Kompressions- und Darstellungsmethoden für hochaufgelöste Volumendaten

– Diplomarbeit –
im
Fachbereich Informatik
der RWTH Aachen

vorgelegt von

Jens Schneider[1]

May 12, 2003

Erstgutachter und Betreuer	:	**Prof. Dr. R. Westermann** Institut für Informatik / I15 Technische Universität München D-85748 Garching bei München
Zweitgutachter	:	**Prof. C. Bischof, Ph.D.** Institute for Scientific Computing RWTH Aachen D-52056 Aachen

[1]e-mail: schneider@glHint.de

Contents

Preface

In this work the following symbols will be used.

Sets and Classes

$\mathbb{N}$:	The set of natural numbers, beginning with 1.
$\mathbb{R}$:	The set of real numbers.
$\Re$:	An alphabet on which a vector quantizer will operate. Usually $\Re \subset \mathbb{R}$.
$\Im$:	A countable index set. Usually $\Im \subset \mathbb{N}$.
I:	An ordered input set to a vector quantizer.
C:	A codebook (an ordered, countable set of vectors).
V:	An ordered set with $V \subseteq I$.
$\mathcal{O}$:	Order of magnitude.

Functions, special vectors

δ:	A distance metric. Usually $\delta \equiv \| \cdot \|_2^2$.
X:	n-dimensional vector taken from I.
Y:	n-dimensional vector taken from C.

Operators

$\circ$:	Concatenation of functions. Read $a \circ b$ as “apply a before b”.
$<,>$:	Standard dot-product.
$\star$:	Convolution between functions.

Fidelity metric

In order to measure the fidelity of a lossy compression process, throughout this document the scalar *signal to noise ratio* (SNR) will be used. Despite the fact that many publications use the peak signal to noise ratio (PSNR), the SNR gives a more intuitive and reliable measurement of the perceived fidelity. The reason may be seen in the fact that the SNR takes into account all input data, whereas the PSNR only accounts for the peak value. The SNR is defined as follows.

Given an input data set $I = \{x_i\}_{i=1}^n$ and a reconstructed data set $I' = \{x'_i\}_{i=1}^n$. Then

$$\text{SNR}(I, I') := 10 \log_{10} \frac{\sigma_I^2}{\sigma_{I'}^2} \text{ dB},$$

where σ_I^2 is the variance of the input set I, and $\sigma_{I'}^2$ is the mean squared error (mse):

$$\text{mse}(I, I') = \frac{1}{n} \sum_{i=1}^n \|x_i - x'_i\|_2^2.$$

Other Conventions

By convention, scalar entities will be lowercase, such as c, d, etc, while vector valued entities will be uppercase, such as X, Y, etc. When not stated otherwise, vectors are interpreted as column-vectors. Matrices will be written calligraphic: $\mathcal{M}, \mathcal{N}, etc$.

Discretely sampled entities will usually have a subscript s, such as Φ_s, Δ_s, etc, in contrast to continuous ones: Φ, Δ, etc.

All implementations were done using the OpenGL API [OGLb] and an ATi Radeon 9700 graphics card [ATi]. Only extensions with multi-vendor support were used, in order to guarantee the resulting code to run with any graphics card supporting the functionality provided by the ARB_multitexture, ARB_vertex_program, ARB_fragment_program and either an OpenGL version of 1.4 or EXT_texture3D. A specification of these extensions can be found in [ARB].

Chapter 1

Introduction

1.1 Abstract

Even with today's amount of memory rapidly increasing, the size of scientific data is growing at least at the same speed. New data acquisition techniques and storage media are the reason for the fact that the amount of data currently surpasses the famous Moore's Law, which states that speed and memory capacity of computers double every 18 months[1]. As a consequence, there is an increasing need of visualization techniques that can efficiently handle datasets that are gigabytes or even terabytes in size.

On the other hand the speed of graphic processing units (GPUs) is currently surpassing Moore's Law as well, doubling processing power about every 12 months. This makes the usage of graphics hardware very appealing for visualization purposes, and consequently great effort has been taken to either map existing algorithms to graphics hardware, or to design entirely new algorithms able to benefit of current graphics hardware. In contrast, video memory is not doubling every 12 months, as very expensive special purpose RAM's are commonly needed to cope with the GPU's sheer processing speed.

As a logical consequence, even with todays computers having more than a gigabyte of main memory and up to 256 megabytes of video memory, compression is mandatory.

The main focus of this diploma thesis is to present an efficient compression scheme for scalar valued volume data that allows for rapid decoding on the GPU itself,

[1]Paradoxically enough, in the field of CFD faster processors produce larger datasets, preventing the data from being displayed at interactive framerates on the very same processor.

thus virtually expanding video memory by a factor of 20 and more.

1.2 Zusammenfassung in deutscher Sprache

Trotz des berühmten Moore'schen Gesetzes, welches besagt, daß sich Speicherkapazität und Geschwindigkeit von Computern alle 18 Monate verdoppeln, scheint sich die Größe wissenschaftlicher Daten mit mindestens derselben Geschwindigkeit zu entwickeln[2]. Als logische Konsequenz ist das Interesse an Visualisierungstechniken groß, die fähig sind Datensätze zu verarbeiten, die Gigabytes oder sogar Terabytes groß sind.

Auf der anderen Seite überholen derzeit auch Grafikprozessoren (GPUs) Moore's Law, hier kann man eine Verdoppelung der Rechenleistung alle 12 Monate beobachten. Diese Tatsache macht den Einsatz aktueller Grafikhardware für Visualisierungszwecke äusserst attraktiv. Im Gegensatz dazu verdoppelt sich die Speicherkapazität von Grafikkarten jedoch nicht alle 12 Monate, da üblicherweise schneller Spezialspeicher eingesetzt werden muß, um mit der immensen Rechengeschwindigkeit der GPU Schritt halten zu können.

Daher ist auch bei heutigen Rechnern, die leicht über ein Gigabyte Arbeitsspeicher und bis zu 256 Megabytes Grafikspeicher verfügen, ein sinnvolles Kompressionsverfahren angeraten.

Das Hauptziel dieser Diplomarbeit ist es, einen effizienten Kompressionsalgorithmus für skalarwertige Volumendaten zu präsentieren. Dieser erlaubt eine schnelle Decodierung auf der GPU selbst, und vergrössert somit die effektive Kapazität des Grafikspeichers um einen Faktor 20 und mehr.

[2]Paradoxerweise führen schnellere Prozessoren im Bereich der CFD zu größeren Daten, die dann auf eben jenen Prozessoren nicht mehr interaktiv dargestellt werden können.

1.3 Contributions

Most approaches made during the last years in the field of volume compression suffer from at least one of the following three problems.

- Datasets of reasonable size take hours, if not days, to compress.
- It is not possible to render the compressed data at interactive framerates.
- The image fidelity is too low to be competitive in any way.

In this thesis previous approaches are combined and extended in order to overcome all three of the above problems. The major novel contributions of this work are

- Implementation of an extremely fast vector quantizer.
- Compression of large data in reasonable time using a novel hierarchical vector quantization scheme.
- Interactive rendering of the data right from the compressed representation.
- Rendering of large time-varying volume data at interactive framerates.

1.4 Contents

The following chapter is intended to provide an overview of common state of the art rendering techniques for scalar valued, uniformly sampled volume data. Over that, an extended model of the OpenGL rendering pipeline will be presented, as introduced by latest consumer class graphics processing units (GPUs). Last but not least algorithms will be discussed that take full advantage of this extended rendering pipeline.

In chapter three, several possible compression methods for volumetric data will be discussed, and it will be motivated why a hierarchical vector quantization scheme was chosen as the back bone for this thesis.

Vector quantization algorithms commonly encountered in the literature are impractical for reasonably sized data due to the associated computational burden. In contrast, chapter four demonstrates how vector quantization can be made fast enough to efficiently encode large data sets. Specifically, is shown that by careful analyzation performance bottlenecks of previous algorithms can be avoided, resulting in a speedup of at least a factor 30.

Chapter five merges the results from chapter three and four and presents a hierarchical vector quantizer that can be used to compress arbitrary volume data sets. For the case of scalar valued volumes it is over that demonstrated that the data can be rendered directly and at interactive framerates from the compressed representation using graphics hardware with programmable fragment processing units.

In chapter six, the results of chapter five will be extended to the compression and rendering of time-varying data. It is shown that by application of a progressive encoding scheme an additional speedup of factor 2 can be achieved, without considerable loss in fidelity.

Chapter seven finally presents conclusions and discusses the results.

Chapter 2

Volume Rendering

2.1 Introduction

Volume rendering is the task of obtaining meaningful images from data sampled on a three-dimensional domain. In this thesis, only scalar valued volumetric data sampled on a regular grid will be regarded, though the compression algorithm presented in chapter 4 extends naturally to vector valued data. The reason for this restriction is that such data can be handled very efficiently by graphics hardware using textures mapping capabilities.

To provide the background necessary to understand the methods presented in this chapter, the next section reviews the latest OpenGL rendering pipeline model, while section 2.3 contains the definitions needed to describe scalar valued volume data. Section 2.4 gives a compact introduction to the volume rendering equation, as far as it is needed for the remainder of this document. The gap between theory and practise is closed in section 2.5 by presenting raymarching as one method to solve the rendering equation. In the final section of this chapter, methods to exploit texture mapping hardware are presented.

2.2 The OpenGL Rendering Pipeline

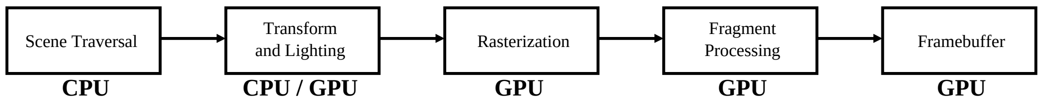

Figure 2.1: *The OpenGL Rendering Pipeline.*

In order to discuss implementations of the techniques presented in the remaining document, it is necessary to understand the rendering pipeline of the OpenGL API. The unextended version is depicted in figure 2.1 and is also referred to as the *Fixed Function Pipeline*, since its functionality is only configurable, as opposed to programmable. During scene traversal the CPU processes the input scene and converts it to rendering primitives. Depending on the capabilities of the graphics hardware these primitives are either transformed and lit on the CPU and then sent across the graphics bus[1] or sent directly across the graphics bus. If they are sent directly to the GPU, transform and lighting takes place on the graphics hardware itself[2]. In either case the transform and lighting step processes each vertex as follows.

- Transform the vertex to world space.
- Evaluate an user-configurable lighting term.
- Perform perspective correction.
- Clip against the viewing frustum.
- Perform the projection to screen space.

The rasterization step performs scan conversion as well as interpolation of colors, texture coordinates *etc*. Fragment processing consists out of the steps texturing, fragment testing and texture compositing (not necessarily in that order). The fragments passing the tests are then written to the framebuffer.

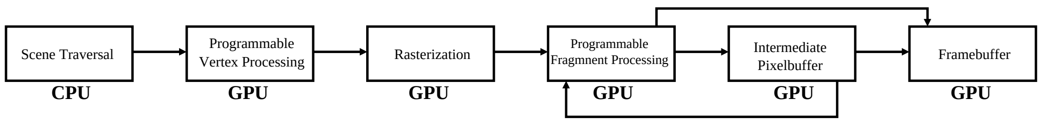

Figure 2.2: *The programmable OpenGL Pipeline. The fixed function pipeline from figure 2.1 is extended by programmable vertex and fragment processing.*

On current consumer class hardware this pipeline model may be extended in various ways. A common trend, however, is to substitute programmability for configurability. Current hardware is quickly developing towards the OpenGL 2.0 pipeline model that will be fully programmable [OGLa]. At an abstract level the standard transform and lighting step is subsumed by a programmable vertex processing unit, and the configurable fragment processing step is subsumed by a

[1]Usually the Advanced Graphics Port (AGP) for consumer class hardware.
[2]Virtually all consumer class graphics hardware supports transform and lighting by now.

programmable fragment processing unit. These functionalities are exposed to the API by the ARB_vertex_program and ARB_fragment_program extensions. Both extensions allow for loading a program written in an assembly language to the graphics hardware by providing an ASCII-string containing that program. The driver then parses and compiles the program to native instructions that will be executed by the GPU. Since current fragment processors are still subject to strong limitations when compared to conventional CPUs, intermediate pixel buffers were introduced that allow for feedback loops of arbitrary length. These pixel buffers are exposed to the API by the ARB_pbuffer extension. All programmable parts perform calculations using high precision up to IEEE 32bit float. Thus the necessity arises for some applications to have high precision texture and buffer formats as well. These are supported via the GL_NV_half_float, GL_NV_float and GL_ATI_texture_float extensions, that are likely to become promoted to EXT or ARB status in the near future. Fragments are only quantized down to 8bits prior to writing them to the framebuffer.

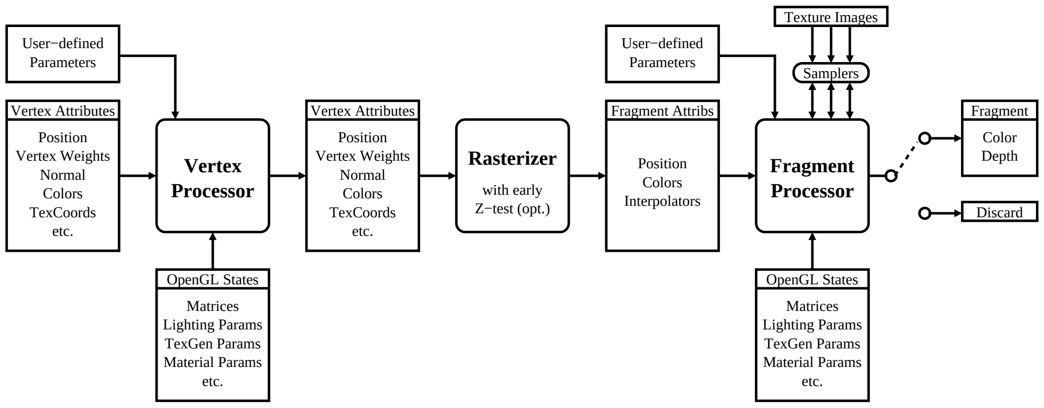

Figure 2.3: *The programmable OpenGL Pipeline in detail.*

At a more detailed level, the vertex and fragment processing parts of the pipeline look as depicted in figure 2.3. Both vertex and fragment processors can access various user-defined parameters that are constant during a glBegin/glEnd pair (see [SA]). Over that, certain OpenGL states are accessible from within each program. These states are needed, since, when enabled, the extended pipeline completely bypasses the standard pipeline. The consequence is that for example transformations to world and screen space have to be performed manually by the vertex program.

Both the vertex and fragment processing units operate on attributes. These are similar to usual processor registers, some being read- or write-only, with the exception that they are written by the user for each vertex or fragment respectively.

The vertex processor takes one set of input vertex attributes and transforms it to a set of output vertex attributes in a way described by the vertex program. For each input vertex exactly one output vertex is generated, there is no way to spawn or to delete vertices, though each vertex may be moved outside the viewing frustum, effectively discarding it when clipping occurs.

During rasterization the set of output vertex attributes becomes converted to a series of fragment attributes. These include fragment position, colors and the so called *interpolators*. The term interpolator refers to any per vertex entity that is interpolated across rendering primitives. This includes the classical texture coordinates, but since the user chooses the semantics of such interpolators they are not limited to just texture coordinates. Current hardware supports up to 8 user-definable interpolators.

The fragment processor subsumes texture fetching and texture compositing of the fixed function pipeline, transforming fragment attributes to a RGBα output color and optionally a depth-value. A fragment program may even choose to discard the fragment based on custom computations. If a fragment program chooses to modify the input fragment depth value, the early depth test that is otherwise performed during the rasterization step is not available.

The classical notion of texture units has proven itself to be too restrictive in the presence of fragment processors, and has been consequently substituted by the notion of *samplers* and *texture images*. Current hardware supports up to 16 texture images that may be accessed by sending texture coordinates to a sampler. This also releases hardware designers of the burden to implement exactly the same number of samplers as there are texture image units. Over that it is a logical improvement of dependent textures, as texture coordinates may be arbitrarily computed - including results from earlier texture fetches.

Limitations of Programmable Graphics Hardware

While the model of having vertex and fragment processors is very powerful, it has some limitations and restrictions that are worth noting.

First, everything that was performed automatically in the fixed function pipeline has now to be programmed by hand. This is only a minor restriction, since programmability offers a lot more flexibility, but it requires some of the instruction slots to be filled with standard transformations.
Second, the length of both vertex and fragments programs is restricted. Vertex

programs currently support up to 65536 instructions, fragment programs up to 1024, either one depending on vendor specifications.
Third, only vertex programs support loops by now. However, these may be unrolled by the driver before translating them to native instructions.
Fourth, branching and conditionals are supported by both vertex and fragment programs, but usually all branches will be evaluated. This also includes that fragment programs do not terminate when the current fragment is being discarded, but are instead fully executed.
Fifth, development of vertex and fragment programs in the assembly language itself is a combinatoric nightmare and very prone to errors. It is thus an important direction to develop a high level shading language. For about two years nVidia performs very active research in this area that led to the development of the *Cg Shading Language*. Cg was inspired by DirectX's HLSL[3] and is itself an influence for GLslang[4]. The Cg and GLslang projects are still under active development [CSF, nVia, OGLa].
Sixth, the alpha test will not be hardware accelerated when using fragment programs. The discard option has to be used instead.
Seventh, access to neighboring vertices and fragments is non-trivial due to pipelining issues.

Besides these general restrictions other vendor specific limitations may exist. For example the ATi Radeon 9700 has a fragment instruction limit of 64. This limit refers to native instructions, however, and a single assembler instruction may be mapped to several native instructions. On the other hand the restriction was alleviated on the Radeon 9800 model. ATi cards also have a *texture indirection* limit of 4. A texture indirection is any texture access in a chain of dependent texture fetches. A single texture fetch has a texture indirection level of 1. These vendor specific limits can be queried along with the program requirements in order to determine if a program will be executed hardware accelerated.

2.3 Scalar valued Volume Data

Scalar valued volume data can be seen as sampling of a three-variate function: $\Phi_s : \mathbb{N}^3 \rightarrow \mathbb{R}$. In the case of a regular grid serving as sampling domain, the volume is partitioned into cubical *voxels*[5] that contain one scalar sample at each corner[6]. The respective samples can thus be accessed at discrete positions

[3]High Level Shading Language.
[4]GL shading language, part of the OpenGL 2.0 proposals.
[5]The term voxel is an artificial abbreviation for "volume element" in analogy to "pixel".
[6]By convention, others such as staggered grids may assign one sample per voxel.

$(i, j, k) \in \mathbb{N}^3$. Such data can be either acquired by measurement using computer tomography (CT), positron emission tomography (PET) or magnetic resonance imaging (MRI) devices, or by physical simulation such as computational fluid dynamics (CFD). To approximate values inside each voxel, an interpolation kernel κ is needed. Usually κ will be a nearest neighbor or trilinear filter. The original data $\Phi(X)$ can then be reconstructed approximately by $(\kappa \star \Phi_s)(X)$.

For scalar valued data the two most common options are either to render the entire volume semi-transparently, assigning an opacity value o to each node depending on properties either known a priori or extracted from the data itself, or to render a so-called *iso-surface*, i.e. to render the set of points $\{X : \Phi_s(X) - s = 0\}$ and discard others [Bli82]. The parameter s is called the *iso-value* corresponding to the iso-surface. The first method is a strict superclass of the second, since a binary opacity function of the form

$$o\left(\Phi(x)\right) = \begin{cases} 1 & \text{if } \Phi(X) - s = 0 \\ 0 & \text{else} \end{cases} \tag{2.1}$$

can be applied that sets all points on the iso-surface to maximum opacity, while all others remain fully transparent. Application of such an opacity function can be extended to the notion of a *transfer function*. A transfer function $\tau : \mathbb{R} \rightarrow$RGB$\alpha$ is commonly defined to map any given scalar sample to a full color vector, where $\alpha \equiv 1 - o$ is called the *transparency*. During rendering, the transfer function is evaluated at each sample, and the resulting color is substituted for the sample. Since the data is reconstructed using an interpolation kernel, there are two possible ways to evaluate the transfer function. These are called *pre-shaded* or *post-shaded*, depending on whether the transfer function is evaluated first:

$$\mathrm{RGB}\alpha_{pre}(X) = \left(\kappa \star \tau(\Phi)\right)(X) \tag{2.2}$$

or the interpolation kernel:

$$\mathrm{RGB}\alpha_{post}(X) = \tau(\kappa \star \Phi)(X) \tag{2.3}$$

In the case of using a nearest neighbor filter for κ, these functions converge to a single one. Usually a post-shaded application of τ results in better visual fidelity, as interpolation errors can be corrected to some extend by a cleverly chosen transfer function.

Designing an appropriate transfer function can be a tedious task, and it is regarded very important by the visualization community that changes to the transfer function can be applied interactively in order to speed up the design process.

2.4 The Volume Rendering Equation

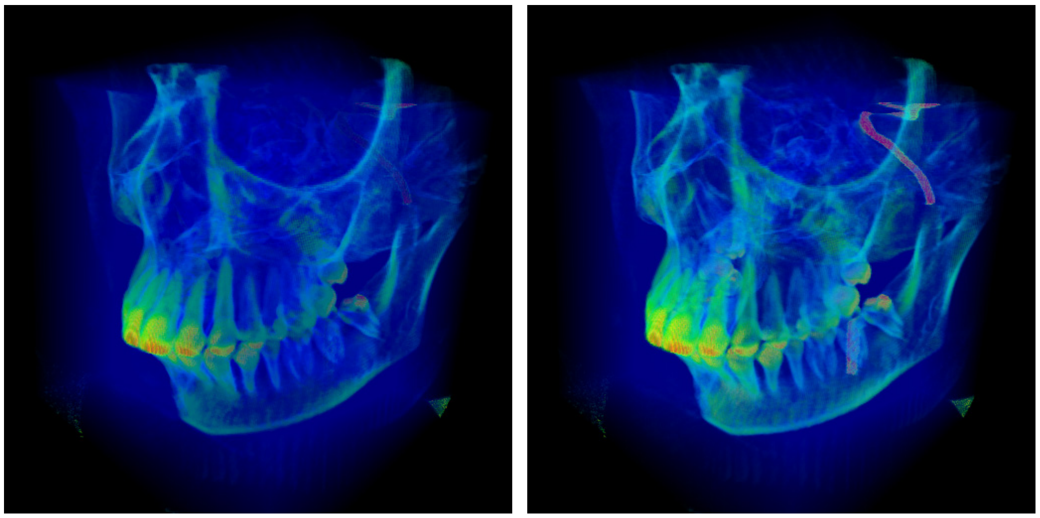

Figure 2.4: *Over versus Under Operator. The left image of the skull dataset was rendered using back to front ordering (Over operator), while the right one was rendered using front to back ordering (Under operator). Both images were made using 400 slices and the same transfer function. Obviously more structures are visible in the right image.*

Once color and opacity is computed for each voxel, these values have to be projected and composited in order to calculate an image on the viewing plane. This is done by solving the *Volume Rendering Equation* that was derived by Kajiya based on physical transport theory in [Kaj86]. Under the assumption that scattering can be neglected, the radiance can be expressed for the post-shaded case as

$$L(\lambda_{end}) = \int_0^{\lambda_{end}} \tau_c(\Phi(X(\lambda))) \cdot e^{-\int_0^{\lambda} \tau_o(\Phi(X(\lambda')))d\lambda'} d\lambda \tag{2.4}$$

where

X: $\{X\}$ describes a ray that is parametrized by λ.

Φ: The reconstructed data, $\Phi(X) = (\kappa \star \Phi_s)(X)$.

τ_c: The color part of the transfer function, $\tau_c : \mathbb{R} \rightarrow$ RGB.

τ_o: The opacity part of the transfer function, $\tau_o : \mathbb{R} \rightarrow \alpha$.

More general optical models can be found in [Max95]. Approximating equation 2.4 by a Riemann sum yields $\left(e^{\sum \cdots} \equiv \prod e^{\cdots}\right)$

$$L(\lambda_{end}) = \sum_{i=1}^{n} \tau_c(\Phi(X_i))\Delta X \cdot \prod_{j=1}^{i} e^{-\tau_o(\Phi(X_j))\Delta X} \tag{2.5}$$

Substituting

$$\begin{aligned} \alpha_i &\equiv 1 - e^{-\tau_o(\Phi(X_i))\Delta X} \\ C_i &\equiv \tfrac{\tau_c(\Phi(X_i))}{\alpha_i}\Delta X \end{aligned} \tag{2.6}$$

in equation 2.5 yields

$$L(X) = \sum_{i=1}^{n} \alpha_i \cdot C_i \cdot \prod_{j=1}^{i} 1 - \alpha_j \tag{2.7}$$

By traversing each ray $X(\lambda)$ back to front, this results in the *Over Operator* [PD84]:

$$C_{n+1} = \alpha_{in} \cdot C_{in} + (1 - \alpha_{in}) \cdot C_n \tag{2.8}$$

where α_{in} and C_{in} refer to transparency and color of the current ray segment. If the ray is traversed in front to back direction, the *Under Operator* is obtained:

$$\begin{aligned} C_{n+1} &= (1 - \alpha_n) \cdot \alpha_{in} \cdot C_{in} & +C_n \\ a_{n+1} &= (1 - \alpha_n) \cdot \alpha_{in} & +\alpha_n \end{aligned} \tag{2.9}$$

Krüger notes in [Krü02] that if C and α are each quantized using 8 bits, the Over and Under Operators no more yield the same result. This is due to the fact that the Under Operator accumulates quantization and roundoff errors at the back end of the volume, while the Over Operator accumulates its inaccuracies at the front. Another reason for preference of front to back compositing is the fact that it uses a higher precision, as an alpha buffer is utilized. Since no term α_n occurs in equation 2.8, the Over operator can not take advantage of an alpha buffer. See figure 2.4 for a comparison.

2.5 Raymarching

In the last section the Volume Rendering Equation was discretized and the Over and Under operators were derived. These describe how a single ray segment contributes to the final image. Rendering the volume can now be reformulated as the task of evaluating either one of these operators. The simplest solution is called raymarching, as formulated for volume densities in [KvH84]. Raymarching is very similar to raytracing. But instead of discretizing each ray only at ray/geometry intersection points, the presence of participating media requires each ray to be discretized based either on some error metric while accumulating $L(X)$, or on a fixed step size. While raymarching itself is relatively slow, it yields superior image quality, and can be significantly sped up, especially when rendering iso-surfaces. This was impressively demonstrated in [WS01, Sev03], where texture mapping hardware was applied to speed up searches for ray/volume and ray/iso-surface intersections. In addition, raymarching is not necessarily restricted to the Over and Under operators that effectively solve the Volume Rendering Equation using a simple Euler integrator. Since the integration step of the raymarching algorithm is usually executed on the CPU, arbitrary numerical integration schemes can be applied, offering good potential for quality/speed tradeoffs. Recent graphics processing units (GPUs) have just reached a level of programmability that allows for performing the entire raymarching algorithm on the GPU. A description and implementation of such a technique can be found in [nVib].

2.6 Hardware Assisted Volume Rendering

Most of the time interactive frame rates are more important than excessively high image fidelity. Hence it is not surprising that a lot of research was done in order to speed up the rendering process. The probably most promising direction is *slice-based direct volume rendering* [CCF94, WVGW94, CN93, WE98]. This class of approaches is based on the fact that if the Over or Under operator is used to obtain images, ends of ray segments with a fixed step size lie either on concentric, equidistant shells around the camera position [7] or equidistant planes[8]. Either case requires resampling the data on these intermediate slices by convolving the sampled data with an appropriate interpolation kernel κ. If only lower order interpolation kernels are needed, this can be efficiently done using either 2D or 3D texturing hardware.

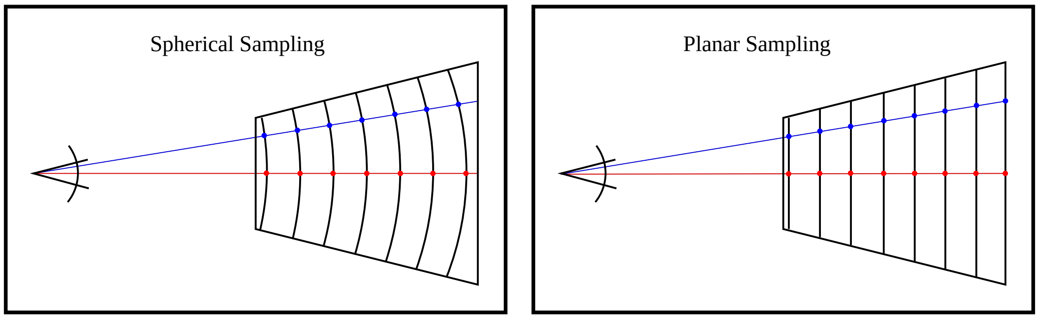

Figure 2.5: *Spherical versus Planar Sampling. Only spherical sampling results in equal intervals on the blue and red rays.*

The shells needed to apply perspective correct projections are expensive, since

[7]If a perspective corrected projection is applied.
[8]If an orthographic projection is applied.

they have to be triangulated[9], possibly resulting in a large geometry processing overhead. While volume rendering speed is clearly bound by the available fill-rate, it is still a very undesirable effect. On the other hand, perspective correction is a very important feature when interactively changing the camera position, because it gives an additional depth cue. As a consequence it has become common practise to approximate the shells by slices even when perspective correction is applied (see figure 2.5).

2.6.1 Rendering using 2D Textures

If only 2D textures are supported on the target platform, the rendering algorithm bears some resemblance of image based rendering. The sampled volume is represented by three axis-aligned stacks of 2D textures, each one containing slices parallel to the YZ, XZ and XY planes respectively. During rendering, the stack whose normal forms the smallest angle with the viewing direction is chosen, rotated, and its slices are sorted either back to front or front to back depending on which compositing operator is to be applied. Each slice is then rendered as a texture mapped quad and blended together with previous ones. The advantage of this method is its sheer speed, since the gaming industry has successfully enforced the development of efficient 2D texturing hardware during the last years. The drawback is that valuable texture memory is wasted, since three copies of the volume have to be stored. In addition artifacts that are due to the switching of stacks can be observed [Krü02], and trilinear interpolation requires additional slices [RSEB$^+$00].

2.6.2 Rendering using 3D Textures

If 3D textures are supported, the rendering algorithm becomes a lot more elegant. The 3D texture contains the entire sampled data set, and can be rotated to arbitrary positions. The 2D image is then constructed by blending a fixed number of textured *clip polygons* together (see figure 2.6). These clip polygons are obtained by clipping planes parallel to the viewing plane against the boundaries of the volume (usually a cube). The resulting polygons are convex, and can thus be efficiently triangulated and rendered as a triangle fan[10]. While it is possible in general to calculate these clip polygons on the graphics hardware, a CPU-based approach yields

[9]Virtually all graphics hardware breaks rendering primitives down to triangles.

[10]A triangle fan refers to a series of triangles connected in a special manner. By removing connectivity redundancies fans can rendered very efficiently.

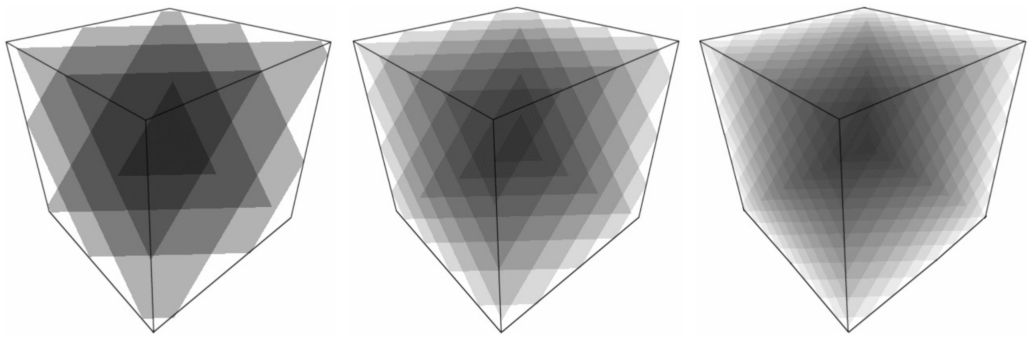
Figure 2.6: *Rendering using 3D textures. From left to right, 5,10 and 20 clip polygons are blended together with constant opacity.*

better performance [Krü02]. Using 3D textures has several advantages. First, it does not waste any texture memory. Second, the resulting image fidelity is higher when compared to a 2D based approach. Third, real trilinear interpolation can be used without requiring additional slices. The drawback is that 3D texture mapping is usually slower than 2D texture mapping by about 20% to 40% on average.

2.6.3 Applying the transfer function

Application of the transfer function can also be realized using graphics hardware. In the OpenGL API several extensions exist that allow for both post-shaded and pre-shaded transfer functions. This is very important, since evaluating the transfer function on the CPU would result in wasting performance and memory, as each voxel had to contain a RGBα-vector instead of a single opacity. On nVidia hardware pre-shaded application is possible via so called *paletted textures*. Paletted textures store an index into a 1D color lookup table for each voxel. These lookup tables are currently restricted to 8bit indices. The hardware then fetches colors from that lookup table in advance to filtering. While being virtually free when it comes to rendering performance, paletted textures are supported via the EXT_paletted_texture extension that is not available on all hardware[11]. On all cards supporting at least pixel shaders 1.3[12] post-shaded transfer functions can be realized using so called *dependent textures*. Dependent textures use the result of a previous texture fetch as texture coordinates, hence the name. With the recent advent of higher texture and pipeline precision of up to IEEE 32bit float, dependent textures are no longer limited to 8bit indices, allowing for higher resolution post-shaded transfer functions. In either case only a small color table has to be sent to

[11]For example it is not supported by current ATi GPUs.
[12]All DirectX 8.0 compliant cards such as nVidia since GeForce3 and ATi since Radeon 8500.

the graphics subsystem when the transfer function was changed, thus providing interactivity. The advantages of multi-vendor support and high precision make post-shaded transfer functions the first choice in state-of-the-art volume rendering. They have been successfully applied even for high quality hardware assisted rendering in [EKE01].

Chapter 3

Compression Methods for Volumes

3.1 Introduction

In this chapter several compression schemes will be presented and the choice of vector quantization as a general purpose compression tool will be motivated. At the current time, only the s3tc standard is supported natively by most hardware vendors, though vector quantization was supported by the now discontinued PowerVR2 chip [Pow], that was the graphics core of SEGA's Dreamcast. All other methods rely on a decoding step on the CPU or, with the recent advent of fragment programs, on the GPU. Since the complexity of algorithms that can be executed on the GPU is currently restricted, a careful analysis of the decoding step is also necessary.

3.2 S3 Texture Compression Standard

The demand for saving valuable texture memory was already recognized a while ago, and consequently current consumer class graphics hardware supports decoding of compressed textures. This lead to the S3 texture compression standard (s3tc), that eventually became a part of the DirectX API and is available via the GL_EXT_s3tc and GL_ARB_texture_compression extensions to the OpenGL API [ARB]. The intention was to offer a simple compression scheme to fit either more or larger textures into texture memory. However, a significant degradation in image fidelity can be easily observed. This lead to the consequence that most applications chose either to use texture compression and higher texture resolutions or to use no texture compression at all. From a numerical point of view the S3 texture compression scheme is catastrophic, and accordingly it is no secret in the community that it is not suited at all to compress normal maps (see figure 3.1).

S3tc implements a special form of so called *block truncation* codes. During the encoder process, the image is decomposed into blocks of $4{\times}4$ pixels. S3tc supports five formats specifying how these pixels are treated.

