High Resolution Video Playback in Immersive Virtual Environments

Han Suk Kim[*]
Computer Science and Engineering
University of California San Diego

Jürgen P. Schulze[†]
California Institute for Telecommunications and
Information Technology
University of California San Diego

Abstract

High-resolution video playback ($>$ 1 megapixel) has become a commodity in homes (Blu-Ray players, internet streaming) and movie theaters (digital HD technology). Immersive virtual reality systems can display tens of millions of pixels today, for instance CAVE-like environments driven by 4k projectors. However, when video is displayed in virtual environments (VEs), where the video screen is part of the virtual world, the resolution of the video is fairly low, and so is its frame rate, typically much lower than standard TV. Allowing high-resolution video playback in VEs can add more realism to the virtual world (e.g., a virtual movie theater), and it can enable a new class of applications which were not possible before (e.g., virtual video surveillance centers).

In this paper, we propose an algorithm based on mipmapped video frames, where each image of the video stream is stored at multiple levels of resolutions, to interactively play high-resolution video in VEs. In addition, we propose an approach to maintain a constant video playback rate, as well as optimizations for the algorithm, such as a memory management mechanism and predictive prefetching of data. Finally, we analyze the playback of three different types of high-resolution video clips in an immersive VE.

Index Terms: I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual Reality H.5.1 [Information Interfaces and Presentation (e.g., HCI)]: Multimedia Information Systems—Video

1 Introduction

Today, most new feature films, TV shows and documentaries, and a rapidly growing number of home and professional videos are shot, edited, and released at high resolution. Typical resolutions are HD, which is usually defined as resolutions between 1280x720 and 1920x1080 pixels, and the new digital movie standard 4K, which features 4096 pixel wide images with 2160-2400 lines. Real-time graphics applications, such as computer games and virtual reality applications, often embed video in their virtual worlds, for instance to visualize a place like Times Square in New York City with its video screens, a virtual movie theater with a movie playing on the big screen in a virtual environment (VE), or a virtual stadium with its video screen. Integrating video into 3D applications can increase the level of realism, and it can bring important information into the virtual world.

However, due to the large volume of data in high-resolution video, which easily exceeds hundreds of gigabytes, it is not an easy task to playback video in real-time. Playing back high-resolution video is already CPU and GPU intensive for today's computers, even when it is displayed in a 2D window on the screen, namely, all traditional video playback tools like Windows Media Player, QuickTime, or VLC. Constant I/O operations are required to load data in a timely manner, and video rendering systems need to manage the limited memory resources. While in the 2D domain the GPUs provide techniques to help speed up playback such as overlaying, playing back video in a VE is even harder. The video is not displayed on a rectangle aligned with the screen, but as its projection on the screens, which is a general quadrangle in an arbitrary orientation. This plays a role, for instance, if a 3D model of a virtual movie theater were displayed with a movie playing on the screen of the screening room. In this publication, we do not cover real-time video sources like video conferencing or video surveillance systems, but some of our algorithms can be applied to those kinds of applications as well.

In this paper, we present an efficient algorithm and its implementation to display high-resolution video in VEs. Our concept is based on using mipmapping [30] for the video frames, on top of which we add various optimizations to allow for constant frame rates under varying viewing conditions and rendering rates determined by the rest of the 3D virtual world, assuming that the video is not the only thing being displayed by the graphics hardware, but that it is embedded in a 3D model the user can move around in (e.g., a virtual movie theater) or look at from all directions (e.g., the visualization of a cell phone with video playback on its display). In our approach, we pre-process the image frames of the video to create frames of successively lower resolution (mipmapping). We also tile these images so that they can be loaded from disk more efficiently. By rendering the lowest resolution tiles which match or slightly exceed the physical display resolution, we can optimize memory usage during playback. Predictive prefetching of data further enhances the performance of our system. All of these approaches and algorithms are software-based and only require a graphics card with a reasonable amount of texture memory, which allows existing VE software applications to easily embed video through graphics libraries, such as OpenSceneGraph [3]. Our algorithm supports the simultaneous rendering of multiple video streams, located in different places of the virtual world.

Digital projectors and DVD players are designed to play video at a constant frame rate. The frame rate for digital cinema is defined as a constant 24 frames per second, which we refer to as the *video frame rate*. However, in interactive computer graphic applications, each frame takes a different time to render, depending on many different factors, such as the amount of data to be loaded, the size of the rendered screen, and cache effect in various places of a computer system. If rendering an *image frame* is faster enough than 1/(video frame rate), then the image frame should be played more than once. On the other hand, if the rendering speed is below video frame rate, then a few images after the current image frame should be skipped. Therefore, the speed of rendering one image frame, the *image frame rate*, should be constantly compared with the *video frame rate* to play a sequence of images as smoothly as in normal video players.

This paper is organized as follows: Section 2 discusses similar approaches proposed for other problems and Section 3 describes the main goals of this work and provides an overview of the video playback system. Section 4 presents implementation of the system and the experiment of our implementation and its results are shown in Section 5. Finally, Section 6 discusses future work and concludes

[*]e-mail: hskim@cs.ucsd.edu
[†]e-mail: jschulze@ucsd.edu

this paper.

2 Related Work

Memory has been the most scarce resource in computer systems and also in graphics processors. In order to overcome the limited resources, graphics communities have developed many algorithms and technologies. The most widely used approaches are level-of-detail and tiling (or bricking, when used with volume data).

