and extended more extensively and on a lower level. We are also working on porting our bicycle simulator FIVIS[21], made in Unity, to HORNET. By offering human-scale visualization to the user, we are expecting to generate a more immersive experience. Another interesting opportunity has recently been provided with the release of Unity 5. Starting from that version, it is possible to build headless applications (without graphical output) for Linux-based systems, thus we become able to leverage the large cluster of twelve nodes (mentioned in section 3) for different complex simulations. For instance, a complex agent-based simulation of road participants (AVeSi project[22]), which is utilized in the FIVIS simulator, might be transferred to the cluster in order to allow for the simulation of a larger number of agents. Further, we would like to explore the possibility of using tracked tablet computers as additional dynamic virtual cameras, as in the work of Spindler et al.[23]

References

1. Andrews C, Endert A, Yost B, and North C. Information visualization on large, high-resolution displays: issues, challenges, and opportunities. *Information Visualization* 2011; **10**(4): 341–355.
2. Ni T, Schmidt GS, Staadt OG, Livingston MA, Ball R, May R. A Survey of Large High-Resolution Display Technologies, Techniques, and Applications. *Virtual Reality Conference,* 2006, pp. 223–236, 25–29 March 2006.
3. Unity, http://unity3d.com/, accessed on 03.07.2015.
4. Haeyong C, Andrews C, North C. A Survey of Software Frameworks for Cluster-Based Large High-Resolution Displays. *IEEE Transactions on Visualization and Computer Graphics* 2014; **20**(8): 1158–1177.
5. Humphreys G, Eldridge M, Buck I, Stoll G, Everett M, Hanrahan P. WireGL: a scalable graphics system for clusters. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques, SIGGRAPH '01* pp. 129–140, New York, NY, USA, 2001. ACM.
6. Molnar S, Cox M, Ellsworth D, Fuchs H. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications* 1994; **14**: 23–32.
7. Humphreys G, Houston M, Ng R, Frank R, Ahern S, Kirchner PD, Klosowski JT. Chromium: a stream-processing framework for interactive rendering on clusters. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques, SIGGRAPH '02,* pp. 693–702, New York, NY, USA, 2002. ACM.
8. Repplinger M, Löffler A, Rubinstein D, Slusallek P. DRONE: A Flexible Framework for Distributed Rendering and Display. In *Proceedings of the 5th International Symposium on Advances in Visual Computing*: Part I, ISVC '09, pp. 975–986, Berlin, Heidelberg, 2009. Springer-Verlag.
9. Lohse M, Winter F, Repplinger M, Slusallek P. Network-Integrated multimedia middleware (NMM). In *Proceedings of the 16th ACM international conference on Multimedia, MM '08*, pp. 1081–1084, New York, NY, USA, 2008. ACM.
10. Eilemann S, Makhinya M, Pajarola R. Equalizer: A Scalable Parallel Rendering Framework. *IEEE Transactions on Visualization and Computer Graphics* 2009; **15**(3): 436–452.
11. Jeong B, Renambot L, Jagodic R, Singhm R, Aguilera J, Johnson A, Leigh J. High-Performance Dynamic Graphics Streaming for Scalable Adaptive Graphics Environment. In *Proceedings of the ACM/IEEE SC 2006 Conference*, pp. 24–24, 11–17 November 2006.
12. Renambot L, Rao A, Singh R, Jeong B, Krishnaprasad N, Vishwanath V, Chandrasekhar V, Schwarz N, Spale A, Zhang C, Goldman G, Leigh J, Johnson A. SAGE: the Scalable Adaptive Graphics Environment. In *Proceedings of WACE 2004*, 23 September 2004.
13. Rehfeld S, Tramberend H, Latoschik ME. An actor-based distribution model for realtime interactive systems. In *6th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS),* 2013, pp. 9–16. IEEE, 2013.
14. Hewitt C, Bishop P, Steiger R. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence,* pp. 235–245. Morgan Kaufmann Publishers Inc., 1973.
15. MiddleVR, http://www.middlevr.com/, accessed on 03.07.2015
16. Sigitov A, Roth T, Mannuß F, Hinkenjann A. DRiVE: An Example of Distributed Rendering in Virtual Environments. In *6th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS),* 2013, pp. 33–40. IEEE, 2013.

17. Sigitov A, Roth T, Hinkenjann A. Enabling Global Illumination Rendering on Large, High-Resolution Displays. *8th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, March 2015.
18. NVIDIA. Frame Lock. http://www.nvidia.com/object/IO_10794.html, accessed on 03.07.2015.
19. OpenMPI - Open Source High Performance Computing, http://www.open-mpi.org/, accessed on 03.07.2015.
20. Unity Coroutines, http://docs.unity3d.com/Manual/Coroutines.html, accessed on 03.07.2015.
21. Herpers R, Scherfgen D, Kutz M, Hartmann U, Schulzyk O, Reinert D, Steiner H. FIVIS - A Bicycle Simulation System. In *Proceedings of the 5th European Conference of the International Federation for Medical and Biological Engineering (IFMBE 09)*, 2009; **25**(4), pp. 2132–2135.
22. Krueger F, Seele S, Herpers R, Bauckhage C, Becker P. Dynamic Emotional States Based on Personality Profiles for Adaptive Agent Behavior Patterns. *11th Workshop Virtuelle Realität und Augmented Reality der GI-Fachgruppe VR/AR,* 2014.
23. Spindler M, Büschel W, Dachselt R. Use Your Head: Tangible Windows for 3D Information Spaces in a Tabletop Environment. In *Proceedings of the 2012 ACM International Conference on Interactive Tabletops and Surfaces, pp. 245–254, ITS '12,* Cambridge, MA, USA, 2012. ACM.