DXT1 Format

The DXT1 format supports compression of RGB-data with an optional binary α support. The 16 pixels of each block are interpreted as 3D-vectors in RGB-space, and a straight line is fitted to them. On this line two points are chosen and stored as 16bit values C_0 and C_1 (565-RGB). Now each block may be stored using one of two modes that can be identified by either[1] (a) $C_0 > C_1$ or (b) $C_0 \leq C_1$.

In mode (a), four static interpolation weights $\left(0, 1, \frac{1}{3}, \frac{2}{3}\right)$ are used to describe a total of four colors C_0 through C_3 on the line $\overline{C_0C_1}$. For each pixel X in that block, the color C_i is chosen that minimizes the metric $\delta(X, C_i) = \|X - C_i\|_2^2$, and the respective index of that color is stored as 2bit value per pixel. In this mode if alpha support is enabled the alpha value is always set to 1.

In mode (b), three static interpolation weights $\left(0, 1, \frac{1}{2}\right)$ are used to describe three colors C_0 through C_2 on the line $\overline{C_0C_1}$. C_3 is set to black. If alpha is supported it is set to 0 if and only if C_3 was encoded, otherwise it is set to 1.

Obviously each block is stored using 64bits, and consequently the compression ratio is 6:1 for RGB and 8:1 for RGBα.

DXT3 Format

The DXT3 format supports only compression of RGBα-data. Compression of the RGB part proceeds as described for the DXT1 format. The alpha part is stored in 64bits per block as 4bit per pixel uncompressed alpha values. The compression ratio is 4:1.

DXT5 Format

This format also supports only RGBα-data. The RGB part is compressed as described for the DXT1 format. The alpha part is compressed similar to the RGB part. First two 8bit alpha values are selected and stored as α_0 and α_1. Each alpha block may be stored using either mode (a) $\alpha_0 > \alpha_1$ or mode (b) $\alpha_0 \leq \alpha_1$.

[1]Interpreting C_0 and C_1 as 16bit scalar value each.

In mode (a) eight interpolation weights $\left(0, 1, \frac{1}{7}, \frac{2}{7}, \frac{3}{7}, \frac{4}{7}, \frac{5}{7}, \frac{6}{7}\right)$ are used to describe values α_0 through α_7 on the line $\overline{\alpha_0\alpha_1}$. For each pixel the closest value is chosen and encoded using 3bits.

In mode (b) the interpolation weights are $\left(0, 1, \frac{1}{5}, \frac{2}{5}, \frac{3}{5}, \frac{4}{5}\right)$. α_6 is set to 0 and α_7 is set to 1. Each value is also encoded using 3bits.

The compression ratio for the DXT5 format is the same as for the DXT3 format.

DXT2 and DXT4

The DXT2 and DXT4 formats are the same as the DXT1 and DXT3 formats, respectively, but store premultiplied alpha values. They are currently not supported by the EXT_texture_compression_s3tc and ARB_texture_compression extensions.

Conclusions

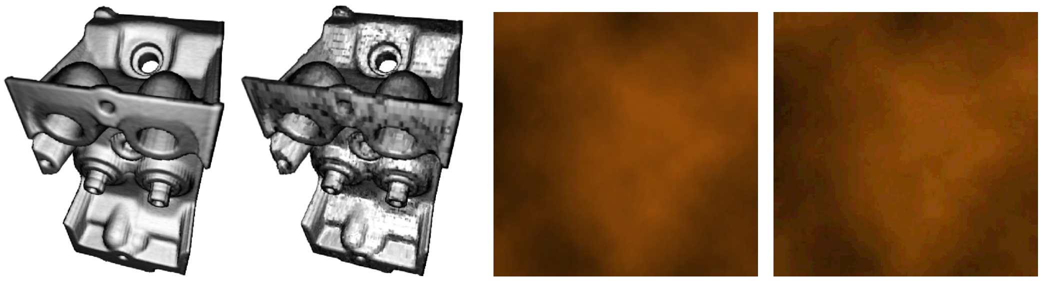

Figure 3.1: *S3tc Examples. The two images to the left show a Blinn-Phong shaded iso-surface of the $256^2 \times 128$ engine dataset. For the first one the engine was rendered uncompressed, while for the second one gradients were stored in a RGB texture and compressed using the s3tc DXT1 format. The two images to the right show a smoothly varying texture. The left one is uncompressed while the right one is encoded using s3tc DXT1.*

As a consequence to the linear fitting process, data not closely scattered along a line will not be reproduced correctly. This is especially true for unit normals (see figure 3.1), since in that case the data is distributed on a sphere. However, s3tc was designed with 2D texture maps in mind, for which the method works quite well, as long as textures show no smooth color gradients. A serious drawback is the fact that only very specific compression ratios are supported, making reasonable fidelity/compression tradeoffs impossible. The advantage of s3tc is that encoding can be done on the fly by the driver, and is usually very fast. The decoding step is virtually free. It might even sometimes be faster to render from a texture in s3tc format since both blocking and compression lead to better cache coherences.

Though it seems intuitively clear how s3tc can be extended to 3D[2], a series of problems arise. First, this extension to 3D textures is only supported through the NV_texture_compression_vtc, that is no multi-vendor extension. Even worse, since all s3tc modes require at least an RGB mode, they are completely unusable for the compression of scalar volume data. Though when extracting iso-values a lighting model is commonly evaluated and thus gradients are needed, the compression artifacts are so severe that it has become common practise to offer s3tc at most as an option in current volume visualization tools.

3.3 Transform based Methods

Transform based methods derive their name from the fact that prior to quantization the input data is transformed from one space into another. The most popular transforms are orthonormal and linear, making a matrix formulation of the transform process very simple:

$$\begin{array}{rlrl} \text{forward} & \text{transform :} & \Theta & = \mathcal{A} \cdot X \\ \text{inverse} & \text{transform :} & X & = \mathcal{A}^t \cdot \Theta \end{array} \tag{3.1}$$

where $\mathcal{A}$ is the matrix of transform coefficients. Since $\mathcal{A}$ orthonormal, $\mathcal{A}^t = \mathcal{A}^{-1}$. X is the vector containing the input sequence, while Θ contains the transformed sequence.

The motivation of such a transform is to achieve as much *energy compaction* as possible. The energy ξ of a real-valued input sequence is defined to be the summed squares of the sequence:

$$\xi := X^t \cdot X \tag{3.2}$$

However, orthonormal transforms are energy preserving, that is

$$\xi' = (\mathcal{A} \cdot X)^t \cdot \mathcal{A} \cdot X = X^t \cdot \mathcal{A}^t \cdot \mathcal{A} \cdot X = X^t \cdot X = \xi \tag{3.3}$$

Hence the efficacy of these transforms does not depend on the total energy ξ, but on how ξ is distributed in the sequence. Thus a meaningful measurement of the energy compaction of a given transform can be defined as the ratio of the arithmetic to the geometric mean of the variances σ_i^2 of the transformed sequence.

[2]Namely by using 4 slices of 4×4 texel blocks.

This entity is called the transform coding gain g_{TC} [JN84]:

$$g_{TC} = \frac{\frac{1}{n}\sum_{i=1}^{n}\sigma_i^2}{\left(\prod_{i=1}^{n}\sigma_i^2\right)^{\frac{1}{n}}} \tag{3.4}$$

The transform coding gain does not depend solely on the choice of the transform, but also on the characteristics of the input sequence to be transformed. The Karhunen-Loéve transformation (KLT) [AR75] can be shown to result in the best energy compaction of all linear transforms, minimizing g_{TC} for any input sequence [JN84]. Obviously the KLT needs to perform some analysis of X before it can be applied in order to adapt to the characteristics of the data. More precisely, the KLT matrix is the eigenvector basis of the auto-covariance matrix of X. These terms will be made clear in section 4.2.1.

Other popular transforms include the discrete cosine transformation (DCT) and discrete sine transformation (DST):

$$\begin{aligned} [A_{DCT}]_{j,k} &:= \begin{cases} \sqrt{\frac{1}{n}}\cos\frac{(2k)(j-1)\pi}{2n} & j = 1, k = 1, 2, \ldots n \\ \sqrt{\frac{2}{n}}\cos\frac{(2k)(j-1)\pi}{2n} & j = 2, 3, \ldots, n, k = 1, 2, \ldots n \end{cases} \\ [A_{DST}]_{j,k} &:= \sqrt{\frac{2}{n+1}}\sin\frac{jk\pi}{n+1} \quad j, k = 1, 2, \ldots n \end{aligned} \tag{3.5}$$

Both can be seen as an approximation to the KLT, the DCT assuming high auto-correlation of X and the DST assuming low auto-correlation of X. Often they are used as complementary transforms. Since the KLT can be shown to be optimal, it has to be motivated why DCT and DST are so popular. From equation 3.5 it is clear that both do not depend on the input data, and can be represented by fixed matrices for fixed dimension n. As consequence only the dimension and not the matrices have to be transmitted to the decoder, saving valuable bits. In addition there exist efficient evaluation schemes that exploit periodic recurrencies of the sine and cosine terms. They are very similar to the Fast Fourier Transform (FFT) [PTVF02], and need only $\mathcal{O}(n \log n)$ instead of $\mathcal{O}(n^2)$.

Conclusions

A general problem with transform based approaches is that it is impractical to compute a single $n \times n$ matrix for the entire sequence. Even if the resulting matrix is sparse, it will not necessarily be diagonal, preventing the transform from being computed in linear time. All practical transform methods thus split the input sequence into blocks of m neighboring input values, hoping that those blocks share similar characteristics. It can hence be argued that arbitrary input sequences are transformed into sequences of m-dimensional vectors. These vectors are then transformed, resulting in only a few components being significantly large[3]. The compression then proceeds by assigning more bits to larger coefficients than to smaller ones. The transformation has been successful if it tends to produced large coefficients at same components of the m-vectors, producing patterns of high energy content that can be compressed well.

Transform methods can be used as a de-correlation step for both lossless and lossy compression, though they are definitely more popular in the context of lossy compression. Especially the DCT and DST decompose the input sequence into bands of different wavelength, similar to the Fourier transform, making it very easy to employ psycho-visual models to accommodate for the fact that the human visual system is less suited to determine lots of small details. This is for example exploited in the jpeg image format [JPEG].

3.4 Vector Quantization

Vector quantization is a very vast field. Though this work is as much self-contained as possible, a more general introduction can be found in [Say00, GN98].

A vector-quantizer is commonly introduced as an *encoder mapping* $\alpha_C : \Re^n \to \Im$ and a *decoder mapping* $\beta_C : \Im \to \Re'^n$, where $\Re$ is the input alphabet[4] and $\Re'$ is the output alphabet of the quantizer. These alphabets need not to be identical, though it is appropriate for most applications to assume that $\Re \equiv \Re' \subset \mathbb{R}$. $\Im$ specifies some index set, usually a subset of $\mathbb{N}$, and C a codebook that may either be generated during the computation of the encoder or may be known a priori.

Obviously, α_C takes a n-dimensional vector as input and maps it to a single index from $\Im$, whereas β_C reverses this process, at least to some extent, since vector quantization is usually a lossy process. Restricting vector quantization to the case where no codebook is known a priori, encoding an ordered input set of vectors is the task of partitioning the input set with respect to α_C *and* to calculate a codebook C. To maintain as much fidelity as possible of the input data, the encoding process should also be designed to minimize the quantization error inherent to $\alpha_C \circ \beta_C$. This suggests treating vector quantization as a non-linear data fitting process to minimize the residual distortion with respect to some error metric $\delta : \Re^n \times \Re^n \to \Re$. The most common distortion metric used for data fitting

[3]This is the practical result of energy compaction.
[4]Since input and output will be represented discretely.

processes is the squared-distance metric $\delta(X,Y) \mapsto \|X-Y\|_2^2$, that provides an intuitive measurement of distortion. The residual distortion of an encoder/decoder pair is then given by

$$d = \sum_{X \in I} \delta(\alpha_C \circ \beta_C(X), X) \tag{3.6}$$

It has been known for long that vector quantization is an useful tool in many application areas, including computer graphics, and accordingly there exists a vast amount of literature dealing with theoretical issues as well as practical ones. However, computer graphics and scientific visualization applications limit the number of usable approaches in several ways. First, such applications are usually very time critical. Thus, the decoding step should be as fast as possible and it should take place on the graphics chip itself in best case, to save valuable bandwidth across the graphics bus and to increase cache coherences. Second, regarding the huge amount of data to be processed, the encoding step should be fast, too, since many applications are sensitive even to preprocessing time.

While the first criterion can be easily met by requesting that the encoder produces a fixed bitrate r, such that $|\Im| = 2^r$, several shortcomings of previous approaches become obvious in the context of the second criterion. These will be discussed in chapter 4.

Obtaining higher Dimensions

While encoding 3D-vectors in RGB-space is intuitive enough, it does not result in the desired fidelity. Encoding scalar values from volumetric data would even be less efficient, as a scalar quantizer would have been sufficient for this task. In contrast it is known from other coding schemes, for example those presented in section 3.3, that grouping adjacent pixels into blocks and treating each block as new input element can be very effective, as these pixels usually show high correlation. The same is true for adjacent voxels in the case of volumetric data. It is thus reasonable to block n^3 voxels together to form a new input vector of dimension n^3 in the case of scalar data and $3n^3$ in the case of RGB-data. Since for RGBα-data, such as pre-segmented RGB images, the RGB channels are usually uncorrelated with the α channel, these two have to be encoded separately.

By such dimension elevation, dramatically better fidelity can be achieved using the same bitrate. This is demonstrated in figures 3.2 and 3.3.
Figure 3.2 shows a 768×512 RGB picture of a parrot. For the middle image, 3D RGB-vectors were quantized using a codebook of 4 RGB-vectors, thus resulting in 2 bits per pixel. For the right image, 2×2 3D RGB-vectors were blocked together to form 12D-vectors that were quantized afterwards using a codebook of

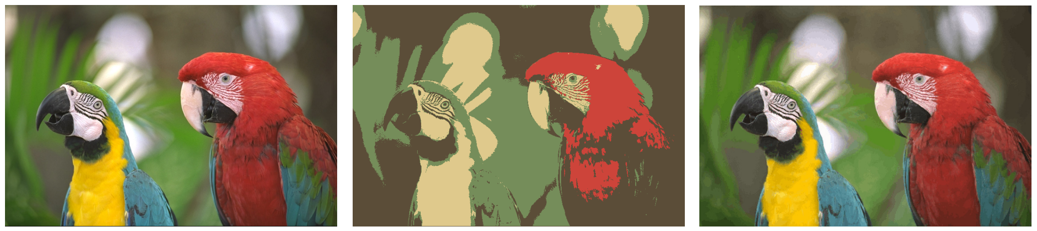

Figure 3.2: *Comparison between per Pixel and per Block Quantization (I). Original parrot image (left), compressed using 3D RGB-vectors (middle) and using 12D* 2×2 *RGB blocks (right). The bitrate was 24bpp (1152KB), 2bpp (96KB) and 2.06bpp (99KB) respectively.*

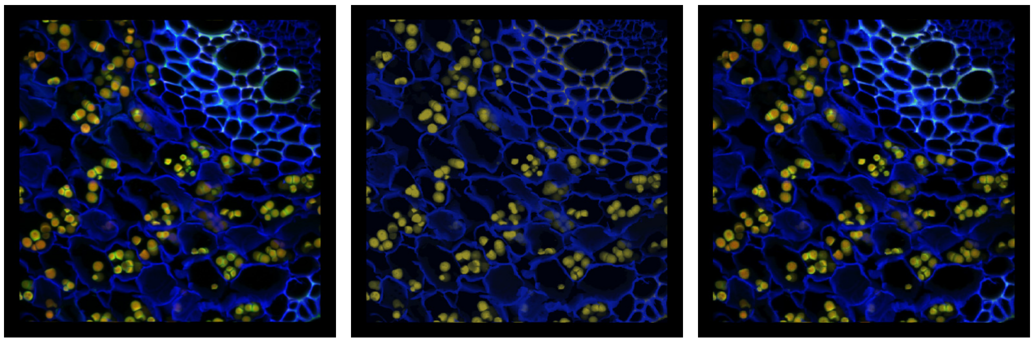

Figure 3.3: *Comparison between per Pixel and per Block Quantization (II). Original confocal microscopy scan (left), compressed using 4D RGB*α*-vectors (middle) and 32D* $2\times2\times2$ *RGB*α *blocks (right). The bitrate was 32bpp (32MB), 2bpp (2MB) and 1.01bpp (1MB) respectively.*

256×4 RGB-vectors, resulting in a bitrate of 2.06 bits per pixel.
Figure3.3 shows a $512{\times}512{\times}32$ RGB confocal microscopy scan. An α channel was generated by applying a smooth luminance-based transfer function. The α channel is thus correlated with the RGB channels and the dataset can consequently be compressed without the need to encode RGB and α channels separately. The resulting datasets were rendered using a slice based direct volume rendering approach. The middle image was quantized as 4D-vectors using a codebook with 4 entries. The right image was quantized by blocking $2\times2\times2$ voxels together to form 32D-vectors. A codebook of size 256×8 RGBα-vectors was used. The bitrates were 2bpp and 1.01bpp, resulting in a compression ratio of 31.75:1 for the right image.

Obviously the fidelity is in both cases dramatically better in the right image. Intuitively this can be explained by the fact that in the left image loss of information automatically means loss in color resolution, while in the right image spatial resolution can also be reduced. At a more theoretical level, this phenomenon may be seen as part of the famous Bucklew's high rate vector quantization mismatch [BW82]:

At high fixed bitrates the Zador rate/distortion theory [Zad66], characterizing the optimal tradeoff for fixed vector dimension and variable codebook size, converges with the Shannon rate/distortion theory characterizing the tradeoff for fixed codebook size and variable vector dimension. This means that generating high vector dimensions by blocking large groups of pixels together has (for high bitrates) the same effect as increasing the codebook size for vectors with fixed dimensionality. Since the parrot was not encoded at a high bitrate, the blocking scheme is superior to the other, but by increasing the bitrate they will eventually converge. As a historical remark, Zador's rate/distortion theory in some way gave birth to practical vector quantization, as it was believed before that vector quantization could only yield better results than scalar quantizers at high dimensions (e.g. several hundreds or thousands) - for which the computational burden of the encoding step becomes overwhelming compared to scalar quantizers.