Level-of-detail (LOD), often called mipmaps, was first introduced by Williams et al. [30]. Precomputed texture data at multiple resolutions, downsampled from the original image, greatly eased the tight texture memory resource budget and the antialiasing problem. The concept of *Clipmaps* [23], which are virtual mipmaps, extended the mipmap concept to load arbitrarily large images into graphics memory by splitting the images up into smaller pieces. The essential observation that led to Clipmapping was that normally only a small portion of the entire mipmap pyramid is used to render a texture. Thus, Clipmapping loads only the portion that is needed to render the current image frame. Clipmapping is very close to our work, except for the fact that it only considers a set of 2D textures of one image frame, not a sequence of frames.

Tiling is a technique that divides an image into smaller, rectangular pieces, called tiles (brick). This concept has been employed in the interaction between main memory and texture memory in graphic processors, and is very similar to paging algorithms of memory management units. The main advantage of the tiling mechanism is that it can help overcome the limit of texture memory size. When rendering images larger than texture memory, which are to be displayed at a size larger than the screen, the renderer will render only those tiles which actually contribute to the visible image.

Tiling and multi-resolution level-of-detail techniques are often combined [13, 4, 10, 27, 5, 19]. There are two different approaches for the level-of-detail selection process: area-and-distance and point-of-interest. LaMar et al. [13] constructed a spatial data structure, a quadtree, to store multi-resolution data sets. The generation of a proper level-of-detail is determined by two factors: 1) the distance from view point p to a tile and 2) the area a tile covers in projected space. Given point p, a tile is selected if the distance from the center of the tile to p is greater than the length of the diagonal of the tile. As the projection transformation matrix transforms objects closer to p to appear larger and those further from p smaller, data points closest to the view point have the finest resolution. Weiler et al. [27] approached the same problem, the selection process of tiles, with a different solution from [13]. Instead of selecting the finest resolution for closest tiles, users can define a focus point. The distance from the focus point to a tile determines the level-of-detail.

Blockbuster [1] is a movie player for high resolution videos, which can run on tiled display walls under DMX (Distributed Xinerama). It plays movies in Lawrence Livermore National Labs' SM format, which supports tiled images, multiple levels of detail and several types of intra-frame image compression. The major difference to our approach is that Blockbuster cannot render to images in a VR environment, where the video image is not rectangular and parallel to the screen.

OpenGL Volumizer [4] is a volume rendering library developed by SGI, but it could potentially be used to render images as well. It supports multiple time steps but it was mainly designed for a single time step and most optimizations, such as caching algorithms, a roaming window and toroidal mapping, focus on volume data with a single time step. These optimizations, however, are irrelevant for video playback, because the image content changes constantly. In addition, OpenGL Volumizer does not have a mechanism to maintain a constant video frame rate, which is critical for video playback.

Preloading algorithms in rendering large scale data sets have been studied with various applications. iWalk [7] and iRun [25] proposed predictive prefetching in walkthrough applications. They enhanced the predictive prefetching by using from-point visibility instead of from-region visibility in [9]. From-point visibility exploits the current camera's position, linear speed, and angular speed, and from the information gathered, it predicts the future camera location and direction.

Octreemizer [18] and its extension [17] provide a more sophisticated paging and prefetching algorithm in volume rendering. The key idea for paging here is that it limits the number of tiles that can be loaded at a given point. Although it may not render with the best possible resolution, it can always guarantee that the time spent on data loading is bounded and that therefore the renderer provides interactivity even when the observer moves fast. Our algorithm is very similar, but Plate et al. load a fixed number of tiles. In contrast, we adjust the limit of tiles to automatically find the best value for the number of tiles to load. The prefetching algorithm in Octreemizer uses a similar approach as in iWalk and iRun, predicting the observer's motion by linear extrapolation from the previous image frame and the current image frame. Through the prediction of the future location, the tiles that may be needed in the near future are loaded in advance.

Another topic related to the subject of this article is video conferencing. The goal of video conferencing is to build an interactive environment enabling face-to-face communication. Typical systems consist of a camera and a computer display at both ends of the data link and display the camera image at the other end [8, 24, 12]. More sophisticated video conferencing systems can display multiple video streams on the same monitor, or on multiple displays. The SAGE system [11, 20] can be used to display multiple video streams on a tiled display wall. However, in all of the above approaches, the video is always played back in a rectangular window which is parallel to the physical displays. In virtual environments, however, the video image needs to be projected onto arbitrarily oriented displays, and projected image is not rectangular anymore, but a general quadrangle.

3 System Overview

In this section, we describe our software system, and we propose a new rendering algorithm for super high resolution video streams.

3.1 Design Goals

Our approach to render super high resolution video in VR environments has three goals: to render large videos (both in image size and duration), optimize frame rate and image quality, and allow for the rendering of multiple concurrently playing videos.

Large Scale Video Texture

In order to create a video texture in a virtual environment, we need to change the texture at least 24 times per second. We want the movie to be of very high resolution, typically exceeding the size of main memory. The resolution of a typical 4k video clip is 4096 x 2160 pixels, which means that a 10 minute video clip at 24 frames per second is 300 Gigabytes of uncompressed data. This is a lot of data to stream into memory from disk and on to the graphics card, and reaches the performance limit of even the highest end machines. Parallelization of loading and rendering has been an approach to achieve the goal of rendering such high resolution videos, but in existing implementations the video is split up into equally sized rectangular pieces which are then processed independently by the nodes of the parallel visualization system. In our case, we need the video to be displayed in a 3D virtual environment, where the location and shape of the video image changes constantly.

