Creating distributed rendering applications

A. V. Bogdanov[a], **A. I. Ivashchenko**[b], **A. I. Belezeko**[c]

Saint Petersburg State University,
7/9 Universitetskaya emb., Saint Petersburg, 199034, Russia

E-mail: [a] bogdanov@csa.ru, [b] aiivashchenko@cc.spbu.ru, [c] alexey.belezeko@gmail.com

This article discusses the aspect of visualization appliance by the usage of distributed computing systems. It describes possible practical scope for several visualization technologies on the basis of an example for the construction such an application exploiting modern technologies and ready-made solutions. An extra attention is paid to the selection of software packages and to the provisioning of a final result to the end user taking in mind the issue of unusual computer graphics output approaches. In the light of these questions this study is carrying out an analysis of implementation's hardware and software features.

Keywords: Distributed Computing, Computer Graphics, Visualization

Research was carried out using computational resources provided by Resource Center "Computer Center of SPbU"(http://cc.spbu.ru/) and supported by grants of Russian Foundation for Basic Research (projects no. 16-07-01111, 16-07-00886, 16-07-01113) and Saint Petersburg State University (project no. 0.37.155.2014).

Introduction

In this report we will be talking about ways and approaches of distributed rendering applications creation and considering about the current state of related core technologies and tools that could be used to scale out, while trying to answer the question "When do we actually need to appeal for distributed rendering?"

First of all we should reach an agreement what we will be calling a «rendering application». In our case it actually doesn't matter if it is a batch rendering system, which produces an animation, video or image, or it is a real-time application we will be talking mostly about. Even a video playback could be called so here. So let's assume that rendering application in our case is a program that is able to use graphics hardware acceleration to produce a visual frame. It is important, because there is a place for software rendering also, which utilizes CPU.

The next question that should be answered is "Why do we need a distributed rendering?" There are two generic reasons, which always cause to appeal for distributed computing. It is lack of computation power and lack of memory. Here they appear as huge array of graphic primitives, which is not handable neither by GPU memory amount, neither by its computation capabilities, or as labor-intensive post-processing scenarios and massive shader subroutines.

There are four common problems of distributed rendering. First of all it is the same out-of-core problem. Even if we a moving forward to the distributed system to achieve the goal, we can still met it when data is really huge. The second important thing is a networking resource, which is always getting used intensively by media streams. Methods of compression and final frame composition, which could help here, are also big topics to talk about and really depend on specific case. So if the system should be a "silver bullet", its configuration possibilities should be as flexible as possible. And the last one is load balancing, because equal distribution of workload doesn't lead to the equal rendering time, due to the fact that real rendering job is defined by camera position and orientation. All these things are getting even heavier to handle if we are talking about interactive applications.

Graphics API

Let's take a view on a low level graphics APIs now, because it is a core technology, which allows communication with GPU. Distributed computing, actually, begins with a parallel processing, so we need to figure out how its done with different standards. Mostly all implementations are supporting a parallel processing but the issue is how it is getting handled. For the Windows environment we will talk about DirectX, because it is native for the system. In a Linux environment OpenGL mostly is the only thing to think about. Also there is a Vulkan API presented few months ago, which is promising true cross platform support with a lower level of computing resource.

There are 3 commonly used versions of DirectX API, 9, 11 and 12, and we are interested about the last one, because it brings a lot of outstanding features, but it is available only on Windows 10 and enabled in most cases only for the latest or hi-end GPUs [AMD DX12; NVIDIA DX12]. One of involvements is a Linked GPU adapter, that allows to work with several cards connected with SLI or CrossFire in a unified space [DX12 Adapter]. It means that finally multiple GPUs can work together to produce a single frame, instead of rendering frames in turn. Another feature is opposite. Unlinked technique allows using cards of different models or even of different vendors in a single application. DirectX 9 could be sort of sapid because of SLI support introduced, but 10th and 11th versions are mostly about shaders.

There are several things that should be said about OpenGL multi GPU usage. First, there is a technique called ping-pong PBO, which enables the transfer of pixel buffers between cards [Biermann et al., 2013]. Each application can have only one active GL context at the same time. It means that if you are using multiple not SLI/CrossFire connected GPUs, you have to run multiple processes. But it is possible to transfer data between contexts and substitute contexts between applications. There is also an opportunity to use GPUs of different vendors for OpenCL computations, but there is no way to get this setup work for graphics. In a short, we need some custom tools here to get parallel.

Vulkan states for low level interface without any abstractions for unified GPU management available. However, it allows to process the pipeline in parallel with a command queue system and has another interesting features, such as a possibility to simultaneously load multiple GPU drivers and interoperability with other graphics API [Vulkan FAQ]. Some features like OpenCL integration and native Multi-GPU are missing right now? but the could be found at roadmap for the next releases [Vulkan FAQ].

