Distributed Rendering

Overview

In this project, I investigate the load balancing techniques used in distributed rendering systems. Distributed rendering system are seen in many applications like scalable displays, multiple projector systems, CAVE rendering systems, stereo systems and render farms. The goal is to achieve more efficient rendering performance through multiple computers. By dividing the workload between computers, we can spread it across many GPU’s and in turn speed up the rendering time. Basically, divide and conquer.

Background

While dividing the workload across multiple computers may sound like an intuitive solution, the overall process is quite tedious and challenging. The first problem we must address is how do we communicate or share the render information (i.e. meshes, render states, scene updates). There are two solutions to this problem. We can send this information over the network each render frame and have each render node render part of the scene. This naïve solution gets out of hand really quick since an animation can easily exceed 5000 frames. We can optimize the performance by only sending update information (i.e. position, orientation, shader parameters), but then this requires that every client have a copy of the whole scene. This leads to our second solution, which is to share the scene data through a distributed filesystem. This in turn centralizes all the scene data. The network is now cleared up and the render nodes can now use it to return render results.

The second problem is how do divide up the workload? There are multiple solutions to this problem. The most common ones are sort-first, sort-last and pixel decomposition.

Naïve – Divides the workload in a per frame basis where each render node renders a whole frame by itself.

Advantages: Each frame is independent of each other. No compositing needed.
Disadvantages: The render time for each frame will take a long time.
**Sort-First** – Divides the workload up by cutting the scene into tiles. It does this by providing each render node a piece of viewing volume of the final render frame.

**Advantages:** It's very easy to encode render tasks over the network. All you need to do it pass along a view and projection matrix, which represents a slice of the final render frame.

**Disadvantages:** Load imbalance occurs frequently due to the scene complexity in each tile can change dramatically.

**Sort-Last** – Divides the workload up by rendering entities on a per render node basis. Instead of cutting the render frame into tiles, sort-last has each render node render an entity and composites the final image comparing the depth of each object.

**Advantages:** Load imbalance is less frequent since you're sending asking each render node to render one entity.

**Disadvantage:** Render nodes not only have to send back the render result, but also the depth buffer. Composition itself also becomes a bottleneck.

Both of these solutions have performance bottlenecks, but how do we solve that? What can we do to balance the workload?

**Hypothesis**

Clever partitioning of scene geometry will yield better performance than time-based load balancing solutions in sort-first distribution approaches.

**Approach**

Since my focus is the load balancing aspect of distributed rendering, I wanted to use an open source project for my test bed. It's impossible for me to write my own distributed rendering system in 6 weeks time. I found a bunch of open source projects that fit my criteria.

- **WireGL** “is an active research project at the Stanford University Computer Graphics Lab to explore cluster rendering systems."

- **Chromium** “is a system for interactive rendering on clusters of graphics workstations. Various parallel rendering techniques such as sort-first and sort-last may be implemented with Chromium. Furthermore, Chromium allows filtering and manipulation of OpenGL command streams for non-invasive rendering algorithms."

- **DrQueue** “is a powerful open source distributed render farm manager, used for a range of applications across the visual effects industry and for general batch
processing jobs in science, engineering and finance. DrQueue is licensed under GNU GPL Version 3.”

I began my research with WireGL through a paper I found off the IEEE Xplore database. As I was investigating the WireGL core, I found very little documentation on how to extend the system. I decided to look into DrQueue, but I was faced with the same problem of the lack of documentation. I then looked into using Chromium and it just so happened to be the successor of WireGL. The community was active and there was documentation for extending the system.

Chromium offers a large repository of examples and tutorials on how to write stream processing units (SPU). SPU’s are small modules that can be added to the system to manipulate incoming and outgoing render data. Using SPU’s, I can build my own load-balancing module where I can then do my analysis. I needed to learn the system before I can build the module. This proved to be a challenge. The Chromium system core was very complicated and writing the SPU involved changing a large deal of an already complex system. The module involved extending parts of the system that wasn’t well documented, so I started talking to the community members. I was going back and forth with the community. Eventually, I was getting the impression that SPU I’m trying to develop is much more complicated then I had first expected. There was no way for me to retrieve statistics from the render nodes without having to change part of the core and there was no documentation on the how to retrieve geometry information. The progress and project seemed grim.

I continued to post on the mailing list and one of the members suggested that I should just use a system that’s called Equalizer. It has a cleaner architecture and a much more extendable framework, but it’s still work in progress. There wasn’t much documentation on what I want to do, but I was in contact with the lead developer. My best bet would be to switch over...

Equalizer “is the standard middleware to create and deploy parallel OpenGL-based applications. It enables applications to benefit from multiple graphics cards, processors and computers to scale rendering performance, visual quality and display size.”

My approach will be to use Equalizer as a test bed and implement two different load-balancing algorithms outlined by the two papers I’ve chosen.