Performance vs. Image Quality

In video playback, especially with sound, it is very important to keep a constant video frame rate. In our tests, it turned out that disk I/O is the most time-consuming part of video playback and our goal is to correctly

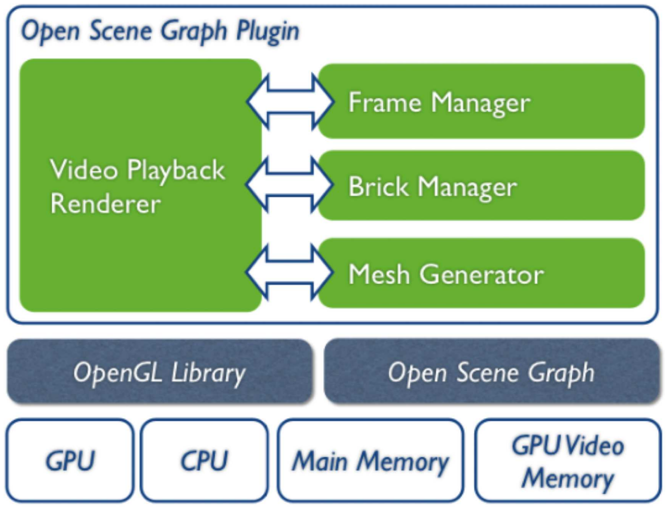

Figure 1: System Overview of Video Playback Rendering.

predict disk I/O requests and to evenly distribute the disk I/O operations over the available rendering time. In the case that a burst of I/O operations has to be processed (e.g., when the viewer moves quickly), a constant video frame rate should be guaranteed even if it leads to a drop in image quality, so that the video stays in synchronization with the audio. This goal can be assessed in two different ways: qualitatively, i.e., the viewer should not see any slowdown during playback, and quantitatively, i.e., the video frame rate should be both as stable and high as possible.

Multiple Instances The third goal is the ability to display multiple videos concurrently, and also display 3D geometry along with the videos. For instance, we want to be able to display a virtual surveillance center with a control room which displays a multitude of videos. This requires that our algorithm uses minimal memory resources for each video stream, so it can co-exist with the other video streams, as well as the rendering of the 3D geometry.

3.2 Software Design

One core approach to achieve the goals we described in Section 3.1 is mipmapping. Instead of rendering the entire high resolution video, we calcualte the best possible resolution for the screen. Because mipmapping reduces data by factors of two, this method reduces the footprint of memory significantly. In addition, although changing the level-of-detail may reduce the image quality, controlling the image quality can greatly lower the amount of disk I/O operations.

Figure 1 shows the main components of our system. The *Video Playback Renderer* cooperates with three other components: *Frame Manager*, *Mipmapped Tile Manager*, and *LOD Mesh Generator*. *Frame Manager* controls which image frame has to be rendered at a given time to synchronize with the video frame rate. *Mipmapped Tile Manager* manages a large number of tiles. It first loads meta information for all tiles and whenever the renderer requests a tile for meta data or texel data, it returns all the necessary data to the renderer. Due to the large size of the video, it is impossible to load all data into main memory at once. Thus, *Mipmapped Tile Manager* swaps requested tiles into main memory and texture memory and removes unnecessary tiles. The decision about cache eviction is also made here. *Mesh Generator* computes the best possible LOD for each region of the playback screen so that the smallest possible amount of data is copied into the texture, which utilizes memory resources and bandwidth more efficiently. The renderer and its accompanying three components are all integrated into an OpenSceneGraph [3] plugin.

Algorithm 1 Rendering Algorithm

```
1: Mesh_Generation
2: for all tile to be rendered do
3:     Data_Load(tile)
4: end for
5: Prefetching
6: for all tile to be rendered do
7:     Render(tile)
8: end for
```

3.3 Rendering Algorithm

The rendering algorithm for a video stream consists of four major steps as shown in Algorithm 1: 1) mesh generation, 2) data loading, 3) prefetching, and 4) rendering tiles. This routine is called for every image frame and it gets as input the image frame to render from the frame manager. The mesh generation algorithm is presented in Section 4.2 and data loading and prefetching are discussed in Section 4.3. Once Algorithm 1 completes mesh generation and data loading, all the necessary data such as texture coordinates and texel data, is ready for rendering each tile. Then the final step is to iterate over the tiles that have to be rendered at the current image frame to draw them with their corresponding texture data.

4 Implementation

In this section, we discuss several implementation issues to achieve high performance rendering even with large data sets, and our solutions. The common goal of the solutions discussed in this Section is to maximize I/O performance and, thus, to stably stream video data. One important issue is the trade-off between resolution and image frame rate. In many cases, sustaining a stable frame rate requires the system to sacrifice some of the resolution of the video stream.

4.1 Mipmap Generation and Tiling

Mipmaps have been widely used in computer graphics since the concept was first introduced in Williams et al. [30]. The main idea is to downsample the original data or texture recursively by factors of two. We utilize mipmaps for the sake of saving resources: when large images are rendered but from a far distance, several texels (texture elements, similar to pixels) may be mapped to one pixel on screen and by using an appropriate mipmap texture the renderer can avoid wasting resources.

In our approach, we need to pre-process the video frames in an off-line step. First, the image files are extracted from the video clip. Each image file corresponds to one frame in the video stream. In typical cinematic movies, 24 images will be extracted for one second of the video. The downsampling process runs on each image file to produce multiple levels of resolution. Since our downsampling process decreases the image size recursively by a factor of two, until the downsampled image is smaller or equal to the tile size (we use 128^2 pixels), the number of recursions depends on the original image size.

