Hybrid Distributed Rendering

Abstract—As data sizes grow, it becomes more appropriate to visualize and analyze them on remote resources due to the scale of the data. Unfortunately, a purely server-based processing regime leaves a considerable amount of capable hardware idle on the client side. Further, it puts visualization and analysis applications at the whim of a sometimes fickle network. Here we describe an in-progress, zero-configuration framework for enabling hybrid rendering in a manner which effectively utilizes all available computational resources.

1 INTRODUCTION

While most previous distributed visualization systems have focused on server-based processing of the visualization pipeline, powerful client devices enable a shift in the workload balance. This provides the possibility of building interactive systems which scale much more effectively without a steep investment in server-side hardware. In essence, the processing power is distributed among the large set of users with capable PCs, tablets, and smartphones, which can mostly be assumed to be available without investment by the system developer. Amortizing this hardware cost over such a large user set means we can design significantly more compute-intensive systems. The challenge, of course, is how we can effectively utilize these client devices.

Although client devices are becoming increasingly powerful, data sizes are also becoming increasingly large and complex. This poses issues both in storing data and processing it in real-time. Unfortunately, the rate at which data sets are increasing exceeds the rate at which client devices are expending their computational capability, and the gap is always widening. Therefore, a server-based component is necessary and it is unlikely that we might be able to remove that component in the future.

We are therefore working on a hybrid rendering environment, which adaptively adjusts where the rendering work is done based on current conditions. The key contributions will be a scheduler for deciding the best workload distribution between client and server, and a generic API for plugging a hybrid & distributed rendering mode into existing visualization systems.

1.1 Related Work

Hybrid rendering can be divided into three approaches; Luke and Hansen [2] give a classification of how rendering work can be distributed. Broadly, images can be cooperatively generated by server and client, generated by the client with continuous input from the server or generated independently on either server or client. Our system follows the last of these classifications, much like that provided by Engel et al. [1].

The approach allows us to keep up interactive visualization on the client independent from a server. This is relevant should the network suffer from high latency or the server be occupied or unavailable. In addition, it allows us to support visualization applications independent of the rendering algorithm.

2 HYBRID RENDERING

A hybrid rendering system splits the rendering workload between client devices and server processes. Transient conditions (such as network status or server load) can greatly influence the desired workload distribution. For this reason, we are developing an architecture which adaptively reacts to the current system conditions and schedules the best workload distribution. In this section, we describe the scheduler which decides at what location rendering should be done.

2.1 Quality Levels

Our approach utilizes a concept we refer to as a quality level. Quality levels are a totally ordered set of visual representations of a data set. The implementation of that ordering is unspecified by the hybrid rendering framework; it only requires that the renderer which is plugged in defines discrete levels which are distinguishable by an integer value. This allows significant flexibility in the definition of a renderer, which might map quality levels to carefully chosen sampling rates; a multiresolution hierarchy; both; or some alternate, unforeseen representation.

Server systems store all of the data necessary to reproduce the visual representation for any available quality level. Clients may choose to support only a subset of these levels, which might be utilized if the clients are computationally limited. At the outset of a session, client and server perform a negotiation handshake to identify, among other things, which quality levels are appropriate for each endpoint.

Fig. 1. An implementation of our scalable hybrid visualization system running in our visualization lab. The display consists of 20 high-resolution 31” monitors and a back projection screen.
2.2 Interleaved transfer

We make no assumptions as to the availability of data on the client side. Therefore, the initial state is one in which all data is rendered on the server side, and the client simply blits a received image into the framebuffer.

To eventually enable switching where quality levels are rendered, we use time division multiplexing to fully utilize our channel by streaming either imagery or data for a quality level at all times. The visual representation is always given priority, and thus all data transfer is postponed when rendered images are available. While rendering intermediate quality levels as well as while waiting for user interaction, the network stream would otherwise be idle; these are the periods which we use for the background task of sending data.

2.3 Negotiating Quality Levels

Once all data has been received for a quality level, hybrid rendering can be performed. The most important decisions here are which quality levels need to be rendered, and on which side to render. Our current scheduling algorithm is outlined here.

