Real Time Rendering of Animated Volumetric Data

by

Luis Valverde, B.Sc. in Computer Science

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

September 2010

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Luis Valverde

September 13, 2010

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Luis Valverde

September 13, 2010

Acknowledgments

First of all I would like to thank my parents because without them nothing I have ever done would have been possible. Special thanks should go as well to my dearest flatmates Steph, Nohema, Fan, Lolo and Sergio, for keeping me healthily alive during all this year; to John Dingliana, for without his supervision and advice this work would not be what it is; to all my IET classmates, for making the whole year such an enjoyable experience, with a special mention to Jorge, Rick and Gianluca; and finally to my friend María Angeles, who I hold personally responsible for my decision of taking this course.

LUIS VALVERDE

University of Dublin, Trinity College September 2010

Real Time Rendering of Animated Volumetric Data

Luis Valverde University of Dublin, Trinity College, 2010

Supervisor: John Dingliana

Animated volumetric data can be found in fields like medical imaging -produced by 4D imaging techniques such as ultrasound-, scientific simulation -for example, fluid simulation- or cinematic special effects -for reproducing volumetric phenomena like fire or water. Real-time rendering of this data is challenging because due to its large size, in the order of gigabytes per second of animation, it requires on-the-fly streaming from external storage to GPU memory (called out-of-core rendering) causing bandwidth between memory subsystems become the bottleneck.

This dissertation work describes the design and implementation of an out-of-core rendering system for animated volumes. A two-stage compression system is used to reduce bandwidth requirements based on a fast lossless compression method in the CPU (LZO) and a hardware supported lossy method in the GPU (PVTC) following previous research [1, 2]. This provides an average increase in FPS of 290% relative to rendering without compression. The system is critically evaluated and compared with a novel GPU compression scheme developed to improve image quality (E-PVTC). Additionally, an assessment of the applicability of these techniques to interactive entertainment and the medical and scientific fields is performed.

Contents

Acknow	wledgments	iv
Abstra	\mathbf{ct}	v
List of	Tables	ix
List of	Figures	x
Chapte	er 1 Introduction	1
1.1	Concepts	2
1.2	Challenges	3
1.3	Goals	4
1.4	Solution	5
1.5	Results	6
Chapte	er 2 Related Work	7
Chapte	er 3 Volume Renderer	9
3.1	Design	9
3.2	Implementation	12
Chapte	er 4 Load System	15
4.1	Design	15
	4.1.1 Extended PVTC	18
4.2	Implementation	20
	4.2.1 CPU Compression	20

4.2.2 GPU Compression	21	
4.2.3 Preprocessing \ldots	25	
4.2.4 Animation Pipeline	27	
4.2.5 Extended PVTC	28	
Chapter 5 Evaluation	31	
5.1 Volume Renderer	33	
5.2 Load System	37	
5.2.1 Rendering Performance	37	
5.2.2 Image Quality \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots	40	
5.3 Assessment of Applicability	43	
Chapter 6 Conclusions and Future Work	46	
Appendices	48	
Bibliography		

List of Tables

4.1	S3TC colour interpolation codes	22
5.1	Datasets used for experiments	32
5.2	Animation performance test parameters	38
1	Acronyms commonly used in the text	49

List of Figures

1.1	Comparison of two and three dimensional data	2
1.2	2D vs 3D data	4
3.1	Textured-based volume rendering	10
4.1	Subsytems and stages in the animated volume rendering system	15
4.2	Load system pipeline	19
4.3	Alpha channel compression with DXTC	23
4.4	Formation of compression blocks in S3TC and VTC	24
4.5	Preprocessing steps	26
5.1	Screen resolution and cut-planes scale impact on rendering performance	34
5.2	Impact of the number of cut-planes on rendering performance \ldots .	35
5.3	Impact of the volume resolution on rendering performance	35
5.4	Impact of the volume compression format on rendering performance	36
5.5	Impact of the number of cut-planes on the image quality	37
5.6	Rendering performance in average frames per second	39
5.7	LZO impact on performance	40
5.8	Avg. stage execution time per time-step	41
5.9	Image quality and compression format	42
5.10	Sensitive transfer function	43
5.11	Image quality with time step	44

Chapter 1

Introduction

This dissertation report is organised in the following way:

In the introduction chapter a general introduction to the most important concepts related with the topic addressed in this work will be given, followed by a description of the main challenges the field poses, the goals they motivated and a brief overview of the solution adopted and the results obtained.

Next, a brief revision of relevant research in the field of real time rendering of time-varying volumetric data will be presented in the chapter about related work. Following it, two chapters for the two subsystems, the volume renderer and the load system, will give the design and implementation details for each one. Design sections will focus on describing the functionality covered by the subsystem and the different options considered, and explaining the decisions made. Implementation sections will give a detailed description of how the system was constructed and of some possible improvements.

The evaluation chapter will present and explain the results obtained and, based on this, give an assessment on the applicability of the technology to different fields. Finally, the conclusions and future work chapters will summarise the main findings of this work and suggest directions for future related research.

1.1 Concepts

Volumetric data is the natural consequence of adding a dimension to flat, two-dimensional data. A data volume is a finite collection of scalar or vectorial values, called *voxels*, distributed in three-dimensional space. Although theoretically voxels can be spatially arranged in any way, in practice mainly regular grid distributions are used because they are easier to store and access. If we imagined an instance of two-dimensional data as a digitalised picture, where each square pixel represents a value, an instance of volume data would be formed stacking several of that pictures on top of each other. The result is usually visualized as a rectangular box divided in equally sized cubical voxels that contain the values of the pixels of the stacked pictures. Figure 1.1 illustrates this concept. An example of practical use of volumetric data is a medical imaging technique called Magnetic Resonance Imaging (MRI) where magnetic fields are used to create three dimensional maps of the different tissues in the human body.



Figure 1.1: Comparison of two and three dimensional data

In the same way that a series of pictures of the same size is used in a film to produce an animation, a collection of volumes with the same dimensions is used to represent the variation through time of the data they contain. We refer to this as *animated volumetric data*. This kind of data is used extensively in the medical field, for medical imaging techniques like 4D ultrasound, MRI and Computed Tomography (CT) that make possible capturing 3D images of body organs in movement; the scientific field, for simulations such as computational fluid dynamics that reproduce the evolution of a fluid; and the entertainment industry, for visual simulation of animated volumetric effects like fire, smoke or water. Rendering a data volume usually implies presenting the information it contains in a two-dimensional display in such a way that it is useful for the viewer. With two-dimensional data this is often as straightforward as printing a picture in a piece of paper, where no especial care has to be taken to present the information in a meaningful way because the destination medium allows doing it naturally. When working with volumetric data, though, techniques have to be applied to map the extra dimension to a two dimensional target. Recovering the picture stack analogy, the question of what should be seen if we look at it from the top arises. Should it be just the picture on the top, the first foreground voxel we encounter for each pixel, or maybe a mix of the colours of all the aligned voxels? The answer will depend on the *volume rendering* technique used.

Volume rendering must also deal with the fact that a voxel value can represent a wide range of concepts from the density of tissue in a human body, to the velocity of the air flowing through a jet engine or the temperature of the gas in a supernova explosion. This values need to be translated to colours before being presented on the screen, e.g. white for bone, red for flesh. The function that maps data values to colours is called the *transfer function*.

Finally, the last essential concept regarding the topic of this work is the idea of *real time*. Volume animations can be rendered and saved as a sequence of two dimensional still images in the form a video. The main disadvantage of this is that it does not allow interactive exploration of the volumes, i.e. changing the point of view, which can be essential to appreciate hidden details or to integrate the animation in a 3d environment. Real time rendering means that each volume is rendered as the animation is being played thus enabling the user to modify interactively the point of view or other parameters that affect image generation like the transfer function.

1.2 Challenges

The main challenge when rendering animated volumes in real time is the size of the data. A 512x512x512 volume of 1 byte voxels, a size easily found and exceeded in medical and scientific applications, takes up 128 MB of space. Figure 1.2 gives a visual reference comparing this size with that of a 512x512 flat image. A 5 second animation at 30 frames per second (FPS) requires 150 volumes, totalling 18.25 GB. Real time volume

rendering techniques commonly require that the volume information is stored in the graphics card memory (VRAM) to produce the rendered image, but current memories have sizes around 512-1024MB, which is insufficient to hold the whole dataset. The first effect of the large data size is then that volumes have to be transferred from the CPU to VRAM continuously while the animation is played, linking rendering performance not only to the GPU rendering speed but as well to the CPU-GPU bandwidth.



Figure 1.2: Graphic representation of the memory required for a 512x512 2D image and for a 512x512x512 3D volume

If the complete dataset cannot be hold in VRAM then it will have to be stored in the CPU's main memory (RAM), but most desktop systems have memories around or below 4GB which, again, is not enough for the whole animation. Consequently, volumes must be read from external storage (usually the hard drive) by the CPU and then transferred to the GPU as the animation is playing, in a process known as *out-ofcore rendering*. This links rendering performance as well to the external storage-CPU bandwidth. Therefore, if real time is to be achieved, especial mechanisms are needed to reduce as much as possible the time taken to transfer a volume from external storage to GPU (graphics card) memory.