Image compositing and compression

Another thing we need to take an eye on is a list of generic operations performed on every generated frame, which are not related to particular task that should be solved.

First of all is compositing methods, because they are straightly affect the workload distribution. There are three basic methods available, which are also used as a basis for more and the first, and the simplest one, is a sort-first [Molnar et al., 1994]. The screen here is getting split to the dedicated camera zones, which are processed separately. So, it called sort-first, because the distribution appears before geometry calculation. Since the camera position is always changing, that method cannot guarantee an equal distribution. It works perfectly, when the most of the screen is covered by image.

The next one is sort-last, and, as it could be guessed from definition, the distribution here is happening after scene calculation, so it distributes objects already [Molnar et al., 1994]. This method cold bring some unnecessary workload too, because some objects could be overlapped by another ones.

Sort middle is a last one, and it is not widely used in real time applications, due to the fact it takes some extra time to produce and network overhead, but it is the most effective [Molnar et al., 1994]. First, each node calculates geometry, and then workload is getting redistributed equally.

The next thing is a compression methods applied to the resulting frame before transfer to reduce bandwidth usage. Right here we actually need to pick between compression speed and possible compression rates, since the speed is a major priority for real-time application. Actually, there are two common algorithms suitable to this criteria [Lin, Hao, 2005]. Run-length encoding stands for the fast and lossless method, however offers slightly low compression rates. Due to the nature of algorithm the highest efficiency could be reached in case of big same colored areas appeared on the frame. Chroma subsampling or YUV encoding is a lossy method, which offers a tight compression based on image color palettes separation, as it could be understood from denotation. The main disadvantage of that approach is an artifact appearance on color layers joint edges.

Tools and libraries

All distributed rendering tools could be split into two categories. The first one is aimed to create a streaming systems, the second is for truly distributed applications.

The most popular streaming system in this area is Chromium [Chromium]. Its workflow is very simple. The bootstrapping tool starts the original application and replaces the real OpenGL library to the fake one. Mothership server provides a configuration for every node involved into the process, where the GL commands are getting streamed.

VTK is a next tool to consider about. This library, probably, is used in the majority of scientific applications as a visualization solution, since its maturity, reliability and stability [VTK]. There are three VTK-based packages that we liked the most. ParaView is a general purpose application, which is capable for processing of data in coming variety of formats commonly used by popular scientific packages, giving an interactive model as a result [Ayachit, 2015]. It deals greatly with big amounts of volumetric data, which is typical for scientific calculations. Other two are VisIt and VisBox. The first one, as a ParaView, is aimed to cover the whole domain, while the second is originally developed to provide a virtual or augmented reality experience for CAVE environment.

One of the most exceptional solutions in this field is an Equalizer library aimed on OpenGL-based applications development. Unlike the Chromium, an application written with the usage of that framework becomes truly distributed, since it gets a capability to be executed on each node involved into the process, thereby increasing application performance and reduces the load on the network [Eilemann, 2013]. Well-designed API allows to integrate Equalizer almost with any graphics engine. Flexible configuration system makes possible to customize the application for the usage with variety of output devices. Two more notable features here are the existence of the load balancing mechanism and native support of InfiniBand protocol.

OpenSceneGraph is a graphics engine based on idea of scene graph data structure [Wang, Qian, 2010]. Encapsulated components could be divided into two general groups: leaves and nodes. The leaves are representing unique entities on the scene: 3D objects, animations, textures, shaders, etc. Nodes are either a group of objects, either their transformation operations. Such an approach not only allows to process large scenes more effectively, but also could help in load distribution process optimization somehow, since at any moment of time the program is able to determine the objects involved into the current scope of view.

Distribution in action

We have tried out the Chromium capabilities on a virtual rendering cluster made for test purposes on GpuTest benchmarking suite [GpuTest, 2014]. Detailed description of that test case and environment could be found in [Bogdanov et al., 2016]. The first one, called FurMark, uses shading subroutines to calculate the fur coating for the transitioning object. The second, called GiMark, spawns multiple similar objects, also in transition.

For the FurMark we were able to increase the frame rate almost in two times, from 34 frames per second up to 77, in sort-first mode with 4 nodes involved. Addition of extra computing nodes led us to the loss of performance due to bandwidth limitation. For the GiMark a sort-last compositing technique had been applied due to its nature. The finest frame rate had been achieved on 6 computing devices, which means that workload is distributed more evenly here, and it was three times higher than the initial and equals to 45 frames. Connection of more nodes also brought us to deterioration.

So, basically, it could be said, that performance of streaming-based distributed rendering system could be improved, but it is extremely depends on cluster's networking capabilities. Nevertheless, it still has an appliance scope where target application's source codes are not available.

