MASTER

Interactive creation of video for large scale volume data

van Dijk, T.A.J.P.

Award date:
2008

Technische Universiteit Eindhoven
Department of Mathematics and Computer Science

Interactive Creation of Video for Large Scale Volume Data

By
T.A.J.P. van Dijk

Supervisors:

Jack van Wijk (TU/e)
Kwan-Liu Ma (UCDavis)

Eindhoven, February 2008

Voor mijn ouders.

"On ne voit bien qu'avec le cœur, l'essentiel est invisible pour les yeux."

-Antoine de Saint-Exupéry

Contents

Preface

The work presented in this report was performed at the University of California, Davis, under the supervision of professor Kwan-Liu Ma, working within his *Visualisation and Design Innovation* (`ViDi`) research group.

I would like to thank the following people in Davis: professor Ma for hosting me, and providing ideas, support, and top quality resources, during my stay. Dr. Chaoli Wang, for providing many ideas, relevant references and helpful insights. The other `ViDi` research group members for their help, friendship, openness, amusing insanity, and for making me feel at home instantly.

Finally, I would like to express gratitude towards my supervisor, professor Jack van Wijk, for giving me this wonderful opportunity and being supportive throughout.

Chapter 1

Introduction

1.1 Problem Description

Through the increasing levels of sophistication of measuring equipment, lowering costs of high capacity storage, and rising processor speeds, data sets generated from various sources such as scans, and scientific experiments are becoming larger and larger, especially when the data is time-varying (i.e., data is given for a number of time steps, as opposed to a single, static data set). In some cases tera-scale, or even peta-scale sizes can be reached. Though storing these data sets in secondary and tertiary storage systems is still feasible, primary storage size and bandwidth have not grown in step, and creating interactive visualizations for entire ultra-scale data sets remains a significant challenge.

Furthermore, the creation of a useful video tour of the data set, which is the most attractive option for data sets that are too large to be rendered interactively, is done with primitive techniques. For example, in `ParaView`[1], an interactive scientific visualization application, it is possible to animate visualization parameters, or the camera viewing position directly, and then output a series of images which represent different views and parameters over time. These images can then be encoded as a video.

However, explicitly specifying a camera path, and visualization parameters over time can be a tedious and difficult task, as the scientist interested in the data might not intuitively know how best to navigate through the parameter space optimally. They may simply be trying to find a number of good views and parameters that show off some features of interest that the video needs to show.

Also, the scientist might be interested in using techniques such as repetition (e.g., to show an interesting feature multiple times, possibly from several different viewing angles), skipping through unimportant time steps, adding annotations, choosing between different interpolation methods for different frames, etc. These functions are usually available in video editing programs, or 3-dimensional modelling and animation software such as `Maya`[2], but are lacking in visualization tools.

[1] http://www.paraview.org
[2] http://www.autodesk.com/maya

Existing remote rendering solutions (see chapter 2) solve some of the problems mentioned above, but require a constant network connection between the client machine and one or more visualization servers. This can be inconvenient in many usage settings, where a permanent network connection of the required bandwidth is not constantly available.

1.2 Goal

Our goal is to design a system that allows a user to interactively explore large scale data using commodity hardware, such as a laptop computer, without requiring a constant high-speed network connection. This system should incorporate the ability for the user to easily and intuitively create high resolution videos of the data, through an intuitive interface, with a high level of configurability.

1.3 Requirements

We would like to have a system that can visualize even ultra-scale data sets interactively. This can be done by generating a simplified representation of the large scale data, such that this simplified representation of the data can be rendered using a less complex rendering technique that can be performed on commodity hardware. However, this representation should still carry enough information to allow the user to identify features of interest in the data set, which is necessary for a user guided video creation process. The simplified representation should therefore satisfy the following requirements:

- The representation is small enough to be downloadable over the internet in its entirety, without requiring a connection with the visualization server after this initial transfer.
- The data in the representation can be rendered interactively, even on low end machines (e.g., a laptop computer), without the need for high end graphics capabilities.
- The representation still allows for the identification of important features.

Furthermore, a video creation interface is needed that is intuitive and easy enough to use, so that a person without specific knowledge on video editing and encoding technology can still guide the creation of a video according to his needs.

A rough overview of the system to be designed is given in figure 1.1. The original large scale data set is stored on the server side, while the user's machine represents the client side. The black box in the middle represents the steps performed to present the data to the user in some simplified form, and it processes feedback from the user, which is then used to generate a suitable video. We need to suitably map these components to either the server or the client side of the system. The exact design of this set of system components is described in chapter 3.

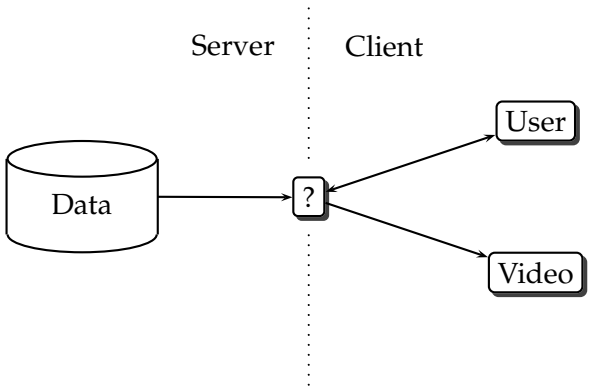


Figure 1.1: An overview of the system to be designed.

1.4 Overview

The rest of this report is laid out as follows:

- Chapter 2 deals with previous work done in areas relevant to our work.
- In chapter 3 we discuss how this work relates to our requirements and describes our design, which aims to meet those requirements.
- Chapters 4 and 5 describe two existing techniques that our used in our system: *image based rendering* and *volume rendering*, respectively.
- A detailed description of our system is given in chapter 6.
- Finally, chapter 7 summarises our work and offers some ideas for future enhancement.

Chapter 2

Previous Work

There have been various approaches to remote rendering systems. These differ mainly in the type of data that is transferred from the server(s) to the client, and which parts of the rendering pipeline (see section 2.1) are performed on the server side, and which on the client.

2.1 The Visualization Pipeline

The visualization pipeline consists roughly of the following processes, as illustrated in figure 2.1:

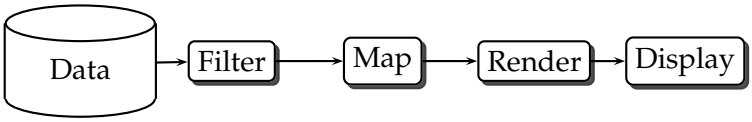


Figure 2.1: The rendering pipeline.

- First, a filtering step is performed, where the data is manipulated in some way (e.g., noise reduction, simplification, etc.)
- In the mapping step, the filtered data is mapped to some geometric representation (e.g., points, lines, polygons, etc.)
- These primitives are then rendered to a 2-dimensional image, using some graphics technique that is suitable for the geometric representation used.
- Finally, the resulting image is displayed to the user.

2.2 Process Mapping Examples

The suitability of each process mapping choice can be evaluated over a number of parameters of a given environment in which the system is to be used. These environment parameters can be divided into categories pertaining to different aspects of the environment. For example, the type of data is important; are we dealing with static data, or data that changes over time? How many and which types of variables are present in the data? We also need to compare the relative capabilities of the server and client machines; is the server much more powerful than the client? Furthermore, the network connection between those machines must be considered. Finally, an important factor is the type of person that will be using the software. This can range from a radiology expert who uses the software as a matter of routine, and has detailed domain knowledge, with the goal of making a diagnosis, to a researcher who is exploring newly obtained simulation data, who might want to present this data at a conference. Table 2.1 lists these environment variables.

We have found the following possible methods of mapping the pipeline processes to the client and server sides:

- The raw data is requested and transferred to the client, possibly from multiple sources. The client performs the entire rendering process, and requests data as needed. Only the data resides on the server; all other steps are performed on the client side.

  Systems like these usually optimize the network communication between servers and client [HHK+03, PF01].

  Here, we are dealing with an environment where network bandwidth is very high, which is necessary for the large amount of raw data that needs to be transferred, and a connection that is constant. The client machine must have enough resources to render the raw data entirely. The only requirement on the server is enough storage space for the data set.

- In other systems, a server performs data access and initial filtering and mapping steps, and sends data in some intermediate geometric representation to the client [COZ98, KKHV02].

  These systems may improve efficiency, for example, by only sending geometry that is currently visible to the user (frustum/occlusion culling).

Data	Server	Network	Client	User
time variability	memory size	bandwidth	memory size	expertise
variable count	processing speed	connectivity	processing speed	goal
size	storage space		storage space	

Table 2.1: Environment parameters.

Again, high network speed and connectivity is needed, while the resource requirement on the client side is reduced. The server must be capable of performing the filtering and mapping steps.

- In the third type of system all filtering, mapping and rendering is done on the server side, and the resulting image data, usually in some compressed form such as `JPEG`, is then transferred to the client for display.

  Bethel et al. have developed a system [BCY] that uses sets of tiled multi-resolution images rendered and encoded in advance on a server machine, to render large scale data sets on the client side. The images are streamed based on user demand. When zoomed in, tiles of higher resolution are requested from the server for increased visual quality, and when zoomed out, tiles of lower resolutions requested, to avoid wasting bandwidth.

  Remote desktop systems such as *Virtual Network Computing* [RSFWH98] and remote mapping systems such as *Google Maps*[1] also fall in this category.

  This method requires minimal resources on the client side, and maximal resources on the server side, as all of the expensive rendering steps are performed there, while client side rendering is reduced to a much simpler process, with lower resource requirements. Network speed must still be high, though this depends mostly on the level of compression achieved in the specific technique. Connectivity requirements also differ, though the aforementioned examples require a constant connection between the server and the client to be maintained. However, it is feasible to design a system that does not have this requirement.