1.3 Goals

Stemming from these challenges, the goals of this project were defined as follows:

- 1. Design and implement a real time animated volume rendering system that allows interactive exploration of large animated datasets using state-of-the art acceleration techniques.
- 2. Perform a critical evaluation of the existing techniques used, namely two-stage compression with PVTC [1, 2], and the new ones implemented (new compression scheme E-PVTC).
- 3. Asses the applicability of the system developed to interactive entertainment and the medical and scientific fields.

1.4 Solution

The solution design and developed to achieve these goals is divided in two subsystems: the volume rendering subsystem and the load subsystem. The first of them is in charge of rendering the volume information once it has been transfer to GPU memory, regardless of its animated or static nature. Volume rendering is performed using a texture-based technique based on [3] that creates a set of view aligned rectangular slices that virtually intersect the volume. Textures coordinates for each slice are calculated dynamically according to the position and orientation of the volume relative to the view point. The coordinates are then used to sample the volume texture and the data value obtained to index the transfer function texture and obtain the final fragment colour. Resulting overlapping fragments are composed in back-to-front order using alpha blending.

The load subsystem is responsible for streaming the volume information from external storage to VRAM. The solution implemented follows the two-stage approach of [1], where doubly compressed volumes are decompressed on-the-fly first in the CPU and then in the GPU. As in the original research, LZO is used as CPU compression/decompression method and Packed Volume Texture Compression (PVTC) for GPU compression/decompression. PVTC makes use of Volume Texture Compression (VTC), a volume texture compression format supported by hardware components in the GPU, to pack the information of three consecutive time steps in the colour channels of a single RGB volume texture. This provides a fixed compression ratio of 6:1 at the cost of image quality degradation. LZO is a general purpose lossless compression format specially designed for real time decompression.

A new volume compression scheme, Extended PVTC (E-PVTC), was designed and developed to improve the image quality achieved with PVTC while retaining the advantages of using GPU supported compression (VTC). This new method works using a different variant of VTC that allows using RGBA volume textures. The alpha channel is employed to encode a Compression Correction Code calculated per voxel that is applied at run time to reduce the errors produced by VTC lossy colour compression.

Load from external storage is performed in parallel to the execution of the main program reducing in this way the total execution time per frame to the maximum of both processes. Pipelining of the CPU decompression stage explained in [2] could not be implemented due to time restrictions.

Finally, a series of preprocessing tools were developed to transform the different test datasets to a common raw format appropriate for use with PVTC and to produce the compressed volume files in PVTC and E-PVTC formats.

1.5 Results

The solution implemented allowed reproduction of previous research results for outof-core rendering with two-stage compression, obtaining over 19 fps for 256x256x256 volume animations with 98 time steps and over 60 fps for 128x128x128 volume animations with 150 time steps. The average improvement in fps relative to rendering without compression was of 290%. E-PVTC proved capable of reducing significantly compression artefacts at the cost of increased compressed volume size and reduced rendering speed relative to PVTC (twice the volume size and 28% slower on average).

Chapter 2

Related Work

Interactive visualization of volume data from time-varying simulations tends to be large in space (number of voxels) and time (number of time steps) and even number of variables. Hardware constrains like limited GPU memory or CPU-GPU bandwith have led to the development of a series of strategies to enable the work with these enormous datasets. The main issues addressed are data encoding, feature extraction and rendering [4]. Data encoding techniques aim to reduce the amount of data necessary for the volume visualization, which provides two advantages. First, it reduces the amount of memory required allowing to fit in texture memory data that otherwise wouldn't have. Second, it reduces the data transference between secondary storage and main memory and between main memory and the GPU.

There are two basic approaches for compressing time-varying volume data. The first one is to separate the time dimension from the spatial dimension. The second is to treat the data as 4D data. The election between one and the other will depended on the characteristics of the data. For example, if the temporal and spatial resolutions differ greatly it would be desirable to keep them separated. Examples of different techniques are time-space partitioning trees (TSP) [5] and wavelet encoding [6].

Regarding feature extraction, automating it using domain knowledge can greatly reduce storage requirements and rendering cost. However this approach has not been generally adopted because it requires understanding the features well enough to make the automation extraction effective. Some successful efforts can be found in [7] and [8].

Efforts in rendering have been focused in automated transfer function generation

and using parallel distributed computing. The former tries to generate automatically transfer functions for features of interest in a time series that exhibit regular, periodic or random patterns [9]. The objective is again to reduce the amount of data to allow for interactive visualization. Parallel distributed computing aims to distribute the data and rendering calculations over a computer network, like in the research [10].

Remaining research challenges in the field are dealing with irregular grids of voxels and multivariable datasets.

Chapter 3

Volume Renderer

3.1 Design

The volume renderer is the subsystem of the animated volume rendering system responsible for drawing the volume information on screen. Its main inputs are a 3D texture representing the volume data and a 1D texture containing the transfer function, both provided by the load subsystem. Its output is the rendered image in the frame buffer.

As volumetric information is not suitable for rendering with traditional polygonbased solutions, specific techniques have been developed for volume rendering (VR). There are two common approaches: textured-based and ray casting VR. Texturedbased volume rendering uses a set of view aligned cut-planes or slices that virtually intersect the rendered volume. The texture coordinates for each slice are calculated dynamically according to the orientation and position of the volume relative to the view point. A 3D texture containing the volume data is sampled using the generated texture coordinates to determine the colour and transparency of the fragments contained in each cut-plane. Finally, alpha blending is used to compose the cut-planes into the frame buffer. Figure 3.1 illustrates this process.

This technique has two main advantages. First, it makes an efficient use of current GPU's acceleration capabilities through alpha blending composition. Second, the use of proxy geometry allows easy integration with other polygonal primitives rendered with traditional techniques. The main disadvantage is the strong dependence of image quality on the number of cut-planes and the impact of this in performance. Each



Figure 3.1: Texture-based volume rendering with view aligned slices. Taken from [11]

additional cut-plane produces as many new fragments as screen pixels, what can quickly make the rasterization stage become the bottleneck in the graphics pipeline.

The second common approach to volume rendering is ray casting [12]. In this technique a virtual ray is created for each screen pixel starting at the eye point. The volume is then sampled at discrete positions along each ray and the data values obtained mapped to colour values with the transfer function and accumulated. The final accumulated values determine the colour for each pixel. This is similar to the way cut-planes sample the volume and are composed with alpha blending in texture-based methods, but there are at least to important differences. First, colour composition in ray casting is performed in front-to-back order, while in texture-based volume rendering it is performed back-to-front. Front-to-back composition allows to implement an optimization know as *early ray termination* that stops ray traversal when the accumulated colour has reached complete opacity, thus reducing the number of samples for that ray. The second difference is the fact that the sampling step size can be different for each ray while in texture-based methods it is constant and given by the number and position of cut-planes. Thanks to this in ray casting it is possible to adjust the step size of a ray to totally skip empty volume regions or quickly traverse uniform regions, a technique called *empty space skipping*, while in textured-based rendering a fragment is generated for each pixel and cut-plane, regardless of uniformity or transparency. These two differences can be summarised saying that ray casting allows making a much more efficient use of volume samples than texture-based VR. The importance of this difference is stressed by the fact that, according to [3], only between 0.2% and 4% of all fragments generated with texture-based VR for typical datasets contribute to the final

image.

If volume ray casting is so much efficient sampling volumes than texture-based techniques, why not use it then? Ray traversal is usually performed in a fragment shader program that is executed for each screen pixel, making use of the massive parallel computing capabilities of modern GPUs. The main problem with this technique is that conditional statements are needed to control ray traversal and these are not well supported by the GPU streaming architecture, producing either heavy execution time penalization in the GPU or the need to offload operations to the CPU. Another disadvantage of ray casting is the difficulty to integrate ray casted volumes in scenes produced with traditional polygon-based rendering techniques.

After considering these pros and cons, a texture-based volume rendering technique based on [13] was chosen for this work. Faster rendering speed to achieve a smoother navigation through the animated volumes was considered more important than image quality for the purposes of previsualization of large medical and scientific animated datasets and the reproduction of small volumetric effects in interactive applications. If better image quality were needed the system could be modified to produce still renderings of the selected time-step using ray casting without impacting overall animation performance. Easier integration with polygon-based rendering was as well a decisive factor from the point of view of the possible uses in interactive entertainment, where a stand-alone volumetric animation has little practical uses. Finally, using texturedbased VR would facilitate the critical evaluation of the previous research on volume animation this work is based on [1, 2], as the results they present were produced using the same technique.

A number of improvements on the basic texture-based rendering approach proposed in [11] were considered but left out due to time restrictions or the possible impact on rendering speed and memory consumption. The principal of them is opacity correction, a technique aimed to correct the opacity variations produced when the number of cut-planes is changed. This is achieved computing on-the-fly a corrected alpha value for each fragment as a function of the current number of slices and the number the transfer function was designed for. A detailed explanation of this issue can be found in the volume renderer evaluation section 5.1. Dynamic proxy geometry is other of the improvements left out. It is an acceleration technique that reduces the number of fragments generated calculating the intersection of the cut-planes with the data volume. The resulting view aligned polygons are used for rendering instead of fullscreen quads. More complex illumination models like Blinn-Phong or Volume Lightning were excluded because they require an additional volume texture for normals and can have an important impact in performance.