Conclusions

Vector quantization offers a very fine fidelity/compression tradeoff. This was demonstrated by datasets compressed with ratios between 12:1 and 31.75:1. Over that, vector quantization can result in images with a fidelity comparable to s3tc at a lower bitrate, or in images showing dramatically better fidelity at the same bitrate. In addition vector quantization is general enough to handle nearly arbitrary data types: scalar-valued, vector-valued, multi-modal, *etc.*, as long as vector components are correlated one with another. Vector quantization was even used to compress normal maps and to perform efficient software per pixel-lighting in [TCRS20].

3.5 Hierarchical Methods

Hierarchical methods perform a decomposition of the data into discrete *levels of detail*. This is usually done by convolving the data with a smoothing and a differentiation filter kernel, resulting in a smooth and a detail part of the original dataset.

Smoothing data means to discard high frequency informations, so the smooth part can consequently be sub-sampled without further loss of information, resulting in a more compact representation. The differentiation kernel is designed such that the detail part still contains the information removed from the smooth part, thus making it possible to reconstruct the original data exactly. However, the detail part has a more compact energy spectrum, making it amenable to compression.

In the field of data compression, hierarchical methods are successfully used to de-correlate the data prior to the application of either lossless or lossy coding schemes. Over that they naturally offer very good opportunities for performance/fidelity tradeoffs during the encoding step, since entire levels of detail can be skipped depending on the desired reconstruction fidelity.

The most popular hierarchical decompositions are *wavelet transforms* and the *Laplace decomposition*. For reasons that will become obvious in section 3.6 wavelet transforms are not further discussed. Refer to [Say00] for a good introduction to the applications of wavelet transforms in the field of data compression.

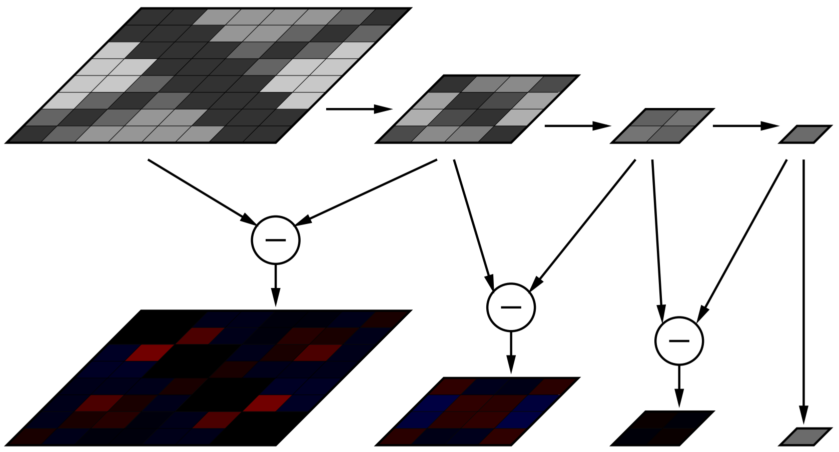

Figure 3.4: *Gaussian and Laplacian Pyramids in 2D. The Gaussian (top)is obtained by repeated application of the* $reduce_2$ *operator, while the Laplacian (bottom) is obtained by calculating the errors* $\Delta_s^{(n)}$ *(blue encodes positive values, red negative ones).*

The Laplace decomposition was successfully applied to volume compression in [GY95]. In the presence of data sampled on a regular grid $\Phi_s = \Phi_s^{(1)}(k_1, k_2, k_3)$ the Laplace decomposition is based on two discrete filter operations:

$$\Phi_s^{(n+1)} = reduce_m\left(\Phi_s^{(n)}\right)(K) := \sum_{k_1'=1}^{m}\sum_{k_2'=1}^{m}\sum_{k_3'=1}^{m} g_m\left(K'\right)\cdot\Phi_s^{(n)}\left(m\cdot K + K'\right) \tag{3.7}$$

where g_m is a normalized Gaussian or a box filter kernel with support m^3, and

$$expand_m\left(\Phi_s^{(n)}\right)(k_1, k_2, k_3) := \Phi_s^{(n)}\left(\left\lfloor\frac{k_1}{m}\right\rfloor, \left\lfloor\frac{k_2}{m}\right\rfloor, \left\lfloor\frac{k_3}{m}\right\rfloor\right) \tag{3.8}$$

Repeated application of $reduce_m$ yields the *Gaussian pyramid*

$$G_m = \left\{\Phi_s^{(i)}\right\}_{i=1}^{n} \tag{3.9}$$

The $expand_m$ operation may be seen as a pseudo inverse, since it reconstructs each next higher level in the Gaussian pyramid with a residual reconstruction error of $\Delta_s^{(n)} = \Phi_s^{(n)} - expand_m(\Phi_s^{(n+1)})$. The set of these $\Delta_s^{(n)}$ together with the smoothest level of the Gaussian pyramid form the *Laplacian pyramid*

$$L_m = \left\{\Delta_s^{(i)}\right\}_{i=1}^{n-1} \cup \left\{\Phi_s^{(n)}\right\} \tag{3.10}$$

Same levels of the pyramids have same resolutions, that is if $\Phi_s^{(1)}$ and $\Delta_s^{(1)}$ have a resolution of (r, s, t), then level j has a resolution of $(\lfloor \frac{r}{m^{j-1}} \rfloor, \lfloor \frac{s}{m^{j-1}} \rfloor, \lfloor \frac{t}{m^{j-1}} \rfloor)$.

This is an important fact, because a full pyramid will have $\log_m(\min\{r, s, t\}) + 1$ levels, and can be constructed in $\mathcal{O}(n \cdot \log_m n)$, where $n = r \cdot s \cdot t$. A full Gaussian and Laplacian pyramid for the 2D case can be seen in figure 3.4.

The reconstruction of data given by a Laplacian pyramid then proceeds by evaluating

$$\Phi_s = expand_{m^{(n-1)}}\left(\Phi_s^{(n)}\right) + \sum_{i=1}^{n-1} expand_{m^{(i-1)}}\left(\Delta_s^{(i)}\right) \tag{3.11}$$

Conclusions

The Laplace decomposition is very attractive as a data de-correlation method, since it can be computed very efficiently. Over that, it offers a rather intuitive level of detail structure that can be exploited for various multi-resolution approaches. As was demonstrated in [GY95], it is possible to assign less bits for higher frequency levels in the Laplacian pyramid than for lower frequency ones. Since the smooth levels contain less samples, such a scheme can result in impressive compression ratios. However, it is not clear how these bits have to be assigned. Possible solutions include scalar and vector quantizers for the case of lossy compression, as well as Huffman codes for lossless compressions. The assignment of bits, and thus the compression of data stored in a Laplacian pyramid representation, is further discussed in chapter 5.

3.6 Decoder Algorithms Reviewed

S3 Texture Compression

For the s3tc, the decoding algorithm can be executed entirely on the hardware. For each texel to be reconstructed the corresponding 4×4 block is fetched. The values C_0 and C_1 are decoded to 24bit RGB-vectors, and the interpolation specified by the 2bit index of the current texel is evaluated. The resulting vector is then applied as RGB color for this texel. The alpha value is decoded similarly.

Obviously, each 4×4 block may be cached, as it is very likely that the next texel rasterized also falls into the same block. Linear interpolation, however, may be expensive because up to 4 blocks have to be accessed for texels lying on a corner. On the other hand, at least two of the four blocks are good caching candidates, as the raster scan will sweep them in the near future. For the case of 3D textures the decoder must only be extended in the presence of a three-linear interpolation kernel, since up to 8 blocks have to be accessed. Due to the fact that the 3D vtc format does not encode $4 \times 4 \times 4$ 3D-blocks, but rather 4 4×4 2D-blocks, no other changes have to be made.

The specification of the EXT_texture_compression_s3tc makes no statement about MIP[5] mapping support, which might be expensive to support.

Transform based Methods

The problem with decoding transform coded data is that a m-block has to be accessed each time a texel is sampled. While this is also true for the s3tc, transform methods rely on a full matrix-vector product for each texel in the block to be executed, thus making it necessary to access *any* entry in the respective block while st3c only accesses *one* entry. This makes transform methods very expensive to decode in hardware, regardless of the actual coding scheme that introduces certain additional processing requirements.

It is thus no surprise that only very specific transforms are handled at all by the hardware. Namely only the DCT and inverse DCT are supported by the hardware, but only for video playback and recording purposes. They have not yet made accessible from within the OpenGL or DirectX APIs for general purpose transformation. It is over that not clear, how the video functionality of current graphic hardware interacts with the fragment processors that are needed to perform the

[5] Multum In Parvo. A filtering method to avoid aliasing during texture minification.

actual decoding of each block before the inverse transform occurs.

Regarding the computational burden that transform methods impose on fragment processors, it should be mentioned that a fast Fourier Transform (FFT) can be implemented using fragment processors [Hea]. The resulting algorithm needs $\log_2 \mathrm{width} + \log_2 \mathrm{height} + 2$ passes. Heart states that it runs in real-time on a 512×512 domain, but the algorithm is not applicable to volume compression, since the associated decoding cost would become overwhelming.

Vector Quantization

From a theoretical standpoint the decoding step of vector quantization is etraordinary easy. As is demonstrated in section 3.4, the decoder only has to apply β_C to the encoded sequence, essentially performing a single lookup into the codebook C. However, in the context of texture compression, dimension elevation is very important, resulting in high dimensional codebooks that are no longer representable by 1D RGBα textures. For example in the case of $2 \times 2 \times 2$ RGB block encoding, the codebook has a dimension of $8\times$RGB=24D. On the other hand, the codebook may still be loaded as a 2D texture, for example $256 \times 8\times$RGB for a codebook with 256 entries. This codebook has then to be accessed using 2D texture coordinates, one component of which specifies the quantization region, and the other one the relative component of the 24D-vector. This can be implemented efficiently using fragment processing capabilities of current hardware. The complete GPU-based decoding algorithm is discussed in chapter 5.

Hierarchical Methods

For hierarchical methods, the hierarchy itself can be decoded very efficiently by the straight-forward approach of sampling a series of differently sized textures (compare to equation 3.11). The cost of this procedure is only one texture fetch per level. But this does not yet make a statement about how the fetched coefficients have to be decoded. This was exactly the reason not to discuss wavelets as decomposition method further. Wavelets can be formulated as transform method, and consequently they require a full transform to be applied that was already argued to be too expensive to compute on the GPU.

3.7 Results

To conclude several results from this chapter, a decoding algorithm can only be ported to current GPUs if only a very compact support of the underlying com-

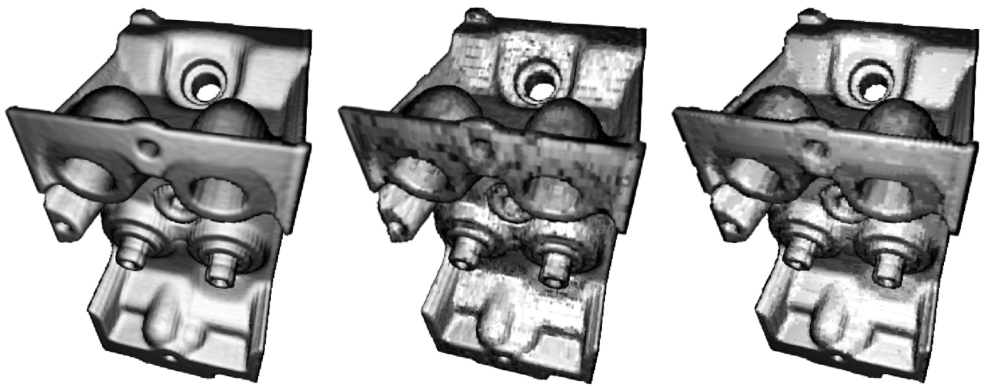

Figure 3.5: *Comparison between S3tc and Vector Quantization (I). Original* $256^2 \times 128$ *engine dataset (left) is 32MB (RGB*α*). Gradients were compressed using s3tc DXT1 (middle) to yield a total size of 12MB, and using a blocked 24D-vector quantizer (right) resulting in only 9MB.*

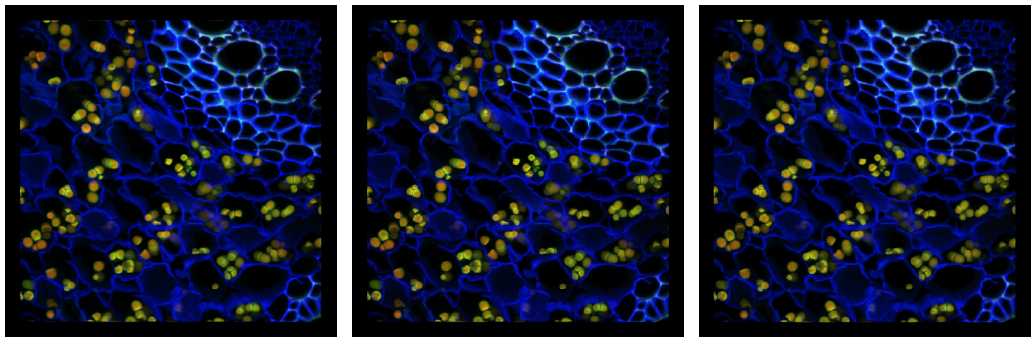

Figure 3.6: *Comparison between S3tc and Vector Quantization (II). Original confocal microscopy scan (left) is 32MB (RGB*α*). The entire volume was compressed using s3tc DXT5 (middle) to 8MB and using a 32D block quantizer (right) to 1MB.*

pression domain is needed in order to reconstruct a single texel. Over that the amount of arithmetic operations that have to be executed before the sample is reconstructed is strictly limited at the time being.

Thus, s3tc and vector quantized textures can be efficiently decoded on the GPU, while transform coded textures can not, as they require both a large support and lots of arithmetic instructions. In the case of hierarchical methods it depends on the choice of the actual compression method. If the compression method allows for GPU based decoding, then hierarchical methods might be as well decodable on the GPU, depending on the amount of levels present.

Since s3tc is too inflexible and totally unsuited to obtain high compression ratios, the use of a vector quantizer is mandated, optionally making use of hierarchical methods to obtain both better fidelity and better compression runtimes. Figures 3.5 and 3.6 demonstrate that vector quantization applied to $2\times2\times2$ blocks of voxels usually results in both better compression ratio and image fidelity when compared to s3tc.

Chapter 4

Making Vector Quantization Fast

4.1 The LBG Algorithm revisited

The compression method proposed in this work is based on a modified version of the *Linde-Buzo-Gray-Algorithm* (LBG) [LBG80]. This algorithm extends the work on scalar quantizers by Max [Max60] and Lloyd [Llo82] and is also referred to as the *Generalized Lloyd-Algorithm* (GLA). As a historical remark, the same algorithm had been used earlier by Hilbert [Hil77], who called it *Cluster Compression Algorithm*.

The LBG Algorithm

1. Start with an initial codebook $C = \left\{Y_i^{(0)}\right\}_{i=1}^{m} \subset \Re^n$. Let $I \subset \Re^n$ be the set of input vectors. Set $k = 0, d^{(0)} = 0$ and select a threshold ε.

2. Find quantization regions $V_i^{(k)} = \{X \in I : \delta(X, Y_i) < \delta(X, Y_j) \quad \forall i \neq j\}$, where $j = 1, 2, ..., m$.

3. Compute the distortion $d^{(k)} = \sum_{i=1}^{m} \sum_{X \in V_i^{(k)}} \delta\left(X, Y_i^{(k)}\right)$.

4. If $\frac{d^{(k-1)} - d^{(k)}}{d^{(k)}} < \varepsilon$ or $k > k_{max}$ stop, otherwise, continue.

5. Increment k. Find a new codebook $\left\{Y_i^{(k)}\right\}_{i=1}^{m}$ by calculating the centroids $Y_i^{(k)} = \frac{1}{\left|V_i^{(k-1)}\right|} \cdot \sum_{X \in V_i^{(k-1)}} X$ of each cell $V_i^{(k-1)}$. Go to (2).

The final output of the LBG algorithm is a codebook

$$C = \left\{ Y_i^{(k)} \right\}_{i=1}^{m} \tag{4.1}$$

and a partition

$$I = \dot{\bigcup}_{i=1}^{2^r} V_i^{(k)} \tag{4.2}$$

of I, where r is the fixed bitrate of the quantization. Each input vector X is then replaced by the index $i : X \in V_i^{(k)}$ of the associated *quantization cell* $V_i^{(k)}$.

Obviously, r bits are needed to encode i, but additional bits are needed to encode the codebook. Thus the compression ratio that can be achieved is limited from above by $\frac{r_{input}}{r}$:1, where r_{input} specifies the bits needed to encode each input vector. To achieve that limit it has to be assumed that the number of bits needed to encode the codebook may be neglected. The logical consequence is that the LBG offers most room for compression/fidelity tradeoffs at high input bitrates - a fact that is commonly exploited by blocking smaller vectors together, as already described in section 3.4.

4.1.1 Obtaining an initial Codebook

The initial codebook for step (1) of the LBG algorithm is usually obtained by means of a so called *splitting technique* originally described in [LBG80]. First, the centroid of the entire input set is placed as single entry into the codebook. Then a second entry is generated by adding a random offset to the first entry, and the LBG algorithm is executed until convergence. This procedure is repeated, until the desired bitrate is achieved. If the desired number of codebook entries is not a power of two, the final step does not perturb all codebook entries by a random offset, but only enough to reach that number of entries. To see that the quantization becomes better with each step, Sayood notes in [Say00] that by including the codebook from the last step it is guaranteed that the new one will be at least as good as the previous one.

Another approach is due to Equitz [Equ89] and is called the *pairwise nearest neighbor* (PNN) algorithm. In the PNN algorithm, each input vector is added to one large codebook, and those codebook-entries that are “closest” to another are subsequentially merged. To be more specific what “closest” means, as codebook entries will eventually represent many input vectors, the PNN algorithm produces clusters of input vectors that increase in size during the process. In each step, the two clusters that result in the smallest increase in distortion are merged. Any desired number of codebook entries can be reached very easy with this procedure, as the codebook size decreases by exactly one in each step.