“A load-balancing strategy for sort-first distributed rendering”
Abraham, F.; Celes, W.; Cerqueira, R.; Campos, J.L.;

“DPBP: a sort-first parallel rendering algorithm for distributed rendering environments”
Huabing Zhu; Kai Yun Chan; Lizhe Wang; Wentong Cai;
I'll then write a render application that will drive the Equalizer server and pass render tasks down to the render nodes.

**Metrics**

The basic metrics to prove my hypothesis will be to measure the amount of the time it takes to render time a frame for each algorithm. The variables will include:

- Number of render nodes – How many render nodes will be used.
- Scene Complexity – How many entities will be rendered in the scene?
- Animation Length – How long the animation will run.

**Paper Analysis**

- “A load-balancing strategy for sort-first distributed rendering”
  Abraham, F.; Celes, W.; Cerqueira, R.; Campos, J.L.;

As mentioned earlier, load imbalance plague sort-first rendering. This paper presents an algorithm that dynamically adjusts the tile sizes by examining the history of previous frames. The algorithm examines the amount of time it takes a node to render a tile. If a node takes a long time to render a tile, the algorithm will shrink the area of which the node is responsible. Essentially, each render node will converge to a point where they all finish rendering at the same time.

All tiles will first be horizontally adjust and then vertically with respect to it’s own new horizontal row. This in turn preserves the spatial coherence of the render frame. The algorithm indirectly divides a complex scene equally into each tile. This algorithm can also be applied to volume renderers since it’s not operating on the scene level.

The paper goes onto describing the architecture of their multithreaded distributed rendering system. It provides good pipeline to assign render tasks to nodes on one thread while on another thread composite returned render results from nodes. This doesn’t apply in the Equalizer system that I’m using as a test bed.

- “DPBP: a sort-first parallel rendering algorithm for distributed rendering environments”
  Huabing Zhu; Kai Yun Chan; Lizhe Wang; Wentong Cai;
This paper presents a different approach to load balancing. Instead of examining the system after the frame has been rendered, this algorithm examines the geometry before it gets rendered. After examining the paper multiple times, the basic algorithm works as follows:

1. **Preprocessing** – Project each entity’s 3D bounding box into screen-space
   
   First, you transform the bounding box into camera space. Next, find the minimum z value of each point. Now, find the min/max xy points. Finally, create a quad using min z for depth and then the min/max xy points. Transform those points into projection space and you’ll have your 2D bounding boxes.

2. **Sorting 2D Bounding Boxes**
   
   Create 2 arrays; the first one will be the size of the width of the screen space and the other will be the size of the height of the screen space. Traverse the bounding boxes and add them to the array at the corresponding location in screen space.

3. **Scan Pixel Bucket**
   
   You will now use the array to build a tree that will equal the number of render nodes you have available for rendering. This is done through a partition rate tree where you want to split the information you gathered into a 1:1 ratio of objects on each side. You continue to split the tree until you have leaves equaling the number of render nodes.

Upon reading this paper and examining the algorithm, I realized that my hypothesis would probably be wrong. One of the main bottlenecks we have in this algorithm is trying to communicate the bounding information to the server’s load balancer. Just examining the complexity, for N entities that you are rendering there will N * size of the bounding data in the packet to send to the server. A scene can easily exceed over 10,000 objects in a scene and easily over 10,000 frames. There are some optimizations that we can apply to this algorithm that I will describe in later sections.

- “Distributed Rendering for Scalable Displays”
  Humphreys, G.; Buck, I.; Eldridge, M.; Hanrahan, P.;
  Supercomputing, ACM/IEEE 2000 Conference 04-10 Nov. 2000 Page(s): 30 - 30
This paper as I mentioned earlier describes a system called WireGL. This paper outlines in detail the foundation for developing a distributed rendering system through examining WireGL. This was a good starting for me. It gave me a good understanding of how distributed systems work.

The paper examines the network layer (the essence of any distributed system of course) of the WireGL distributed rendering system. It basically shows how it encapsulates OpenGL calls into a network packet. The packets can all sorts of information from vertex information and lighting to matrix transformations and OpenGL attribute settings. This paper was the initial seed that led me to consider WireGL as my testbed.

WireGL is a very network heavy system. The amount of geometry and render information sent between across the network can be both faulty and dangerous. We can't trust that a packet will get across the network and by increasing the amount of network traffic you also increase the margin of error for faults.

**Software Design**

Since I'm using Equalizer as a testbed, my system is divided into two layers. That is the application layer and server layer.

For the application layer, I'll be writing a rendering application that will be wrapped up by Equalizer API library. Equalizer will be able to break down the render frame into smaller piece. Upon initialization of the Equalizer library, the system will connect to the eqServer to start the rendering.

For the server layer, I will be implementing two equalizer classes. The classes will consist of the dynamic load-balancing tile algorithm and the DPBP load-balancing algorithm, respectively.

There is one caveat to the server though. The server has no knowledge of the scene geometry, so I had to extend the core and add new functionality to implement the DPBP algorithm. This actually took quite a bit of time. I was waiting mostly on the lead developer to tell me what’s going on in the core since there are no examples or documentation on it.