3.2 Implementation

The texture-based volume renderer subsystem was implemented in C++ using OpenGL as the graphics API and The OpenGL Utility Toolkit (GLUT) to manage the application window. Fragment and vertex shaders were created with Cg shader language and handled from the OpenGL application through NVIDIA's Cg run time API [14].

Before the application can start rendering volumes a series of setup steps need to be performed. First, the cut-planes are created as a set of view aligned quads evenly spaced along the Z axis in eye space. The number of cut-planes can be either configured by the user or calculated automatically as the longest diagonal of the volume, to ensure that there is at least a quad per voxel regardless of the volume orientation. An OpenGL display list is created to encapsulate the calls necessary to draw all the quads in backto-front order.

Next alpha blending is enabled and set to back-to-front with a call to the OpenGL function glBlendFunc with parameters GL_SRC_ALPHA and GL_ONE_MINUS_SRC_ALPHA. In this blending mode, if C_s and A_s are the source colour and alpha -i.e. the colour and alpha of the fragment being processed- and C_d is the destination colour -i.e. the colour for the corresponding pixel already present in the framebuffer- the final blended colour C_f is computed this way:

$$C_f = A_s C_s + (1 - A_s) C_d (3.1)$$

Following, textures are created for the transfer function (1D) and the volume data (3D) with linear interpolation modes. The transfer function texture is loaded as well in this stage because it will not change during rendering. Volume texture loading will be performed by the loading subsystem (see 4.1).

The last setup step is initializing the Cg components. First, a Cg context is created with the cgCreateContext function to allow invocation of the Cg run time. Then vertex and fragment shaders are loaded and some of their parameters are initialized (mainly texture references).

Once the system has been initialized the volume rendering system performs the following steps each frame: First, the CPU sets variable shader parameters like zoom and transformation matrices and issues commands to the GPU for drawing the cutplane quads with a call to glCallList. Next, the vertex shader computes the cutplanes vertices clip position and texture coordinates using the transformation matrices. Finally, the fragment shader fetches the data values for fragments from the 3D texture using the interpolated texture coordinates of the vertices of the triangle they belong to. The colour channel read depends on the volume texture format (see 4.2.2). That recovered data value is used to perform a texture fetch in the transfer function texture and obtain the final fragment colour. When the volume texture is encoded with E-PVTC an extra step must be performed before using the recovered value to index the transfer texture. This is detailed in 4.2.5. Example Cg code for the basic vertex and fragment shaders is included below:

```
{
    // Volume texture lookup
    float dataValue = tex3D(volumeTexture, textureCoordinates).r;
    // Transfer function lookup
    finalColour = tex1D(transferFucntionTexture, dataValue);
}
```

Chapter 4

Load System

4.1 Design

The load subsystem is the component of the animated volume rendering system responsible for transferring time-varying volume data from an external storage device (e.g. HDD or DVD) through system memory (RAM) to video memory (VRAM). Once there, the volume texture data will be read by the volume rendering subsystem to produce the final volume image. Figure 4.1 shows this whole process as a sequence of 3 stages: data load (from external storage to RAM), texture transfer (from RAM to VRAM) and volume rendering. The data load and texture transfer stages are performed by the load system in the CPU while the volume rendering stage is performed by the volume renderer in the GPU.



Figure 4.1: Subsytems and stages in the animated volume rendering system

Given this definition of responsibilities the load subsystem performance can be described with the following parameters:

- **P1** Load time. Time taken to read a time-step from external storage and write it to RAM in the format it will be transferred to the GPU
- **P2** RAM footprint. RAM space used to read a time-step from external storage and transform it to the texture format in which it will be transferred to the GPU.
- P3 Texture transfer time. Time taken to transfer a time-step from RAM to VRAM.
- P4 VRAM footprint. Space used to store a time-step in VRAM in the format it is received from the GPU plus any additional VRAM space that may be required to decode that format.
- P5 Texture fetch penalization. Additional time spent by the GPU to obtain a filtered voxel value from the texture in the format received relative to the time taken for a normal texture fetch.

The load system was designed taking into account the optimisation of these parameters.

The most common approach for loading systems of these characteristics is to use some kind of on-the-fly decompression, either in the CPU or the GPU. GPU decompression presents the disadvantage that, due to GPU streaming architecture limitations, branching instructions penalize performance heavily. This greatly limits the range of decompression algorithms that can be executed in the GPU without a great impact on P4 or P5. CPU decompression on the other hand helps improve P1 but, as raw data is stored in RAM and transferred to VRAM, it does not improve P2, P3 or P4. Therefore a two-stage decompression method using both CPU and GPU seems like the best alternative to optimise all performance parameters.

The two-stage compression method used in this work is the one developed in [1, 2]. Nagayasu et al. name the compressions methods for GPU and CPU as M_g and M_c respectively and define the following coherences to be exploited in time-varying data compression:

Temporal coherence The correlation between voxel values observed at different moments in time.

Spatial coherence The correlation between voxel values at different voxels in space.

The following three requirements are defined to guide the design and evaluation of the compression scheme:

- **R1** Both M_c and M_g should achieve a compression ratio high enough to make the reduction in transference time greater than the decompression time.
- **R2** M_c and M_g should exploit different coherences
- **R3** M_g output should keep the coherence that M_c exploits

The GPU compression method selected is based on VTC (Volume Texture Compression), a 3D texture compression format supported by hardware components in the GPU. VTC compresses blocks of 4x4x1 RGB voxels, encoding them as a pair of 16-bit representative colours and a 2-bit interpolation code for each voxel in the block, giving a fixed compression ratio of 6:1. The data for three consecutive time steps is packed in the three colour channels of a 3D texture and compressed with VTC, in a format called Packed Volume Texture Compression (PVTC). Details about the technique are given in the implementation section. Decompression hardware support helps keep texture fetch penalization low (P5) and VRAM memory usage at a minimum (P4), as it is decompressed on the fly. Smaller texture sizes will help as well to reduce load time, RAM usage and texture transfer time (P1, P2 and P3). Regarding R2, PVTC simultaneously exploits time coherence between every three consecutive time steps and spatial coherence within the 4x4x1 compression block. The high fixed compression ratio and fast decompression speed assures R1 fulfilment for M_g .

For CPU compression the LZO (Lempel-Ziv-Oberhumer) algorithm is used [15]. LZO is a lossless compression method designed for fast decompression taking advantage of large CPU caches. Compression is performed substituting frequent data sequences within a 64KB sliding window for references to a dictionary, exploiting in this way spatial coherence inside the window. This ensures that spatial coherence between 64bit PVTC compressed blocks is exploited (R2 and R3). Fulfilment of R1 for M_c is not assured, as the compression ratio is not fixed, and will depend on the input dataset. This will have an effect as well in P1 and P2.

The two-level compression system introduces a new stage between data load and texture transfer, the decompression stage. An additional processing stage for GPU decompression is not required as it is performed automatically by hardware components when values are fetched from the texture.

The four stages of the process can be pipelined to work in parallel with each other as long as there is enough memory for the buffers connecting each stage. As shown in figure 4.2, in this manner the data load stage would load the first time step -actually the first group of three time steps because PVTC encodes three time steps in a single volume-, store it in a RAM output buffer and proceed to load the next time step. Once the first volume is loaded the decompression stage is notified, reads it from the data load stage output buffer and decompress it to its own output RAM buffer. Following, it notifies the texture transfer stage and proceeds with the next volume in the data load stage output buffer or, if there is none, blocks until it is notified that a new volume has been loaded. After a new decompressed volume has been produced by the decompression stage, the texture transfer stage copies it to VRAM memory, signals its availability to the rendering stage and continues processing the new decompressed volume. Finally, when a GPU compressed texture has been transferred to the GPU memory the volume renderer stage displays the three time steps contained in it sequentially and blocks until the next texture is available.

Nagayasu et al. in [1] pipeline data load and in [2] add pipelining of the decompression stage reporting performance improvements ranging from 33% to 122% relative to the first work. Pipelining of the volume rendering stage is mentioned as a future improvement. In this work only data load is pipelined due to time restrictions. Details will be given in the corresponding implementation section 4.2.4.

4.1.1 Extended PVTC

Early tests of the animated volume rendering system showed that the image quality achieved by PVTC could be an issue. The underlying lossless compression system, VTC, takes 8-bit values and encodes them as a linear interpolation of a pair of representative 5 or 6-bit values -the precision depends on the colour channel- calculated for each 4x4x1 voxel block. There are just four possibilities for the linear interpolation: it can be either just one of the representative values or a weighted sum where one value gets half the weight of the other. As a result of this, the number of different values that can be represented in a block is limited to four and the number of possible values



Figure 4.2: Full pipelining of the two-level compression system. Compressed volumes contain three consecutive timesteps

limited by the reduced precision of the representative colours. When a PVTC encoded volume is rendered to screen these limitations translate into blocky artefacts and loss of smoothness in gradients. Furthermore, as the texture values are not presented in screen themselves but mapped to colour values with a transfer function, the sensitivity of the produced image to compression errors can be greatly increased. More details about this can be found in the section on image quality in the evaluation chapter 5.2.2.