For each level of the mipmap images, the preprocessing step divides the image into multiple smaller squares, called a *tile* [26, 6, 28, 14, 29]. A tile is later on used as the atomic unit to read and write between the different levels of the memory hierarchy. Figure 2 describes the layout of the tiles, which are stored in separate TIFF files. An image is divided into a 2D grid and the origin of the grid is shown at the bottom-left. Tiles at the rightmost column and at the topmost row are padded with zeros so that all tiles have a uniform size. Using a uniform size simplifies the rendering process, so there is no need to distinguish boundary tiles. Each tile is stored as a single multi-page block in the TIFF file so that it can be viewed

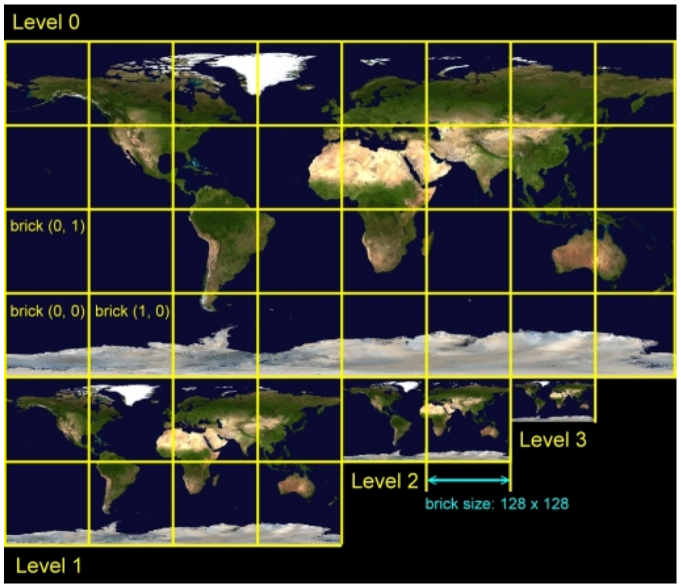

Figure 2: Layout of tiles at multiple mipmap levels. The image is from NASA's Blue Marble data set [21].

Algorithm 2 Mesh Generation

```
 1: PriorityQueue aQueue $\leftarrow$ Tile(0, 0, 0)
 2: List output
 3:
 4: **while** aQueue.empty() or output.length() $>$ tileLimit **do**
 5:     Tile parent $\leftarrow$ aQueue.pop()
 6:     **if** parent is shown too small on screen to be subdivided **then**
 7:         output.insert(parent)
 8:     **else**
 9:         **for all** aChild $\leftarrow$ parent.child() **do**
10:             **if** aChild is inside screen **then**
11:                 aQueue.enqueue(aChild)
12:             **end if**
13:         **end for**
14:     **end if**
15: **end while**
```

by normal image viewers (useful for debugging, etc). However, the format itself is not restricted only to the TIFF specification and it can be easily converted to any other image format. Although the TIFF multi-page block specification is flexible enough to put the data block in any location in the file and locate each page using a pointer, we store the tiles sequentially. The tile at $(0,0)$ is followed by $(0,1)$ and so on, to optimize the disk read buffer in secondary storage or the networked disk buffer.

4.2 Mesh Generation

The first step of rendering is to subdivide the playback screen into a set of tiles, which we call the mesh. The mesh is comprised of multiple tiles of different mipmap levels. The goal of subdividing the screen is to allocate the best possible mipmap level to each region with a limited number of tiles.

Algorithm 2 describes how to generate a mesh. The main idea of Algorithm 2 is to render areas closer to the viewer at higher resolution and those farther away at lower resolution. Rendering at lower resolution does not hurt the overall image quality because, after perspective projection in the VE, the tiles farther from the viewer are rendered smaller and the downsampled mipmap texture is still detailed enough to render this tile correctly without a noticeable change of the image quality. Figure 3 shows an example of the output of Algorithm 2. Depending on the location and rotation of the plane, the playback plane has several levels of detail.

Algorithm 2 is based on quadtree traversal. Starting from the root node, which is the maximum mipmap level of the image, the algorithm checks whether or not the tile visited can be subdivided further. The area, $area(b)$, of tile b after transformations, i.e., modelview, perspective projection and viewport transformation, is used in the decision rule for the subdivision. Let $tileSize$ denote the size of a tile. Then, if one tile of a certain mipmap level occupies about $tileSize \times tileSize$ pixels on viewport screen, the subdivision of this tile cannot further improve the image quality of the region. In VEs, the decision rule can be relaxed by adding a constant value α as follows:

$$area(b) > \alpha \times tileSize \times tileSize \tag{1}$$

where α can be any float value larger than 1. Algorithm 2 subdivides one tile if Predicate 1 is true and stops if false. The constant α controls how detailed the image is rendered. If α is one, the texel to pixel ratio of the rendered tiles is near one. On the other hand, large α makes the mesh algorithm stop the subdivision even if one texel of each tile maps to more than one pixel, which creates an image of lower resolution. α is introduced to control the system between high frame rate and the best image quality. In case of not having enough frame rate on low end machines, the system is designed to keep up with the desired frame rate, sacrificing image quality.

Another variable, $tileLimit$, controls the number of tiles to be rendered on a physical display screen. Tiles in the output list grow exponentially along the traversal of the quadtree. However, $tileLimit$ guarantees that the rendering system does not have excessively many tiles on the rendering list. The ideal number for $tileLimit$ is different from hardware configurations and a realistic number often used is around 40 128^2 tiles. That is, 40 tiles on one display screen corresponds to $40 \times 128 \times 128$ texels, which is about 640K texels.

With $tileLimit$, not all tiles can have the most desired mipmap level. Some tiles still can be subdivided into four smaller tiles to have higher resolution. Algorithm 2, therefore, has to rank all tiles so that it can choose one tile among multiple possible choices of tiles given the bounded $tileLimit$ value. In order to give priorities to each tile, a cost function is employed as follows:

$$cost(b) = \frac{area(b)}{distance(e,b)} \tag{2}$$

$cost(b)$ denotes the cost for tile b and $distance(e,b)$ measures the distance between the viewer's location and the center of tile b. The viewer's location is given by the tracking position in the VE. Intuitively, tiles occupying a large area on screen have higher priorities so that no large tiles of low resolution are left on the list. The denominator, $distance(e,b)$, gives higher priority to tiles closer to the viewer. Namely, this term provides a point-of-interest mechanism; as the viewer walks toward to a specific part of the playback screen, the region around the viewer is set to higher resolution.