2.3 Video Generation

We are not aware of any system that currently allows users to generate videos of arbitrary time-varying data sets. We feel there is a clear need for a method of user guided video content creation for large scale volume data, in terms of key frames that are specified by a user with only domain knowledge of the data in question, and not necessarily with expert knowledge of video authoring technology.

Rößler et al. have developed a system for use in medical data analysis, where the data in question originates on the client side and must be uploaded to the server, connected to a GPU-cluster, which then generates video material [RWIB+07]. However, their work focuses on creating a standardised video generation process, specifically for static medical data. The video generation happens automatically, using heuristics derived from specific domain knowledge of the medical field of radiology, and knowledge of the structure of the medical data set, as opposed to being guided by the user, and is not based on any kind of exploration of the data beforehand. In fact, the video material generates is meant as a replacement for a real-time interactive exploration of the data.

[1] http://maps.google.com

In contrast, our system should not require any knowledge of the nature or size of the data, and this data may represent anything, as long as it can be rendered to two-dimensional images. The data used may also change over time.

Chapter 3

Design Considerations

3.1 Environment Assumptions

We would like to develop a system that would be useful in an environment where ultra scale data sets are used. Data sets are growing larger and larger, as mentioned in section 1.1. We will assume an environment where a powerful server machine (or cluster of machines) is available, but the user is restricted to a much simpler machine, such as a laptop, with relatively low processing speed, memory size, and storage availability.

To allow the user to roam, which is especially useful in the case where the client is a laptop, we assume a network situation where the client does not always have access to the network. For example, the client could be a laptop connected to the internet some of the time, but not constantly. This means the required data must be downloadable over an internet connection in its entirety, so that after an initial data download step, the data can be viewed even when the client is off-line. For example, the user might be a scientist who wants to present the data at a conference where no internet access is available.

3.2 Rendering

Traditional graphics rendering algorithms usually have a time and space complexity that is a function of the size of the geometric data that is to be rendered. Thus, in the case of ultra scale data sets, achieving interactive rendering speeds on commodity hardware is difficult, especially when available memory size is also low, as is the case on the client system used in our assumed environment.

Rather than attempting to render the entire ultra-scale data set using one of these traditional techniques, we would like to use some method where speed and memory requirements do not depend on the original resolution of the ultra scale data set.

An established technique that satisfies this requirement is image based rendering (IBR). The general idea of this technique, along with two specific instances of it, are described in detail in chapter 4. A motivation is also given for the decision to use one

of these two specific methods in the system.

An important and widely used technique used in scientific and medical visualization, is *volume rendering*. We have decided to focus on this technique, because it is useful for many applications, but the system is designed in such a way that other visualization methods could be used as well, using the same general ideas. Volume rendering is explained in chapter 5.

3.3 Communication

In our system, we assume that the network connection between the server and client is not persistent; the server prerenders the image data for a specific data set, which is then downloaded to the client side once. At this point, the user can explore the data off-line interactively. This is possible because the image data is not generated on demand, based on the current viewing state. Instead, the image data is generated to accommodate an image based rendering technique on the client side.

This image data generation is effectively a subsampling of the original volume data. This sampling must be done sparsely, such that the resulting image data is small enough to fit in the client machine's memory.

3.4 Client Side Exploration

On the client side, the user should be able to interactively view the data. This includes being able to manipulate the camera angles to be able to view the data from any direction, as well as the ability to zoom into, and away from the centre of the data set.

Additionally, the user should be able to have at least some limited ability to manipulate the *transfer function* (see chapter 5) applied to the data, in the case where the original data contains scalar values. This is crucial to obtain a meaningful insight into the volume data, as with the use of any fixed transfer function, there will always be parts of the data that would be obscured by others, and not all important features in the data can be revealed. It's also very important to make sure that the user can specify different transfer functions to be used throughout the video that is to be generated, to highlight different features of the data set. A detailed description of how such a limited form of transfer function manipulation has been implemented for scalar data sets can be found in section 6.4.

3.5 Video Generation

In addition to being able to interactively view the data, the user should be enabled to guide the creation of a video, by specifying views of interest, which are recorded as key frames, and may be accompanied with additional parameters to indicate options and preferences. Information associated with each key frame may include camera angles, camera zoom factor, transfer function settings, and a textual annotation to appear in

the video as a subtitle during the display of that frame. Other parameters such as repetition, interpolation type at specific time intervals, etc. may also be specified. The data for these key frames and the additional options are then sent to a server machine, which has direct access to the complete data in its original representation. The video is then generated by interpolating between the views of interest according to the user's specification. Once the video is complete, it can be downloaded to the client machine.

A more detailed illustration of the system architecture is given in figure 3.1.

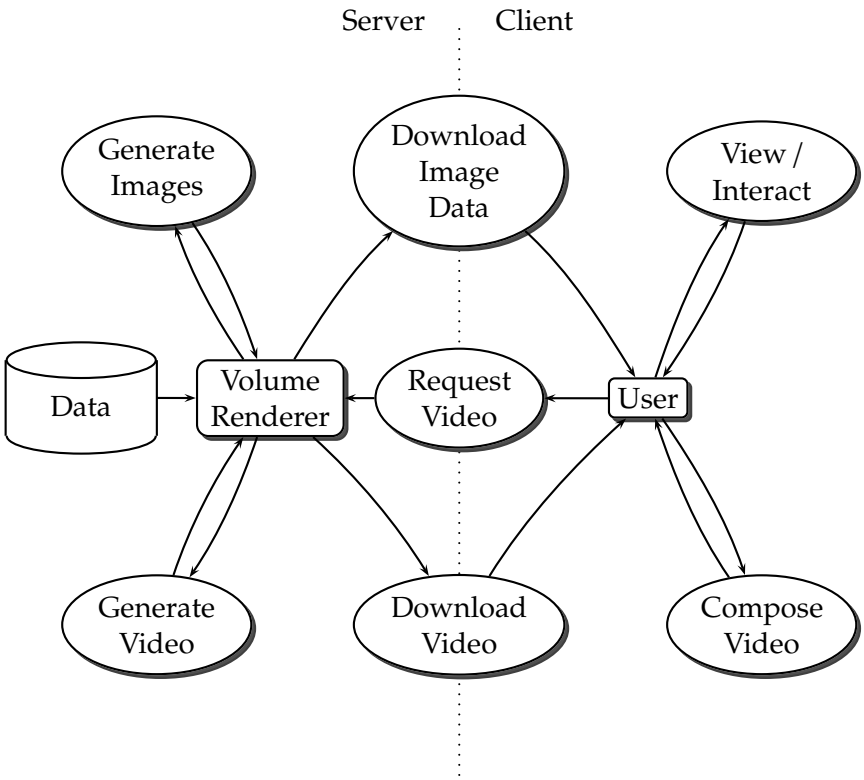


Figure 3.1: Architecture and operation of the system.

Chapter 4

Image Based Rendering

Image based rendering (IBR) is a collection of techniques used in the field of computer graphics that differ from traditional techniques in that they use a collection of sampled images as the scene representation, as opposed to the more traditional approach of using a collection of geometric primitives.

The various IBR techniques differ with respect to the sampling method and density, the rendering process, and the amount and type of auxiliary data (e.g., geometry data, optical flow, etc.) used. Shum and Kang offer an extensive review of existing IBR techniques, differentiating between techniques that require no, implicit, and explicit geometric information about the scene [SK00]. Techniques that use explicit geometry require the least amount of image data (in the most extreme case, polygonal data is available, and only image data in the form of texture maps is necessary), whereas ones that use no geometric information at all require the greatest amount of image data, and therefore also the most storage space, as image data in this case will be more data intensive.

Since we have no direct geometric information available to us in volume rendering, we have to rely on a technique that does not require any. Those techniques can be described as being various approaches to *plenoptic modelling*, introduced by McMillan and Bishop [MB95], using one form of the *plenoptic function* [AB91].

Section 4.1 describes some general theory behind various forms of image based rendering. Sections 4.2 and 4.3 describe two specific techniques that we used for our system: *light field rendering* and *object movie rendering*. Finally, section 4.4 gives a comparison between these two techniques.

4.1 The Plenoptic Function

The original plenoptic function is:

$$I = P(\theta, \phi, \lambda, V_x, V_y, V_z, t) \tag{4.1}$$

It is a 7-dimensional function that gives the intensity I, of the pencil of light passing

through an idealised eye at every location (V_x, V_y, V_z), at every angle (θ, ϕ), for every wavelength λ, at every time t, as illustrated in figure 4.1.

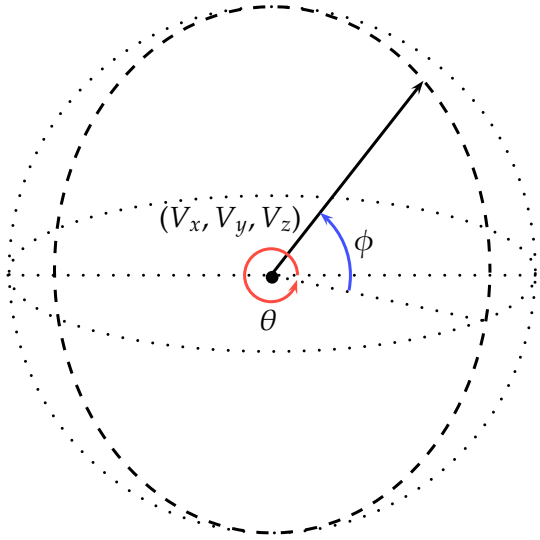


Figure 4.1: The plenoptic function gives the light intensity at given wavelength λ, arriving at a given location (V_x, V_y, V_z) at given angles (θ, ϕ) at time t.

One possible representation of a scene would be a complete sample of the plenoptic function, for all positions, angles, wavelengths, over time. With such a representation, any desired view could be generated by substituting given values for $V_x, V_y, V_z, \theta, \phi, \lambda$ and t. However, in practice it is unfeasible to obtain such a complete sampling over a 7-dimensional continuous space. Instead, we can approximate such a representation with some form of discrete sampling.

Furthermore, the high dimensionality of the plenoptic function can be reduced in various ways. For static scenes, with fixed lighting conditions, we can omit parameters λ and t, and have the function produce a triple of intensity values (e.g., using an RGB colour space):

$$\vec{C} = P'(\theta, \phi, V_x, V_y, V_z) \tag{4.2}$$

In the situation where we have a fixed viewpoint, we can simply reduce the plenoptic function to:

$$\vec{C} = P''(\theta, \phi) \tag{4.3}$$

In this context, a regular photo can be interpreted as an incomplete sample of the plenoptic function, for a certain discrete number of points within a range of values for θ and ϕ.

This approach is used by Szeliski and Shum to create cylindrical and spherical panoramas from a sequence of images, by warping them into cylindrical or spherical co-ordinates [SS97].

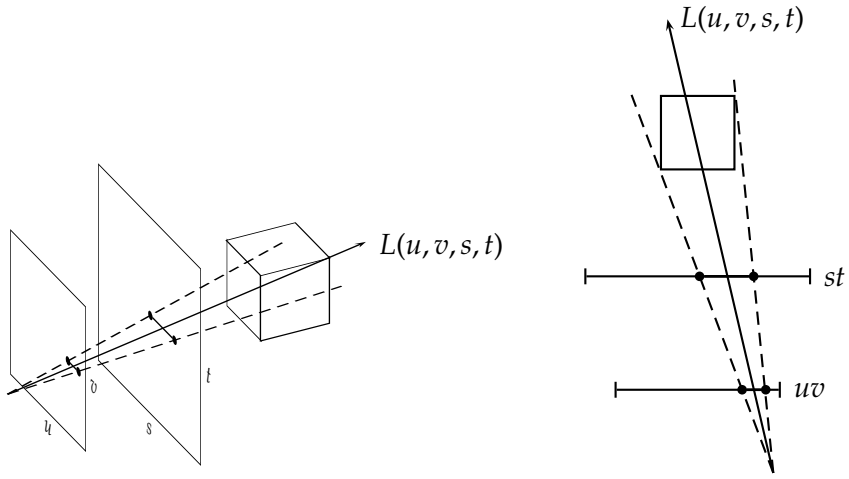
Light Field Rendering is another IBR technique that uses a different version of the plenoptic function. This technique is described in detail in section 4.2.

4.2 Light Field Rendering

Levoy and Hanrahan [LH96] make the observation that the intensity of a ray of light does not change along its length, as long as there is no occlusion between the source of the ray and the observer (i.e., if the observer stays outside the convex hull of the object being observed), the plenoptic function can be simplified to 4 dimensions. Under this restriction, they change the parameters of the plenoptic function as follows:

$$I = P_{LF}(u, v, s, t) \tag{4.4}$$

where (u, v) and (s, t) form a local co-ordinate system for two planes, possibly of different size, positioned somewhere in space (see figure 4.2), parallel to each other. The collection of all light rays that pass through both of these planes is called a *light slab*.



(a) Perspective view. (b) Top view.

Figure 4.2: An example of a light field *slab*, formed by planes parallel to one side of the object's bounding box.

4.2.1 Creation

Generating the image data required for light field rendering entails defining the geometry for a set of light slabs, positioned around the object of interest, and then rendering

views of the object from a number of camera positions along the u and v axes on one plane (the UV-plane), aimed at the other plane (the ST-plane), for each slab in the set. The number of slabs, their position, orientation, and the sizes of their planes can be chosen, along with the sampling density along each of the four dimensions. Each slab offers a view from one direction.

This yields a collection of images that each represent a slice of the 4D light slab. Each pixel in an image represents a point (s, t) on the ST-plane, for a camera position (u, v). Therefore, the image dimensions chosen for the rendering corresponds to the sampling frequency in s and t. The number of images rendered corresponds to the sampling frequency in u and v (see figure 4.3).

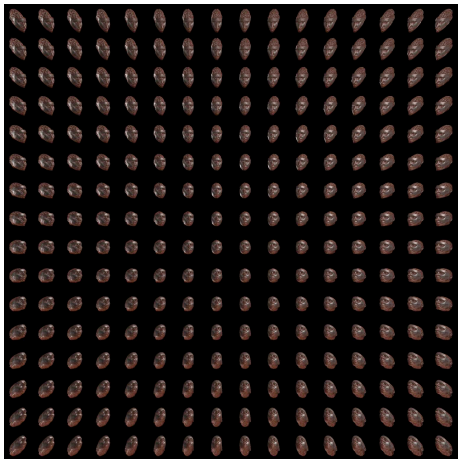


Figure 4.3: An example of a light slab, in a 2D array of 2D images representation. Each image corresponds to a point on the UV-plane.

Alternatively, the images can also be generated from the real world, by using a camera on a gantry. This is however not applicable to our system.

4.2.2 Rendering

Once all light slabs have been created, arbitrary new views from any point outside of the object's convex hull can be rendered from the data obtained. This is done by resampling the light slabs for each pixel on the screen. A ray from the eye point through the centres of each of those pixels is cast into the scene, and if it intersects with any of the light slabs, the intersection points with each of its two planes is found. The co-ordinates for these two points ((u, v) and (s, t)) are then used to index into the acquired images to find an appropriate pixel colour. The (u, v) co-ordinate indicates the image in the slab, the (s, t) co-ordinate indicates the pixel within that image.

Section 6.2.1 describes the light field rendering implementation in more detail.

4.2.3 Compression

Light field rendering requires the use of a large amount of data. For an uncompressed light field, we need the following amount of bytes:

$$nsl * ns_u * ns_v * ns_s * ns_t * bpp \tag{4.5}$$

where nsl denotes the number of slabs used, ns_x the number of samples in dimension x, and bpp the number of bytes stored per pixel, usually one byte per colour component. In the case of time varying data, this number must be multiplied with the number of time steps. This number can be very large, even for modest choices of these variables.

Vector Quantisation

Levoy and Hanrahan propose a vector quantisation [GG91] scheme to alleviate this problem. A pixel colour can be interpreted as a 3-dimensional vector, if the colour is broken down into its components (e.g. red/green/blue, hue/saturation/value). These vectors can be used to train a codebook of vectors, such that each of the pixels in the light field can be replaced by an index into the codebook, that points to the closest matching vector. Using a codebook with cb_size vectors, this index would take $\lceil \log_2(cb\_size) \rceil$ bits to store, which offers a reasonable improvement. at the cost of some quality loss.

To further reduce storage requirements, instead of quantising individual pixel colours, tiles of pixels can be grouped together into a higher dimensional vector, and then quantised. For example, a 2-dimensional tile of four pixels can be interpreted as a single 12-dimensional vector (assuming 3 colour components), or, extending this idea to all four dimensions, 4-dimensional tiles of sixteen pixels can be used. This means that these groups of pixel values can now be replaced with a single index into the codebook. If we define ts_x as the tile size in dimension x (e.g., for 4-dimensional tiles of sixteen pixels this would be 2 for each of the four dimensions), the total storage requirement in bytes per time step becomes:

$$nsl * \left\lceil \frac{ns_u}{ts_u} \right\rceil * \left\lceil \frac{ns_v}{ts_v} \right\rceil * \left\lceil \frac{ns_s}{ts_s} \right\rceil * \left\lceil \frac{ns_t}{ts_t} \right\rceil * \left\lceil \frac{\log_2(cb\_size)}{8} \right\rceil \tag{4.6}$$

An advantage of vector quantisation is that it still allows for random access to vectors in compressed form, which is necessary during the light field rendering step.

Obviously, choosing higher values for ts_x will severely reduce the visual quality of the final rendering output. See section 4.4 for more on this.

Entropy Coding

Following the vector quantisation step, more storage space can be saved by using an entropy coding scheme, such as one based on Lempel-Ziv coding [ZL77] on the resulting indices. This is especially true since large parts of the images will have an identical background colour, which means that those background pixels will result in an identical index value.

The drawback of an entropy coding scheme is that random access to the pixel data is no longer possible, so decompression must happen before the rendering process can start, and no memory space is saved once rendering starts. However, this technique is still useful, as it will reduce the time needed to download the light field data from the server.

4.3 Object Movie Rendering

QuickTime VR [Che95] is a much simpler system than light field rendering. It has two modes; in *panoramic mode*, a static viewer in the centre of the scene, looks out, and is able to rotate the view and look all around into a panorama from this fixed position.

The other mode, which is called *object mode*, uses rows of images rendered (or photographed) from different points along a circle around a scene object, where each row of images represents a different elevation. Given these images, the system can then give the illusion of interactive exploration, by selecting the image from this set that is the closest approximation to the current view as specified by the user.