These findings motivated the decision to design an alternative GPU compression method that provided better image quality but still worked well in the current twostage approach. Maintain the use of a VTC-based texture compression format was a priority because it is the only volume texture compression format supported by GPU hardware and it had already proved to work well with LZO compression. This limited the options to the three VTC variants available: DXT1, used in PVTC, which provides compression for RGB textures; DXT3, which provides compression for RGBA textures with uncompressed alpha information; and DXT5, which offers compression for RGBA textures with compressed alpha information. It was important as well to keep the decompression cost low, as it would have to be performed for each fragment in the fragment shader, and avoid the use of additional textures because it would require extra texture transferences and fetches.

Finally an extension to PVTC using the DXT3 variant was designed called Extended PVTC (E-PVTC) that packs the voxel information of three consecutive time steps in the RGB channels, as PVTC does, and adds a Compression Correction Code (CCC) in the alpha channel. This code is calculated per voxel to minimize the error produced by PVTC compression and is applied in the fragment shader after fetching texture values. Details are given in the corresponding implementation section 4.2.5.

It was expected that this new approach would improve image quality because it would allow a wider range of values per block, as a consequence of the CCC being calculated and applied per voxel, and produce images closer to the original. On the down side, the size of the compressed textures doubles, due to the inclusion of uncompressed alpha information in the DXT3 format, and thus affects RAM/VRAM memory foot-print and data/texture transference times. Rendering performance could be directly affected as well because the shader has to decode the Compression Correction Code from the alpha channel and apply it to the voxel value before mapping it with the transfer function.

4.2 Implementation

The load system was implemented in C++ mainly in the form of a statically linked Windows library (.lib). The next sections will give details about the implementation of the main functionalities as well as some relevant technical background information.

4.2.1 CPU Compression

The compression and decompression of volume data in CPU was implemented using the LZO library [15], a portable lossless data compression library written in ANSI C and published under GPL license. It offers very fast decompression, about at a third of the speed of raw memory access for the same data size.

The algorithm scans the input data in blocks, substituting frequent substrings within the block for references to a dictionary of frequently encountered substrings. When dealing with uncompressible data, LZO expands the input block by a maximum of 16 bytes per 1024 of input. The LZO library provides multiple variants of the algorithm that differ in compression and decompression performance and block size. The variant used for the implementation of volume data compression and decompression is LZO1X, which uses a block size of 64KB and is recommended by the algorithm author as the best performing option in most cases. The reduced size of the block allows the CPU to keep it in level 2 cache while decompressing, greatly improving performance.

When LZO1X is used to compress a PVTC encoded volume 8192 consecutive PVTC blocks are read into a single compression block and scanned for frequently repeated data. VTC, and thus PVTC too, compresses voxels in groups of 4x4x1 adjacent blocks producing an output that maintains close together the information of voxels that are close in the original volume (more details in 4.2.2). LZO takes advantage of this preserved spatial coherency in PVTC output using its bigger block size to find and compress patterns in larger contiguous regions of the original volume.

4.2.2 GPU Compression

PVTC, the technique used for GPU compression of animated volumes, relies on Volume Texture Compression (VTC) for the underlying compression. VTC is a 3D texture compression format supported natively in OpenGL through extensions [16] and in DirectX through the DDS (Direct Draw Surface) format[17]. Decompression is implemented in hardware components in current GPUs from the main vendors (NVIDIA and ATI). This allows fast on-the-fly decompression in texture fetches without any changes in shader code. Compression is not usually implemented in hardware but instead delegated to the device driver or performed as a preprocessing step with external applications such as NVIDA's Texture Tools or AMD's Compressonator.

S3TC

VTC is based on S3TC (S3 Texture Compression), also now as DXTC (DirectX Texture Compression), a 2D texture compression format supported in OpenGL through the EXT_texture_compression_s3tc extension [18]. S3TC is a lossless compression method for RGB and RGBA 2D textures with 8 bits per channel. It works in blocks of 4x4 pixels and handles colour (RGB) and alpha information, when present, separately.

Control Code	Encoded Color
0	color0
1	color1
2	(2color0 + color1)/3
3	(color0 + 2color1)/3

Table 4.1: S3TC colour interpolation codes. Interpolation is performed separately for each channel

The colour component is handled in the same way regardless of the S3TC variant. Two 16-bit representative colours (color0 and color1) are calculated per compression block (16 pixels) and each pixel in the block is encoded with the linear combination of the two colours from table 4.1 that best approximates the original value. When the pixel colour is decoded from the compressed data, linear interpolation is performed separately for each channel. Representative colours are encoded as UNSIGNED_SHORT_5_6_5 values [19], with 5 bits for each the red and blue channels and 6 for the green channel. The linear combination for each pixel is represented with a 2-bit control code. The final result is a 64-bit compressed colour block where all pixels are considered to be totally opaque. A 4x4 texture block contains 16 RGB pixels with 8 bits per channel giving a total uncompressed block size of 48 bytes and a resulting compression ratio of 6:1.

The three main S3TC variants, DXT1, DXT3 and DXT5, differ in the way the handle the alpha channel. When DXT1 is used with a RGBA 2D texture it uses the order of the representative colours inside the compressed colour block to indicate whether the original alpha value is closest to 0 or to 1. If the first representative colour (color0) is less or equal to the second (color1), both interpreted as 16-bit unsigned integers, and the pixel control code is 3, then the alpha value for that pixel will be 0, otherwise it will be 1.

DXT3 uses an additional 4-bit value per pixel to store the original alpha value. These alpha values are packed in a 64-bit (16 pixels x 4 bits per value) integer and included at the beginning of the compression block doubling the size to 128 bits and thus a compression ratio to 4:1. This method provides a wider range of possible alpha values, 16 possibilities versus 2 in DXT1, at the cost of increased block size. DXT5 works in a similar way to DXT3, in the sense that it uses an additional 64-bit per block to encode the alpha channel information. The difference is that with this variant two 8-bit representative alpha values are calculated per block and interpolated with the best out of eight possible ways to approximate to the original alpha value per pixel. This means that an additional 3-bit control code is required per pixel to encode the interpolation mode. Figure 4.3 shows the alpha channel obtained after compression with the three different DXTC variants.





PVTC works only with RGB textures but E-PVTC makes use of the alpha channel to store the Compression Correction Code, as it will be seen later.

VTC

VTC is an extension of S3TC to three dimensional textures. It operates as well with RGB or RGBA voxels with 8 bits per channel. Voxels are grouped in blocks of 4x4x1 and compressed as an S3TC 4x4 block, with the same three possible variants: DXT1,

DXT3 and DXT5. S3TC blocks are grouped in blocks of four corresponding to the same X and Y range and growing Z coordinate, forming a VTC block. Figure 4.4 illustrates how S3TC and VTC blocks are constructed.



Figure 4.4: Formation of compression blocks in S3TC and VTC

PVTC

Packed Volume Texture Compression (PVTC) is a technique developed in [1] to take advantage of VTC in the compression of time-varying volumes. It works with a collection of volumes all with the same dimensions and 8 bit voxel values that represent the different time steps in a volume animation. This technique packs the data from three consecutive time steps in the red, green and blue channels of a 3D RGB texture and compresses it with VTC, variant DXT1. In other words, the values of a voxel in a given position during three consecutive time steps are transformed into the components of a virtual colour assigned to that voxel. When VTC/S3TC computes the representative colours of a 4x4x1 block it is actually calculating the two most representative voxel values for each of the three time steps packed. The interpolation mode calculated for a specific voxel is influenced by both the values of that voxel in each time step and the way they change compared to the other voxels in the block. Compression quality will be determined then by how uniform voxel values are in each time step, which allows the calculation of better representative values for each channel, and how similar their evolution is in time, which reduces the errors produced by using the same interpolation mode for each voxel in the three channels/time steps.

At rendering time, the resulting 3D VTC compressed texture is transferred to the GPU and the voxel information recovered from the colour channel corresponding to the time step being displayed. The selection of the colour channel is made from the CPU, changing the fragment shader used to draw the current frame. This way, costly branching instructions are avoided in the shader code.

4.2.3 Preprocessing

Volume data preprocessing utilities had to be developed to satisfy two needs: conversion of test datasets obtained in different formats to a common raw format suitable for PVTC compression and compression of the raw format files with PVTC and LZO. The raw format used is a binary sequence of unsigned byte values representing scalar voxels in a single volume, with X being the fastest running direction and Z the slowest. This means that if the value of the voxel in the position (x, y, z) is noted as $V_{x,y,z}$ a 2x2x2 volume will have the following layout in memory:

$$V_{0,0,0}, V_{1,0,0}, V_{0,1,0}, V_{1,1,0}, V_{0,0,1}, V_{1,0,1}, V_{0,1,1}, V_{1,1,1}$$

$$(4.1)$$

Converting from a different raw format involved performing some or all of the following steps:

- 1. Remove header information from source data.
- 2. Invert byte order according to source endianess and data type.
- 3. Compute global maximum (S_{max}) and minimum (S_{min}) values for the whole dataset -i.e. the volumes for all time steps.
- 4. Convert source values in each time step $t(S_{x,y,z}^t)$ to unsigned bytes $(V_{x,y,z}^t)$ using the following formula:

$$V_{x,y,z}^{t} = (S_{x,y,z}^{t} - S_{min}) / (S_{max} - S_{min}) * 255$$
(4.2)

Padding was only required in some datasets to avoid texture compression issues related with dimensions. Values mapped to zero opacity by the transfer function were used to perform it.