Figure 3 shows an example of a mesh generated by Algorithm 2. The image plane is tilted in such a way that the bottom right corner of the plane is set closer to the viewer and the top left corner of the plane is farthest from the viewer. As shown in Figure 3, tiles around the bottom right corner have a smaller size, which results in a higher resolution.

Due to the viewer's constant movement, $distance(e,b)$ returns updated values at every frame and that makes it impossible to reuse the mesh generated during the rendering of the previous frame. Thus, this process has to be called when rendering every frame.

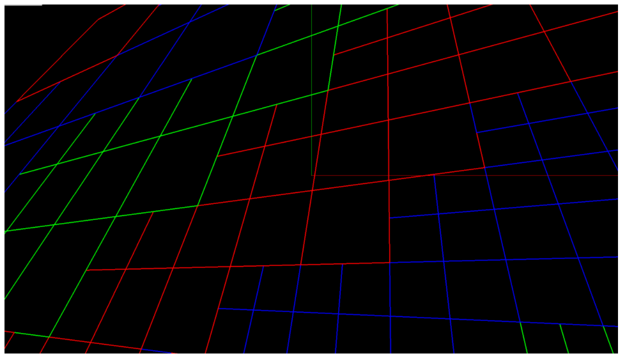

Figure 3: Dynamically generated multi-resolution tiled 2D volume.

This slows the rendering process because tiles that need to be visited may grow exponentially in a quadtree. The view frustum culling test, however, reduces the cost of quadtree traversal by significantly pruning unnecessary nodes. One simple rule is that if a parent node is culled from the view frustum, all the child nodes are also outside of the view frustum. Therefore, when the playback screen is zoomed in, i.e., when the traversal has to go all the way down to the leaf nodes, a large portion of the octree is pruned at the earlier stage of the traversal. As the video moves away from the viewer, the traversal does not need to go down as much anymore as the video frame can be rendered with a smaller number of tiles.

4.3 Data Loading and Prefetching

The mesh generation algorithm described in Section 4.2 produces a set of tiles covering the region of the playback canvas shown on one display screen. The next step is to load the texture data from secondary storage. For small videos, it may be possible to preload all the texture data either in memory or even in texture memory in the GPU. However, high resolution video streams require hundreds of gigabytes of data and even short clips can consist of more than a few gigabytes of uncompressed data. Therefore, it is often impossible to load all the texture data into memory before playing. Instead, the image data is loaded right before rendering.

Loading a few megabytes of data from secondary storage as well as copying data from main memory to texture memory for every frame slows down the rendering process as the bandwidth for data read from secondary storage is much lower than other interfaces, like memory bus, memory-GPU interface, etc. Thus, we implemented three optimization methods.

Prefetching Although a variety of prefetching algorithms has been proposed in the past [18], [25], [7], [9], which are based on a prediction of what is to be displayed next, it is difficult in a VE to correctly predict the viewer's motion and to load the predicted data for the next frame. Moreover, a sophisticated prediction scheme generates a large computational overhead and thus we decided to minimize the computation time for the prediction. The key observation we made was that often times, once viewers in the VE zoom in to the displayed video, they stop to watch the video clip. In VEs, there is always a slight change of the viewer position due to head tracking, but the movement does not affect the mesh generation algorithm as much as when the viewer walks around: in our experience, the difference between the tile meshes of two successive frames is at most four tiles when viewers are stationary.

Another issue is to predict the video frame from which the tiles are to be prefetched. After rendering the n-th video frame, due to the synchronization described in Section 4.4, the next video frame is the $(n+k)$-th frame, where k can be any positive number, which means that we skip frames $(n+1)$ to $(n+k-1)$. At every rendering step, k has to be estimated as correctly as possible, otherwise it causes the system to prefetch unnecessary tiles. Again, we adopted a simple, computationally light scheme based on reinforcement learning [22]. We estimate the next frame by looking at the history of frame rates. If the system has been skipping, for instance, every other video frame, we estimate that in the next image frame we are going to skip a video frame again. More formally, let A_n denote the current estimate of how many frames the system will skip and a_n be the current observation of the skip. Then, the next estimation of A_{n+1} is the weighted average between A_n and a_n.

$$A_{n+1} = \alpha a_n + (1-\alpha)A_n$$

where α is a parameter representing how fast the algorithm adapts to new information a_n as opposed to the history A_n. We use the rounded values of A_n for the estimation of how many steps to skip. In order to further improve the accuracy, the $(n+k-1)$-th and $(n+k+1)$-th frames are also prefetched. The number of tiles prefetched is conservatively kept low, from one to four tiles, to prevent prefetching from generating too much load for the entire system and to utilize only the idle time of the I/O thread without delaying immediate requests from the rendering process even in the case of misprediction.

Asynchronous I/O In order to accelerate data transfers between main memory and texture memory, a separate thread is spawned and dedicated to asynchronous disk I/O operations. Every disk read request is sent to the I/O thread via a message queue and the I/O thread reads data whenever it finds a message in the queue. There are two queues: a tile request queue and a prefetch request queue. The tile request queue contains the request from the main thread, which is for texture data of a tile that is needed to render the current frame. The prefetch request queue contains requests for texture data of a tile which will be needed in the near future. To provide texture data to the main thread in a timely manner, the messages from the tile request queue always have a priority over the messages from the prefetch request queue. In addition, the request for data loading is made as soon as the main thread finds a tile which will be needed for rendering. By posting disk I/O requests as early as possible, the overlap between the rendering process and disk operations can be maximized. Another message used for communication between the main thread and the disk I/O thread forwards the current frame number. Synchronization makes the main thread skip one or two image frames to keep the correct video frame rate and the I/O thread should not spend time on prefetching tiles of the skipped frames.