Next, I had to setup the render farm itself. This took even more time since the documentation was outdated and direct links on the website that describes the process. I had to go back a bunch of revisions to find the article that described how to do so. I will try to walk you through the whole process in the next section.

Also, Equalizer already has a 2D tile load-balancing algorithm implemented. I've been using that to make sure the render farm itself is working.

**Developer's Manual**
There are no binary releases on the Mac OS X platform for the current versions, so everything must be from scratch. All development has been done in Mac OS X.

The following two dependencies must be compiled and built before we can use eqOSG:

- **Equalizer** – Both XCode and VS solution files are provided so you can build them
- **OpenSceneGraph** – Uses CMake to generate solution files. Simply build the project using the generated files.

Instead of using the CMake provided to build

- **eqOSG** – Reference the built Equalizer and OpenSceneGraph libraries.

**User Manual**

Before I can describe how to run the software, you must understand how to use the configuration files of Equalizer. Depending how many nodes you’re running in your cluster all of this must be described in that file including the connection information. Also, you must map one node to one GPU on the machine. You can’t run multiple nodes on the same machine if you don’t have enough GPUs.

First and foremost, you must specify all library paths to point to both Equalizer library binaries and OpenSceneGraph library binaries.

First, you need to start up all your render nodes first.

```
./eqOSG --model=<model filename> --eq-client --eq-listen <ipaddress>:<port>
```

Example: `./eqOSG --model=avatar.osg --eq-client --eq-listen 129.21.34.18:12345`

- `<model filename>` Name of the model to load
- `<ipaddress>` Render node’s address
- `<port>` Listening port

After starting up all the render nodes, you start the actual server. The server will automatically connect to all the render nodes as specified in the configuration file.

```
./eqServer <config-file>
```

Example: `./eqServer config/2-node.2D.eqc`

- `<config-file>` Configuration file (.eqc)

Lastly, start the application and connect to the server.
./eqOSG --model=<model filename>
Example: ./eqOSG --model=avatar.osg

$model filename$  Name of the model to load

Measurement Data

I regret to say that I don’t have much data regarding the analysis of the algorithm. I’m still in the midst of implementing DPBP because one of the developers just got back to me. I’m not going to lie and give you false data.

I’m hoping to finish everything before my presentation on Wednesday and possibly submit additional material to you.

Future Work

I plan on completing this project and submit the code to the Equalizer community once it is done. I’ll continue to work towards my original goal and integrate Ogre3D with Equalizer. Currently, I’m using OpenSceneGraph, which is alien to me, but I’m able to make progress due to my understanding of overall render system designs and concepts.

I plan to improve the documentation for both developers and users. Even as I described how to set up the system to compile and run above, I feel that it is too complicated.

Optimizations

A few optimization come to mind to the overall system that I have in mind.

• Culling before dividing the scene tiles. This way each render nodes do not have to re-cull the geometry.
• Build or employ a distributed filesystem that can be shared between all render nodes and also the server.
• Expose the server to the application layer, so it can grab more information.
• Extend the distributed object system to the server. Add general event dispatcher and event listeners to eq::node, instead of the command system that’s currently in place.

What I learned

I have always been taught to not reinvent the wheel in the computer science world, but learning how to use the wheel can be just as hard if not time consuming. Throughout this project, I’ve learned a lot about distributed rendering systems.
This project at a glance seemed trivial, but in hindsight the amount of work to even get to the state of where I can actually implement the algorithms is tremendous.

**Team Members**

I’m the only one working on this project and I started this project during week 4 of the school quarter.

**References**

- “Distributed Rendering for Scalable Displays”
  Humphreys, G.; Buck, I.; Eldridge, M.; Hanrahan, P.;
  Supercomputing, ACM/IEEE 2000 Conference 04-10 Nov. 2000 Page(s): 30 - 30
- “A load-balancing strategy for sort-first distributed rendering”
  Abraham, F.; Celes, W.; Cerqueira, R.; Campos, J.L.;
- “DPBP: a sort-first parallel rendering algorithm for distributed rendering environments”
  Huabing Zhu; Kai Yun Chan; Lizhe Wang; Wentong Cai;
- “Sort-First, Distributed Memory Parallel Visualization and Rendering”
  Bethel, E.W. Humphreys, G. Paul, B. Brederson, J.D.
  Lawrence Berkeley Nat. Lab., CA, USA
  Symposium on Publication Date: 21-21 Oct. 2003 Page(s): 41 - 50
- “An open source Grid based render farm for Blender 3D”
  Patoli, M.Z. Gkion, M. Al-Barakati, A. Zhang, W. Newbury, P. White, M.
  Dept. of Inf., Univ. Sussex, Brighton
  Publication Date: 15-18 March 2009 On page(s): 1 - 6

- Equalizer Documentation:
- Chromium Documentation: