Extended Light Fields and their Application to Volume Rendering

Patrick Ohly[1]

January 8, 1999

[1] Patrick.Ohly@gmx.de

Abstract

Image based rendering with light fields has been limited to static scenes so far. This project extends light fields so that they become capable of capturing scenes with changing attributes. These additional attributes can be changed interactively while rendering the scene with an extended light field. Interactive frame rates are achieved for extended light fields with at least two additional parameters.

To reduce the time to generate an extended light field and the amount of memory to store it, algorithms to calculate a good sampling and reconstruction of the additional parameters were developed. An irregular sampling pattern with non-linear interpolation is used.

Extended light fields are applied to volume rendering by mapping density to opacity with a simple ramp. Two parameters, its position and steepness, specify this ramp. Knowledge about the volumetric data is used to improve the efficiency of sampling these two parameters. Depending on the speed of the original renderer, extended light fields achieve a speedup of 10 or allow interactive frame rates at all.

Acknowledgements

I would like to thank my supervisor in Edinburgh, Eric McKenzie, for his help with this project and the thesis and Gordon Watson for proposing the idea of volume rendering using light fields and introducing me to this area.

I also thank the Department of Clinical Neurosciences, Western General Hospital, for granting their permission to use the head data set. Philippe Lacroute and Sandy Napel have kindly provided two further data sets.

Marc Levoy has been very helpful and without him and Lucas Pereira making the `lightpack` source publicly available, this project would have been hardly possible.

My thanks also go to Professor Schmitt and Peter Oel for providing the opportunity to continue this project as a part of my studies in Germany.

Contents

Chapter 1

Introduction

Compared to traditional techniques, image based rendering techniques have the advantage that rendering speed does not depend on the complexity of the scene. New images are simply generated from existing ones by resampling them. This only depends on the size of the images, not on their contents.

This advantage has been used before to enable interactive viewing of volumetric data in [FLN90]. Possible viewing positions were placed on the surface of a sphere around the volume and focused on its center. This surface was subdivided into regular triangles. Each vertex of this mesh was chosen as the view point for a volumetric rendering of the data and the resulting images were stored. Interactive rendering was done purely by linear interpolation between the three closest precalculated images.

This approach was limited to rotations around the center of the volume and a constant distance, though. Light fields allow arbitrary viewing transformations.

1.1 Light fields

1.1.1 Parameterisation and Generation

Light fields [LH96] respectively Lumigraphs [GGSC96] both treat the flow of light through space as a function that maps position and direction of a light ray to its intensity. This general *plenoptic function* $L(x, y, z, \theta, \psi)$ ([AB91]) has five dimensions.

Because the intensity of a ray does not change if no obstructions are between the observer and the origin of a ray, four dimensions are sufficient to describe this function: the rays on the surface of a cube around an object hold all the information about this object (figure 1.1). The same is true for rays entering an empty cube around the observer, but for the sake of simplicity this report only mentions the case of a convex object from now on.

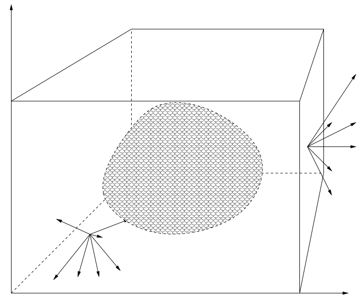


Figure 1.1: The cube holds all radiance information.

In [LH96] and [GGSC96] two planes are used to parameterise the reduced 4D space of rays. Each ray is specified by its intersection with the camera plane (uv) and focal plane (st) (figure 1.2). The lower left edge of each plane corresponds to $(0,0)$ and the upper right one to $(1,1)$ with continous values in between. The two planes parameterise the four dimensional *line space*. The position, size and sampling density of these planes define a *light slab*. One slab cannot cover views from all sides of an object, but several slabs can be combined in one light field to allow a full reconstruction.

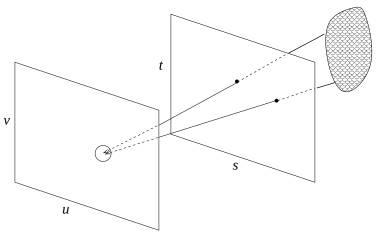


Figure 1.2: The light slab representation.