Memory Pool and Cache The third optimization we implemented is to pre-allocate a pool of memory blocks so data loading can save time for allocating memory blocks. The pool of memory blocks consists of a list of blocks, each of which can store the texture data of one tile. And the pool is initialized both in main memory and in texture memory. Whenever a new tile needs to load its data into the GPU, it finds an available memory block in main memory and texture memory and binds them. As video streams keep playing and as more memory blocks are requested, the pool runs out of free blocks at some point. Then, the two pools remove the least-recently-used blocks. For videos about the size of main memory, this means that most blocks end up staying in the pool after the first playback and the pool mechanism works as a cache and improves the data loading process when a video stream is played multiple times.

4.4 Synchronization

The time for rendering an image frame varies between frames, mostly depending on the number of tiles loaded for each frame. This causes two types of synchronization problems: synchronization 1) between frames and 2) between CAVE nodes. The first problem is that, without a synchronization scheme, the video frame rate changes depending on how many tiles are rendered for the frame, which varies depending on the viewer's location. Therefore, if one image frame is rendered quickly and the time spent on the frame is shorter than that of the video frame rate, e.g., 1/24th second, the video frame needs to be rendered again until the sum of the rendering times reaches 1/24th second. On the other hand, rendering one image frame may take longer than 1/24th second. For example, when the total time to render a frame is, for instance, 3/24th seconds, the next two video frames have to be skipped.

The second synchronization problem occurs because in a multi-node VE all nodes usually have different workloads and cause an imbalance in rendering times. For those display nodes that do not render much data, the rendering time is short, whereas other nodes might need more time for an image frame update than the video frame rate allows for, so that video frames have to be skipped. In our CAVE system, which is based on 17 computers and 34 passive stereo screens with head tracking, we update the images on all nodes at the same time, so that the update rate is equal to the frame rate of the slowest node.

Our software provides a synchronized time which is the same on all nodes. Using this clock, we measure the time passed since the start of rendering the first frame, $t_{elapsed}$. Then, the desired video frame number, d, can be easily computed with the following formula for a 24 frames per second video clip:

$$d = d_{base} + \left\lfloor \frac{t_{elapsed}}{1/24} \right\rfloor$$

d_{base} denotes the frame number of the first frame. d_{base} will change when a video stream is paused and later unpaused. This approach solves the two problems because the above formula enforces frames to change neither too fast nor too slow, which solves the first problem, and because $t_{elapsed}$ is measured from the globally synchronized clock, which is the solution for the second synchronization problem.

5 Results

We have implemented the above described video playback algorithm for virtual environments running on PC clusters by writing a C++ plug-in for the COVISE software framework. In this section, we describe the hardware environment we used for the experiment and the data sets we used. Then, we evaluate the algorithm by presenting frame rates for three different resolution settings.

5.1 Experiment Environment

We tested three different videos in our StarCAVE virtual environment. Our VE consists of five walls, each of which has three rear projected screens, and a top projected floor. Each screen is projected on by a pair of JVC HD2K projectors (1920 x 1080 pixels each), which are connected to an Intel quad core Dell XPS computer running ROCKS [16], with 4GB of main memory and dual Nvidia Quadro 5600 graphic cards. We copied the video data to the hard drives of each of the nodes.

5.2 Data Sets

In our experiment we used three different video clips: 1) one was a 4k (3840 x 2160 pixels) clip showing the result of a tornado simulation created by the National Center for Supercomputing Applications (NCSA) [2], 2) the same tornado clip at 2k (1920 x 1080) resolution, and 3) a series of light microscopy images (14914 x 10341 pixels) from the National Center for Microscpy and Imaging Research (NCMIR) [15] showing a mouse hippocampus cell. We preprocessed each of the video clips with our tiling and mipmapping tool. All three clips have 24 bit RGB colors and each image from three clips is divided into 512 x 512 pixel tiles. Only 1200 frames out of 3000 frames of the original tornado simulation video were used for experiments due to storage limitations, and the light microscopy clip consists of 24 frames.

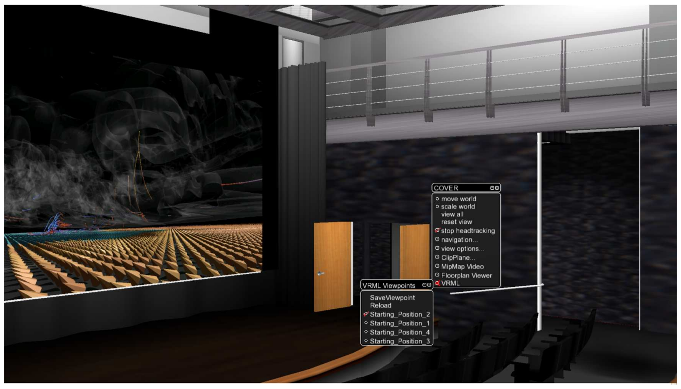

Figure 4: The video playback plugin embedded into a virtual theater VR application. Users can navigate into the theater and watch high resolution videos. The virtual theater application renders 201,688 polygons.

