Distributed Unity Applications

Evaluation of Approaches

Anton Sigitov[1(✉)], Oliver Staadt[2], and André Hinkenjann[1]

[1] Institute of Visual Computing, University Bonn-Rhein-Sieg of Applied Sciences, Sankt Augustin, Germany
{Anton.Sigitov,Andre.Hinkenjann}@h-brs.de
[2] Institute of Computer Science, University of Rostock, Rostock, Germany
Oliver.Staadt@uni-rostock.de

Abstract. There is a need for rapid prototyping tools for large, high-resolution displays (LHRDs) in both scientific and commercial domains. That is, the area of LHRDs is still poorly explored and possesses no established standards, thus developers have to experiment a lot with new interaction and visualization concepts. Therefore, a rapid prototyping tool for LHRDs has to undertake two functions: ease the process of application development, and make an application runnable on a broad range of LHRD setups. The latter comprises a challenge, since most LHRDs are driven by multiple compute nodes and require distributed applications.

Unity engine became a popular tool for rapid prototyping, since it eases the development process by means of a visual scene editor, animation libraries, input device libraries, graphical user interface libraries etc. However, it will charge developers with a high fee in order to make an application LHRD compatible. In our previous work, we developed an extension for Unity engine that allows to run Unity applications on LHRDs.

In this work we consider different static vs. dynamic camera/world conditions of distributed applications; and propose and evaluate different Unity specific approaches within the scope of 2D and 3D applications for these scenarios. The primary focus of the evaluation lays on world state synchronization, which is a common issue in distributed applications.

Keywords: Unity · Distributed rendering · Large, high-resolution displays

1 Introduction

Rapid prototyping tools are of substantial value for researchers since they lower overhead of hypothesis/concept evaluation significantly. Unfortunately, a choice of the rapid prototyping tools for large, high-resolution displays is scarce. To address this issue, we developed a lean, easy to use extension [1] for Unity game engine[1]. It enables developers to create and build Unity applications for different LHRD setups. For the purpose of world state synchronization, we developed a set of routines. These differ mainly in how visibility of virtual objects is determined for a particular application instance.

[1] http://unity3d.com, last access on 7.04.2016.

DOI: 10.1007/978-3-319-40548-3_23

In this work, we evaluated three Unity specific approaches for world state synchronization in distributed applications. We considered different scenarios, e.g. dynamic/static camera, 3D/2D world, different rendering and lightning conditions, in order to achieve more expressive evaluation results.

2 Related Work

Chung et al. [2] classified software frameworks for LHRDs in two categories based on task distribution models: distributed application and distributed renderer. Another way to discriminate these tools is by focusing on the type of information that they distribute across individual application instances. Here we highlight three types: draw-call, pixel data, and change list.

Using a draw call distribution approach, a framework intercepts draw commands executed by a *manager instance*. It processes and subdivides them into sub draw-calls in appliance with the system configuration. Next, it conveys the sub draw-calls to *output instances*. The representative frameworks that implement this approach are: Chromium [3], ClusterGL [4], and CGLX [5].

Within the scope of a pixel data distribution approach, one or more remote nodes generate image data. Based on the LHRD configuration, the *manager instance* splits data and distributes it among *output instances*. The frameworks SAGE [6] and DMX[2], to name but a few, implement this approach.

Following a change list distribution approach, the *manager instance* submits changes made on virtual objects to *output instances*. Using this information individual *output instances* become able to update the state of their virtual world replicas. Implementations of that approach could be found, for instance, in OpenSG [7], Garuda [8], and DRiVE [9].

Our extension [1] combines both task distribution models. The extension requires multiple application instances with each having its own copy of a virtual world. One instance undertakes a manager functionality and becomes responsible for event and world state synchronization. Such configuration reflects the distributed application model. At the same time, the *manager instance* takes on all world simulation routines, thus decoupling rendering from simulation and making other application instances to pure *output instances*. This is characteristic for the distributed renderer model. In addition to its hybrid task distribution model, our extension makes use of the change list approach for world state synchronization. Thanks to Unity's inherent abilities and our extension, we created a promising rapid prototyping tool for LHRDs.

3 Approaches

We developed three Unity specific approaches for world state synchronization in distributed Unity applications: *Naïve*, *Adaptive_F* (Frustum-based), and *Adaptive_C* (Camera-based). The approaches handle virtual objects that own an *MPIView*

[2] http://dmx.sourceforge.net/, last access on 7.04.2016.

component. This component tags an object as distributable and encapsulates a unique object ID that ensures a proper object match across individual application instances.

Naïve. The *Naïve* approach broadcasts state information of all distributable objects. It does not matter if a particular application instance requires this information for a current frame. The approach provides the highest integrity level for applications, as all instances have the same world state at any time. This, for instance, allows correct shadow calculation (for objects that lay outside the instance's partial frustum, but their shadows within) on *output instances*. However, *Naïve* causes high network overhead, which depends on both number of virtual objects and number of application instances.