The distributed application case is presented with a simple molecular structure viewer we have developed with Equalizer and OpenSceneGraph [Eq and OSG, 2010]. For the test purposes we have took a beta-galactosidase (5A1A) molecule in rotation, without any compression in a FullHD resolution [Bartesaghi et al., 2015]. Atomic structure appearance, which includes almost 33000 elements, is generated in a procedure way, and workload is distributed in a sort-last manner.

On the side-by-side comparison of rotation animation produced on one node with approximately ten frames per second and on eight nodes with approximately sixty frames difference feels really well. Since that application is truly distributed, we were able to get such improvement rates. There is a lot of space for optimization, which could be made, including compression, advanced compositing techniques and of course in image generation process. The network during the test had been in use at the level of 40 percent, so we are still able to add several nodes. Taking also in mind that our hardware is designed for other purposes, and the generic setup will actually have several more powerful GPUs per node, processed scenes potentially could be more demanding and the image quality and resolution would be just outstanding.

Conclusion

This paper represents some analysis of computer graphics and distributed computing relation area. Moreover, examples of interaction with the deployment-ready visualization software, middleware tools and libraries for distributed rendering applications development have been shown. Provided cases are illustrating the possibility to increase the frame rate and thereby positively affect the interactivity without taking any compromises towards the image quality. It was also found that the main limiting factor for that type of computations is a network throughput capability due to nature of the real-time applications.

As a further possible area of work we can note the development of full stack infrastructure solution for visualization of scientific computations results and simulations of various processes. Another interesting task is to check the possibilities of Tesla K40 and newer models of NVIDIA's general purpose graphics accelerators since they are supporting a rendering mode. Taking into account the fact that these cards have 2 SoC on board with shared memory, it would be fancy to check a scenario whether one of chips performs a simulation and the other goes straight for result visualization.

Список литературы

AMD DirectX 12 Technology. — http://www.amd.com/en-us/innovations/software-technologies/directx12. — Accessed: 14.04.2016.

Ayachit U. The ParaView Guide: A Parallel Visualization Application. — Kitware, 2015.

Bartesaghi A., Merk A., Banerjee S. et al. 2.2 A resolution cryo-EM structure of beta-galactosidase in complex with a cell-permeant inhibitor. — 2015. — may. http://dx.doi.org/10.2210/pdb5a1a/pdb.

Biermann R., Carter N., Cornish D. et al. Pixel Buffer Object. — https://www.opengl.org/registry/specs/ARB/pixel_buffer_object.txt. — 2013. — October 8,. — Accessed: 06.04.2016.

Bogdanov A., Ivashchenko A., Belezeko A. et al. Building a Virtual Cluster for 3D Graphics Applications // Computational Science and Its Applications – ICCSA 2016: 16th International Conference, Beijing, China, July 4-7, 2016, Proceedings, Part II. — Cham: Springer International Publishing, 2016. — P. 276–291.

Chromium Documentation. — http://chromium.sourceforge.net/doc/index.html. Accessed: 17.01.2016.

DirectX 12 Supported GPUs GeForce. — http://www.geforce.com/hardware/technology/dx12/supported-gpus?field_gpu_type_value=All. — Accessed: 14.04.2016.

Eilemann S. Equalizer Programming and User Guide. — Eyescale Software GmbH, 2013. — July 26,.

Frequently Asked Questions - LunarG. https://lunarg.com/frequently-asked-questions/. Accessed: 07.04.2016.

GpuTest - Cross-Platform GPU Stress Test and OpenGL Benchmark for Windows, Linux and OS X | Geeks3D.com. http://www.geeks3d.com/gputest/. 2014. March 4,. Accessed: 16.11.2015.

Lin T., Hao P. Compound image compression for real-time computer screen image transmission // IEEE Transactions on Image Processing. — 2005. — aug. — Vol. 14, no. 8. — P. 993–1005.

Molnar S., Cox M., Ellsworth D., Fuchs H. A sorting classification of parallel rendering // IEEE Computer Graphics and Applications. — 1994. — jul. — Vol. 14, no. 4. — P. 23–32.

Multi-Engine and Multi-Adapter Synchronization. https://msdn.microsoft.com/en-us/library/windows/desktop/dn933254(v=vs.85).aspx. Accessed: 11.04.2016.

OpenSceneGraph and Equalizer: Tech. rep. — Neuchâtel, Switzerland: Eyescale Software GmbH, 2010. — April.

VTK - The Visualization Toolkit. — http://www.vtk.org/. — Accessed: 19.01.2016.

Wang R., Qian X. The Journey into OpenSceneGraph // OpenSceneGraph 3.0: Beginner's Guide. — Packt Publishing, 2010. — P. 7–18.