5.3 Results

Table 1 shows the frame rates for various settings. Because our VE displays stereoscopic images, the frame rate here is defined as a reciprocal of the time to render two images, one for the left and the other for the right eye. We could double the numbers in the table to show an update rate, the interval between two buffer swaps. However, we decided not to due to the fact that right eye images are rendered faster than left eye images. This is because, once texture data is loaded into memory for rendering a left eye image, it can be reused for rendering a right eye image for the video frame. All measurements were averaged over a full playback cycle of each clip.

We set the size of the playback canvas in two ways. One is the size of one display panel of our CAVE, which displays about 2M pixels (1920 x 1080 pixels). All three video clips were set to read 2M texels for a frame. With this size, the 2K video is played at its original resolution, whereas the 4K video is rendered at half its resolution (1/4 the number of pixels).

The second setting is for the 4K video. We zoomed in so that it occupied the area of 2x2 display screens in the CAVE. In this setting, the 4K video is displayed at its full resolution, reading 8M texels every frame. 2K video is streamed at the highest resolution - 2M texels - but it does not add more detail as the size of the playback canvas exceeds the size of the video frame. The video of the microscopic data set was displayed with 12M texels, and the result shows that our system is useful for scientists who want to visualize and examine ultra-high resolution time-series of images in a VE.

In addition to different screen sizes, we compared the frame rates of our optimized system (first column) to those of an unoptimized system (second column). The optimization includes asynchronous I/O, prefetching, and DXT compression while the unoptimized system loads uncompressed texture tiles at the time they are requested for rendering. The system with optimizations shows a 3 to 4 times speedup in all cases, compared to the one without these optimizations.

The third column for each video source shows the frame rate when the video is played at a lower resolution, i.e., 1/4 of the number of texels used for the first and second column. This is implemented in such a way that the quad tree traversal stops at an earlier stage than the original algorithm so that it produces lower resolution images. This functionality provides a preference between image quality and frame rate. If one wants to watch a video clip as fast

Video Clip	2K Video 1920 x 1080			4K Video 3840 x 2160			Microscopy 12941 x 10341		
Configuration	opt	no opt	LOD	opt	no opt	LOD	opt	no opt	LOD
2 x 2 walls	23.7	7.5	44.2	9.4	2.7	26.0	8.9	2.8	26.7
single wall	20.4	5.0	59.6	18.0	4.9	45.1	21.8	6.1	58.4

Table 1: Frame rates from three different video sources. Frame rate (frames per second) is the reciprocal of the time to render two images (stereoscopic image for left and right eye) and is averaged over a full playback cycle of each clip. Optimizations enhance the overall performance by a factor of 3 to 4.

Prefetching hit ratio	98.5 %
Percentage of preloaded tile	90.3 %
Co-executed with virtual theater	12.0 % slower

Table 2: Prefetching hit ratio is the ratio of how many times the prefetching algorithm successfully prefetched video frames. Prefetching and asynchronous I/O hide the latency of loading. The percentage of preloaded tiles tells us how many tiles were able to be loaded before rendering of the tile begins. The rest were stalled until the I/O process finished loading texture data from disk.

as possible, e.g., at 24 fps or more, not wanting to skip any video frames, then by using a lower LOD, the system achieves high frame rates. However, in some case, especially when investigating large scale scientific data sets, the full detail of the images is what scientists are the most interested in, as long as it is played at a reasonable frame rate to convey motion.

Table 2 shows how our optimization works and how our system can co-exist with other 3D geometry. The first row reports the prefetching hit ratio. The prefetching algorithm predicts the next frame and loads one to four tiles depending on the availability of system resources. The ratio indicates that it predicted correctly 98.5% of the time. The second row describes the percentage of tiles that are already in the main memory before the copying from memory to GPU begins. If a tile is not ready by the time of GPU offloading, the system stalls and waits. Both prefetching and asynchronous I/O work to prepare all the data needed before rendering, running parallel to the rendering process. Only about 10% of the tiles were unable to be prepared. The last row shows the performance difference when it runs with a VR application. We used a virtual building application and put our video playback canvas on the big movie theater screen inside the building. Figure 4 is the screen shot for this experiment. The 3D model of the virtual movie theater consists of 201,688 polygons. Compared to the original frame rate we measured without the plugin, the framerate decreased by only 12.0%.

6 Conclusion and Future Work

We showed and discussed the design and implementation of high resolution video textures in VEs. In order to achieve a constant video frame rate, we created multiple levels of detail and dynamically subdivide the video into a set of tiles with different LODs. For efficient disk I/O operations, we assume that the plane will not change too much between image frames and prefetch tiles for the next frame. This helps overlap rendering with texture copying. In addition, synchronization was considered to sync the speed of rendering image frames and the video frame rate. Our experiments have shown that our system provides constant frame rates and usable video playback performance.

The main disadvantage of the current implementation is that data has to be preprocessed to generate multiple resolutions. Downsampling increases the disk space requirement, and real-time video streams cannot be displayed. Since the downsampling process, however, is highly parallelizable and modern GPUs provide hardware supported parallel computing (e.g., CUDA), real-time downsampling might allow us to render real-time video streams in VEs in the future. In addition, real-time processing is possible to downsample only the regions actually rendered by the respective node, which will further reduce the overall computational cost of downsampling.

The current system needs to convert video clip data to a series of TIFF images and the downsampling process runs on the TIFF images. Supporting more image formats or even generating downsampled data from compressed video clips will significantly improve the usability of this video playback system.

Acknowledgements

This publication was made possible by Grant Number (NCRR P41-RR004050) from the National Center for Research Resources (NCRR), a part of the National Institutes of Health (NIH). Its contents are solely the responsibility of the authors and do not necessarily represent the official views of the NIH.

References

[1] Lawrence Livermore National Laboratory Blockbuster. `https://computing.llnl.gov/vis/blockbuster.shtml`.