Adaptive_F. The *Adaptive_F* approach tests distributable objects against each virtual camera's frustum. It conveys state information of an object to an application instance only if the object lays within a frustum of the instance's camera. *Adaptive_F* lowers network overhead. Dependences on number of virtual objects and number of application instances remain though, since each object undergoes the frustum check for each virtual camera. This in turn increases computational overhead on the *manager instance*. The approach does not ensure proper shadow calculation, as it cannot detect shadows within a frustum.

Adaptive_C. The *Adaptive_C* approach enforces the manager instance to create and maintain replicas of all virtual cameras. Each camera replica will raise the Unity callback *OnWillRenderObject* for each virtual object it is going to render in a current frame. In this way, *Adaptive_C* determines object visibility for a particular camera. Hence, application instances receive state information of only visible distributable objects. *Adaptive_C* abolishes dependence on number of virtual objects. Dependence on number of application instances remains due to virtual camera replicas. Also, overhead on the *manager instance* rises, since each camera replica renders an image. *Adaptive_C* and *Adaptive_F* have equal network overhead. Similar to *Adaptive_F, Adaptive_C* does not safeguard correct shadow generation.

At the heart of our extension lies the centralized control model [10]. The entire world simulation process takes place only on the *manager instance*. The m*anager instance* distributes a new world state across *output instances* at the end of each frame. The o*utput instance* is responsible for state receiving, state applying, rendering, and output to a connected display unit.

4 Evaluation Setup

The evaluation was performed using a large curved tiled display wall comprising 35 LCD displays (Fig. 1), ordered through a seven (column) by five (row) grid. Each of the columns has a relative angle difference of 10 degrees along the Y-axis to adjacent columns, as such creating a slight curvature. The LCD displays are 46" panels with a 1080 p resolution, resulting in a total of 72 megapixels. The installation is driven by a cluster of three PCs, each equipped with three GeForce GTX 780 Ti, providing a total of twelve outputs per PC.

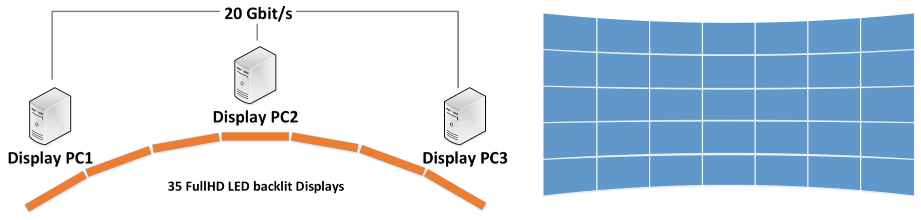

Fig. 1. The Hornet system: (left) view from above; (right) frontal view

We implemented two types of 3D evaluation scenarios (static (s) and dynamic (d)) with three conditions (standard (1), shadow (2), and multiple lights (3)) each. In the standard condition, there was one directional light, which had a random position in a virtual scene and cast no shadows. This was the only difference to the shadow condition, where the directional light cast shadows. In the multiple lights condition, there were one directional light and eight point lights. Neither of them cast shadows.

There were five stages in all type-condition combinations. There were 64 distributable virtual objects at the beginning of the first stage. The number of objects doubled with each subsequent stage. For every stage, we logged lowest frame time, highest frame time, average frame time, and total stage time. We ignored stage preparation frames in order to avoid interference of administrative overhead. Also, we turned the dynamic batching and occlusion culling methods off in order to mitigate impact of Unity's inter routines on the evaluation results.

The static scenarios comprised static, not synchronized virtual cameras. The cameras shared the same origin and orientation. Each camera's frustum made up a part of a large mutual frustum. Distributable virtual objects (cubes of the same size and different colors) appeared at random position and with random orientation within the bounds of the mutual frustum. Each object revolved one degree per frame around its local Y-axis.

The dynamic scenarios had a camera setup comparable to one in the static scenarios. However, the cameras were dynamic and synchronized. The mutual cameras' origin lay at the local center of a ring-shaped volume. Distributable virtual objects emerged within the ring-shaped volume. Each object had a random position and orientation. The objects and the cameras rotated one degree per frame around their local Y-axis. Hence, only a subset of the objects lay within the cameras' mutual frustum at a time.

For a 2D case, we implemented only one type-condition combination, namely static-standard. The 2D scenario incorporated a set of cameras with an orthogonal projection instead of perspective projection, like in the 3D scenarios. It also made use of Unity Sprites instead of 3D cubes.

We determined two baselines for every type-condition variation. The baseline (B1) reflects the application's performance at the standard condition with frame synchronization only. Additionally, *MPIView* components were disabled in order to prevent Unity to make any calls on them. The baseline (B2) shows the performance at a specific condition with frame synchronization and enabled *MPIView* components. Although *MPIView* components were active no world state synchronization took place, as class methods contained no logic.

5 Results

Figure 2 depicts the evaluation results. At every condition the *Naïve* approach was less efficient in comparison to *Adaptive_F* and *Adaptive_C*. The average frame time increased linearly with the number of distributable objects. With 1024 distributable objects it performed with 4 frames per second. However, it is the only approach that ensures proper shadow visualization currently. Moreover, it adds no computational overhead on the *manager instance*.