4.3.1 Creation

Creating the data for an object movie entails taking views of an object from a number of camera locations on a circular path around the object of interest. To additionally allow for different elevations, samples at several paths around the object, positioned at different heights, have to be obtained. In other words, sample views are created from different points on the surface of a sphere centred at the object's centre. Each point corresponds to a *zenith* ($0 < \phi < \pi$) and an *azimuth* ($0 < \theta < 2\pi$) angle. The number of pixel samples can be expressed in the number of sample images along each of these two dimensions, ni_ϕ and ni_θ, multiplied by the number of pixels per image: $ni_\phi * ni_\theta * ns_w * ns_h$. Figure 4.4 shows this sampling configuration for $ns_\theta = 18$ sample points along the circular path, at $ns_\phi = 5$ different elevation angles.

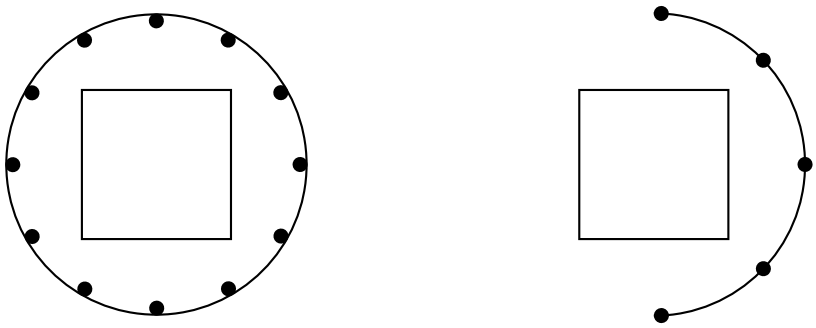
4.3.2 Rendering

To render a given view of the object, the closest matching sampled view is chosen from the image data and displayed. Using a typical trackball style camera interaction style, where the mouse is used to manipulate the azimuth and zenith angles, this can be done by comparing those angles to the ones used for each image in the sampling process, and selecting the most closely matching image for display.

4.3.3 Compression

The uncompressed image data size in bytes, with *bpp* the number of bytes per pixel, is:

$$ni_\phi * ni_\theta * ns_w * ns_h * bpp. \tag{4.7}$$



(a) Top view. (b) Side view.

Figure 4.4: Example camera positioning used to generate samples for object movie rendering.

To compress the image data, each row of images corresponding to an elevation angle can be stored as an image strip, encoded as a `JPEG` image, where the quality parameter for the `JPEG` compression process determines the trade-off between image quality and compression ratio.

4.4 Comparison

Table 4.1 lists some of the advantages and disadvantages of the techniques described above.

	Light Field Rendering	**Object Movie Rendering**
Advantages	high quality possible smooth interaction flexible zooming	easy to implement requires less storage no graphics hardware required simple and fast compression
Disadvantages	hard to implement requires more storage requires graphics hardware compression difficult and slow	jerky interaction pixel zoom only

Table 4.1: Advantages and disadvantages of the two image based rendering techniques used.

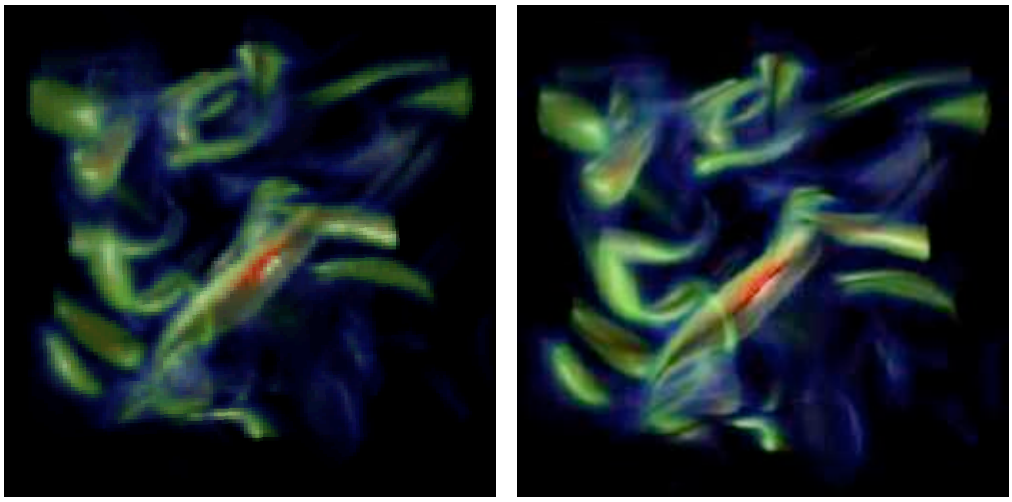
We have created implementations for both the light field technique, and for object movie rendering. With the former technique, we saw that although the image quality is

very good for high sampling densities, the image data for time-varying data sets is too large to be feasibly downloaded and stored in memory on the client side. Additionally, it is difficult and very time consuming to find a suitable codebook for the quantisation process, such that it represents a good trade-off between image quality and compressed image data size.

Figure 4.5 shows the same vorticity data set rendering using the two techniques. In this static view, image quality is comparable, though interaction using light field rendering is much smoother, as the light field is simply resampled for each pixel in the view, whenever the viewing conditions change. This also includes zooming. With object movie rendering, interaction is more jerky, which is noticeable when changing the viewing angles, as a result of the switch from one sample image to the next. Zooming in and out is done using a simple pixel zoom, which results in noticeable pixelation when zooming in too close.

However, there's a big difference in the size of the data sets used. The light field data set consists of six slabs (one view for each of the six faces of the cube enclosing the volume), each using the following sampling parameters: $ns_u = 32$, $ns_v = 32$, $ns_s = 256$, $ns_t = 256$, and the following vector quantisation parameters: $ts_u = 1$, $ts_v = 1$, $ts_s = 2$, $ts_t = 2$, $cb\_size = 4096$. Using these values, equation 4.6 gives us a size of 192MB for one time step. Using entropy coding this can be brought down to about 30MB. The size of the accompanying codebook image is negligible.

The sampling parameters used in the object movie data set are as follows: $ni_\phi = 5$, $ni_\theta = 18$, $ns_w = 256$, $ns_h = 256$, $bpp = 3$. Using equation 4.7 gives a size of 16.88MB. Using JPEG compression with a quality level of 60, this size is reduced to about 500kB.



(a) Rendered using light field rendering. (b) Rendered using object movie rendering.

Figure 4.5: Comparison of image based rendering techniques.

Clearly, this size difference weighs heavily in favour of object movie rendering. The

technique is also easier to implement, and requires no dedicated graphics hardware. Furthermore, the JPEG compression used is straightforward and fast. We therefore chose to use that technique for the implementation of the system.

Chapter 5

Volume Rendering

5.1 Background

Volume rendering [DCH88] is a technique used when values in the data set are discretely sampled within a volume of space. A *voxel* (volume element) is one such discrete, three dimensional cube-shaped sample point in space, analogous to the concept of a pixel (picture element) in a two dimensional image, it is the fundamental unit of a volumetric data set. This is illustrated in figure 5.1.

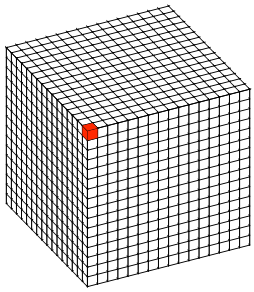


Figure 5.1: A three dimensional subspace, discretized into a collection of voxels. One voxel is highlighted.

Typical examples of situations where this type of data is obtained, are CT and MRI scanning, seismic measurements, and simulations performed in the fields of physics. Each voxel in the data set contains either a single value (mono-variate), or a set of values (multi-variate). Values can be in the form of scalars (e.g., temperature, density, velocity magnitude, etc.), or sometimes in the form of vectors (e.g., velocity of fluid flow) or tensors (e.g., stresses inside a solid body).

5.2 The Transfer Function

In order to visualize the data, a *transfer function* is defined, usually by the scientist who has domain knowledge of the data, which maps the values of the data variable to a colour. Given a data set where the data variable is temperature, a simple example would be to make the lowest value map to blue (cold), and the highest value to red (hot), with a gradient from blue to red for the values in between.

Data values are also mapped to an accompanying opacity value, ranging from 0 (fully transparent) to 1 (fully opaque). This opacity value is necessary to enable the user to see the data inside the volume, which would otherwise be obscured by objects of a different material surrounding it. Take for example a scan of a human head; in order to see the brain, the range of values that correspond to the skull would have to be mapped to a mostly transparent opacity value, so one can see through it.

A variety of algorithms can then be used to render the volume to a 2D image [MHB+00]. Figure 5.2 shows an example of a volume rendered data set. The transfer function has been chosen such that the skin is mapped to a partially transparent colour, thus revealing the underlying bone structure.

Figure 5.2: Volume renderings of a human foot. The bone structure is visible in a lighter colour.

5.3 Optimization

To speed up the rendering process, clusters of computers can be used to perform the rendering in parallel, in the following ways:

- Dividing the volume data into subvolumes, which are then sent to the nodes for rendering. The resulting image data is then collected, and composited into a final image.
- Dividing the image to be rendered into subregions, and letting each node take care of each region, followed by stitching the resulting images together.

In modern implementations, programmable graphics hardware is exploited to vastly speed up the rendering process. This can be done very effectively because of the highly parallel nature of video hardware, which lends itself well to direct volume rendering algorithms such as ray casting.

In this technique, rays are cast from the eye point into the volume, for each pixel on the screen. Along this ray, sampling is done for each voxel that is encountered in the volume. A colour and a transparency value is accumulated after application of the transfer function. After each sample, a step size is added to the current position along the ray and the next sample is obtained. This continues until the position along the ray is outside the volume, or until the accumulated transparency value reaches 1, in which case no further samples would affect the final pixel value [DH92, KW03]. The step size used is an important parameter of the algorithm, as it greatly affects execution time and image quality. A smaller step size slows the system down, while a greater size will result in sampling artefacts, as clear gaps appear in the rendered image.