[2] National Center for Supercomputing Applications. `http://www.ncsa.uiuc.edu`.

[3] OpenSceneGraph. `http://www.openscenegraph.org`.

[4] P. Bhaniramka and Y. Demange. OpenGL volumizer: a toolkit for high quality volume rendering of large data sets. *Proceedings of the 2002 IEEE symposium on Volume Visualization and Graphics*, Jan 2002.

[5] I. Boada, I. Navazo, and R. Scopigno. Multiresolution volume visualization with a texture-based octree. *The Visual Computer*, 17(3):185–197, May 2001.

[6] S. Bruckner and M. Gröller. VolumeShop: An interactive system for direct volume illustration. *Proceedings of IEEE Visualization*, Jan 2005.

[7] W. Correa, J. Klosowski, and C. Silva. iWalk: Interactive out-of-core rendering of large models. Technical Report TR-653-02, Princeton University, 2002.

[8] Y. Ebara, N. Kukimoto, J. Leigh, and K. Koyamada. Tele-immersive collaboration using high-resolution video in tiled displays environment. *ainaw*, 2:953–958, 2007.

[9] T. Funkhouser, C. Séquin, and S. Teller. Management of large amounts of data in interactive building walkthroughs. *Proceedings of the 1992 symposium on Interactive 3D graphics*, Jan 1992.

[10] S. Guthe, M. Wand, J. Gonser, and W. Strasser. Interactive rendering of large volume data sets. *Visualization*, Jan 2002.

[11] B. Jeong, L. Renambot, R. Jagodic, R. Singh, J. Aguilera, A. Johnson, and J. Leigh. High-performance dynamic graphics streaming for scalable adaptive graphics environment. *Supercomputing, 2006. SC '06. Proceedings of the ACM/IEEE SC 2006 Conference*, pages 24–24, Nov. 2006.

[12] Z. Jiang, Y. Mao, B. Qin, and B. Zang. A high resolution video display system by seamlessly tiling multiple projectors. *Multimedia and Expo, 2007 IEEE International Conference on*, pages 2070–2073, July 2007.

[13] E. LaMar, B. Hamann, and K. I. Joy. Multiresolution techniques for interactive texture-based volume visualization. In *VIS '99: Pro-*

ceedings of the conference on Visualization '99, pages 355–361, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[14] M. Meißner, U. Hoffmann, and W. Straser. Enabling classification and shading for 3D texture mapping based volume rendering. *vis*, 00:32, 1999.

[15] National Center for Microscopy and Imaging Research. `http://ncmir.ucsd.edu`.

[16] P. Papadopoulos, M. Katz, and G. Bruno. NPACI: rocks: tools and techniques for easily deploying manageable Linux clusters. In *Cluster Computing, 2001. Proceedings. 2001 IEEE International Conference on*, pages 258–267, 2001.

[17] J. Plate, T. Holtkaemper, and B. Froehlich. A flexible multi-volume shader framework for arbitrarily intersecting multi-resolution datasets. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1584–1591, 2007.

[18] J. Plate, M. Tirtasana, R. Carmona, and B. Fröhlich. Octreemizer: a hierarchical approach for interactive roaming through very large volumes. In *VISSYM '02: Proceedings of the symposium on Data Visualisation 2002*, pages 53–ff, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[19] S. Prohaska, A. Hutanu, R. Kahler, and H.-C. Hege. Interactive exploration of large remote micro-ct scans. In *VIS '04: Proceedings of the conference on Visualization '04*, pages 345–352, Washington, DC, USA, 2004. IEEE Computer Society.

[20] R. Singh, B. Jeong, L. Renambot, A. Johnson, and J. Leigh. Teravision: a distributed, scalable, high resolution graphics streaming system. *Cluster Computing, 2004 IEEE International Conference on*, pages 391–400, Sept. 2004.

[21] R. Stockli, E. Vermote, N. Saleous, R. Simmon, and D. Herring. The blue marble next generation – a true color earth dataset including seasonal dynamics from modis. Published by the NASA Earth Observatory, 2005.

[22] R. Sutton and A. Barto. *Reinforcement Learning*. The MIT Press, first edition, 1998.

[23] C. C. Tanner, C. J. Migdal, and M. T. Jones. The clipmap: a virtual mipmap. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 151–158, New York, NY, USA, 1998. ACM.

[24] H. Towles, W. chao Chen, R. Yang, S. uok Kum, H. Fuchs, C. C. Hill, N. K. J. Mulligan, L. Holden, B. Zeleznik, A. Sadagic, and J. Lanier. 3d tele-collaboration over internet 2. In *International Workshop on Immersive Telepresence, Juan Les Pins*, 2002.

[25] H. Vo, S. Callahan, N. Smith, C. Silva, and W. Martin. iRun: Interactive rendering of large unstructured grids. *Eurographics Symposium on Parallel Graphics and Visualization*, Jan 2007.

[26] W. Volz. Gigabyte volume viewing using split software/hardware interpolation. *Proceedings of the 2000 IEEE symposium on Volume Visualization*, January 2000.

[27] M. Weiler, R. Westermann, C. Hansen, and K. Zimmermann. Level-of-detail volume rendering via 3D textures. *Proceedings of the 2000 IEEE symposium on Volume Visualization*, Jan 2000.

[28] D. Weiskopf, M. Weiler, and T. Ertl. Maintaining constant frame rates in 3D texture-based volume rendering. *Computer Graphics International*, Jan 2004.

[29] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 169–177, New York, NY, USA, 1998. ACM.

[30] L. Williams. Pyramidal parametrics. *SIGGRAPH Comput. Graph.*, 17(3):1–11, 1983.