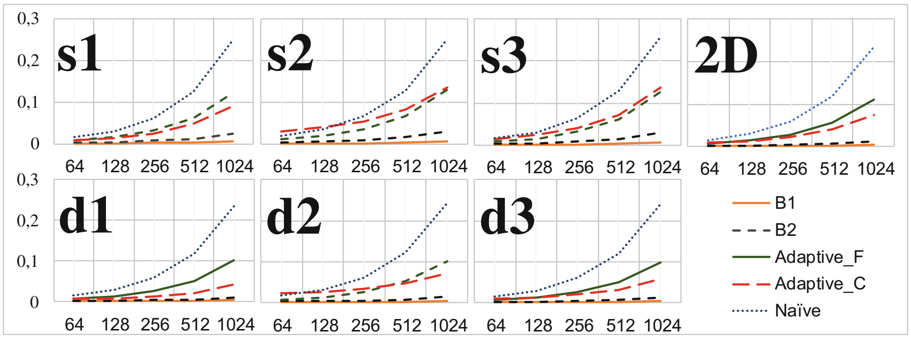

Fig. 2. Evaluation results: (s1) static – standard; (s2) static – shadow; (s3) static – multiple lights; (d1) dynamic – standard; (d2) dynamic – shadow; (d3) dynamic – multiple lights; (2D) static – standard – 2D. The X-Axis shows number of distributable virtual objects. The Y-Axis shows the average frame time in seconds (Color figure online)

Likewise, *Adaptive_F* depends on number of distributable objects strongly. It performed at least twice as fast as the *Naïve* approach though. This ascribes to lower visibility computation overhead in comparison to synchronization overhead. By small numbers of distributable objects, it outperformed the *Adaptive_C* approach too. However, the approach is incongruous if shadow visualization is desirable.

With a low number of distributable objects, *Adaptive_C* performed less efficient than the *Naïve* and the *Adaptive_F* approaches. This is due to rendering overhead caused by virtual camera replicas. The effect is well observable at the shadow conditions (s2) and (d2). It is more resistant to a number of distributable objects in a virtual scene though. As a result, with increased number of objects it performed significantly better in comparison to other two approaches. The transcendence is less apparent in the variations (s2) and (s3). However, inspection of logged data for both variations disclosed that average frame time of the *Adaptive_F* approach had been increasing at a faster pace. Similar to *Adaptive_F, Adaptive_C* ensures no proper shadows.

6 Conclusion

In this work, we evaluated three Unity specific approaches for world state synchronization in distributed applications. We applied the approaches to the different scenarios and to the different virtual world conditions to make the evaluation more comprehensive and expressive. The evaluation revealed that the *Naïve* approach performs worst. It is the only alternative to ensure correct shadow generation on *output instances* though. The comparison between *Adaptive_F* and *Adaptive_C* elucidated that *Adaptive_F* is more due for applications with a small number of visible distributable objects, while *Adaptive_C* behaves better with a large number of distributable objects.

References

1. Sigitov, A., Scherfgen, D., Hinkenjann, A., Staadt, O.: Adopting a game engine for large, high-resolution displays. Procedia Comput. Sci. **75**, 257–266 (2015)
2. Chung, H., Andrews, C., North, C.: A survey of software frameworks for cluster-based large high-resolution displays. IEEE Trans. Vis. Comput. Graph. **20**, 1158–1177 (2014)
3. Humphreys, G., Houston, M., Ng, R., Frank, R., Ahern, S., Kirchner, P.D., Klosowski, J.T.: Chromium: a stream-processing framework for interactive rendering on clusters. ACM Trans. Graph. **21**, 693–702 (2002)
4. Neal, B., Hunkin, P., McGregor, A.: Distributed OpenGL rendering in network bandwidth constrained environments. In: Eurographics Symposium on Parallel Graphics and Visualization (2011)
5. Doerr, K., Kuester, F.: CGLX: a scalable, high-performance visualization framework for networked display environments. IEEE Trans. Vis. Comput. Graph. **17**, 320–332 (2011)
6. Jeong, B., Renambot, L., Jagodic, R., Singh, R., Aguilera, J., Johnson, A., Leigh, J.: High-performance dynamic graphics streaming for scalable adaptive graphics environment. In: SC 2006 Conference, Proceedings of the ACM/IEEE, pp. 24–24. IEEE (2006)
7. Roth, M., Voss, G., Reiners, D.: Multi-threading and clustering for scene graph systems. Comput. Graph. **28**, 63–66 (2004)
8. Nirnimesh, H.P., Narayanan, P.J.: Garuda: a scalable tiled display wall using commodity PCs. IEEE Trans. Vis. Comput. Graph. **13**, 864–877 (2007)
9. 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), pp. 33–40. IEEE (2013)
10. Chalmers, A., Davis, T., Reinhard, E. (eds.): Practical parallel rendering. AK Peters, Natick (2002)