Figure 4.5: Preprocessing process to convert source volume data into RAW, PVTC and LZO formats

Using a linear interpolation between the maximum and minimum values for conversion assures that the loss of precision is uniform through the whole source data range. Tests show that in some cases it may be interesting to assign higher precision to certain ranges of the source data where the most relevant information is located. Figure 5.10 in 5.2.2 shows the histogram of the data values in a time step of one of the datasets with the transfer function overlaid on it. It can be seen how most values are concentrated in the middle region and how the transfer function only shows a narrow range around the histogram peak. If this information is taken into account when the source values are converted to 8 bits, higher precision can be assigned to the most relevant value ranges. The transfer function can be modified to take advantage of this and show the regions of interest with more detail. Another advantage of this transformation method is that it would reduce the number of compression artefacts caused by close original values being mapped to the same compressed one. Sadly, there was no time to implement these ideas in this project, remaining an interesting direction for work in the future. Back to the preprocessing tools, PVTC compression was performed in the following steps:

- 1. Read 3 consecutive raw volumes and store them interleaved in a single RGB array.
- 2. Load the array as a DXT1 compressed texture using OpenGL function glTexImage3DEXT with internal format GL_COMPRESSED_RGB_S3TC_DXT1_EXT. This will make the device driver compress the texture as it is loaded.
- 3. Retrieve the compressed data from the texture with a call to OpenGL function. glGetCompressedTexImageARB

When the total number of time steps in the dataset is not a multiple of three, the last PVTC block is completed replicating the last time step. It is important to use actual volume data to fill in these empty channels because the representative values and interpolations calculated by VTC will be affected by it, even if the channels are not used for rendering.

Finally, LZO compression was performed using LZO's library function lzo1x_1_compress with a raw or PVTC compressed volume as the source data. Figure 4.5 illustrates the whole preprocessing process.

4.2.4 Animation Pipeline

As explained in the design section 4.1, the application stage that loads volume files from external storage (usually the hard drive) has been pipelined with the rest of stages (CPU decompression, texture transfer and volume rendering). When the animated volume renderer is started a loader thread is created with the Windows function _beginthreadex. This thread will read the current dataset volume files in sequence and store them in a circular buffer. The main thread blocks until the next volume is available in the buffer and then decompresses (if compressed with LZO) and transfers it to the GPU memory, marks the buffer position as free and renders the time-steps contained in the volume texture. In this way, the average time taken to render a frame is the maximum of the times taken by the two threads, i.e. if the data load is slower than volume decompression, texture transference and volume rendering together, it will be the bottleneck; otherwise the rendering speed will be determined by the main thread. Synchronization between the loader thread and the main thread follows the producerconsumer scheme [21]. Access to the shared circular buffer is regulated using two semaphores: *fillCount* and *emptyCount*. *fillCount* is increased by the producer (i.e. the loader thread) when a new item is inserted in the buffer and decreased by the consumer (i.e. the main thread) when an item is read. If there are no pending items in the buffer, the consumer blocks until a new one is inserted by the producer. *emptyCount* is increased by the consumer after removing an item from the buffer and decreased by the producer before inserting a new item. If the buffer is full -*emptyCount* is zero- and the producer tries to insert a new item, it will block until the consumer removes an item.

Both semaphores are created with the Windows function CreateSemaphore. *fill-Count* has an initial value of 0 and a maximum value equal to the size of the buffer in items; *emptyCount* is initialized to the size of the buffer in items and has the same maximum value. Up (increase) and down (decrease) operations on the semaphores are performed using ReleaseSemaphore and WaitForSingleObject functions respectively. The number of volumes that can be stored in the buffer is configurable, although values higher than two only serve to dampen irregularities in stage execution times and a size of one totally defeats the purpose of the pipeline, because it forces either the producer or the consumer to block while the other one is working.

Pipelining of the remaining stages could be implemented without big structural changes. Following the current architecture, the decompression stage should be moved to a separate thread, consuming from the load stage output buffer, and a decompressed volume buffer should be created. Another thread should be added for the texture transfer stage that would read from the decompressed volume buffer and transfer the texture to GPU memory. Further changes would be needed in the fragment shaders and the main (volume rendering) thread to use the appropriate volume texture when rendering. Pipelining of the texture transfer stage would only be possible with this architecture if there is enough space in GPU memory for two volume textures.

4.2.5 Extended PVTC

Extended PVTC (E-PVTC) was implemented making use of the DXT3 variant of VTC. As explained before in 4.2.2, DXT3 allows compression of RGBA textures treating the colour components the same way DXT1 does but encoding the alpha channel as a 4bit uncompressed value per voxel. Compared with PVTC, this effectively doubles the volume size but provides an extra 4-bit value per compressed texture voxel to play with. E-PVTC packs three consecutive time steps in the RGB channels of a 3D texture, like PVTC, and uses the alpha channel to store a Compression Correction Code (CCC) that will be applied to the voxel values fetched from the compressed texture by the volume renderer fragment shader. Let $R_{x,y,z}^t$, $R_{x,y,z}^{t+1}$ and $R_{x,y,z}^{t+2}$ be the values of the voxel at position (x, y, z) in time steps t through t + 2 and $C_{x,y,z}^t$, $C_{x,y,z}^{t+1}$ and $C_{x,y,z}^{t+2}$ the values for the same voxel and time steps obtained after PVTC compression. The CCC for the corresponding E-PVTC packed voxel is the value of x that minimises the sum of squared compression erros per time step/channel:

$$(R_{x,y,z}^{t} - (C_{x,y,z}^{t} + x))^{2} + (R_{x,y,z}^{t+1} - (C_{x,y,z}^{t+1} + x))^{2} + (R_{x,y,z}^{t+2} - (C_{x,y,z}^{t+2} + x))^{2}$$
(4.3)

Deriving with respect to x and equating to zero yields the following equation for the Compression Correction code for a PVTC packed voxel representing the values at position (x, y, z) through time steps t to t + 2 $(CCC_{x,y,z}^{[t,t+2]})$:

$$CCC_{x,y,z}^{[t,t+2]} = \left(\left(R_{x,y,z}^t - C_{x,y,z}^t \right) + \left(R_{x,y,z}^{t+1} - C_{x,y,z}^{t+1} \right) + \left(R_{x,y,z}^{t+2} - C_{x,y,z}^{t+2} \right) \right) / 3$$
(4.4)

The value produced by this formula is a signed real number that has to be encoded in the 8 bit alpha component of the packed voxel that will be latter reduced to a 4 bit precission value by DXT3 compression. This means that only 16 different values can be used, thus the decision to clamp the CCC to the range [-7, 8] before encoding it. For this, the computed CCC is rounded to the nearest integer, clamped to the range [-7, 8], added 7 and mutiplied by 16 to convert it to the range [0, 255] covered by the 8 bit alpha channel. The equation below shows the CCC encoding process:

$$CCC_{encoded} = (clamp(round(CCC), -7, 8) + 7) * 16$$

$$(4.5)$$

Once a volume RGBA texture has been created for a sequence of three time steps with the voxel information in the colour channels and the CCC in the alpha channel, it is compressed calling the OpenGL function glTexImage3DEXT with internal format GL_COMPRESSED_RGBA_S3TC_DXT3_EXT. This causes the device driver to compress the raw data into DXT3 format. The compressed information is recovered, as in PVTC, using the function glGetCompressedTexImageARB.

After the E-PVTC compressed texture has been transferred to the GPU the fragment shader will obtain a decoded CCC floating point value in the range [0, 1] from the alpha channel. The value fetched will have to be transformed to the range [-7/255, 8/255]before being used to correct the voxel value read from the colour channel corresponding to the current time step. The following formula summarises the process:

$$Voxel_{corrected} = Voxel_{PVTC} + (CCC_{decoded}/16) - (7/255)$$

$$(4.6)$$

A different version of each of the three Cg fragment shaders -one per time step encoded in a PVTC volume texture- was developed to apply the CCC while rendering. The right fragment shader is selected from the main program according to the compression format and current time step. This avoids unnecessary and costly conditional statements being executed in the shader program for each fragment processed.

Finally, a note about interpolation. When values are fetched from the volume texture in the fragment shader, linear interpolation is applied. This applies not only to the voxel values read from the colour channels but as well to the Compression Correction Code extracted from the alpha channel. CCC values are calculated to minimise the compression error of individual voxel values, so it seems necessary to evaluate if they still work well when interpolated and applied to an interpolated voxel value. Let's imagine a simple scenario where two voxels with values V_a and V_b and CCC CCC_a and CCC_b are linearly interpolated with weights a and b (a+b = 1). The following formula shows how correcting the interpolated values is equivalent to interpolating the corrected values:

$$(a * V_a + b * V_b) + (a * CCC_a + b * CCC_b) = a * (V_a + CCC_a) + b * (V_b + CCC_b)$$
(4.7)

It is quite straightforward to prove that the same holds true regardless of the number of terms used in the interpolation.