Chapter 6

System Description

6.1 Image Generation and Compression

The image data is generated from a modified volume renderer, for each time step of the original volume data. This renderer uses the ray casting technique, where ray casting is performed on the GPU [KW03]. Volume rendering is explained in chapter 5. These images are then compressed and made available for download through a regular HTTP server. The images are generated and compressed differently for the two methods of client side rendering used.

6.1.1 Light Field Rendering

Images are generated according to a given set of light slabs, as described in section 4.2.1. For each point (u, v), an image is rendered of the ST-plane, representing a slice of the 4D light slab. This is done using a sheared perspective projection, to ensure that each pixel corresponds exactly with a sample point (s, t) on the ST-plane.

As described in section 4.2.3, the image data is compressed first using vector quantization. An open source library for advanced compression algorithms called QccPack[1] is used to train a codebook using a subset of the image data as the training set. The codebook obtained is then used to quantize the image data.

Finally, commonly available entropy coding software is used to further compress the data without loss of quality. Any kind of entropy coding scheme could be used, but experimentation with other options showed that the best results are achieved using the LZMA (Lempel-Ziv-Markov chain-Algorithm), a modern extension to the classic Lempel-Ziv [ZL77] algorithm.

The compressed image data files and their associated codebook file, as well as the slab configuration details are described in an XML file, which is read by the viewer application (see section 6.2.1).

[1] http://qccpack.sourceforge.net/

6.1.2 Object Movie Rendering

Images are rendered from viewpoints along a number of circles around the volume, as described in section 4.3. If transfer function manipulation is used, the image data is generated in a special way, to accommodate separate data for different value ranges in the data set. This is described in section 6.4.

A script is used to execute ImageMagick[2] command line tools which are used to tile the images into strips, and to convert them into a compressed JPEG format. Each strip of images corresponds to the set of viewpoints along a circle around the data set.

6.2 Client Side Rendering

6.2.1 Light Field Rendering

A standalone light field rendering application was developed to render light fields created by the modified volume renderer. The renderer can handle either uncompressed image data, or compressed light fields. Sampling the light field is done for each pixel on the screen independently, and thus can be done in parallel. It is therefore an excellent candidate for implementation in the form of a fragment shader; a program that is compiled and executed on the *Graphics Processing Unit* (`GPU`), which is highly parallel in nature. We have developed a new light field rendering implementation that exploits the parallel nature of programmable graphics hardware commonly available today.

Decompression and Loading

Firstly, the light field data files are loaded from disk, one file per slab per time step. To allow exploration all around the object, six slabs are needed, each one representing a view of one of the six faces of a cube. The files can be compressed for transfer using the `LZMA` compression algorithm. These compressed files are decompressed first, which is necessary because we need random access to the light field data in later stages.

File Structure

Each light field comes with an accompanying description file, structured in an `XML` format. This file defines each slab of the light field, and contains information for each time step including in it. For each slab, it defines the geometry of the UV- and ST-planes, indicates the number of samples along each of the four dimensions, and refers to external files that contain the actual light field data. It also contains information about one or more codebooks that were used for the vector quantization step; the tile setup, number of tiles, and a reference to an external codebook file.

Once decompressed, the files for each slab are structured as a 2D array of smaller 2D grids of indices. Each index refers to a codebook entry. This can be a specific pixel colour, or a tile of pixel colours. For example, if the tile setup is give in the description

[2] http://www.imagemagick.org

file as $ts_x = 1 \wedge ts_y = 2$ for $x \in \{u, v\}$ and $y \in \{s, t\}$, then each index refers to a tile of $1 * 1 * 2 * 2 = 4$ pixels.

The codebook itself is stored as a `PNG` image. `PNG` (Portable Network Graphics) is a lossless bitmap image file format. Each tile of the codebook is stored as a consecutive sequence of pixels in the image. Using the above example, each of these sequences would be four pixels long. Ideally, the image would be one pixel high, that is, it would simply be a one-dimensional strip of tiles. However, in practice, video hardware imposes size limitations on images that are to be loaded as textures. If we were to use a consecutive strip of tiles representation, we could easily exceed a texture width restriction. To overcome this, multiple rows can be used for the codebook image if necessary (figure 6.1).

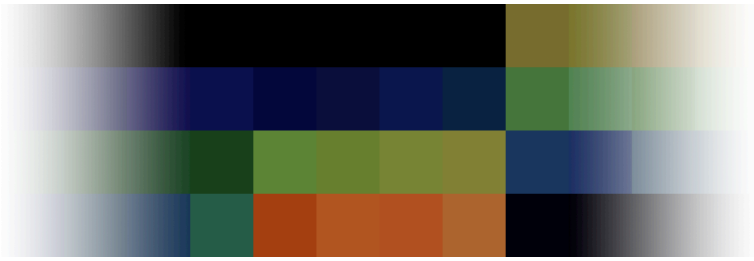


Figure 6.1: A small fragment of an example codebook file.

Internal Representation

Internally, each slab of the light field is represented as a 2-dimensional array of 2-dimensional images, as illustrated in figure 4.3. Here we can again run into the texture size limitation problem mentioned before. To circumvent this, the grid texture is split up into multiple subgrids as necessary, where each subgrid contains a subset of the UV images.

Initial Rendering Stage

When rendering the light field, we first need to find out, for each pixel on the screen, the corresponding set of (u, v, s, t) coordinates, given the current viewing parameters. We can then use these coordinates to sample pixel values from the light field in a later stage.

Tracing a ray from the eye point, through each screen pixel, into the scene and finding the intersection points with the UV- and ST-planes, would be a costly operation. We can simplify this step by observing that the transformation from screen coordinates to both the (u, v) and (s, t) coordinates is a projective map, and that this mapping can be done using the standard texture mapping technology widely available on graphics hardware today.

For each slab in the light field, we draw quadrilaterals for both the UV- and ST-plane, and assign texture coordinates to each corner vertex. The coordinates will then be interpolated over the surfaces of the quadrilaterals by the standard rendering pipeline. The interpolated values are then available to a simple fragment shader, which encodes those values into the output fragment colour (the u value corresponds to the red component, v to green, etc.)

We use the widely available `OpenGL` extension `GL_EXT_framebuffer_object` to render the result of this pass into a texture. This gives us a texture which contains a set of (u, v, s, t) coordinates, encoded as texel values, for each pixel on the screen. This texture is used in the second rendering pass.

Rendering on the GPU

In the second rendering pass, we start out by drawing a quadrilateral which covers the screen exactly, and bind the "coordinate texture" that was generated in the first pass to this quadrilateral. This allows us to use a fragment shader that has access to the coordinates values for each pixel on the screen. This shader also has access to the texture (or textures) that represent the light field, and the texture that represents the codebook that was used during the vector quantization phase.

The actual lookup to determine the colour for the current fragment in the fragment shader is then performed as follows:

1. Sample the coordinate texture with the current texture coordinates to obtain the (u, v, s, t) coordinates for the current fragment.
2. Use these coordinates values to compute an offset into the light field index texture.
3. Sample the index texture to obtain an index into the codebook.
4. Use this index to compute an offset into the codebook texture and an offset to the pixel within the corresponding tile.
5. Sample the codebook texture to obtain a colour value for the fragment.

The lookup step is illustrated in figure 6.2.

Sampling and Interpolation

By default, simple point sampling is used to determine a colour for each pixel on the screen. That is, the closest sampled point in each of the four dimensions is used. Alternatively, the user can choose to use one of three methods of interpolation; bilinear interpolation, either over the UV-plane, or over the ST-plane, which uses four samples (two in each of the chosen dimensions), or quadralinear interpolation over all four dimensions, which uses sixteen samples and interpolates between them.

Interpolation makes the shader more complex, depending on the number of samples used, but offers a nicer image quality, as it greatly reduces aliasing. Quadralinear

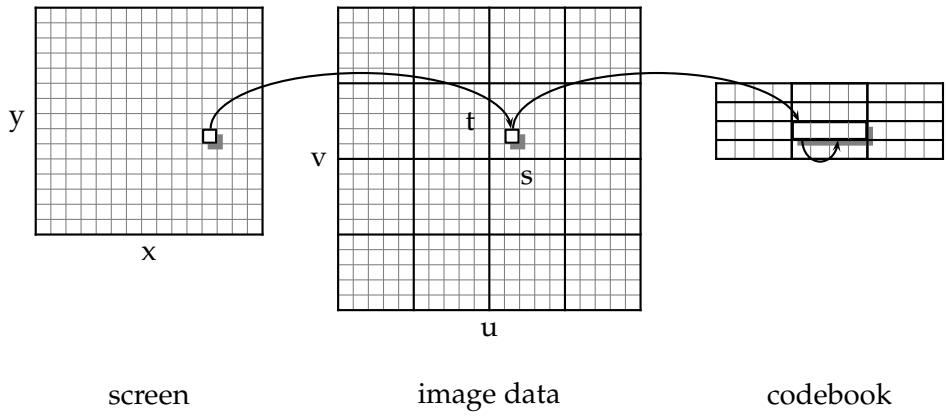


Figure 6.2: A codebook lookup is performed by first finding the tile in the codebook that corresponds with the value stored in the index texture for the current screen pixel. Then an offset is calculated for each individual pixel within the tile.

interpolation is most effective and leaves few artefacts, but is also the most expensive. In cases where the sampling density in one plane is much higher than it is in the other, bilinear interpolation in only one of the two planes can already be sufficient to reduce aliasing artefacts.

