Large Scale Visualization Techniques for Huge Datasets (January 2014)

Asad Ullah Naweed

Abstract—This technical document highlights the problem of visualizing large datasets on multiple large displays. The work was undertaken as part of an RA project and an independent study from August 2013 to December 2013. A complete server-client system was built in this period, with support for rendering many different kinds of datasets, with a primary focus on 3D models in various formats reconstructed from images and video sequences using computer vision techniques.

Index Terms—rendering, visualization, reconstruction

I. INTRODUCTION

This technical report outlines and details the approaches taken to display 3D reconstructed models on a large multi-panel display for an immersive viewing experience for multiple viewers. The motivation behind the development of such a system was to create a Telepresence Wall, which would act somewhat like a “window” between two rooms that are 3D-scanned in real time. There is one large display wall in each room, and we assume that a live 3D scan is available for each room, and the system is aware of how many viewers are in a single room, and their respective head positions are tracked in real time as well. The actual data used was reconstructed offline, but the system developed has support for receiving and rendering a live 3D scan stream in real time.

Another motivation lies in rendering of large architectural structures for the purposes of virtual tourism or educational programs. Historical buildings, such as the Colosseum in Rome, are highly popular tourist sites and providing a realistic virtual tour of such structures could be a potential use for such a large scale system.

The main problem addressed by the system is that of rendering from the viewpoint of multiple people in the room. Such a rendering is non-trivial, and has to be implemented such that the viewing experience is as acceptably close to reality for each individual as possible. This is exacerbated by the fact that the display is a standard 2D display, and the users are not wearing any stereoscopic eyeglasses. This makes it exceptionally hard for the users to be convinced of the illusion of a “window into another room”.

Much work has been done in the field of multiview rendering of large datasets. Even with modern 3D TVs and auto-stereoscopic displays, there are numerous issues that need to be targeted while rendering volume from multiple points of view simultaneously. Work has been done in content generation [1], antialiasing and image refinement [2] and single-pass rendering [3] for 3D displays. However, the problem of rendering data on standard 2D displays such that multiple viewers can be provided an experience which is satisfactory enough, has been largely unaddressed. This document highlights a system developed to carry out a series of experiments to potentially discover such a rendering scheme, and to discuss the pros and cons of each scheme attempted.

The remainder of the report is organized as follows. Section II describes the system and setup in detail, including the hardware, platforms and APIs used, along with important classes and interfaces in the system. Section III describes the various approaches used and experiments performed, and their outcomes. Finally, Section IV contains conclusions and suggestions for future work.

II. SYSTEM OVERVIEW

A. Hardware

The display system comprised of four large LCD screens mounted vertically and adjacent to each other on a single wall. Two computers controlled these four panels, with each computer controlling two panels. Both the machines were connected to the same network subnet to minimize network latency when data was communicated between processes. Panels were connected to the computers’ graphics cards using DVI-D cables. Three of the panels were identical, with a resolution of 768x1360 pixels, and the fourth one, which was on the extreme left, had a resolution of 1080x1920 pixels. Each machine had 64 GB of RAM and an NVIDIA Titan GPU.

B. Platforms and APIs

The code was written entirely in C++ for 64-bit Windows 7. The environment used was Microsoft Visual Studio 2010, and the Win32 API was used for all window management, thread management and network communication. OpenGL 3.3 was used for rendering, and only in the case of the server application, wxWidgets 3.0 was used for GUI programming.

Asad Ullah Naweed is a second-year graduate student in the Computer Science department at the University of North Carolina at Chapel Hill. (email: asad@cs.unc.edu).
C. Software Design

The system was designed using the client-server paradigm. Each client process controls and renders to one panel, and these processes may reside on arbitrary systems. Each panel had its own XML file, which contained data about the panel, including device name, pixel resolution and real world measurements and location among other thing. Each process could uniquely determine a display to render to and the frustum parameters from a given panel descriptor file. Each process also had access to all the data being rendered and all rendering was done client side. A server-side rendering approach was attempted, in which the entire scene was rendered to a texture, and then pieces of the texture sent to the clients for simple displaying, but this was later abandoned due to the large amount of time needed to copy large textures from the GPU, compress them using JPEG encoding, and then send them over the network. Rather, the server process now simply provides each client with a set of rendering parameters, and each client process manages its own screen, given that it has access to the data locally, which is easily achieved.

The data to be rendered was in two different formats, and parsers were written for both formats. The first format, CSurface, was the format used to store models reconstructed from multiple Microsoft Kinect cameras scanning a dynamic 3D room with moving objects [4]. The second format was VRML (Virtual Reality Modeling Language), which consists of a tree-like structure of files, with parts of the model split up among different files. This format was used to store the 3D model of the Colosseum in Rome reconstructed from hundreds of pictures of the arena [5]. This format has been, in recent years, been superseded by the X3D format for storing 3D models for use in Virtual Reality applications.

D. Class Overview

The backbone of the client-side rendering resides in the Win32GLWindow class, which abstracts away all the complexities of setting up device contexts and canvases in Win32. The constructor simply needs the device name of the monitor to be used, and it automatically sets up a fullscreen OpenGL canvas on that monitor. This class also contains code to render all supported 3D model formats.