4.2 Performance Issues

When talking of vector quantization, it may be regarded as common sense that the encoding process is very slow, in contrast to the decoding process which is virtually free. While being basically true, it is exactly this asymmetry that is useful in the context of hardware supported decoding. However, by carefully analyzing the performance requirements of the encoding step, several very costly subroutines can be clearly identified, and subsequently be substituted by more sophisticated ones.

4.2.1 Routines for the initial codebook

While being reasonably fast, the splitting technique suffers from the so called *empty cell problem*. Empty cells are the result of collapsing codebook entries during refinement steps. These entries are not detected automatically by the LBG algorithm, and as a consequence many codebook entries might be wasted. While it is possible to detect and delete such empty cells explicitly, this results in more LBG steps, because the number of codebook entries is not being doubled in each splitting step. One method to overcome this problem is to randomly select an input vector from the cluster with the largest distortion, but this does not guarantee stable entries during the rest of the process. Over that, even detecting empty cells may be expensive, as the entire input vector set has to be scanned for unused codebook entries in the worst case. This usually prohibits rapid encoding of reasonably sized input data, though the runtime is $\mathcal{O}(n \cdot \log_2\ m)$, where n is the number of input vectors and m the desired number of codebook entries.

The pairwise nearest neighbor algorithm suffers even more from nearest neighbor searches. As a matter of fact the runtime is $\mathcal{O}(n^2)$, where n is the number of input vectors. This entirely prohibits the use of the PNN algorithm with reasonably sized data, no matter how fast each search can be made.

4.2.2 The PCA Split

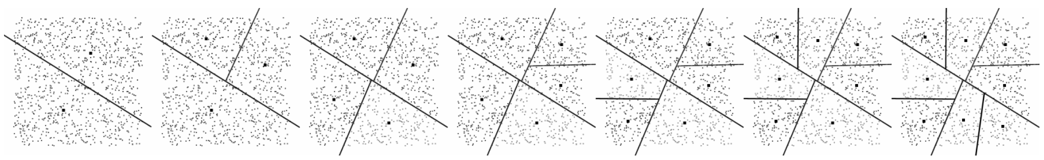

Figure 4.1: *PCA Splits. A series of PCA splits to obtain a first codebook.*

As a logical consequence, a technique very similar to the splitting technique was chosen to obtain a first codebook. But instead of random perturbation a principal component analysis (PCA) of the underlying input data was chosen to drive the split. This even allows to apply LBG steps as post-processing only, i.e. when a first codebook was already obtained. In contrast the original splitting technique relies on LBG steps even to obtain the first codebook.

The PCA split (see figure 4.1) allows to choose an optimal splitting plane in each step with regard to the variances of the disjunct clusters already produced. The exploitation of such a splitting process has been described in various applications ranging from data clustering to load balancing, and it is essentially the technique used in [PGK02] and [LKHS01] for the hierarchical clustering of point sets and scanned BRDFs. The method is quite similar to the Karhunen-Loéve transformation (KLT) introduced in section 3.3.

The splitting technique proceeds as follows.

Initializing the PCA Split

1. Generate a first quantization cell $V_1 = I$ containing the entire input set I.
2. Calculate a first codebook entry $Y_1 = \frac{1}{|V_1|} \cdot \sum_{X \in V_1} X$.
3. Calculate the initial distortion $d_1 = \sum_{X \in V_1} \delta(X, Y_1)$.
4. Initialize a double-linked *to-do list* by inserting the new *group* defined by $(d_1, Y_1, \Im_1 := \{i \in \Im : X_i \in V_1\})$ into this list.

The idea behind the to-do list is to keep it sorted in descending order with respect to the distortion d_j. In each following PCA step the group j with the largest distortion d_j is selected to be split further on. This *highest error* selection is actually a heuristic to predict the *highest gain* in fidelity that can be achieved by splitting the respective group. An accurate calculation of this gain would require to perform one split in advance, but the heuristic performs very good and is fast. In tests the heuristic yielded only insignificantly lower signal to noise ratios when compared to the highest gain selection, but was almost twice as fast.

The PCA Split

1. Pick the group j with the largest residual distortion d_j from the to-do list.
2. Calculate the auto-covariance matrix $\mathcal{M} = \sum_{i \in \Im_j} (X_i - Y_j) \cdot (X_i - Y_j)^t$.

3. Calculate eigenvector E_{max} corresponding to the largest eigenvalue of $\mathcal{M}$.

4. Split the original group into a *left* and a *right* group with following indices:

$$\begin{aligned}\Im_{left} &= \{i \in \Im_j : \ < (Y_j - X_i), E_{max} > \ < \ 0\} \\ \Im_{right} &= \{i \in \Im_j : \ < (Y_j - X_i), E_{max} > \ \geq \ 0\}\end{aligned}$$

5. Calculate new centroids Y_{left} and Y_{right} and new distortions d_{left} and d_{right}.

6. Insert the two new groups into the to-do list.

7. If number of groups equals 2^r, stop, else go to (1).

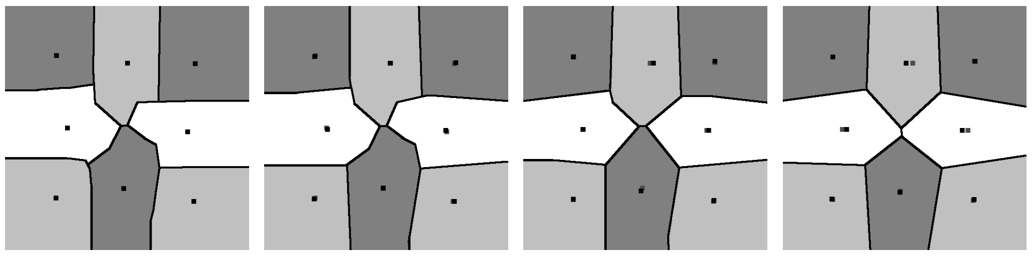

Figure 4.2: *LBG steps. A series of post-refining LBG steps applied to the codebook from figure 4.1. Current centroids are marked with a dark point, while the lighter ones mark the previous centroids. Only a few steps are needed to establish stable Voronoi regions.*

This procedure adds exactly one codebook entry per split. Since the splitting hyperplane passes through the old centroid, the sum of the new distortions will be small compared to the old one. This was the reason for choosing a double-linked list for the to-do list. Such a list allows for scanning the list starting with the last element, while offering constant access to the first element. Though there is some overhead associated with maintaining a double-linked list, the probability that only a small fraction of the list has to be scanned in each step is very high.

Once the splitting procedure terminates, the codebook along with the residual distortion can be directly obtained from the to-do list. This initial codebook can then be refined using a series of LBG steps (see figure 4.2).

Calculation of the largest eigenvector is best implemented using an iterative solver. Knowing from linear algebra that the sequence

$$E_{n+1} = \frac{\mathcal{M} \cdot E_n}{||\mathcal{M} \cdot E_n||_2} \tag{4.3}$$

will converge towards the largest eigenvector E_{max} very quickly if the initial vector E_1 is sufficiently close to E_{max}, a solid guess is necessary for E_1. Such a guess can be obtained by choosing the column[1]-vector of $\mathcal{M}$ that has the largest length. This can be motivated by interpreting the resulting guess E_1 as the vector E_0 of a standard basis for which $\mathcal{M} \cdot E_0$ results in the largest increase in length. Obviously that vector is the standardvector minimizing $< E_{max}, E_0 >$. This simple iterative scheme is easy to implement and outperformed Jacobi rotations [PTVF02] both in speed and precision. The iteration terminates when

$$\frac{E_{n+1} - E_n}{E_n} < \varepsilon \quad \text{or} \quad n > n_{max} \ , \tag{4.4}$$

generating an error in the second case.

The benefits of the PCA approach are manifold.
First, a cell with low residual distortion will never be split, since it is always appended to the end of the list. In particular this includes cells that contain only one data point and thus have a residual distortion of 0. This is an important fact, since empty cells are only generated by the PCA split if the input sequence is exactly represented by the codebook entries already generated. Second, if a cell is split it is divided into two sub-cells of roughly equal residual distortions. Third, applying some LBG steps as post-refinement to relax the centroids quickly converges to stable Voronoi regions. Fourth, during the examples that were done for this thesis, empty cells during the LBG post-refinement could not be observed. This is worth mentioning, since the PCA split only guarantees that the initial codebook is free of empty cells. But the PCA split places centroids always into densely populated clusters, thus heuristically avoiding empty cells. Fifth, the algorithm is extremely fast, because it avoids expensive LBG steps during the splitting process. It even retains the runtime of $\mathcal{O}(n \cdot \log_2 \ m)$ of the original splitting technique. Sixth, the algorithm is easy to implement and can handle any dimension. Moreover, since profiling sessions indicate that more than 80% of the total execution time are spend during distortion evaluations, the algorithm offers huge potential for further optimizations based on latest SIMD-technology, such as SSE/SSE2 and 3DNow. Though these optimizations were not implemented in the current version, using an optimizing compiler with SSE-support, such as the GNU C++ compiler [Gcc], showed an increase in speed of about 15%.

4.2.3 Fast Searches

Regardless of the mentioned improvements the total run-time of the algorithm is still dominated by the LBG steps, which are employed to relax the codebook

[1]$\mathcal{M}$ is symmetrical.

entries. On the other hand, these refinements are necessary, because they significantly increase the resulting fidelity. By carefully analyzing the LBG performance, it becomes obvious that the LBG algorithm spends nearly all of its execution time during nearest neighbor searches.

Nearest neighbor searches are originally performed taking the entire codebook into account. Obviously the procedure can be sped up by cleverly picking only relevant entries and performing a nearest neighbor search that is restricted to these entries. By further observing that it is very likely for each input vector either not to leave the associated quantization cell or to migrate only to neighboring cells, in the current implementation the search is restricted to a *k-neighborhood* of the original cell. All k-neighborhoods are pre-calculated for each LBG step and are then used during the nearest neighbor searches. To do so, mutual distortions are calculated for each pair of centroids, and references to the k nearest neighbors are established for each entry. Once the mutual calculations are done, the k nearest neighbors are found by means of a modified QuickSort algorithm. This algorithm is essentially the same to find the k^{th} largest element of a list. In contrast to the original QuickSort, this sorting procedure only recurses for the partition that overlaps the k^{th} entry, and consequently runs in linear time. Now the nearest neighbor search is performed as before, but restricted to the list of adjacent centroids. Because this procedure will converge to sub-optimal distortions for small ks, the search radius is continuously increased by some value whenever the rate of convergence[2] drops below 5%. In the current implementation the radius is increased by 1 to 5, depending on how much the rate of convergence dropped below 5% since the last step. This seems to be a reasonable choice, as it allows to save up to 85% of the time that is needed to perform an exhaustive search. In any case the algorithm may be terminated when a search radius k_{max} is reached or when an user defined distortion ε is reached (whatever happens first).

4.2.4 Partial Searches

For high vector dimensions, evaluating a single distortion term $\delta(X,Y)$ of the form $(X-Y)^2$ can easily consume a huge amount of computation time. Because each distortion term is basically a scalar product $< X - Y, X - Y >= \sum_{i=1}^{dim}(X_i - Y_i)^2$, and each term of that sum is positive, the computation can be stopped whenever the next contribution $(X_i - Y_i)^2$ leads to a higher value than an initial distortion $\delta(X, Y_{init})$. Obviously, the correct selection of Y_{init} is very important because it triggers the number of computations to be performed. Since it is very likely for a data point not to change the associated quantization cell, the

[2]The rate of convergence was defined in step 4 of the LBG algorithm: $\frac{d^{(k-1)}-d^{(k)}}{d^{(k)}}$

distortion is initialized with respect to the quantization cell the point was previously associated with. Even for 3D-vectors about 50% of the computations could be saved. But on the other hand, on current processors this does not necessarily mean a speedup, since a conditional has to be inserted at the innermost loop. As a consequence only for large dimensions (greater than 20), an actual speedup could be observed. These so called *partial searches* are enabled in the current implementation via an optimization flag set by the user at compile time.

4.3 Results

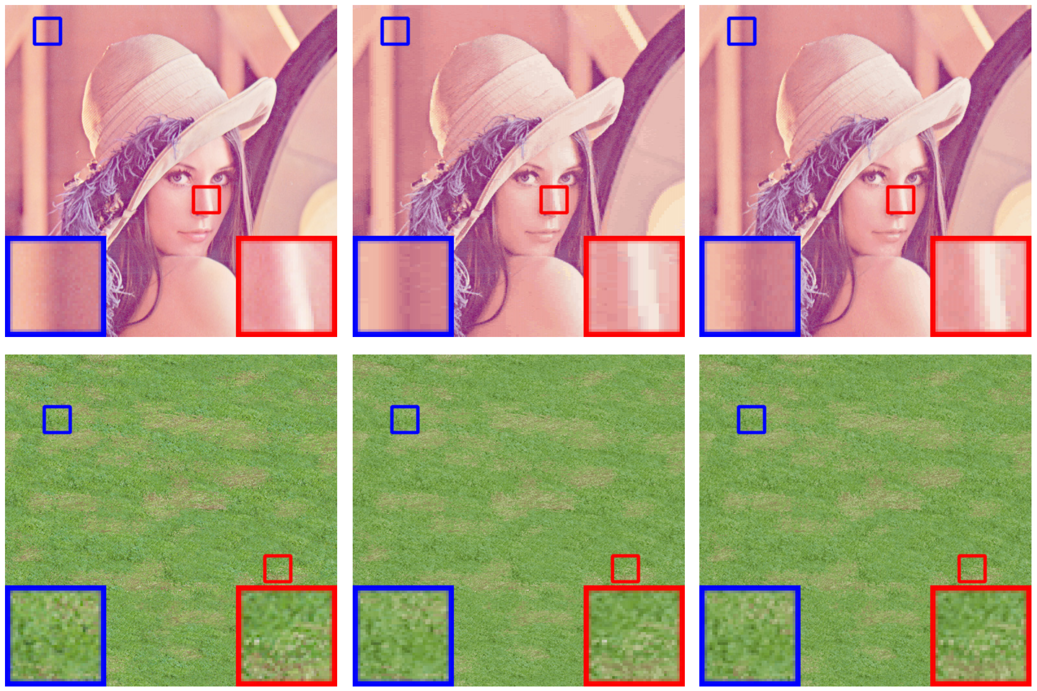

Figure 4.3: *Comparison to Gamasutra Quantizer. The Lena and grass images were obtained from [Iva]. From left to right: original image,* 2×2 *block encoding using the vector quantizer from [Iva] and the method suggested in this work.*

The implementation of a fast vector quantizer is the back bone of this work. All methods in the next chapter rely on a vector quantizer, that is fast and general enough to quantize data sets containing millions of vectors with arbitrary dimensionality.

The described method was implemented using C++, and was compared against previous implementations, such as the vector quantizer available from the Gamasutra webpage [Iva] and QccPack [Fow]. The Gamasutra Quantizer and the one presented here were compiled using gcc for CygWin [Red] on a Celeron 600MHz with SSE. Optimization for SSE was done automatically by gcc. The 2D Lena-image and the 2D grass texture were obtained from [Iva]. Both images have a resolution of 512^2 and 2×2 RGB-pixels were grouped in order to form 12D-vectors. The results, using a codebook with 256 entries, can be seen in figure 4.3. In the Lena image both zoomed areas show how much an increase of only one dB in the SNR can affect the visual fidelity, while the three grass textures are nearly indistinguishable.

The Gamasutra quantizer took about 128 seconds to quantize the grass texture, and 173 seconds to quantize the Lena image, while the method presented in this work took under 4 seconds in both cases. The signal to noise ratio that could be achieved with the Gamasutra quantizer was 30.10dB for the grass and 34.77dB for the Lena image. The implementation presented here produces signal to noise ratios of 30.13dB and 35.74dB respectively. It is over that worth noting that the Gamasutra quantizer produced codebooks containing empty cells in both cases. These cells are detected and consequently removed, at the expense of speed.

To compare the described method against the QccPack, a scalar 128^3 volume containing a distance volume of the famous Stanford Bunny was blocked to $2 \times 2 \times 2$ voxels in order to form 8D-vectors. Quantization using QccPack was performed on a R12000/R12010 processor running at 400MHz, and took about 22 minutes. The fast vector quantizer described in this thesis took under 35 seconds on a Celeron 600MHz. The SNR was nearly identical in both cases.

As a preliminary résumé, the vector quantization method described in this thesis is at least of a factor 30 faster than other general purpose implementations while producing at least the same fidelity. For some data sets the fidelity may be by far superior, as the LBG algorithm converges towards a local minimum, and is thus sensitive to the choice of the initial codebook.

Chapter 5

Hierarchical Vector Quantization

5.1 Introduction

Combining the results from the previous chapters, in this chapter the *L3VQ compression scheme* is presented. The L3VQ compression scheme is a rapid hierarchical vector quantizer that can be used to compress various data types. In the next section the encoding procedure is presented, while section 5.3 covers the basic decoding algorithm. Section 5.4 shows how decoding of L3VQ encoded volume data can be done on the GPU. The results of this chapter are discussed in section 5.5, while in section 5.6 extensions such as the application of L3VQ to RGB-data or 2D images are proposed.

5.2 L3VQ Encoding Algorithm

The L3VQ encoder for the case of scalar valued volume data is depicted in fig 5.1. All numbers in braces in the text refer to the circled numbers in that figure. The levels of the encoded pyramid are also referred to as L_3 (bottom), L_2 (middle) and L_1 (top) in analogy to MIPmap level enumeration.