Generating a light field means sampling the continuous intensity function $L(u, v, s, t)$. For rendered images, a virtual camera is placed within the uv plane so that the viewing ray is perpendicular to the st plane. Now rays are collected that pass through the st plane and reach the center of the camera.

They form a sheared picture of the object, *i.e.* sampling $L(u, v, s, t)$ can be done by tradional rendering of pictures with the correct view transformation at several different camera positions.

The sampling density of the st plane is the resolution of these pictures and determines how large the reconstructed images can be made without a loss of quality. The sampling density of uv determines the quality of view point interpolation during the reconstruction. This sampling introduces aliasing, as the light field is likely to have high frequencies in both st (at edges of the object) and in uv (parts of the object that are out of focus and change their position in the images more rapidly as the camera moves). This can be avoided by combining a pixel filter for the st plane and an aperture filter for the uv plane.

A different representation is proposed by [TGFV98]. He suggests polar coordinates and claims that this representation allows the collection of ray intensities that are more relevant for an object, thus increasing the quality of the reconstruction without increasing the resources, as this representation can also be reduced to four dimensions and is as simple to handle as the light slab. [CLF98] also uses a different parameterisation and sampling pattern to overcome the problems caused by uneven sampling of the line space with slabs. However, generating such light fields cannot be done any more with conventional rendering engines unless they support the calculation of single rays.

This project therefore uses the light slab representation, but for the theory behind it it is only relevant that there is a sampling and reconstruction technique that captures a static scene. This is now called a *generic light field*. Several additional parameters that control the appearance of the object or its rendering, like time, illumination, surface attributes *etc.* are fixed when generating a generic light field.

For some applications however it is as important to change the object itself as it is to choose different point of views. Volume rendering is an example for that.

1.1.2 Vector Compression

In the Stanford implementation, vector quantisation ([GG92]) is used to compress the light fields. This takes advantage of the fact that light fields are highly redundant in all four directions. Within the image plane, adjacent pixels are likely to be similar. Adjacent values in the camera plane capture rays that depart into similar directions, so their intensity is usually similar, too.

This compression is done by dividing the light field into *tiles*. All tiles have a fixed size in each direction. They can differ between the different directions. Usual examples are 2x2x2x2 for u, v, s, t or tiles that do not stretch over both planes (2x2x1x1 and 1x1x2x2). For each tile, a vector of intensity values is constructed from the uncompressed light field by placing the intensity values of all rays within the tile one after the other.

Vector quantisation takes a subset of the complete amount of vectors during the *training phase* and finds a fixed number of vectors so that the mean-squared error between these *codewords* and the training set is minimal. Compression is done by replacing the intensity values of each tile in the light field with the index of the codeword that is closest to the vector constructed from this tile. Decompression is done by finding the tile a ray belongs to, looking up the codeword based on the index stored for this tile, and taking the intensity values that belong to the ray from this codeword.

For a tile of 2x2x2x2 and 16384 codewords this results in a vector length of $2^4 * 3 = 48$ bytes and a compression rate of 24:1, because the 48 bytes can be replaced by a two byte index number.

1.1.3 Rendering Light Fields

Although texture mapping can be utilised to speed up light field rendering ([SCG97]), this requires special hardware and thus goes against the intention of this projects. Therefore the performance restrictions of a pure software implementation have to be kept in mind.

This project is based on the Stanford `lightpack 1.0a`[1] source and performance measurements have been done with this implementation, but the general issues are independent from the actual implementation.

After sampling the continuous $L(u, v, s, t)$ function, u, v, s, t now represent integer values in the range from 1 to nu, nv, ns respectively nt. Because of the uniform sampling, each index can be mapped to the originally continuous value by dividing by the number of samples of the parameter.

Rendering a light field is done by extracting those rays from the light field that pass through the image plane into the camera centre (figure 1.3). As pointed out in [LH96], mapping from screen coordinates x, y to the line space

[1]http://www-graphics.stanford.edu/projects/lightfield/

with light field coordinates u, v, s, t and vice versa is a projective map and only involves linear algebra (multiplying with a 3x3 matrix). More details can be found in [GGC97].