The VRMLLoader and VRMLModel classes are used to load the entire VRML tree in a recursive fashion, and then store it in a tree-like structure in memory. This structure mimics the logical structure it has on disk, but is not very efficient for rendering purposes because a separate draw call is needed for each node in the tree to be rendered, which means that data has to travel multiple times across the GPU bus for the whole model to be rendered. The model could potentially be consolidated into one large model. However, since each submodel has its own texture image(s), and texture coordinates with respect to that image, constructing a texture atlas and baking new UV pairs for each vertex at runtime is prohibitive for such a large model. This could, however, potentially be done offline with the help of a tool such as Unity or any other game engine, provided the appropriate plugins for VRML support. The CSurface class is used to load and render CSurface objects that are stored either in binary or ASCII format, and function similar to the VRML classes.

E. Experiment Setup

For the purposes of this experiment, the people in the room were not tracked live. Rather, the position of the viewers in the room was set manually via a GUI. The GUI designed is shown in Figure 2. The experiments were always conducted for a total of 4 viewers in the room. The bounds of movement were restricted to a rectangle in front of the screen, so positioning
the viewers around on the GUI was intuitive. This provided for better control and helped to build a better unit test for the rendering system. The setup was first tested for correctness using single-view rendering parameters, and then the rest of the experiments were done. A few of these viewpoints are shown in Figures 4, 5 and 6.

III. IMPLEMENTATION

A. Naïve Mean Location Viewpoint

The first approach taken was to render the scene from the viewpoint of the average position of the four viewers in the room. The motivation behind such an approach was to mitigate the effect of the movement of one viewer on the viewing experience of the others, assuming they remained stationary. This is not a bad assumption, considering the use case of remote meetings via the screen. In such meetings, participants are not actively moving around, but rather tend to stay in one place.

The approach proved to be effective in certain scenarios. If three of the viewers were seating close to each other, or even in a single line, the movement of the fourth did not cause a largely adverse effect on the rendered image. However, this was highly dependent on the position of the data being rendered in the 3D world, as well as the position of the three static viewers. It was found that movements along the z axis caused the least change in the image on the screen, and changes in x and y caused a significant change, especially if the fourth person was moving very quickly, in the worst case. These sudden changes in the image tended to greatly affect the viewing experience of the three static viewers.

B. Smooth linear interpolating average

This method maintained a track of the current location of the viewpoint, and smoothly moved towards the average over time. This smooth linear interpolation is an extension of the first attempt, and it attempted to smooth out the sudden discrepancies caused by sudden movements of the fourth viewer. This resulted in the image transitioning more smoothly than the first scheme, and the smoothness caused the viewing experience of the three static viewers to be significantly improved over the first scheme. However, while it was an improvement over the first method, it still suffered from the same flaws of being very data and viewer dependent, meaning that it would behave very well for some data and not for others. Also, the effects of movement were not of the same level for all axes.

C. Image feature movement restriction

In this scheme, we attempted to restrict the speed of the camera viewpoint such that no feature in the rendered image moves by more than a certain amount of pixels per second. In
order to do this, the scene was rendered to a frame buffer with two render targets, specified as textures in GPU memory. The first target was the standard color image. The next target was the world position of each pixel rendered to a texture. This texture was then extracted, and the pixel which would be displaced on the image the most by moving the camera to the center was located. The camera speed was then adjusted to throttle the pixel speed to the specified threshold.

This method has obvious advantages. We used a physically accurate method to correctly limit the movement of features on the screen rather than employ heuristics like smooth interpolation. This is because the position of the data is being accounted for, and the method always throttles the camera speed the same in all the axes and the discrepancy is not noticed, as long as the threshold is small.

The biggest disadvantage of this method is that it is very slow. Because of the need to extract the world position texture from the GPU on every rendering pass, the method causes dynamic datasets to have very choppy animation sequences, though static datasets look fine. This problem is a major one in the existing implementation mainly because

IV. CONCLUSIONS AND FUTURE WORK

The many experiments and schemes that were implemented show that any solution to this problem might just not be good enough for a good user experience. The utter lack of work done on this is also a sign. Even with our best estimates, users were still not completely convinced of the virtual reality. The problem also stems from the fact that many visual cues humans need to perceive reality are missing from this 2D system, and the “anchoring” of the surrounding visual cues in the physical world further diminish the quality of perception.

As far as the code base goes, the system in its current state is not portable. The many things for which the Win32 API is being used could very well be written completely using the wxWidgets API, therefore making the system cross-platform. Offline atlas compositing of the Colosseum model is also something that can greatly benefit performance. All these are things that are doable, but could not be completed due to time constraints.

Figure 3 - The blue taped rectangle show the bounds within which the viewers can move. This blue rectangle corresponds directly to the rectangle shown in the Server-end GUI in Figure 2
REFERENCES


Figure 4 - The Roman Colosseum. The model comprises of 484904 vertices, with each vertex consuming 6 floats of space, and 747914 triangles defined on those vertices. Additionally, the model was fully textured, using up 356 MB in JPEG compressed form on disk. This figure shows a view of the stadium from the right side of the room.
Figure 5 - The Colosseum from the Left Side

Figure 6 - A back view of the colosseum
Figure 7 - A virtual person sitting close to the screen near the Colosseum.

Figure 8 - The same person moved slightly closer to the Colosseum.