Compositing

Performing the rendering steps described above will yield only a (possibly empty) set of foreground pixels for each slab. In order to create the final image, the pixels for all slabs must be combined into one final image. To facilitate this, the stencil buffer is used during the initial rendering stage, to mask out all parts of the screen where the projections of the two planes for the current slab do not overlap. In doing so, in the later stage, rendering is restricted to only the area where those areas do overlap. This has the additional benefit that the fragment shader will only be run for those fragments where a meaningful foreground colour can be selected from the light field, eliminating the need to waste resources on background pixels. See figure 6.3 for an illustration of this step.

In this way, each slab contributes an area of screen pixels that are then blended together to form a final image. In practice, these areas will be mutually exclusive, unless a redundant slab configuration was used.

Figure 6.4 shows a screenshot of the application.

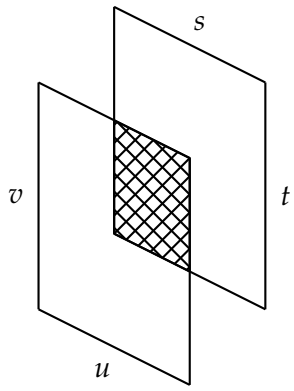


Figure 6.3: Only the region of intersection of the projections of the two slab planes is rendered.

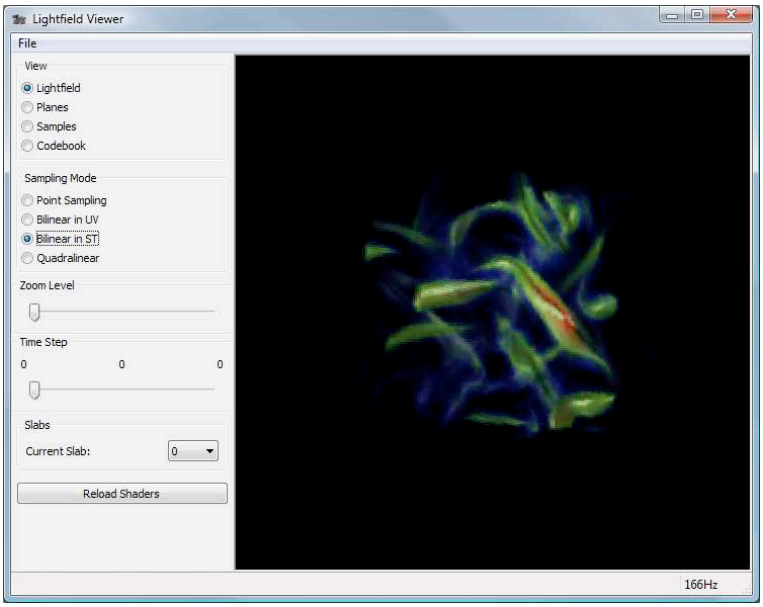


Figure 6.4: A screenshot of the light field viewer, displaying a light field of the vorticity data set.

User Interaction

The user can interact with the light field viewer application by using the mouse to change the viewing orientation using the standard *trackball* technique, by pressing and holding the left mouse button and moving the mouse. Similarly, the user can zoom in and out by holding the right mouse button and moving the mouse up and down.

A slider is used to set the current time step, and the user can also select from the different interpolation techniques described above.

6.2.2 Object Movie Renderer

The object movie renderer is implemented as a `Java` applet, which is hosted on a website the user can navigate to using a regular web browser. The applet is embedded in an HTML document that also includes the parameters needed by the applet.

Downloading Image Data

After reading the parameters given to it by the HTML file, the applet will start downloading images for the available time steps. These images are hosted through a regular HTTP server, as is the applet itself. Time steps that have been downloaded are available for viewing immediately.

Internal Representation

The strips of JPEG images are downloaded and stored internally as a two-dimensional array of pixel data for each image in the set.

Rendering

As described in section 4.3.2, rendering is done by selecting the image out of the set whose parameters most closely match the current viewing state. Which image is to be rendered is determined by comparing the current zenith and azimuth angles to the angles used when generating the images.

The image is rendered at a size that corresponds to the current zoom level. The closer the zoom level, the more the image is stretched and thus the more of the image will fall outside of the visible viewing area.

Figure 6.5 is a snapshot of a browser with the applet loaded.

User Interaction

The user can view the data by manipulating the current camera angles with the mouse, in typical trackball fashion. Zooming can be done either by using the mouse wheel, if one is present, or with keyboard controls otherwise. Holding the right mouse button down and moving across the viewing area horizontally will move through the available

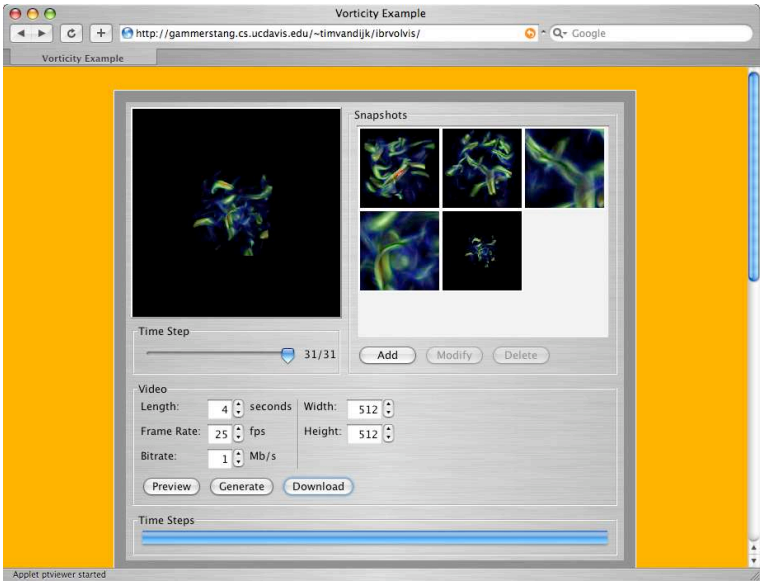


Figure 6.5: A screenshot of the viewer applet.

time steps of the data. Alternatively, a slider control can be used for more precise control.

There is a button in the applet window that allows the user to add the current view (the combination of camera parameters) to a list of "keyframes" that should appear in the video that is to be generated by the server. This list is displayed on the right side, where each keyframe is represented by a thumbnail preview. There are also buttons for editing the selected keyframe and for deleting it, and the user may use the mouse to drag keyframes to a different position in the list.

Textual annotations can be specified for any of the keyframes. These can be used by the user to provide some information about what is currently visible in the sequence, to point out an interesting feature, for example. These annotations will be included in the video in the form of a subtitle track.

Finally, the user can specify parameters that are used in the generation of the video by the server. These include the duration of the video, the frame rate, the bit rate, and dimensions of the video.

6.3 Video Generation

Once the user is happy with the indicated collection of views, he can click a button, and the applet will send descriptions of these views, in `XML` format, along with any textual annotations, and video preferences such as frame rate, bit rate, resolution, etc, to the visualization server, which will then use the same modified volume renderer used for image generation (section 6.1) to render high resolution views of the original

volume data, based on a linear interpolation of the parameters (camera angles, zoom level) that define the views that were indicated by the user on the client side. Time steps are also interpolated. The closest discrete time step number is determined for the fractional time offset associated with each frame of the video to be generated. The data set corresponding to that time step is then loaded by the volume renderer if it hasn't been already, and used for the image generation.

These images will then be encoded into a video and made available for download by the user. To encode the video, the system uses `FFmpeg`[3], a command line interface to a range of libraries that together are capable of encoding video using any of a large number of different codecs. Among these is the modern `H.264` (also known as `MPEG-4 AVC`) standard, which offers high video quality at substantially lower bit rates than older standards.

The resulting video track is embedded in a Matroska[4] container file. This is a very modern and flexible container format, allowing any number of video and audio tracks, as well as tracks for subtitles, which is used by the system to conveniently store the user annotations, after converting them to a recognised subtitle stream format.

This video file is then made available for download to the user, through the applet. Figure 6.6 shows an example frame from a video generated by the system.

6.4 Transfer Function Manipulation

6.4.1 Introduction

As mentioned in section 3.4, we would like to give the user the ability to manipulate the transfer function in some limited way on the client side. A single fixed transfer function might not show all features and the user might want to experiment with different transfer functions to try and gain a better insight into the data. The result of this manipulation can then also be used in the later video generation step. This section describes the method used in the system to achieve this functionality.

6.4.2 Sampling Method

The idea is to subdivide the value range of the data set into subranges, six in the case of the running example used in this section, and define a Gaussian bell-shaped transfer function for each subrange. This way, each subrange will have a natural representation, with the biggest contribution formed by the value at the centre of each range, and the images for one subrange will gradually transition into images for the next.

The general Gaussian function is given in equation 6.1. It has three parameters: a, b, and c. Parameter a is the height of the Gaussian peak, b is the position of the center of the peak and c is related to the *full width at half maximum* of the peak according to $FWHM = 2\sqrt{2\ln(2)}c$.