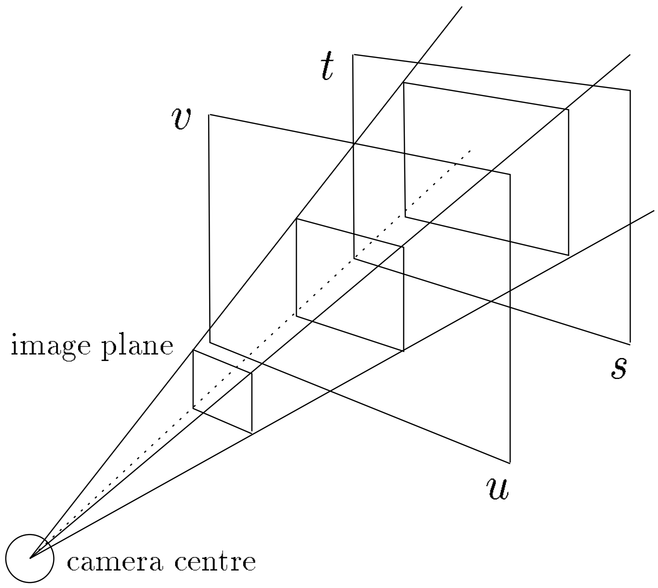


Figure 1.3: Rendering an Image from a Light Field.

Rendering an image from a light field involves two steps: *rasterisation* of x, y coordinates in the image plane and their mapping to the discrete u, v, s, t line space coordinates of each slab as available in the sampled light field. For each slab, this can be implemented efficiently using a scan line approach. For each row in the image plane start and end point are converted to line space. Because the mapping is a linear projection, the u, v and s, t values between them form two lines again and can be easily computed by continuously adding the difference between their end points divided by the number of pixels in the image row.

In the Stanford implementation, one function calculates this scan line of u, v, s, t indices for each row in the image. It also determines the differences between the desired continuous u, v, s, t values and the left respectively bottom neighbour in the discrete raster of samples and uses this difference to calculate the factor for a linear interpolation.

The second step is *resampling* of the intensity function for each pixel. For each scan line, a second function is called with the information calculated in the first step. It extracts and/or interpolates the intensities for each pixel. If the nearest neighbour is used for all four parameters, only one entry in the array of rays is extracted. For full interpolation in both planes, 16 rays are extracted and in doing so quadrilinear interpolation is performed to calculate the actually requested ray intensity. In practice, the *st* plane is usually sampled at a higher density to allow large images, so the nearest neighbours can be used in this plane, whereas *uv* should be interpolated to avoid artifacts.

The light field itself is organised as an ordinary array:

```
Entry_t LightField[nu][nv][ns][nt];
```

The entries of the light field are either the intensity values for the red, green and blue colour channel or, if vector compression is used, an index into a table that contains these intensities. In both cases extracting a ray is a simple address calculation plus one more table lookup for vector compression. It is not possible to optimise cache usage within this array by arranging the entries in a different order, because the required scan lines hardly ever align to the borders of the *uv* or *st* plane.

Profiling shows that during rendering the vast majority of time is spent within these two functions (over 90%). Merging them does not improve performance because overhead for the function call is neglectible compared to the time spent in both functions. Actually merging the two loops themselves that both iterate over the scan line is complicated by the initialisation of the resampling loop. Increased register pressure because of the increase of loop invariant variables might spoil the improvements gained by higher instruction level parallelism (ILP), so this optimisation has not been tested.

The ratio between both functions largely depends on the amount of interpolations done during resampling. For simply extracting the nearest neighbour, over 75% are spent on rasterising the scan line. With full interpolation the ratio is inverted with over 60% of the time doing the resampling.

1.1.4 Examples

Generating a light field is always a tradeoff between quality and size: larger planes capture more views of the object and higher sampling density improves the quality. Sampling at a low density has to be compensated by filtering the original images and thus increases the blurring of the resampled images.