Chapter 5

Evaluation

This evaluation has several goals, the first of them being to compare the results obtained rendering animated volumes with those of the original research papers [1, 2]. For this, the results in the section about the compression system will be compared against the ones provided in the most recent research [2]. The newer paper is used because it provides measurements of the original techniques developed in the older paper but using more recent hardware. This sets a fairer comparison point, even if the pipelining improvements described in this second paper have not been included in the dissertation work.

The second goal is to assess the improvements achieved with the new E-PVTC compression format. For this image quality and rendering speed tests will be performed with E-PVTC and the results compared against the ones obtained with volumes in raw and PVTC formats. Finally, the evaluation should provide data to support the assessment of the applicability of the techniques used to real scenarios such as interactive entertainment and medical and scientific visualization.

All tests were performed in a desktop PC with an Intel x86 Core Duo CPU running at 2.6GHz and 4GB of RAM. The graphics card was an NVIDIA Quadro FX 580 with 512 MB of VRAM. The operative system used was Windows XP SP3.

Datasets D1 to D4 were obtained from [22]. The first two represent the simulation of a turbulent vortex, with D2 being an upscaled version of D1. D3 and D4 are the result of the simulation of a turbulent jet and its upscaled version respectively. D5 is a static dataset of a human head taken from [13]. All of them contain 1 byte voxel

Dataset	Resolution	Time steps	Raw time-step size (MB)
D1: Vortex Small	$128 \ge 128 \ge 128$	98	2.00
D2: Vortex Big	$256 \ge 256 \ge 256$	98	16.00
D3: Jet Small	$104 \ge 129 \ge 129$	150	1.65
D4: Jet Big	$208 \ge 258 \ge 258$	150	13.00
D5: Head	$256 \ge 256 \ge 256$	1	16.00

Table 5.1: Datasets used for experiments

values, as bigger types are not supported in the PVTC compression format. Whenever the original data was not available with that precision, the original values where linearly normalized between the maximum and minimum values of the whole dataset and converted to the range 0-255 (see 4.2.3 for details). Upscaling was performed by linear interpolation of the closest voxels.

When benchmarking an out-of-core rendering system -i.e. rendering systems where data is streamed in real-time from external memory, such as a hard drive- care has to be taken to avoid the effects of file caching. Usually there are at least two complementary kinds of file caching in a modern desktop PC: the caching performed in the storage device -which cannot be easily overridden- and the one performed by the operating system. This second caching can be avoided in Windows through a series of functions provided to access files without system caching [23]. To prevent both types of file caching two tests with the same dataset were never performed without a system reboot between them.

A particular issue when benchmarking out-of-core systems in Windows is the effect of a system service called Windows Prefetch. This background service monitors the execution of programs and keeps track of their frequently accessed files. After marking a file as frequently used by a certain program, Window Prefetch will preload the file whenever it detects the program is started, saving loading time for a file that it is probably going to be loaded anyway at a later point. This service is active by the default and can affect the performance of the test application if a dataset is identified as a frequently used file. To prevent any interference from Windows Prefetch it was deactivated through the Windows Registry before performing any tests.

Finally, CPU-GPU parallelism has to be taken into account for a proper timing

of the texture transfer and rendering stages. OpenGL commands are usually queued and executed in batches to optimize performance [24]. Therefore the time taken to execute an OpenGL command is the queuing time, as the actual execution in the GPU is usually delayed to a later moment in time. To avoid this issue, the execution of the command must be forced with a call to the OpenGL function glFinish, that causes all pending commands to be executed. To take texture transfer and rendering times the corresponding stages were enclosed in glFinish commands in the testing program, ensuring in this way that timings reflect the actual GPU execution time at the cost of a slight penalty in the frame rate.

5.1 Volume Renderer

In this section the effect of different parameters in the volume renderer performance will be evaluated. The measures presented were produced rendering a single timestep -no animation- to isolate the performance of the texture-based rendering system from that of the load subsystem. The goal was to establish a correspondence between the different parameters affecting performance and the number of frames rendered per second (FPS).

First, the impact of the fillrate, understood as the number of fragments (pixels with depth information) processed per frame, was tested. The fillrate is affected mainly by three parameters in our scenario: screen resolution, number of view aligned cut-planes and their scale. The screen resolution determines the total number of pixels to be renderer and thus has an impact in the number of fragments. Each cut-plane adds a fragment for every screen pixel it covers. Scaling up the cut-planes means they cover a bigger area of the screen, up to a scale of x1 that represents that the cut-plane corners coincide with the corners of the screen. Therefore, increasing the scale up to x1 produces an increment in the number of fragments too.

Figure 5.1 shows the impact of the screen resolution and the scale of cut-planes on FPS, figure 5.2 shows the impact of the number of cut-planes. Screen resolution and scale of cut-planes figures were produced rendering the first time-step of dataset D1 with 221 cut-planes. Screen resolution tests were carried out with a fixed scale of x1; cut-plane scale tests used a fixed screen resolution of 1024x1024. The tests with varying number of cut-planes were performed using the static dataset D5 and fixed screen resolution of 512x512 and scale of x1. Data was pre-cached in system memory to make sure external storage access speed did not affect the results.



Figure 5.1: Screen resolution and cut-planes scale impact on rendering performance

It can be seen that even though the number of fragments to be processed increase quadraticaly with the resolution and the cut-planes scale the impact in the frame rate is not quadratic. A reason for this is the unified shader architecture present in the GPU that dynamically allocates shader units to the vertex, geometry or pixel/fragment stages of the graphics pipeline according to the current workload. As the number of fragments increases but the number of vertices remains constant the GPU decides to dedicate more shaders to the fragment stage, thus reducing the impact of the increment in the number of fragments. The same effect can be seen in the results for different numbers of cut-planes because the variation in the number of vertices (4 for each additional plane) is relatively small compared with the increment in the number of fragments.

The second batch of tests performed was aimed to find out if the volume resolution has an impact in the FPS, being the rest of parameters constant. It was expected that it would not have an important impact as the number of fragments and texture fetches would remain the same. The first time-step of dataset D1 was rendered with a screen resolution of 512x512 and 256 cut-planes at x1 scale (full screen). The other volumes used were upscaled or downscaled versions of D1. Figure 5.3 shows how rendering performance remains almost constant with increasing volume resolutions from 64 to



Figure 5.2: Impact of the number of cut-planes on rendering performance

256 cube side, but experiments a noticeable drop of around 25% when a side of 512 is used. This could be caused by the inefficiency of GPU texture caches hiding the latency of texture memory access with large volume textures [11].



Figure 5.3: Impact of the volume resolution on rendering performance

Other of the tests carried out was designed to check if the volume compression format had an impact on rendering performance. For this, versions of the first timestep of dataset D1 in the three compression formats -raw (no compression), PVTC and E-PVTC- were rendered with a 512x512 screen resolution and 256 cut-planes at x1 scale. As figure 5.4 shows, there are no significant differences in the FPS figures achieved with each format, with the slowest one being around 0.3% worse than the



fastest.

Figure 5.4: Impact of the volume compression format on rendering performance

Finally, dataset D5 was rendered with different numbers of cut-planes to get a visual assessment of its influence in image quality. A 512x512 screen resolution and a scale of x1 were used. Figure 5.5 shows the results obtained ordered from left to right and top to bottom by growing number of slices, with the last image being a raycasted render of the dataset created with the Voreen volume renderer [25].

Two effects of the increment in the number of cut-planes can be noticed in the images. The first one is the apparent brightening of the colours as the amount of cut-planes grows. The cut-planes can be thought of as semi-transparent coloured slides that are stacked on top of each other. The more slides, the more opaque the image we see through the stack will be. This effect is usually undesired and happens when the transfer function used has been designed for a different number of cut-planes or sampling rate. It can be avoided computing a corrected transparency or alpha value for each cut-plane using a technique known as opacity correction [26] that has not been included in this volume renderer due to time restrictions.

The second visible effect of an increment in the number of slices is the improvement in the smoothness of the surfaces. This improvement though tends to be less appreciable when the number of cut-planes exceeds the depth resolution of the volume. Comparison with the raycasted image is difficult because of the colour differences caused by the different sampling rates. The main difference that can be appreciated is the absence of the curve artefacts produced in textured-based rendering by the intersection of the cut-planes with the volume.



Figure 5.5: Impact of the number of cut-planes on the image quality

5.2 Load System

The purpose of these compression systems tests is to evaluate the performance of the system rendering animations with the different compression formats and to assess the visual quality achieved with PVTC and the new compression scheme E-PVTC.

5.2.1 Rendering Performance

To evaluate the out-of-core rendering performance of the system with animated volumes measures of the average FPS achieved were taken. Besides, execution time was registered broken down in three stages. The first is the data load stage that comprises loading the volume data files from external storage (HDD) to main CPU memory

Dataset	Screen Resolution	Number of cut-planes
D1	256×256	221
D2	512x512	443
D3	256 x 256	209
D3	512x512	419

Table 5.2: Test parameters for the volume animation rendering performance tests

(RAM) and, when applicable, LZO decompression by the CPU. The second stage is the texture transference from RAM to GPU memory. Finally, the last stage is volume rendering in the GPU. There are two important points to consider when working with these broken down times. One is that the data loading and LZO decompression stage is executed in parallel with the other two stages and thus the total execution time is the maximum of both. The other is that when PVTC or E-PVTC compression is used each volume file contains three time-steps, so the average time per frame for data load and texture transfer is calculated dividing the times taken by three.