[3] http://ffmpeg.mplayerhq.hu
[4] http://www.matroska.org

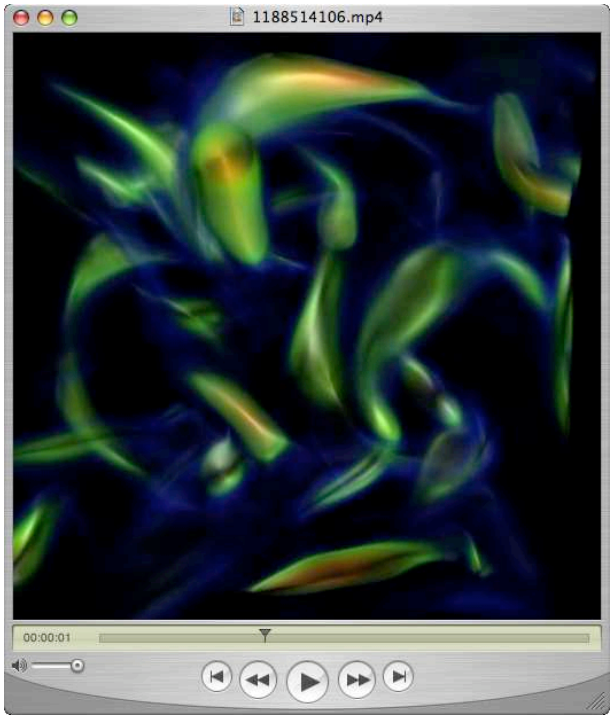


Figure 6.6: A frame of an example video, created using the viewer applet, of the vorticity data set.

$$\texttt{Gaussian}(a, b, c, x) = a \exp\left(-\frac{(x-b)^2}{2c^2}\right). \tag{6.1}$$

In our case, for a we can use a value of 1, so the peak of the function corresponds to an opacity of 1, fully opaque. For parameter b, we can use a set of different values that correspond to the centre of each bell curve, according to the number of curves we are using. For c, we need to choose a value that gives a good representation for each curve. The system uses a value such that curve $n + 1$'s left side approaches zero at the centre of curve n. See the curves in figure 6.7.

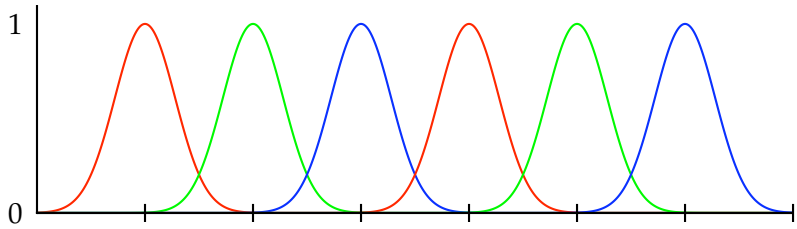


Figure 6.7: Gaussian bell curves used in the example.

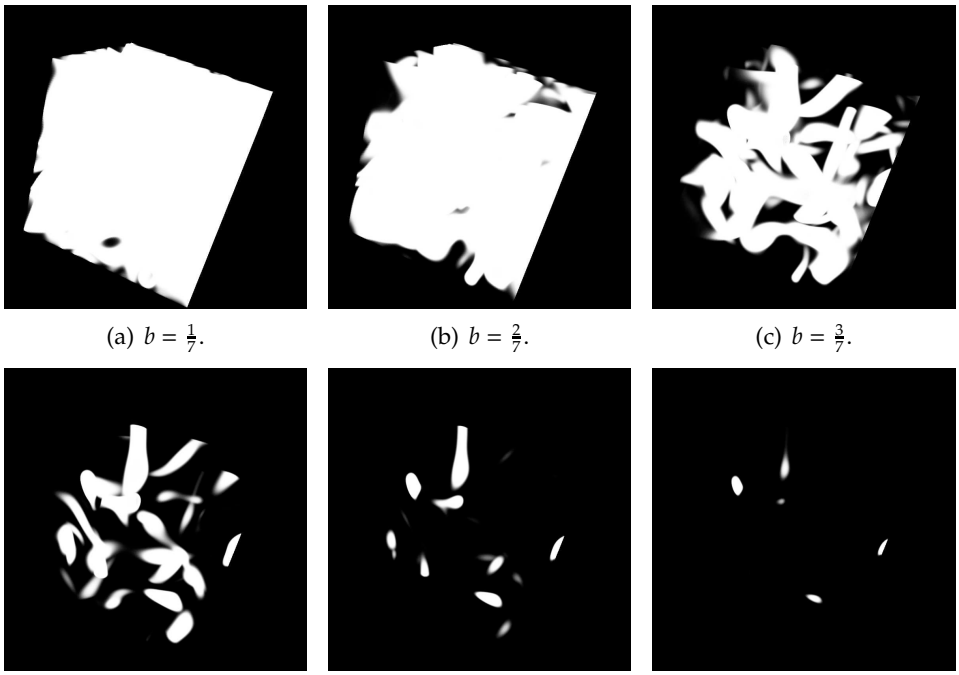
6.4.3 Data Generation

For each of the bell curves, an image is generated by the server, that corresponds to the transfer function given by that curve, such that this function only defines an opacity value. In other words, the transfer function only maps from a data value to an opacity value, instead of a colour with associated opacity value. We do not yet create any kind of mapping from data values to colours. We output a greyscale image for each curve, whose pixels indicate the accumulated opacity at that point. Black corresponds to an opacity of 0, white to an opacity of 1. Figure 6.8 shows the opacity maps obtained for the six curves used in this example.

The six greyscale images can be packed into two `RGB` colour `JPEG` images, such that each of the three channels in each image (six channels in total) stores one of these six maps. An example of this is given in figure 6.9. This means that in the case of six curves, the required storage space only doubles in comparison to using a single fixed transfer function, and in the case of three curves, no extra space would be needed at all.

6.4.4 Client Side Rendering

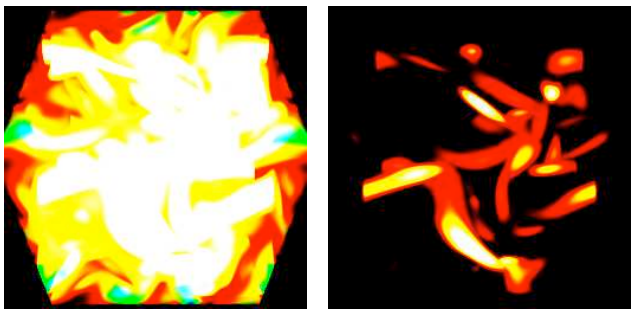
On the client side, rendering can be done as usual, except now, instead of directly taking colour pixels from the input images, the system interprets the values in each channel in each input pixel (two input pixels in the case of six curves) as blending weights to be applied to a colour that is selected by the user. In other words, the output colour $\vec{C_{out}}$ for a pixel at position (x, y) becomes



(a) $b = \frac{1}{7}$. (b) $b = \frac{2}{7}$. (c) $b = \frac{3}{7}$.

(d) $b = \frac{4}{7}$. (e) $b = \frac{5}{7}$. (f) $b = \frac{6}{7}$.

Figure 6.8: Example opacity maps for the vorticity data set. $a = 1$ and $c = 0.04$ for all images.



(a) $b \in \{\frac{1}{7}, \frac{2}{7}, \frac{3}{7}\}$. (b) $b \in \{\frac{4}{7}, \frac{5}{7}, \frac{6}{7}\}$.

Figure 6.9: The example opacity maps packed into two colour images.

$$\vec{C_{out}}(x, y) = \sum_{i=0}^{N-1} I_i(x, y) * \frac{\alpha_i}{\alpha_T} * \vec{C_i}$$

with

$$\alpha_T = \sum_{i=0}^{N-1} \alpha_i,$$

where N is the number of channels, $I_i(x, y)$ is the intensity value at position (x, y) in channel i, and $\vec{C_i}$ is the colour chosen by the user for channel i. The user may also specify an additional opacity value α_i for each channel, so the user can, for example, hide a subrange of values that is less interesting, or stress the importance of a subrange by increasing this opacity value for that curve.

Figure 6.10 shows a composited image for the running example, using the images in figure 6.8, with colours and opacity values defined for each of the six subranges.

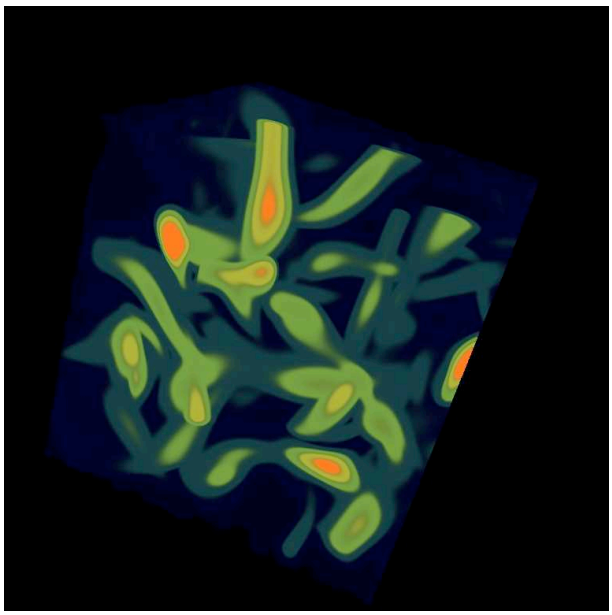


Figure 6.10: Composited image result.

6.4.5 User Interface

To present this functionality to the user, the applet used on the client side includes a simplified transfer function editing widget (see figure 6.5. The parameters $\vec{C_i}$ and α_i for each subrange are represented by a vertical slider control, which controls α_i,

and a button with colour $\vec{C_i}$. The colour for each range can be edited by clicking the corresponding button and selecting a colour in a colour picker dialog that pops up. To change the opacity value, each slider can be dragged vertically with the mouse. The result of each change is displayed immediately, so the user receives instant feedback.

6.4.6 Video Generation

Once the user is content with the transfer function settings for each keyframe in the list, clicking the video generation button will send those settings to the visualisation server, along with the other settings for each key frame.