Light fields with a good quality in the Stanford light field archive[2] are about 8MB large with vector compression and can be reduced to 25%–50% with `gzip` for storage on disk.

For volume rendering aperture prefiltering is not supported by either `VolVis` or `Volpack`, so in the light field archive example of a light field generated with `Volpack` a very dense sampling of the uv plane had to be used, increasing its size to 9.5MB with an image size of only 128x128.

Synthetic aperture could be added to `VolVis` with the same methods used for ray tracing (*stochastic sampling*), but it is difficult to implement in `Volpack`, because the shear-warp technique relies on the fact that it does not have to cast single rays.

Figure B.1 (see appendix B for colour figures) compares the rendering of light fields produced from the volumetric data of a head. The same slab geometry was used, but the sampling density for u in the right image was twice as high as for the image on the left. The edges of the skull move rapidly when rotating the object around its vertical axis. Because of the low sampling and the missing synthetic aperture this part of the object "jumps" between the sampled positions.

With the higher sampling this artifact is less obvious, but it cannot be avoided. Prefiltering would avoid it by blurring these critical areas. In this project, the effect of using a synthetic aperture was approximated by generating five images for each sampled position and averaging them. This allows a lower sampling to be used and reduces the number of images that are stored, but not the number of images that have to be generated.

One image is placed in the centre of the camera lense and four are spread across the area of the lense, one in each direction. Each image captures rays that would pass through the lense at different positions and onto the image plane, so averaging them approximates the integral that would have to be calculated to simulate a lense with finite aperture ([LH96]).

This reduces the artifact, as the middle image shows, but it also increases the blurring of the images in other parts. For volume rendering fine details might be important, so in the rest of this project prefiltering was not used. A high sampling density is assumed instead.

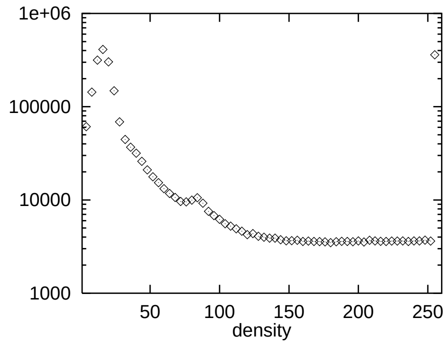
[2] `http://www-graphics.stanford.edu/software/lightpack/lifs.html`

1.2 Volume Rendering

1.2.1 Introduction

Volume rendering produces images from a 3D array of normally sampled density values. Instead of constructing a surface from the data and rendering the surface, the data is rendered directly by casting rays through the volume and accumulating light intensities. The volume is divided into small cubes (*voxels*) and a density value is obtained for each one. The illumination model assumes that each voxel is filled with pixels of one or more materials. The contents of a voxel can be *classified* based on its density value if the densities of the materials are known. Then different colour can be assigned to materials and therefore voxels. [Kau96] is a comprehensive introduction to this technique and gives an overview of recent developments.

Figure 1.4 shows the density histogram of the main example of this project, a partial CT scan of a a human head[3]. Three major materials can be identified that contribute to the histogram: bone, brain, and blood vessels.



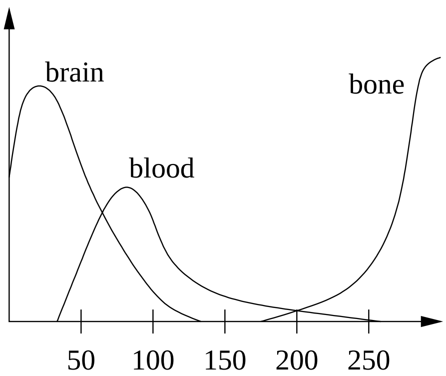


Figure 1.4: Histogram of the human head data set (logarithmic scale) and the materials contributing to it.

Different colours were assigned to the materials to distinguish voxels with different densities. A greenish colour was used for the brain to distinguish it from both blood (red) and bone (grey).

[3]Courtesy Department of Clinical Neurosciences, Western General Hospital.