For these tests datasets D1 through D4 were used. Screen resolution was 256x256 for D1 and D3 and 512x512 for D2 and D4. The full animation was played once, with a scale of x1. The viewing direction was initially set to the -Z axis direction and then rotated 2 degrees around the X and Y axes every frame. The number of cut-planes used for each dataset was set to the diagonal of the volume to make sure that, regardless of the orientation, there was at least one cut-plane per voxel. Table 5.2 summarizes the parameters used.

Figure 5.6 shows rendering performance in FPS for the first four datasets with three different GPU compression possibilities: raw (no compression), PVTC, and E-PVTC. LZO compression is not included in these tests to keep the number of configurations low. Results will be shown later with and without LZO compression for dataset D2. It can be seen how PVTC provides an average increase in FPS of 290% with respect to the raw format, the minimum improvement being the 200% obtained with dataset D3 -from 21 to 63 FPS- and the maximum the 345% obtained with dataset D2 -from 4 to 18 FPS. These results do not deviate significantly from the ones published in [2].

E-PVTC gives an average speed improvement of 177% with respect to the raw format, with a minimum of 114% in dataset D3 and a maximum of 224% in dataset



Figure 5.6: Rendering performance in average frames per second

D1. The difference with the results for PVTC can be explained by the increased size of the data volumes in E-PVTC -double the size of PVTC- due to the inclusion of the uncompressed alpha channel where the Compression Correction Code is stored. Whether this loss in speed is acceptable or not will depend on the use of the animation. If improved image quality is more important than faster animation, E-PVTC can be a better option than PVTC.

The impact of LZO in rendering performance is shown in figure 5.7 for dataset D2. It can be seen how the addition of LZO compression produces a slight increment in FPS of around 7% in the case of PVTC and a decrement of 9% for E-PVTC. In general, the speed improvement obtained with LZO depends on the compression achieved. If the compression ratio is under a certain threshold the overhead of decompression dominates over the reduced data transfer times produced by the smaller file size. In the case of dataset D2, the average compression ratio obtained with LZO for the PVTC files is 1.35 : 1 while for E-PVTC is 1.26 : 1. This could explain the differences in the performance obtained for both formats.

Finally, figure 5.8 shows the breakdown of the average execution time per timestep of each stage -data load, texture transfer and rendering- for dataset D2 in all the available compression formats, with and without LZO. As expected from the results obtained in the volume renderer tests, the rendering time does not vary significantly



Figure 5.7: Performance impact of LZO compression in FPS with dataset D2 (Vortex Big)

with the compression format used. Texture transfer time is reduced around a 28% with PVTC and around a 20% with E-PVTC relative to the times taken with the raw format. The performance gap between PVTC and E-PVTC is caused by the volumes compressed with the latter being double the size of the ones compressed with the former. Data loading times present the biggest variations with compression formats. PVTC performs 81% faster than the raw format, 89% with added LZO compression, while E-PVTC results improve the ones obtained with the raw format by a 68%, 65% with LZO. The increased load time when using LZO with E-PVTC would explain the results obtained previously in the study of the impact of LZO in rendering performance.

5.2.2 Image Quality

The issue of image quality arises from the fact that Volume Texture Compression (VTC), the GPU hardware supported texture compression format used in PVTC, is lossless. Summarising section 4.2.2, VTC compresses the original RGB volume in blocks of 4x4x1 voxels computing two RGB_5_6_5 representative colours for the each block and expressing each individual voxel as one of the four possible linear combinations of the representative colours. Combined with the fact that each colour channel in PVTC contains a time step, this means that the original 8-bit precision voxel data values have to be reduced to four possible different 5 or 6 bit values -depending on the time step.



Figure 5.8: Breakdown of average stage execution time per time-step for dataset D2 (Vortex Big) with different compression methods

per compression block.

Despite this reduction in the precision of the texture values VTC usually produces relatively good results when it is applied to volume textures that contain the actual colours to be rendered. This is because the error on the colours shown on the screen is directly proportional to the errors produced by the compression. When the volume texture contains data values -such as density, velocity, vorticity- that are not colours, they must be mapped to colours applying a transfer function when rendering to screen. This means that if two close data values are mapped to very different colours even a small deviation produced by VTC compression can have a big impact on the image produced.

As explained in 4.2.5, E-PVTC aims to improve image quality computing a 4-bit Compression Correction Code that minimises the compression error for each voxel colour. This allows a wider range of values by compression block which helps to blur the limits between blocks and increases the smoothness inside them.

Figure 5.9 shows renderings of frame 40 of dataset D3 with raw, PVTC and E-PVTC compression formats. The values in this dataset represent vorticity magnitude. The images were produced with a screen resolution of 512x512 and 209 cut-planes. Strong blocky artefacts can be seen in both left and right sides of the PVTC image caused by the small number of different values allowed per 4x4x1 block. E-PVTC greatly reduces artefacts in both sides thanks to the wider range of values that can be



Figure 5.9: Comparison of image quality with raw, PVTC and E-PVTC compression formats. Strong blocky artefacts in PVTC are almost totally supressed in E-PVTC

represented. This greater variety of colours helps as well to produce smoother contours that resemble more the original ones, as it can be seen in the augmented region shown in the bottom row of pictures in figure 5.9. The histogram of the data values of the volume rendered is shown in figure 5.10 with the transfer function overlaid on it. It is a clear example of a transfer function that assigns well differentiated colours to data values that are very close together making the rendered output very sensitive to compression errors.

Finally, there is another visual artefact produced in both PVTC and E-PVTC by the different precision of the RGB channels used in VTC. As mentioned before, the representative colours of each compressed block are encoded as RGB_5_6_5, meaning that 5 bits are assigned to both the red and blue colour values and 6 to the green one. (E-)PVTC packs three consecutive time steps in the RGB colour channels, what results in the second of every three time steps -i.e. the one packed in the green channel- having



Figure 5.10: Histogram of the values in time step 40 of dataset D3 (Jetflow Small). The transfer function is overlapped with the colour indicated by the node colour and the opacity by the Y axis position

better precision that the other two. Even though some extra definition would not seem negative, when the time-steps are rendered in sequence the difference in precision from one step to the next produces an impression of lack of continuity or smoothness in the animation. To illustrate this effect, three consecutive time steps of dataset D3 are presented in figure 5.11 rendered with raw and PVTC compressed formats. It can be seen how the second step of the PVTC series, the one packed in the green channel, is the most similar to the uncompressed one.

5.3 Assessment of Applicability

The previous sections aimed to produce and analyse measures of the system performance. In this section the knowledge gained in that process will be used to assess the possible application of the technologies developed to different fields.

Pregenerated animated volumetric effects such as burning fires, smoke clouds or animated glows are frequently used in interactive entertainment -i.e. videogames- and commonly implemented as animated 2d textures projected into planes. This has the disadvantage that the effect is always shown for the same point of view -or from a reduced number of points of view- regardless of the position of the camera with respect



Figure 5.11: Images of time steps 40 to 42 of dataset D3 (Jetflow Small) rendered with raw and PVTC compression formats. Notice the higher quality of the second time step in PVTC.

to it, impacting negatively on the perceived realism. Real time volume rendering of animated volumetric data allows introducing in interactive applications volume animations that can be presented from any point of view.

Let us consider a typical scenario where a volume animation four seconds long, possibly periodic, with 20 frames per second is required for the flame a burning torch. Results obtained in previous sections show how out-of-core rendering is possible at 65 fps for 128x128x128 animations. Nevertheless, this cannot be translated directly to interactive entertainment because a high number of elements has to be renderer each frame, so only a small fraction of it can be spent on volumetric effects. Consequently, speed has to be favoured over quality. Tests showed that, thanks to texture interpolation, 128x128x128 volume resolutions produce acceptable results when viewed at 512x512, which is approximately half the resolution of a high definition screen (1080x720), a common screen resolution for interactive entertainment applications.

Halving volume resolution to 64x64x64 would allow displaying animations covering a quarter of a typical game screen while retaining decent quality, what may be acceptable for background effects like that of a burning torch, and decrease rendering time thanks to the reduction in the number of cut-planes required. Besides, the full GPU compressed animation will take only 5 MB of memory eliminating the need to stream information from external storage and making possible the storage of the full animation in the GPU memory. This could greatly improve rendering performance because all data transferences could be performed off-line and will allow rendering simultaneously multiple instances of the same animation with the only cost of the additional volume rendering. Finally, seamless integration of the volume animations with the polygonal environment should not require much effort thanks to the texture-based volume rendering approach.

Summarising, real time volume rendering of multiple instances of a small sized volumetric background animation around 64x64x64, fully integrated with the polygonal environment, should be feasible for interactive entertainment applications in current hardware.