First, the original volume Φ_s is decomposed into a Laplacian pyramid

$$L = \left\{ \Delta_s^{(1)}, \Delta_s^{(2)}, \Phi_s^{(3)} \right\} \text{ (see (1), enumerated from top to bottom)}$$

as described in section 3.5. To do so, the $reduce_4$ operator is applied using a 4^3 box-filter, resulting in the lowest resolution level $\Phi_s^{(3)}$. In [GY95] a 5^3 Gaussian kernel was used to perform the Laplace decomposition, but this is not necessary, since the unweighted average of the samples results in the optimal approximation with respect to the metric δ. The lowest resolution volume is then quantized to

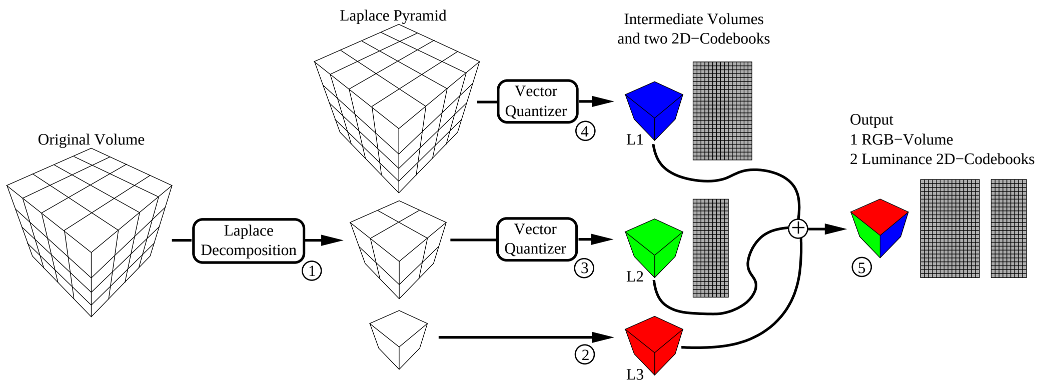

Figure 5.1: *The L3VQ Hierarchical Vector Quantizer. See text for description.*

8bits, yielding L3 (2).

The difference

$$\Delta_s^{(2)} = reduce_2(\Phi_s - expand_4(\mathrm{L3})) \tag{5.1}$$

is calculated[1] and its scalar values are normalized. Normalization is done to fully exploit 8bit texture channels, and to avoid negative values. The intention is to fit the resulting quantization L_2 into 8bit textures without problems. It is proceeded by calculating a $bias_{L_2}$

$$bias_{L_2} = \min\left(\Delta_s^{(2)}\right) \tag{5.2}$$

and a $scale_{L_2}$

$$scale_{L_2} = \begin{cases} \max\left(\Delta_s^{(2)}\right) - \min\left(\Delta_s^{(2)}\right) & \text{if } \max\left(\Delta_s^{(2)}\right) \neq \min\left(\Delta_s^{(2)}\right) \\ 1 & \text{else} \end{cases} \tag{5.3}$$

$\Delta_s^{(2)}$ can then be normalized by

$$\Delta_s^{(2)}(i,j,k) \mapsto \frac{\Delta_s^{(2)}(i,j,k) - bias_{L_2}}{scale_{L_2}} \tag{5.4}$$

As a consequence, each value of $\Delta_s^{(2)}$ will be in the range $[0,1]$. Now $\Delta_s^{(2)}$ is sent to an 8D-vector quantizer (3) as described in chapter 4. The result is the scalar volume L_2 and a 8bit luminance 2D-codebook containing 256×8 entries.

To obtain L_1, it is proceeded similarly. First

$$\Delta_s^{(1)} = \Phi_s - expand_2(decode(L_2)) - expand_4(L_3) \tag{5.5}$$

[1]Since essentially $\Phi_s^{(3)} \equiv L_3$, L_3 does not have to be decoded.

is calculated, where $decode$ has to reverse the normalization of L_2 by application of a scale and bias step. $\Delta_s^{(1)}$ is normalized exactly like $\Delta_s^{(2)}$, using equations 5.2 through 5.4. In order to obtain the scalar volume L_1 and a 8bit luminance 2D-codebook, $\Delta_s^{(1)}$ is sent to a 64D version of the vector quantizer (4). The respective codebook has a resolution of 256×64.

L_1, L_2 and L_3 have the same resolution, resulting in $\Delta_s^{(1)}$ being encoded at the lowest bitrate, while $\Phi_s^{(3)}$ is encoded at the highest. Now the three levels can be interleaved to form a 8bit RGB-volume (5). This RGB-volume is then stored together with the two 2D-luminance codebooks and a header containing the two scale and bias scalars and the data resolutions.

It is important to note that the L3VQ compression scheme accumulates neither roundoff nor quantization errors. This is clear from equations 5.1 and 5.5: Each time a difference level is calculated, a full decoding step is performed, preventing quantization error accumulation. Roundoff errors are avoided by subsampling the input data each time a level of the Gaussian pyramid is needed rather than applying the $reduce_2$ operator successively. As a result, the image fidelity that can be achieved by the L3VQ scheme is quite high. The compression ratio is limited from above for scalar valued data by 64:3≈21.33:1. Both codebooks together are exactly 18KB in size, such that even compression of 128^3 volumes results in a compression ratio of nearly 18:1. This is substantially better than the any of the compression ratios offered by s3tc.

5.3 L3VQ Decoding Algorithm

Decoding of the hierarchy basically proceeds as formulated in equation 3.11, but reversion of the normalization step has to be performed:

$$\Phi_s \approx expand_{2^{(n-1)}}(L_n) + \sum_{i=1}^{n-1} scale_{L_i} \cdot \beta_{C_i}(L_i) + bias_{L_i} \tag{5.6}$$

where β_{C_i} was the decoding process of the vector quantizer using codebook C_i (see section 3.4).

Since C_1 and C_2 are a mappings $\Im \rightarrow \Re^{64}$ and $\Im \rightarrow \Re^{8}$ respectively, the necessary expansion of L_1 and L_2 is performed implicitly by the decoder. Accordingly, the decoding algorithm has to be modified.

The modified decoder then proceeds as follows to decode the sample at (i_1, i_2, i_3).

1. Fetch samples s_3, s_2, s_1 from L_3, L_2 and L_1:

   $s_3(i_1, i_2, i_3) = L_3(i_1 \bmod 4, i_2 \bmod 4, i_3 \bmod 4)$
   $s_2(i_1, i_2, i_3) = L_2(i_1 \bmod 4, i_2 \bmod 4, i_3 \bmod 4)$
   $s_1(i_1, i_2, i_3) = L_1(i_1 \bmod 4, i_2 \bmod 4, i_3 \bmod 4)$

2. Obtain a scalar value $v_3 = s_3$.

3a. Calculate the correct component o_2 of the 8D C_2-codevector:

   $o_2 = \left\lfloor \frac{i_1-(i_1 \bmod 4)}{2} \right\rfloor + 2 \cdot \left\lfloor \frac{i_2-(i_1 \bmod 4)}{2} \right\rfloor + 4 \cdot \left\lfloor \frac{i_3-(i_1 \bmod 4)}{2} \right\rfloor$

3b. Decode sample s_2 to obtain a scalar value v_2:

   $v_2 = < \beta_{C_2}(s_2), E_{o_2} >$,

   where E_{o_2} defines the o_2^{th} standard vector.

4a. Calculate the correct component o_1 of the 64D C_1-codevector:

   $o_1 = (i_1 - (i_1 \bmod 4)) + 4 \cdot (i_2 - (i_1 \bmod 4)) + 16 \cdot (i_3 - (i_1 \bmod 4))$

4b. Decode sample s_1 to obtain a scalar value v_1:

   $v_1 = < \beta_{C_1}(s_1), E_{o_1} >$

5. Perform scale and bias operations:

   $v_2 \mapsto scale_{L_2} \cdot v_2 + bias_{L_2}$
   $v_1 \mapsto scale_{L_1} \cdot v_1 + bias_{L_1}$

6. Sum up the values v_1, v_2 and v_3:

   $\Phi_s(i, j, k) \approx v_1 + v_2 + v_3$

5.4 Compression Domain Volume Rendering

5.4.1 GPU based Decoder

The GPU based L3VQ decoder proceeds just as described in the previous section. The RGB-volume containing the interleaved levels L_1 through L_3 is loaded as a 3D RGB texture, and the two codebooks are each loaded as 2D luminance textures. In order to save the expensive modulo and remainder operations[2] from the previous approach, a 3D RGBα *decoder texture* is issued as fifth texture image. The decoder texture is 4^3 voxels in size and contains pre-calculated codevector

[2]A modulo operation requires at least 3 native fragment instructions at the time being.

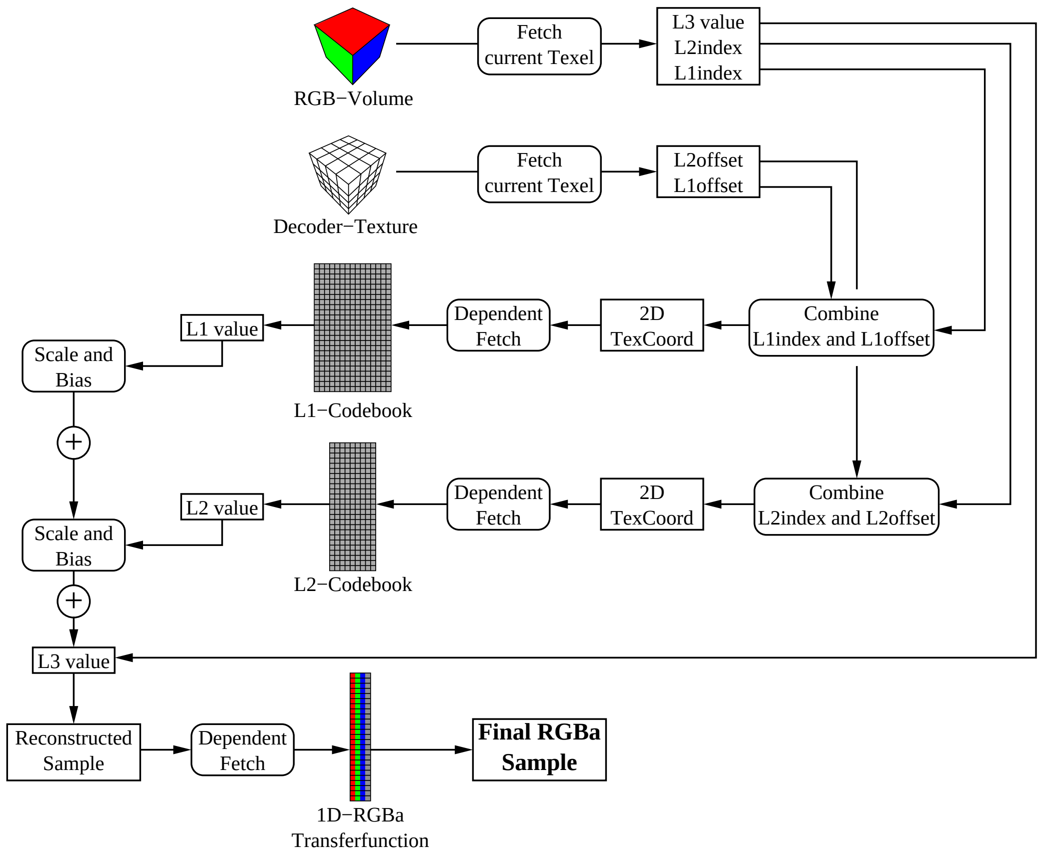

Figure 5.2: *The GPU based L3VQ Decoder. See text for description.*

components as in steps 3a and 4a. To be more precise, the R- and B-channel hold values

$$r(i_1, i_2, i_3) = \frac{(i_1 - 1) + 4 \cdot (i_2 - 1) + 16 \cdot (i_3 - 1)}{63}, \quad i_1, i_2, i_3 = 1, 2, 3, 4 \quad (5.7)$$

and

$$b(i_1, i_2, i_3) = \frac{(i_1 - 1) + 2 \cdot (i_2 - 1) + 4 \cdot (i_3 - 1)}{15}, \quad i_1, i_2, i_3 = 1, 2, 3, 4 \quad (5.8)$$

The modulo operation can then be performed by mapping this texture using a mode of GL_REPEAT for GL_WRAP_S, GL_WRAP_T and GL_WRAP_R. In order to overlap each voxel of the RGB-volume with exactly one voxel of the decoder texture, texture coordinates $(0, 0, 0)$ (lower left) through $\left(\frac{n_1}{4}, \frac{n_1}{4}, \frac{n_2}{4}\right)$ (upper right) have to be issued for all fetches made from the decoder texture. The RGB-volume is still mapped using cardinal texture coordinates from $(0, 0, 0)$ to $(1, 1, 1)$. Both the decoder texture and the RGB-texture must only be accessed using GL_NEAREST filtering, as interpolation is invalid in the compression domain[3].

The GPU based decoding strategy is depicted in figure 5.2.
First, a RGB sample is fetched from the RGB-volume using standard texture coordinates. Then the decoder texture is accessed, using special texture coordinates as described above. The obtained RGB sample contains a triple (s_3, s_2, s_1), as described in the previous section. s_3 can be immediately used as a reconstruction value v_3, while the other two are indices into the respective codebooks (called "L1index" and "L2index" in the figure). The fetch made from the decoder texture contains the associated "L1offset" and "L2offset" specifying the component of the codevector. The L1index and the L1offset are then combined to a set of 2D coordinates, and the same is done for the L2index and the L2offset. The rationale behind storing the offsets in the R- and B-channels of the decoder texture was to allow a single instruction to perform both combinations. Using these texture coordinates, a dependent texture fetch is made from the L1- and L2-codebooks, resulting in two scalar values v_1 and v_2. A scale and bias operation is then applied to v_1 and v_2, using the scale and bias values obtained before. They can be issued as program parameters and are accessible as a constant from within the fragment program. The three values v_1, v_2 and v_3 are then added up, and a last texture fetch from the transfer function (stored as a 1D RGBα texture) is made, resulting in a full RGBα sample.

If front to back rendering is used, the RGB channels have to be multiplied by the alpha value (see equation 2.9), since separate color and alpha blending functions are not supported in OpenGL as a multi-vendor extension. Another option would be to supply premultiplied colors in the transfer function, but it is strongly advised against it. The reason is that the result of such a premultiplied transfer function are small color values that suffer a lot from 8bit quantization. The fragment program on the other hand supports full float precision, and usually colors get only quantized to 8bit after blending.

For front to back ordering of the clip polygons, the Under operator (equation 2.9) is applied, requiring an OpenGL blending function of

(GL_ONE_MINUS_DST_ALPHA,GL_ONE)

as well as an alpha buffer. For back to front ordering of the clip polygons, the Over operator (equation 2.8) is applied, requiring a blending function of

(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA)

[3]The compression domain is related to the cardinal 3D-domain by a non-affine transformation.

For the Over operator no alpha buffer is required.

The final fragment program in Cg syntax [nVia] looks as follows.

```
CG_PROFILE_ARBFP1

void main(uniform float4      scalebias  : C0,
                  float3      StdCoord   : TEXCOORD0,
                  float3      DecCoord   : TEXCOORD1,
          uniform sampler3D   RGBvol     : TEXUNIT0,
          uniform sampler3D   Decoder    : TEXUNIT1,
          uniform sampler2D   L2code     : TEXUNIT2,
          uniform sampler2D   L1code     : TEXUNIT3,
          uniform sampler1D   transfer   : TEXUNIT4,
          out     float4      color      : COLOR0)
{
    float   L2,L1;
    float3  indices;
    float4  offsets;

    // Fetch indices from RGB-volume using standard coordinates.
    indices=tex3D(RGBvol,StdCoord).rgb;

    // Fetch offsets from the decoder texture (in r and b components).
    offsets=tex3D(Decoder,DecCoord);

    // Combine indices and offsets for L1 and L2 simultaneously.
    offsets.ga=indices.rg;

    // Decode L2 and L1 values. Perform scale and bias operations.
    L2=scalebias.x*tex2D(L2code,offsets.ba)+scalebias.y;
    L1=scalebias.z*tex2D(L1code,offsets.rg)+scalebias.w;

    // Add the three values together. Apply transfer function.
    color=tex1D(transfer,saturate(L1+L2+indices.r));

    // Front to back only: multiply rgb channels with alpha.
    color.rgb*=color.a;
}
```

5.4.2 Deferred Decoding

Rendering volumes generates a tremendous amount of fragments. Even if the volume is rendered using as few as 200 clip polygons, for a 512×512 viewport as many as 22.4 millions of fragments may be generated. If the number of polygons is increased to a value that avoids undersampling the volume (450 slices and more), the amount of fragments grows linearly. This limits interactive decoding, as the fragment program has to be executed for each generated fragment. A technique commonly applied in cases where an expensive fragment program is present is called *deferred fragment processing*. The basic idea is to discard as much fragments as possible by a cheap fragment program first, and to apply the expensive fragment program only to those fragments surviving the first one.

Obviously, the fragments that result in an alpha value of 0 can be safely discarded, since they do not contribute to the final image. But these fragments are not easily identified, as the entire decoding step has to be performed before the alpha value is known. Luckily, for some transfer functions entirely transparent 4^3 blocks can be identified using only a single texture fetch. Based on the fact that the lowest level $\Phi_s^{(3)}$ of the Laplacian pyramid was subsampled using the $reduce_4$ operator, it will contain values of 0 only where the input data Φ_s contained a 4^3 block with very low values. More precisely, only blocks with

$$\sum_{i,j,k=1}^{4} block(i,j,k) < \frac{32}{4^3} \cdot \frac{1}{2^8} = \frac{1}{16}$$

will result in a value of 0 in the $\Phi_s^{(3)}$ level, since the average of these blocks will be rounded down. This is based on the fact that the level is quantized uniformly using 8bits per block. That means that whenever a value of 0 is encountered in L_3, it can be safely assumed that not much structure is lost, if any at all, by the prediction that the final reconstruction of Φ_s will be 0 as well. In most datasets a value of 0 corresponds to free space, and consequently the entire block can be discarded.