1.2.2 Opacity Mapping

In the volume illumination model the opacity of each voxel is important to determine how much light is reflected from the voxel itself and how much can pass through it. The opacity is derived from the density of a voxel by a mapping function. This function could have any shape, but a simple ramp is versatile enough to produce good results as it follows the reality by making less dense material translucent first and allows sufficient control over the opacity of interesting parts of the material.

The advantage of a ramp for this project is that it can be specified by only two parameters, its position and steepness. The relevant values for the position are determined by the resolution of the density values in the volumetric data. This is usually a byte, so values from 0–255 are possible. The steepness is measured by the width of the linearly increasing part of the ramp, so a lower limit is zero. An upper limit and the resolution are arbitrary, but usually the integer values up to 255 are useful.

1.2.3 Examples

In figure B.2 the right hand side shows pictures of a human head, rendered with different combinations for steepness and position. The most interesting area is the lower left corner where the brain becomes translucent and reveals the blood vessel. For completely opaque material, the brain is very shiny and empty parts are clearly visible. As soon as the start of the ramp is placed beyond the density of the blood vessel, the images hardly change any more. For the largest values of position and steepness the bone becomes less opaque and its surface is therefore more dull.

On the left hand side every pixel in the images represents the average of one (coloured images) or all (the second grey image) colour channels of a complete picture created with the corresponding ramp position and steepness.

The average overall intensity has been calculated without any bias, *i.e.* it is simply the average intensity of the three basic colours for each pixel. Each image was normalised to emphasise the differences, so even areas that are now black represent images that are not black but only dimmer, and areas which are very bright are less bright in the real image.

The image above the average intensities is the average opacity for each parameter pair. For each density value, the number of voxels with this density is multiplied with the opacity for this density as defined by the current ramp.

These products are added up and divided by the total number of voxels. Because of the large number of empty voxels, this average opacity is rather low, so the image was normalised again.

This example illustrates some imporant aspects: the effect of each parameter is not linear and they depend on each other. Changing the steepness with a ramp at a low position affects the opacity of the brain and therefore causes large changes in the images. A very steep ramp causes more rapid changes, again more in the area relevant to the brain.

The amount of change is correlated with the average opacity. If the opacity changes, the images change, too. This correlation is limited, though, by the colour mapping: in this example the colour intensity of the brain has been reduced for voxels near the density of the blood, so moving the ramp at the beginning leaves voxels partly opaque that are very dark and effectively hide the background until they, too, become translucent enough to reveal the blood vessel and the bone. In general, making one material translucent to reveal another one can have a considerable effect on images, even if the average opacity does not change as much.

1.2.4 Implementations

The inherent problem of volume rendering is that an implementation has to look at the complete data set to render an image. For large data sets (*e.g.* $256^3 = 16$ MB or more) this inevitably becomes slow.

There have been several attempts to improve the rendering performance. Some use special hardware, other parallelise the rendering and/or distribute it over several machines. Solutions without this special requirements use precalculation with the drawback of increased memory demand and slower startup times.

Two software solutions are available for free non-commercial usage. Both have been used in this project.

1.2.4.1 `VolVis`

`VolVis` ([AHH+94]) is a general purpose implementation which may use parallel processing, but otherwise is not meant to be fast. Its main strength are the flexible rendering methods. It casts rays through the original data and has been extended to support ray tracing effects. For example, mirrors can be placed in the scene to show parts that would be invisible otherwise. Interactive rendering rates are completely impossible with this system even on a fast PC or workstation.

1.2.4.2 `Volpack`

`Volpack` ([LL94]) uses a *shear-warp* factorisation of the viewing transformation to render images. This speeds up rendering because rays are aligned to the original data and a scanline-based approach can be used. This produces distorted images that have to be warped to produce the final image.

It also *preclassifies* the volumetric data based on the current opacity and colour mapping. This produces an compressed and more efficient representation of the data that can be rendered faster. This classification has to be calculated each time the mapping changes and takes considerably longer than rendering an image. Again, this classification can be speeded up by calculation a *min-max octree* of the original data. Its size can be chosen, but useful sizes can become quite large. On a machine with 64MB main memory even the swap space was exhausted when trying to classify original data of approximately 30MB.