Compared with interactive entertainment, typical medical and scientific animated volume visualization applications can afford spending most of their process time in volume rendering but use datasets that are larger and have higher precision. It is not unusual to find 512x512x512 4 byte precision time-varying data in the medical field while in scientific applications very high resolutions and precision can be obtained thanks to simulation software, well beyond 4096x4096x4096 and 8 byte precision. The use of PVTC compression requires reducing this data precision to 1 byte and applying a lossy compression technique (VTC) that can produce very noticeable artefacts (see 5.2.2). E-PVTC can be used to reduce image degradation to some extent at the cost of a 50% data size increase and an average 28% speed loss relative to PVTC. Besides, results show real time rendering is not currently achievable with resolutions much higher than 256x256x256 (19 fps), although speed improvements between 33% and 122% with further pipelining are reported in [2]. Overall, despite the precision and resolution restrictions and artefacts due to lossy compression, the two-stage compression system is still a valuable tool to allow interactive navigation of large animated datasets and could be used in conjunction with other methods to provide higher quality still images when required.

Chapter 6

Conclusions and Future Work

The two-stage compression approach to out-of-core real time rendering of animated volumes has yielded an average 290% frame rate improvement relative to rendering of uncompressed volumes. This is achieved thanks to the reduced external storage to CPU and CPU to GPU transfer times obtained through double compression of volume data and the use of compression methods that leverage the different characteristics of CPU and GPU hardware and exploit efficiently time-varying data coherency. This improvement allows real time rendering of time-varying data with resolutions up to 256x256x256 at 19 fps in current hardware. Further optimizations could be obtained pipelining the CPU decompression stage ([2] report between 33% and 122% increase in fps). A novel and interesting addition would be testing the effect on performance of pipelining the texture transfer stage (see 4.2.4).

Image quality is affected by the use of a lossy compression method (PVTC) for GPU compression. Blocky artefacts appear and they become very strong with sensitive transfer functions. The new compression method developed to improve image quality, E-PVTC, has proved capable of reducing significantly compression artefacts at the cost of increased compressed volume size and reduced rendering speed relative to PVTC (twice the volume size and 28% slower on average). Animation discontinuities caused by the higher precision used to encode the second of every 3 time-steps remains an issue that could be addressed in future work. Approaches to take could include changing compression methods to use the same precision in all channels or take advantage of new GPU supported volume compression formats.

There is margin for improvement as well in the way high precission source voxel values are converted to the 8 bit precision required by PVTC. As suggested in 4.2.3, information about the relevant data ranges could be extracted from the transfer function and used to assign higher precision to regions of interest when converting the source data to single byte format. Another limitation of PVTC based compression techniques is that they only allow working with scalar data values. New methods to make use of GPU supported volume compression (VTC) with vectorial time-varying values remain to be explored.

Finally, the assessment of applicability has shown that the techniques used can be useful for rendering small volumetric effects in interactive entertainment applications and for navigating large datasets, in conjunction with complementary techniques to produce higher quality snapshots, in medical and scientific visualization.

Appendix

Acronym	Definition
CCC	Compression Correction Code
CPU	Central Processor Unit
DXTC	DirectX Texture Compression
E-PVTC	Extended Packed Volume Texture Compression
FPS	Frames Per Second
GB	Giga Byte
GPU	Graphics Processor Unit
HDD	Hard Disk Drive
LZO	Lempel-Ziv-Oberhumer
MB	Mega Byte
PVTC	Packed Volume Texture Compression
RAM	Random Access Memory
RGB	Red Green Blue
RGBA	Red Green Blue Alpha
S3TC	S3 Texture Compression
VR	Volume Rendering
VRAM	Video RAM
VTC	Volume Texture Compression

Table 1: Acronyms commonly used in the text

Bibliography

- [1] D. Nagayasu, F. Ino, and K. Hagihara, "Two-stage compression for fast volume rendering of time-varying scalar data," in *GRAPHITE '06: Proceedings of the* 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia, (New York, NY, USA), pp. 275–284, ACM, 2006.
- [2] D. Nagayasu, F. Ino, and K. Hagihara, "Technical section: A decompression pipeline for accelerating out-of-core volume rendering of time-varying data," *Comput. Graph.*, vol. 32, no. 3, pp. 350–362, 2008.
- [3] J. Kruger and R. Westermann, "Acceleration techniques for gpu-based volume rendering," in VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03), (Washington, DC, USA), p. 38, IEEE Computer Society, 2003.
- [4] K.-L. Ma, "Visualizing time-varying volume data," Computing in Science and Engg., vol. 5, no. 2, pp. 34–42, 2003.
- [5] H.-W. Shen, L.-J. Chiang, and K.-L. Ma, "A fast volume rendering algorithm for time-varying fields using a time-space partitioning (tsp) tree," in VIS '99: Proceedings of the conference on Visualization '99, (Los Alamitos, CA, USA), pp. 371–377, IEEE Computer Society Press, 1999.
- [6] R. Westermann, "Compression domain rendering of time-resolved volume data," in VIS '95: Proceedings of the 6th conference on Visualization '95, (Washington, DC, USA), p. 168, IEEE Computer Society, 1995.
- [7] R. Samtaney, D. Silver, N. Zabusky, and J. Cao, "Visualizing features and tracking their evolution," *Computer*, vol. 27, no. 7, pp. 20–27, 1994.

- [8] D. C. Banks and B. A. Singer, "A predictor-corrector technique for visualizing unsteady flow," *IEEE Transactions on Visualization and Computer Graphics*, vol. 1, no. 2, pp. 151–163, 1995.
- [9] T. J. Jankun-kelly and K. liu Ma, "A study of transfer function generation for time-varying volume data," in *In Proceedings of Volume Graphics Workshop 2001*, pp. 51–68, Springer-Verlag, 2001.
- [10] K.-L. Ma and D. M. Camp, "High performance visualization of time-varying volume data over a wide-area network status," in *Supercomputing '00: Proceedings* of the 2000 ACM/IEEE conference on Supercomputing (CDROM), (Washington, DC, USA), p. 29, IEEE Computer Society, 2000.
- [11] L. Ikits, Kniss and Hansen, GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics, ch. 39: Volume Rendering Techniques. Pearson Higher Education, 2004.
- [12] M. Hadwiger, J. M. Kniss, C. Rezk-salama, D. Weiskopf, and K. Engel, *Real-time Volume Graphics*. Natick, MA, USA: A. K. Peters, Ltd., 2006.
- [13] T. Sumanaweera, GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics, ch. 40: Applying Real-Time Shading to 3D Ultrasound Visualization. Pearson Higher Education, 2004.
- [14] NVIDIA Corporation, "Cg toolkit gpu shader authoring language." Retrieved September 10, 2010, from http://developer.nvidia.com/object/cg_toolkit. html, 2010.
- [15] M. F. Oberhumer, "Lzo real-time data compression library." Retrieved August 27, 2010, from http://www.oberhumer.com/opensource/lzo/, 2008.
- [16] NVIDIA Corporation, "VTC texture compression extension," tech. rep., Open GL Extension Registry, 2004. Retrieved August 26, 2010, from http://www.opengl. org/registry/specs/NV/texture_compression_vtc.txt.
- [17] Microsoft, "Dds, direct draw surface file format." Retrieved September 8, 2010, from http://msdn.microsoft.com/en-us/library/bb943990(VS.85) .aspx, 2010.

- [18] NVIDIA Corporation, "S3TC texture compression extension," tech. rep., Open GL Extension Registry, 2001. Retrieved August 26, 2010, from http://www. opengl.org/registry/specs/EXT/texture_compression_s3tc.txt.
- [19] Apple, "Apple_packed_pixels opengl extension specification." Retrieved September 8, 2010, from http://developer.apple.com/graphicsimaging/opengl/ extensions/apple_packed_pixels.html, 2002.
- [20] D. Rosen, "Dxtc texture compression." Retrieved September 8, 2010, from http: //blog.wolfire.com/2009/01/dxtc-texture-compression/, 2009.
- [21] M. Ben-Ari, Principles of Concurrent and Distributed Programming, ch. 6.7 The producer-consumer problem. Addison Wesley, 2 ed., 2006.
- [22] K.-L. Ma, "Time-varying volume data repository." Retrieved August 31, 2010, from http://www.cs.ucdavis.edu/~ma/ITR/tvdr.html, 2010.
- [23] Microsoft, "Msdn file buffering in windows." Retrieved August 31, 2010, from http://msdn.microsoft.com/en-us/library/cc644950(v=VS.85).aspx, 2010.
- [24] R. S. Wright, B. Lipchak, and N. Haemel, OpenGL(R) SuperBible: Comprehensive Tutorial and Reference (4th Edition). Addison-Wesley Professional, 4 ed., June 2007.
- [25] University of Münster, "Voreen volume rendering engine." Retrieved September 3, 2010, from http://http://www.voreen.org/, 2010.
- [26] J. van Scheltinga, M. Bosma, J. Smit, and S. Lobregt, "Image quality improvements in volume rendering," in *Visualization in Biomedical Computing* (K. Hhne and R. Kikinis, eds.), vol. 1131 of *Lecture Notes in Computer Science*, pp. 87–92, Springer Berlin / Heidelberg, 1996. 10.1007/BFb0046940.