We maintain a list of quality levels and their associated rendering times on both the client and server. Server-based times include the time required for image transmission. Timings are subject to fluctuation, especially for server levels where the result is affected by load and network status. In our system, a timing represents the expected value of a normal distribution (Figure 2). This allows us to schedule in a monte-carlo fashion by comparing levels in terms of probabilities. The scheduler endeavors to provide the highest quality possible whilst respecting a client-provided update frequency. There are two scenarios to be concerned with: updating after a client event, and progressive rendering to increase quality during idle periods.

![Timing distributions for two quality levels.](image)

Fig. 2. Timing distributions for two quality levels.

After a user interaction, clients iterate through the list of quality levels which are available and choose the highest level which can be rendered with high probability given the time constraints, whether it be from the client or the server. The probability used to decide can be set up as a parameter and must be greater than 50%.

Once the rendering of this first quality level is available on the client side, we would like to render increased quality levels to visibly improve the results. Here we have client and server render different quality levels in parallel. When deciding which one of two quality levels shall be rendered, we use the probability that one level completes before the other. The scheduler attempts to create a schedule which 1) displays the highest quality as soon as possible, and 2) displays as many quality levels as possible. The latter is difficult in the face of uncertainty that is inherent in any distributed system. Even if the schedule predicts the client will finish quality level three before the server finishes quality level four, the server might finish first. Should one endpoint outpace the other, the former’s results are used. We will however still consider timings achieved for quality levels which complete late. Rendering may be aborted due to user interaction though, in which case we expect the render system to return an estimation of the render time.

With this algorithm, it is conceivable that our predictions for rendering times on either the client or server would never be complete. However, the monte-carlo approach allows us to guarantee every level will eventually get scheduled. This is important when considering changing conditions, especially load and network status, which heavily affect the predictions for server quality levels. Our approach is able to adapt to these changes. Quality levels with low schedule probability, for example caused by server load, will be scheduled at an increasing rate until a new distribution reflecting the improved server state has been reached.

3 Hybrid Rendering API

We are also working on a pluggable framework which allows developers to enable hybrid rendering modes in their applications. The API abstracts away the connection information and event processing. Instead, applications register callback functions which respond appropriately to events. On the client side, the API is extended to allow user interaction events to be inserted into the system.

There are a modicum of function callbacks which must be registered. For example, the application is notified whenever a new image is ready for display. Also, the framework requires the renderer to provide a list of available data sets. In our prototype implementation, this is simply a list of filenames obtained by scanning a directory for files of a particular type.

For those who simply desire a remote rendering solution, we have created a simplified client API and programs to utilize it. These clients are less capable, as they cannot perform local rendering (negotiated automatically during the handshake phase), but they are considerably simpler to get set up. They also provide a quick way to test server integration. Currently, we have created DirectX and iOS versions of these clients, and we hope to expand to Qt/OpenGL and Android versions in the future.

4 Preliminary Results

Our initial results plug-in a volume renderer for very large data. The renderer suits this environment, as it utilizes a multiresolution representation of the data which maps well to the framework’s quality levels. To provide more flexibility, we define additional quality levels which render the lowest resolution data with decreased screen resolution and/or sampling rate. This ensures we can always render something interactively, thereby providing immediate feedback. While we feel that a visualization tool requires this level of interactivity, we note that we have not encoded any real-time requirements into the framework.

Our current renderers are interruptible by batching the render requests. That is, we submit the rendering in blocks, and test for events between each block. While this is not a requirement of our framework, we have found this significantly increases the interactivity of the system; it is often the case that a request for a new rendering comes in while an existing rendering is in progress. Interruptible renderers allow us to cancel partially-completed renders of a given quality level and refocus on the more relevant request.

In the future we plan to look at isosurfacing and particle tracing techniques. Both visualization techniques will challenge our notion of quality levels. For the former, very fast renderers are available, so it is not clear that an extensive set of quality levels will be necessary; for many data sets, we expect only one quality level to be relevant. The latter is computationally complex, and it is not immediately clear that informative reductions could be generated which would map well to our notion of quality levels.

References