It is thus tested by the rendering application wether the first entry of the transfer function maps to an alpha value of 0. If so, deferred decoding is activated, and a cheap discard program is executed prior to the full decoder. If not, deferred decoding is disabled. This has the additional nice property that for dense data the deferred decoding can be disabled very easily in order to avoid the overhead of having two fragment programs executed. If deferred decoding is enabled, the algorithm proceeds as follows.

```
For each clip polygon:
    Enable depth mask and disable depth test.
    Disable color mask.
    Render the clip polygon.
    For each fragment:
        Fetch a sample from the RGB-volume.
        If the R-channel of that sample is not 0, discard the fragment.
    Now the depth buffer contains the current depth where decoding is obsolete.
    Enable depth test and set the depth function to GL_GREATER.
    Enable the color mask.
    Render the clip polygon with the full decoder program enabled.
```

This procedure relies on the early Z-test (see figure 2.3) to be present on the target architecture. The deferred decoding has proven itself to be very effective for sparse data. For example in frames of the shock-wave simulation (see figure 6.1) structures are present in approximately one third of the entire dataset. As a consequence a speedup of roughly a factor 3 could be observed. For dense data the deferred decoding simply becomes an unnecessary overhead, and consequently a performance loss of 10% to 15% could be observed. The user may then chose to disable the test by assigning an alpha value other than 0 to the first transfer function entry.

5.5 Results

A L3VQ encoder and a decoder were implemented in C++. The encoder implementation is based on the fast vector quantization method provided in chapter 4. The GPU based decoder was implemented in C++/Cg as part of a slice-based volume renderer. The renderer supports uncompressed as well as L3VQ compressed volumes, post-shaded transfer functions and both front to back and back to front compositing. While it would have been possible in general to apply a trilinear interpolation kernel for both uncompressed and compressed volumes, this would have been expensive in the case of the compressed volume, since 8 decoding steps would have to be performed. The renderer is consequently restricted to nearest neighbor interpolation for compressed volumes. In order to provide comparable images, the uncompressed volumes were rendered using nearest neighbor interpolation as well.

Encoding performance as well as compression domain rendering performance were extensively tested. The results are documented in figures 5.3 through 5.5. The engine dataset has a resolution of $256^2 \times 128$ voxels, while the skull has a resolution of 256^3. Encoding times for the L3VQ compression were 89 and 250 seconds on a Celeron 600MHz. While the $2{\times}2{\times}2$ block quantization is roughly of a factor 4 faster than the L3VQ, considerable gains in the compression ratio can be made without too much loss in fidelity by application of the hierarchical method. In case of the engine 1.4dB were lost, while in the case of the skull approximately 3.2dB were lost. However, for noisy data such as the skull the perceived image fidelity is a lot better than the low SNR would suggest. The reason may be seen in the fact that the noise is mainly contained in the highest frequency level of the Laplacian pyramid, that is encoded using the fewest bits. As a consequence, the noise can only be unsufficiently reconstructed, resulting in an increase of the mean square error and a decrease in the SNR. This can also be seen an additional benefit of the hierarchy, since noise in the data will be automatically removed by the L3VQ scheme to some extend. If the amount of noise becomes too large, it may be appropriate to apply an additional noise removing operation such as a thresholding or Wiener filtering [PTVF02] to the data prior to encoding.

All rendering performances were measured on a $512{\times}512$ viewport with 200 slices and front to back compositing. For the L3VQ compressed data deferred decoding was enabled. This is no option for the uncompressed and block quantized versions, since in both cases only the transfer lookup can be saved by such a method.

For the engine rendering performance was about 19fps for uncompressed, 14fps for block quantized and 16fps for L3VQ compressed data.The skull was rendered at 14fps (uncompressed), 10fps (block quantized) and 11fps (L3VQ).

While this indicates that 20% to 40% performance are lost when rendering from L3VQ data, there are also some datasets for which rendering from a L3VQ compressed representation actually speeds up the process. This usually happens when the dataset is sparse enough to discard about 50% of the fragments prior to decoding. The shock-wave dataset (see fig. 6.1) is such an example that results in a performance gain of nearly 20%.

5.6 Possible Extensions to the L3VQ Scheme

Since the L3VQ scheme is based on a general purpose vector quantizer, some interesting extensions can be implemented.

First, the scheme could be modified to compress 2D RGB images by building a 2D Laplacian pyramid, effectively based on 4^2 blocks of pixels. The $\Phi_s^{(3)}$ level then contains RGB-vectors at 24bit per block resolution. The $\Delta_s^{(2)}$ and $\Delta_s^{(1)}$ lev-

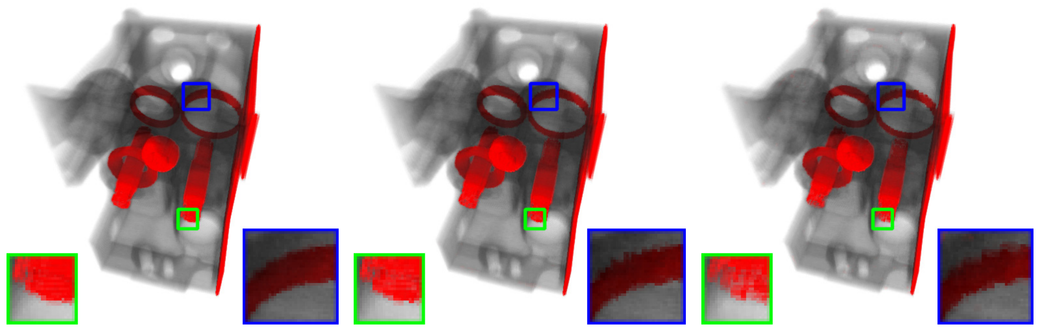

Figure 5.3: *Comparison between per Block Quantization and L3VQ (I). From left to right: original engine dataset, encoded by $2 \times 2 \times 2$ block quantization and encoded by L3VQ. Compression ratios are 1:1, 7.99:1 (SNR=24.29dB) and 20.38:1 (SNR=21.87dB) respectively. Encoding took approx. 25 seconds for the block quantization and 89 seconds for the L3VQ.*

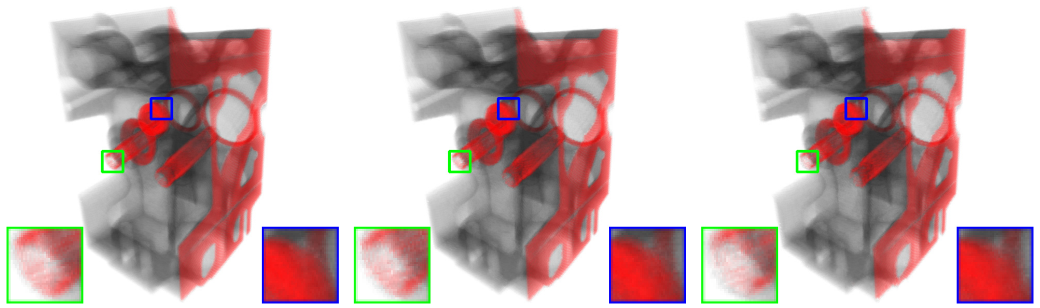

Figure 5.4: *Comparison between per Block Quantization and L3VQ (II). From left to right: original engine dataset, encoded by $2 \times 2 \times 2$ block quantization and encoded by L3VQ. Compression ratios, SNRs and encoding time as in figure 5.3.*

els contain RGB-vectors as well, essentially resulting in a dimension of 24D- and 192D-vectors respectively. The resulting codebooks would then store RGB instead of luminance values. The maximum compression ratio that can be achieved by this scheme is 48:5=9.6:1 for the case of 8bit codebook indices. If the images do not have to be decoded on the GPU, higher compression ratios are achievable, as the number of bits per index can be reduced. For the case of RGB-volumes the maximum compression ratio is 192:5=38.4:1. To demonstrate the effectiveness of 2D RGB-L3VQ, figure 5.6 shows a series of images showing that RGB-L3VQ offers the best image fidelity at low bitrates. This becomes especially clear for the red zooming area. Only the RGB-L3VQ compressed image and the original show correct colors. Encoding took 11 seconds for the 3D RGB encoding (SNR=16.25dB), 24 seconds for the 2x2 block encoding (SNR=25.97dB), and 50 seconds for the RGB-L3VQ (SNR=26.02dB). All timings were done on a Celeron

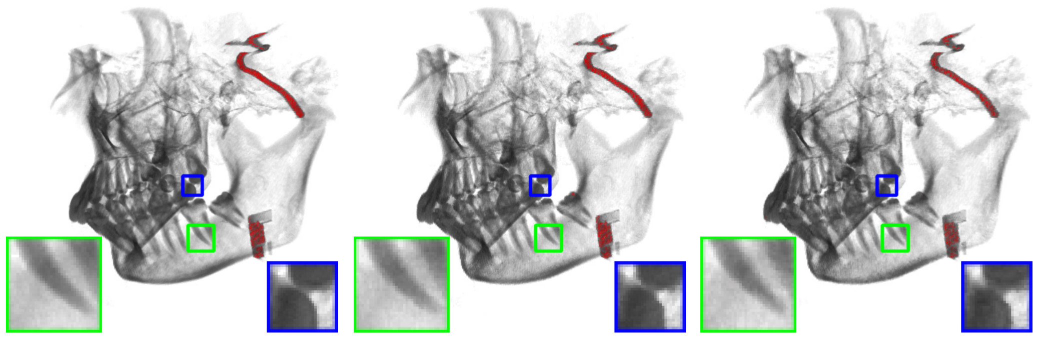
Figure 5.5: *Comparison between per Block Quantization and L3VQ (III). From left to right: original engine dataset, encoded by $2 \times 2 \times 2$ block quantization and encoded by L3VQ. Compression ratios are 1:1, 8.0:1 (SNR=14.86dB) and 20.84:1 (SNR=11.70dB) respectively. Encoding took approx. 61 seconds for the block quantization and 250 seconds for the L3VQ.*

600 processor. In the case of the RGB-L3VQ each level was encoded using 3bits per block. The image was decoded on the CPU.

Second, since the alpha channel of the RGB index-volume is unused, a 4 level Laplacian pyramid could be computed, essentially operating on 8^3 blocks of voxels. This would add another codebook with resolution 256×512 for the case of 8bit codebook indices. Since 512 entries can not be encoded using only 8bits, the decoder texture would have to use one channel for L_2 and L_3 respectively, and two for L_1. The fragment program would have to be modified to calculate an additional 2D texture coordinate and to perform an additional texture fetch from the new codebook. This would make a “L4VQ” scheme considerably more expensive to decode on the GPU, but the compression ratio could be as large as 512:4=128:1. This could be especially interesting for very large volumes, as the combined codebooks would be 146KB in size for scalar valued volumes. L4VQ has not yet been implemented.

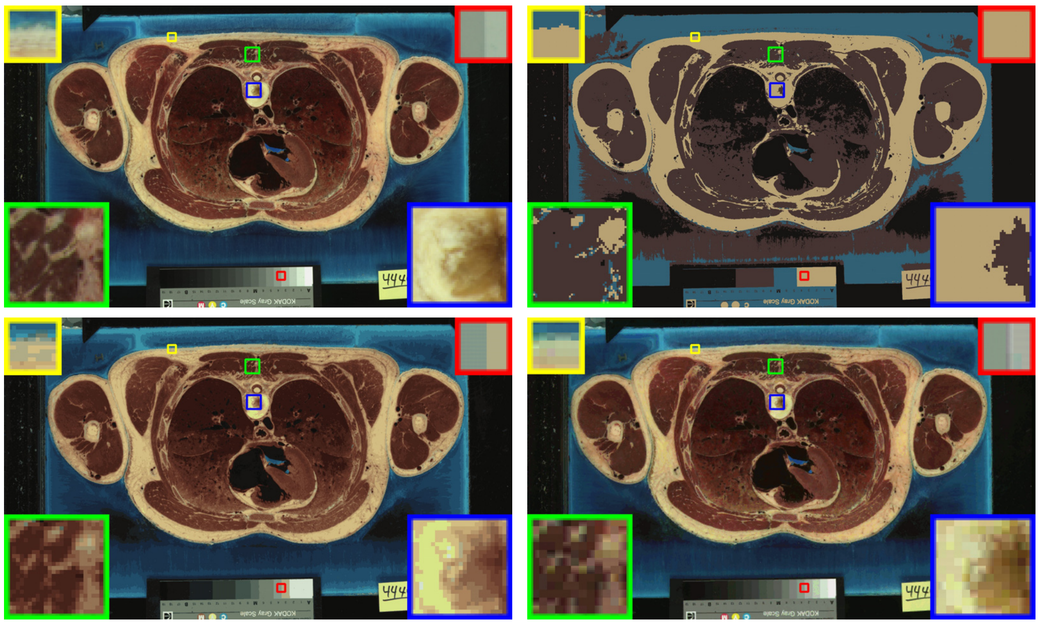

Figure 5.6: *One Slice of the Visbile Human Project [VHP] encoded using RGB-L3VQ. Left to right, top to bottom: original 24bpp RGB image, image encoded using 2bpp by 3D RGB-vector quantization,* 2×2 *block quantization at 1.25bpp and RGB-L3VQ using 1.11bpp.*

Chapter 6

Time-varying Volume Data

6.1 Introduction

Time-varying data results from sampling a four-variate function $\Phi(x_1, x_2, x_3, t)$ and is commonly generated by CFD applications. It is of utmost importance to scientists working in that field to obtain a feeling for how a certain velocity or pressure field evolves over time. Thus the underlying space-time domain is discretized both spatially and temporally, and a solution for each timestep is calculated. The result is a series of volumetric data sets that have to be displayed in some way in order to get insights from the data. Here a major problem arises. Volumetric data sets that are tens or hundreds of megabytes in size are not uncommon, and producing one such volume for each timesteps even multiplies that amount of data. The result are datasets with overwhelming storage requirements that can not be displayed interactively on most consumer class workstations. Even on today's supercomputers providing interactive investigation of such datasets is a non trivial task, since large amounts of data have to be transferred to the graphics subsystem.

6.2 Compression of Time-varying Volume Data

To solve the storage requirement problem, compression has been proposed for time-varying volume data before. Both hierarchical data structures and difference encoding schemes have been proposed, for example in [Wes95, SJ94, SCM99]. But none of these techniques applied a GPU based decoding technique. Consequently these methods do not solve the bandwidth problem, as each timestep has to be decoded on the CPU prior to being sent to the graphics subsystem. Though current graphics busses have bandwidths of up to 2GB/s, the conversion to internal texture formats severely limits the interactivity of systems transferring large amounts of data to the GPU.

This bottleneck was recognized and GPU based decoding scheme for time-varying data was supposed not so long ago by Lum in [LMC01]. The approach applies a DCT to the original dataset, quantizing the tansformed data and encoding it into hardware assisted color tables. These color tables can then be reloaded at interactive rates in order to generate animations. The drawback of this method besides the need for hardware accelerated color tables is the fact that either good compression ratios or high image fidelity could be achieved. The reason is that graphics hardware did not offer enough flexibility to support fine compression/fidelity tradeoffs at that time.

Vector quantization on the other hand is suited very well to compress time varying-volume data. First, because it is general enough to handle arbitrary dimensions, making both spatial and temporal encoding of the data possible. Second, vector quantization automatically calculates a reduced set of coefficients that best fits the data. In general it performs a lot better than the DCT when used to obtain representative coefficients of the data, since the DCT does not adapt to intrinsic data characteristics. Third, vector quantization provides good compression/fidelity tradeoff possibilities.

In order to use the hierarchical L3VQ vector quantization scheme to compress time-varying volume data, a straight-forward approach is to encode each single frame without taking temporal coherences into account, producing a compressed volume and two codebooks each timestep. While this already solves both the storage requirement and bandwidth problems, the encoding process can be sped up considerably by applying a progressive encoding scheme.

6.3 Progressive Encoding

Progressive encoding introduces I-,P- and B-frames, similar to the MPEG compression standard. I-frames are independent of other frames, while P-frames are predictively encoded, meaning that they depend on prior frames. B-frames are bidirectionally predictive, depending on a fixed number of previous and future frames. B-frames usually result in the best compression, but are most expensive both to encode and decode, as they must rely on the largest support of all three types. Since for vector quantization the encoding step is still expensive when compared to MPEG, it is reasonable to support only I- and P-frames.

There are two ways to extend the vector quantizer to accommodate for I- and P-

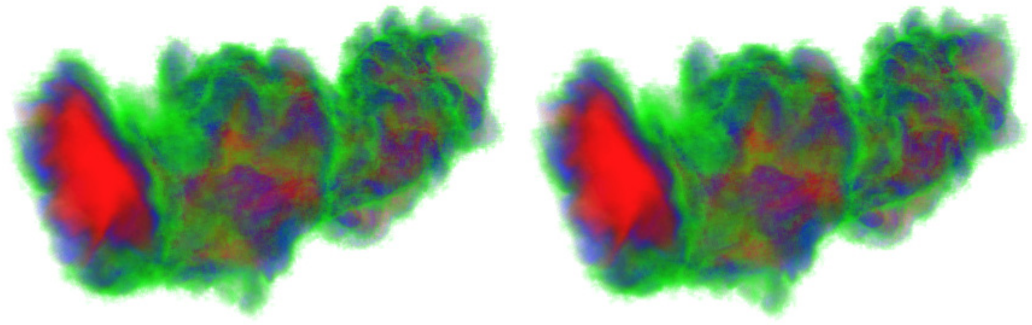

Figure 6.1: *Comparison I- and P-frames. Both images show the 85^{th} timestep of the shock-wave sequence. The left image was encoded as an L3VQ compressed I-frame, while the right one is the 30^{th} in a sequence of P-frames.*

frames. First, for each I-frame a codebook is calculated by performing both PCA split and LBG relaxation. This codebook is then re-used for the next P-frames simply by fitting each P-frame vector to the codebook. Second, instead of predicting the final codebook, only the initial codebook for the LBG step is predicted. For each I-frame both PCA split and LBG relaxation are executed, but the resulting codebooks and indexsets are not discarded. Instead they are treated as entry point for the next P-frame that only performs LBG relaxation. Since even the indexset of the previous frame is re-used for P-frames, fast and partial searches can be applied as well.