The transfer function used for generating the images that will be encoded into a video, is constructed from the values of $\vec{C_i}$ and α_i specified by the user for each key frame. The function becomes a weighted sum of the Gaussian curves used for each subrange of the data values, when the image data was initially generated:

$$\alpha_{TF}(x) = \sum_{i=0}^{N-1} \alpha_i * \exp\left(-\frac{\left(x - \frac{i+1}{N+1}\right)^2}{2c^2}\right).$$

The value for Gaussian parameter a is set to 1, while b is set to the appropriate value along the data range, which is the centre of each subrange. An example of such a function is given in figure 6.11, for example values of α_i for $i \in [1, \ldots, 6]$.

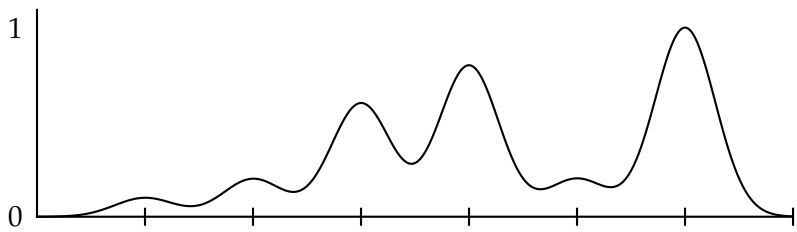


Figure 6.11: Example of transfer function opacity values in the form of a weighted sum of Gaussian curves.

The colour map used is created by creating a colour node at each of the subrange centre points (in this example at $\frac{1}{7}$, $\frac{2}{7}$, etc.) and setting its colour to the corresponding value of $\vec{C_i}$. The rest of the value range is then obtained by linearly interpolating between those colour nodes, just as is normally done in volume rendering applications.

The values for $\vec{C_i}$ and α_i are linearly interpolated between the key frames to obtain values for frames in between, just as the camera parameters are.

The transfer function that is constructed using this approach leads to a result that approaches the user's expectations closely, in terms of which value ranges are visible during the different stages of the video.

6.4.7 Limitations

Unfortunately, depth values for each pixel are not retained in the image data, so it's not possible to correctly blend the various subranges together (higher values are not necessarily in front of lower ones, or vice versa) in the applet. This can lead to misleading results, as can be seen in figure 6.10. However, this is alleviated through spatial cues during spatial exploration by the user, and the added value of being able to manipulate the transfer function on the client side is still useful, even with this shortcoming.

Chapter 7

Conclusions

7.1 Achieved Goals

As stated in section 1.3, our aim was to develop a system that allows its users to interactively explore large scale data sets on commodity hardware, without high end graphics capabilities. Our system achieves this by employing a rendering technique with an execution time and memory complexity that is independent of the size of the original data set.

Through a novel method of image based, user-changeable transfer functions, the user can still identify important features of interest in the data.

An intermediate data representation is used, which may be many times smaller than the original data set, small enough to be downloaded over an internet connection. After this download, the network connection with the server is no longer necessary for exploration of the data. It is only needed when a video of the data needs to be generated.

High quality videos can be generated through a novel interface in the system, without the need for video technology expertise, using a straightforward interface. These videos can then be downloaded and reviewed by the user, or demonstrated to others, using common video player software available on any system.

We have also implemented a novel light field rendering method that takes advantage of the high parallelism of current programmable graphics hardware technology.

7.2 Future Work

There are a number of ways the system could be extended or modified to increase its usefulness or improve the existing functionality.

7.2.1 Light Field Compression

As mentioned in section 4.4, the light field rendering approach is currently unfeasible for this system, mainly because the compressed form of the required amount of image data remains very large, especially for time-varying data sets, and finding a good codebook

for the quantisation process is a very slow and difficult process, which is required for each individual data set.

However, it might be possible to improve the compression used for this method, such that using this approach becomes feasible. It would be especially important to exploit the coherency in the temporal dimension in the case of time-varying data sets, in addition to the coherency in the four spatial dimensions.

7.2.2 Video Generation Options

In its current state, the system uses a single type of linear interpolation between key frames, for generating the video on the server side. This could be extended to include more options, so the user can achieve some more interesting effects. Examples include giving the user the ability to specify that a specific key frame should remain static for a certain amount of time (i.e., a pause in the video), that a selected sequence of frames should be repeated a give number of times, or that certain transitions should be faster or slower than others. Another option is to be able to specify the type of interpolation between frames, so that a non-linear interpolation can be chosen.

Note that all of these effects can already be achieved now, simply by manually inserting extra frames. For example, a pause in the video can be achieved by copying a key frame a certain number of times. However, these extra options would make the specification of parameters for the video much easier, faster, and more intuitive.

7.2.3 Improved Sampling

Currently, the sampling method used for object movie rendering, as described in section 4.3.1, is somewhat wasteful. The sampling density along circles that are closer to the poles of the sphere around the object is the same as it is closer to its equator. Since the surface area of each sample is always the same, there is more overlap closer to the poles, which means that pixels are wasted.

If fewer samples were made near the poles, or a different sampling scheme were used, the resulting data could be made smaller, or the sampling overall density could be increased with no extra storage cost, with obvious benefits.

7.2.4 Transfer Function Editing

The current implementation of client side transfer function editing has some limitations in regards to depth values (see section 6.4.7). This cold be solved or alleviated using a more sophisticated sampling method that also stores depth values for each value subrange in some efficient way.

In the current implementation, only mono-variate (one variable per voxel) scalar data sets can be used for the client side transfer function editing scheme. Multiple variables could be supported by storing separate image data for each variable, and then allowing the user to blend these together on the client side. Non-scalar data types (e.g.

vectors and tensors) could be supported too, as long as there is some way of mapping the values to image data.

Finally, more work is needed to evaluate the performance of other types of server side transfer functions used for encoding the subranges of data to a series of images, other than the current Gaussian bell curves method.

List of Figures

Bibliography

[AB91] Edward H. Adelson and James R. Bergen. The plenoptic function and the elements of early vision. In Michael S. Landy and J. Anthony Movshon, editors, *Computational Models of Visual Processing*, pages 3–20, Cambridge, 1991. MIT Press.

[BCY] Wes Bethel, Jerry Chen, and Ilmi Yoon. Interactive, internet delivery of visualization via structured, prerendered multiresolution imagery. *IEEE Transactions on Visualization and Computer Graphics : Accepted for future publication*.

[Che95] Shenchang Eric Chen. QuickTime VR: an image-based approach to virtual environment navigation. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 29–38, New York, NY, USA, 1995. ACM Press.

[COZ98] Daniel Cohen-Or and Eyal Zadicario. Visibility streaming for network-based walkthroughs. In *Graphics Interface*, pages 1–7, 1998.

[DCH88] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 65–74, New York, NY, USA, 1988. ACM Press.

[DH92] John Danskin and Pat Hanrahan. Fast algorithms for volume ray tracing. In *VVS '92: Proceedings of the 1992 workshop on Volume visualization*, pages 91–98, New York, NY, USA, 1992. ACM.

[GG91] Allen Gersho and Robert M. Gray. *Vector quantization and signal compression*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[HHK+03] H. Hege, A. Hutanu, R. Kähler, A. Merzky, T. Radke, E. Seidel, and B. Ullmer. Progressive retrieval and hierarchical visualization of large remote data. In *Proceedings of the Workshop on Adaptive Grid Middleware*, September 2003.

[KKHV02] Dieter Kranzlmüller, Gerhard Kurka, Paul Heinzlreiter, and Jens Volkert. Optimizations in the grid visualization kernel. In *IPDPS '02: Proceedings*

of the 16th International Parallel and Distributed Processing Symposium, page 237, Washington, DC, USA, 2002. IEEE Computer Society.

[KW03] Jens Krüger and Rüdiger Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings IEEE Visualization 2003*, 2003.

[LH96] Marc Levoy and Pat Hanrahan. Light field rendering. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 31–42, New York, NY, USA, 1996. ACM Press.

[MB95] Leonard McMillan and Gary Bishop. Plenoptic modeling: an image-based rendering system. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 39–46, New York, NY, USA, 1995. ACM Press.

[MHB+00] Michael Meißner, Jian Huang, Dirk Bartz, Klaus Mueller, and Roger Crawfis. A practical evaluation of popular volume rendering algorithms. In *VVS '00: Proceedings of the 2000 IEEE symposium on Volume visualization*, pages 81–90, New York, NY, USA, 2000. ACM.

[PF01] Valerio Pascucci and Randall J. Frank. Global static indexing for real-time exploration of very large regular grids. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 2–2, New York, NY, USA, 2001. ACM Press.

[RSFWH98] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[RWIB+07] F. Rößler, T. Wolff, S. Iserhardt-Bauer, B. Tomandl, P. Hastreiter, and T. Ertl. Distributed Video Generation on a GPU-Cluster for the Web-Based Analysis of Medical Image Data. In *Proceedings of SPIE Medical Imaging 2007: Visualization and Image-Guided Procedures*, pages 650903 1–9, 2007.

[SK00] Heung-Yeung Shum and Sing Bing Kang. Review of image-based rendering techniques. In King N. Ngan, Thomas Sikora, and Ming-Ting Sun, editors, *Proc. SPIE Vol. 4067, p. 2-13, Visual Communications and Image Processing 2000, King N. Ngan; Thomas Sikora; Ming-Ting Sun; Eds.*, volume 4067 of *Presented at the Society of Photo-Optical Instrumentation Engineers (SPIE) Conference*, pages 2–13, May 2000.

[SS97] Richard Szeliski and Heung-Yeung Shum. Creating full view panoramic image mosaics and environment maps. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 251–258, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.