Obviously the second method results in the better fidelity, since each frame has an own codebook. Over that, random access becomes especially easy, as each frame is essentially a volume of its own. Furthermore, since the fully converged solution of the previous step is used, exceptionally long sequences of P-frames (see figure 6.1) can be generated, as long as the volume does not show rapid changes. These however do rarely occur in scientific data, since in contrast to movies no cutting is applied.

6.4 Results

The hierarchical L3VQ vector quantization scheme was extended by the progressive encoding method described in the last section. Two 8 bpp test datasets were used, a $256^3 \times 89$ shock-wave simulation (1.4GB) and a $128^3 \times 99$ vortex dataset (200MB). Both were first quantized using the L3VQ method described in chapter 5, and then using the progressive encoder.

In both cases using only a single I-frame resulted in no perceivable performance

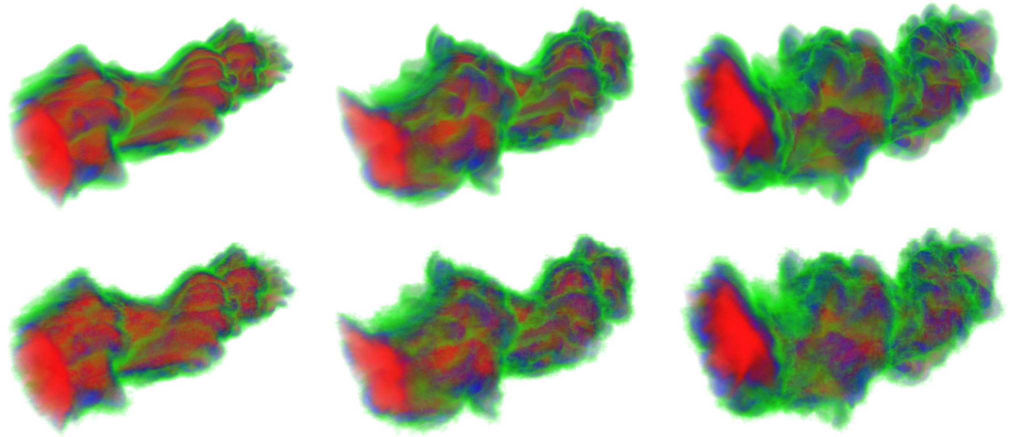

Figure 6.2: *Shock-wave dataset* $(256^3 \times 89)$. *From left to right timesteps 65, 75 and 85 are shown. The top row was rendered using the original dataset (1.4GB), while in the second row L3VQ compressed data was used (70GB).*

loss. This is due to the facts that neither data set shows rapid changes and that each P-frame yields a fully converged quantization. The only reason that some loss may occur is that the LBG is sensitive to the input codebook, as it converges towards a local minimum.

On a Celeron 600 the shock-wave simulation could be encoded in approximately 3 hours, while encoding of the vortex data took 54 minutes. If the progressive encoder was used, the shock-wave sequence took 1 hour 22 minutes while the vortex data took only 24 minutes. The maximum difference in the SNR between the two methods was in both cases under 0.15dB. SNRs were between 23.46dB and 43.68dB for the shock-wave frames and between 19.98dB and 20.32dB for the vortex frames. Compression was 20.84:1 for the shock-wave and 17.98:1 for the vortex data, resulting in roughly 70MB and 11MB. Rendering performance was about 22fps for the shock-wave and 13fps for the vortex dataset. Rendering from the uncompressed data achieved roughly 18fps in both cases. All rendering performance was measured on a 512×512 viewport using front to back compositing and 200 slices. See also figures 6.2 and 6.3.

In order to merge the many single frames into a single file, they were additionally packed into a gzipped tarball archive. Interestingly enough, for the shockwave this lossless compression step resulted in an output file of 3.5MB in size, as the volume is very sparse. The vortex data set did not compress well, resulting in a file of 10.5MB in size. This can be explained by the interesting property of vector quantization to substitute similar patterns of voxels by the same indices. Thus if large scale regularity was present in the input file, it is also likely to be present

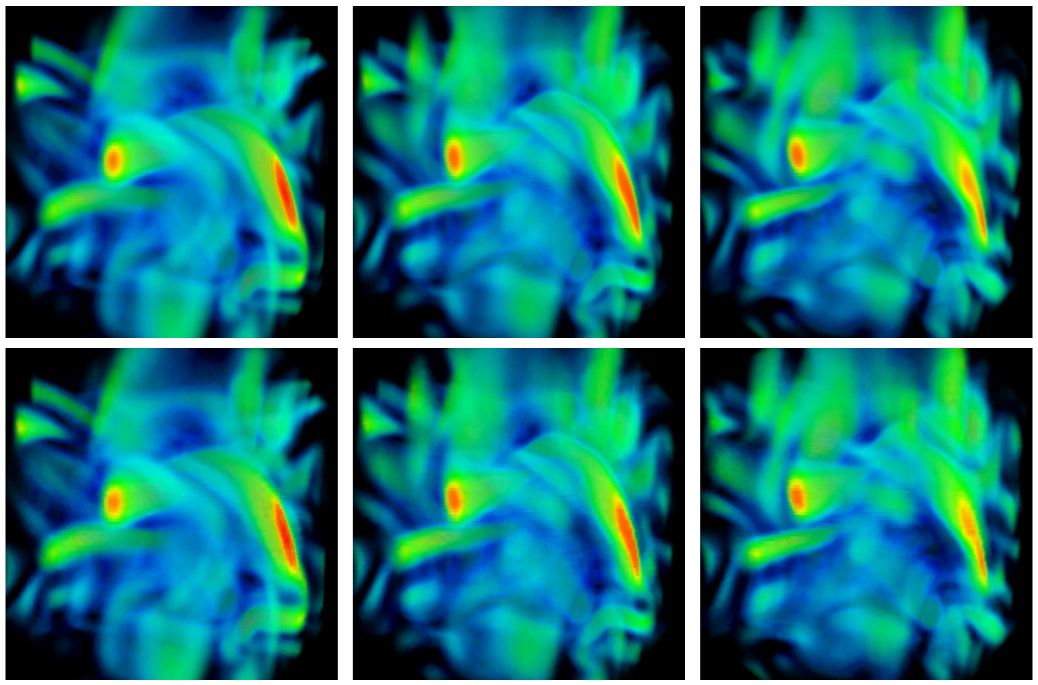
Figure 6.3: *vortex dataset* $(128^3{\times}99)$. *From left to right timesteps 1,5 and 9 are shown. The top row was rendered using the original dataset (200MB), while in the second row L3VQ compressed data was used (11MB).*

in the compressed file. While vector quantization only implicitly profits of such structures, zip algorithms must rely on it to provide good compression ratios. The relatively rapid encoding of the shockwave when compared to the vortex can be explained by the fact that large empty spaces are quantized *exactly* during one of the first PCA splits, consequently never being touched again for the remainder of the PCA algorithm. The LBG algorithm also profits of such large empty spaces, as these areas show excellent cache coherence. The reason lies within the internals of the LBG algorithm, that maintains lists of indices. These lists get more and more fragmented during the encoding process. The exception are those lists that contain same indices at adjacent positions.

Chapter 7

Results and Discussion

7.1 Results

In this diploma thesis it was demonstrated that vector quantization can be used as a general purpose compression tool for scientific data, not necessarily limited to scalar valued volume data. For reasonably sized data an efficient implementation that avoids common performance bottlenecks of previous vector quantization tools is needed. An implementation that is by a factor of 30 faster than previous ones was described in chapter 4.

In chapter 5 it was shown that the Laplace decompostition can serve as an efficient de-correlation step prior to sending the data to the vector quantizer. Furthermore, data available in such a representation can be rendered directly from the compression domain at interactive framerates. Over that, since the Laplacian pyramid is a hierarchical decomposition, various approaches that require a multi-resolution representation of the data, become possible. This was demonstrated by the deferred shading algorithm that discards entire blocks of voxels based only on the lowest level of the hierarchy.

Application to time-resolved volume sequences as well as an efficient progressive encoding scheme was demonstrated in chapter 6. The approach solves both the storage requirement and bandwidth problem associated with large time-resolved datasets, virtually increasing graphics memory by a factor of 20. In addition sequences encoded in the proposed format allow for easy random access.

The methods proposed in this thesis have been implemented in C++ using OpenGL and Cg. The sourcecode may be obtained by contacting the author.

7.2 Future Directions

In chapter 4, the performance analysis of the LBG algorithm showed that approximately 80% of the runtime was spend computing averages and mean squared errors. An important direction to additionally speed up the encoding would thus be the mapping of these operations to SIMD technology such as SSE or 3DNow. However, the GPU is currently approaching a general purpose SIMD processor very quickly. It might consequently be interesting to employ graphics hardware even to the task of compressing datasets.

As was shown in this thesis, the hierarchical vector quantizer can be extended in order to handle vector valued volume data. However, it is not clear how such volumes should be rendered in order to obtain meaningful images. A successful framework for rendering time-resolved vector valued data thus needs to employ novel rendering algorithms that could be based on implicit data analyzation[1] properties of the vector quantization tool.

For the compression of time-varying data, it has not yet been investigated how temporal coherences can be exploited in order to achieve higher compression ratios. It might be viable to re-use lowest frequency levels as well as codebooks of I-frames for some number of consecutive P-frames, thus reducing storage requirements. Another way to improve compression ratios would be to calculate the hierarchical compression directly on the underlying space-time domain, using a 4D Laplacian pyramid.

[1]Vector quantizers classify the data into blocks sharing similar characteristics. From a mathematical point, quantization cells are discretely sampled Voronoi regions.

Acknowledgements

I would like to thank my supervisor, R. Westermann, for many fruitful discussions and advices.

Over that I would like to thank my colleagues B. Sevenich and M.O. Westerburg for the various hints that helped me improve the results of the quantizer. Thanks must also go to J. Krüger for providing me with volume rendering sample code, for testing the renderer and helping me getting started with Cg.

I would also like to thank K.L. Ma and D. Silver for providing me the vorticity dataset, as well as P. Kipfer for the shockwave dataset.

Last but not least I would thank A. Neuendorf for proof-reading this document.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, daß ich die vorliegende Diplomarbeit selbständig und ohne fremde Hilfe verfasst habe. Ich habe dazu keine weiteren als die hier aufgeführten Hilfsmittel benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet.

Aachen, May 12, 2003 Jens Schneider

Bibliography

[AR75] N. Ahmed and K.R. Rao. *Orthogonal Transforms for Digital Signal Processing*. Springer-Verlag, New York, 1975.

[ARB] The OpenGL Architecture Revision Board (ARB). http://www.opengl.org/developers/about/arb.html.

[ATi] ATi Technologies, Inc. ATi Developer Relations and Whitepapers. http://www.ati.com/developer.

[Bli82] J.F. Blinn. A Generalization of Algebraic Surfaces. In *ACM Transactions on Graphics*, number 1, pages 235–256, 1982.

[BW82] J.A. Bucklew and G.L. Wise. Multidimensional Asymptotic Quantization Theory with rth Power Distortion Measures. In *IEEE Transactions on Information Theory (IT)*, volume 28, pages 239–247, March 1982.

[CCF94] B. Cabral, N. Cam, and J. Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *Proceedings ACM Symposium on Volume Visualization 94*, pages 91–98, 1994.

[CN93] T.J. Cullip and U. Neumann. Accelerating Volume Reconstruction with 3D Texture Hardware. Technical Report TR93-027, University of North Carolina, Chapel Hill N.C., 1993.

[CSF] Cg Shader Forum. Cg Shaders Website. http://www.cgshaders.org.

[EKE01] K. Engel, M. Kraus, and T. Ertl. High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In *Proceedings Eurographics / SIGGraph Workshop on Graphics Hardware*, 2001.

[Equ89] W.H. Equitz. A New Vector Quantization Clustering Algorithm. In *IEEE Transactions on Acoustics, Speech, and Signal Processing*, volume 37, pages 1568–1575, October 1989.

[Fow] J.E. Fowler. QccPack - Quantization, Compression, and Coding. http://qccpack.sourceforge.net.

[Gcc] The GNU Project. The GCC Homepage. http://gcc.gnu.org.

[GN98] R.M. Gray and D.L. Neuhoff. Quantization. In *IEEE Transactions on Information Theory*, volume 44, pages 2325–2384, 1998.

[GY95] M. Ghavamnia and X.D. Yang. Direct Rendering of Laplacian Pyramid Compressed Volume Data. In *Proceedings of IEEE Visualization '95*, pages 192–199, 1995.

[Hea] E. Heart. Cool Stuff (that does not fit elsewhere). Presentation at the Games Developer Conference 2003, http://www.ati.com/developer.

[Hil77] E.E. Hilbert. Cluster Compression Algorithm - A Joint Clustering Data Compression Concept. Technical Report, Washington DC: NASA, 1977. JPL Publication 77-43.

[Iva] I.A. Ivanov. Image Compression with Vector Quantization. Gamasutra feature article. http://www.gamasutra.com/features/20010416/ivanov_01.html.

[JN84] N.S. Jayant and P. Noll. *Digital Coding of Waveforms*. Prentice Hall, Englewood Cliffs, NJ, 1984.

[JPEG] Joint Photographic Experts Group. JPEG. http://www.jpeg.org.

[Kaj86] J. T. Kajiya. The Rendering Equation. *Computer Graphics*, 20(4):143–150, August 1986.

[Krü02] J. Krüger. Echtzeitsimulation und -darstellung von Wolken. RWTH Aachen, University of Technology, Germany. Diploma thesis, 2002.

[KvH84] J.T. Kajiya and B.P. von Herzen. Ray Tracing Volume Densities. In *Proceedings SIGGraph '84*, volume 18, pages 165–174, 1984.

[LBG80] Y. Linde, A. Buzo, and R.M. Gray. An Algorithm for Vector Quantization Design. In *Proceedings IEEE Transactions on Communications*, volume 28, pages 84–95, January 1980.

[LKHS01] H. Lensch, M. Kautz, Goesele, W. Heidrich, and H.P. Seidel. Image-based Reconstruction of Spatially Varying Materials. In *Proceedings of the 12th Eurographics Workshop on Rendering*, pages 104–115, 2001.

[Llo82] S.P. Lloyd. Least Squares Quantization in PCM. In *IEEE Transactions on Information Theory*, volume 28, pages 127–135, 1982.

[LMC01] E. Lum, K.L. Ma, and J. Clyne. Texture Hardware assisted Rendering of Time-varying Volume Data. In *Proceedings IEEE Visualization '01*, 2001.

[Max60] J. Max. Quantizing for Minimum Distortion. In *IRE Transactions on Information Theory*, volume 6, pages 7–12, 1960.

[Max95] N. Max. Optical Models for Direct Volume Rendering. In *IEEE Transactions on Visualization and Computer Graphics*, volume 1, 1995.

[nVia] nVidia Corporation. The Cg Homepage.
http://developer.nvidia.com/view.asp=?PAGE=cg_main.

[nVib] nVidia Corporation. nVidia's Developer Relations and Whitepapers.
http://developer.nvidia.com.

[OGLa] The OpenGL Organization. OpenGL 2.0 Proposals Website.
http://www.3dlabs.com/support/developer/ogl2/index.html.

[OGLb] The OpenGL Organization. OpenGL Website.
http://www.OpenGL.org.

[PD84] T. Porter and T. Duff. Compositing Digital Images. In *Proceedings SIGGraph '84*, volume 18, pages 253–259, 1984.

[PGK02] M. Pauly, M. Gross, and L.P. Kobbelt. Point Primitives for Visualization: Efficient Simplification of Point Sampled Surfaces. In *Proceedings IEEE Visualization '02*, pages 163–170, 2002.

[Pow] PowerVR. PowerVR Website. http://www.powervr.com.

[PTVF02] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C++*. Cambridge University Press, second edition, 2002.

[Red] RedHat. Cygwin. http://www.cygwin.com.

[RSEB$^+$00] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and Ertl. T. Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures And Multi-Stage Rasterization. In *SIG-Graph/Eurographics Workshop on Graphics Hardware*, 2000.

[SA] M. Segal and K. Akeley. The OpenGL Graphics System: A Specification (Version 1.4). http://www.opengl.org/developers.

[Say00] K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann Publishers, 2929 Campus Drive, Suite 260, San Mateo, CA 94403, USA, second edition, 2000.

[SCM99] H.W. Shen, L. Chiang, and K.L. Ma. A Fast Volume Rendering Algorithm for Time-varying Fields using Time-space Partitioning. In *Proceedings IEEE Visualization '99*, pages 371–377, 1999.

[Sev03] B. Sevenich. Paralleles Volume Rendering auf PC Clustern mit lokal verfügbarer Grafikhardware. Diploma thesis, RWTH Aachen, University of Technology, Germany, 2003.

[SJ94] H.W. Shen and C. Johnson. Differential Volume Rendering: A Fast Volume Rendering Technique for Flow Animation. In *Proceedings IEEE Visualization '94*, pages 180–187, 1994.

[TCRS20] M. Tarini, P. Cignoni, C. Rocchini, and R. Scorpigno. Real-time, Accurate, Multi-featured Rendering of Bump Mapped Surfaces. In *Proceedings Eurographics '00*, page 2000, 112-120.

[VHP] The Visible Human Project. VHP Male Fullcolor CCD Slices. http://www.nlm.nih.gov/research/visible.

[WE98] R. Westermann and T. Ertl. Efficiently using Graphics Hardware in Volume Rendering Applications. In *Proceedings SIGGraph '98*, pages 291–294, 1998.

[Wes95] R. Westermann. Compression Domain Volume Rendering. In *Proceedings of IEEE Visualization '95*, pages 168–176, 1995.

[WS01] R. Westermann and B. Sevenich. Accelerated Volume Ray-Casting using Texture Mapping. In *Proceedings IEEE Visualization '01*, pages 271–278, 2001.

[WVGW94] O. Wilson, A. Van Geldern, and J. Wilhelms. Direct Volume Rendering via 3D Textures. Technical Report UCSC-CRL-94-19, University of California, Santa Cruz, 1994.

[Zad66] P.L. Zador. Topics in the Asymptotic Quantization of Continuous Random Variables. Bell Laboratories Technical Memorandum, 1966.

List of Figures

Index