Interactive rates are possible for smaller or down-sampled data, though, but no special effects can be rendered.

1.3 Aims of this Project

As image based rendering techniques in general, light fields are not dependent on the complexity of a scene. They are versatile enough to replace traditional rendering of a scene. Volumetric rendering on the other hand is very complex both because the data sets can be very large and because special effects are expensive to calculate.

The idea is that a combination of both techniques might provide interactive rendering of volumetric data without requiring special hardware or extremely powerful machines. This defines the key issues of this project. Extend light fields to volume rendering so that:

- an image rendered from an extended light field shall be similar to the one rendered from the volume directly (*quality*).

- the extensions are simple enough to use "average" machines for rendering of an extended light field at interactive rates (*performance and memory demand*).
- Special hardware might be used, but should not be necessary.

Volume rendering is not the only example where changeable parameters affect the scene, so another issue is desirable:

- Keep the extension general enough to handle other parameters and rendering methods.

1.4 Overview

In the next chapter, the basic approach for extending generic light fields to varying scenes is described and consequences for performance and resource requirements are discussed. The following chapter explains how the additional parameters can be sampled efficiently. Then the implementation of this theoretical discussion is presented and finally theory and implementation are evaluated.

Chapter 2

Extended Light Fields

2.1 Basic Approach

To allow varying scenes, more information than just the intensities of rays into different directions of 3D space have to be stored in a light field. There are two possible approaches:

2.1.1 Enrich Information per Ray

Instead of just storing the intensity of light along each ray for one (grey scale) or three channels (colour), problem specific information could be stored that allow the calculation of the light intensities for this ray based on the current values for the additional parameters.

This has been done by [WHON97] to realise controllable illumination: they represent the bidirectional reflectance distribution function (BRDF) using spherical harmonics. 16 to 25 coefficients per light source and ray proved to be enough to reconstruct the light intensities of every ray in their examples sufficiently well. Several light sources can be handled at increased computational and storage costs. Their prototype supports up to three light sources at interactive frame rates.

In general, the calculation of the light intensities could be done either each time a ray is extracted or for the whole light field at once each time a parameter is changed. In the second case, rendering performance without changing parameters is the same as for generic light fields because effectively a generic light field has been derived from the extended one. The drawback is that changing a parameter can take considerably longer. On the other hand, calculating the light intensities while extracting rays would decrease the frame rate permanently. A compromise could be to calculate rays as required and to cache this information.

The approach of storing additional information per ray has not been pursued in this project. If an efficient solution could be found for volume rendering at all, it would probably restrict the illumination model and the effects that could be used. It could not be applied to other forms of image rendering either.

2.1.2 Precalculated Intensities for Different Parameters

Strictly speaking, generic light fields do not sample a function $L(u, v, s, t)$ but rather a function of higher order with the additional parameters fixed at a constant value: $L(u, v, s, t, p_1, p_2, \ldots, p_n)$[1].

A straight-forward attempt could extend the sampling and reconstruction of this function to this $4 + n$ dimensional domain. Each parameter $\#i$ could be sampled within a certain range at np_i regularly spaced positions $P_{i,j}, 1 \leq j \leq np_i$. This results in $\prod_{i=1}^{n} np_i$ n-tuples of parameter values that all require a complete generic light field.

Figure 2.1.2 illustrates this for two parameters by representing a generic light field with a three dimensional box. Both parameters are not necessarily discrete or the sampling complete, so that there are parameter values that are not covered by these generic light fields.

Rendering an image from such an extended light field could be done by linear interpolation between the light intensities extracted from the generic light fields. To obtain the intensity of a ray that belongs to the tuple $(\overline{p}_1, \overline{p}_2)$, the intensity of the rays with the same u, v, s, t values from four generic light fields are required. Two interpolations along one parameter and one along the other have to be done. In the general case, 2^n original rays and $1 + 2 + \ldots + 2^{n-1} = 2^n - 1$ interpolations are necessary.

Alternatively, rendering could simply pick the nearest neighbour for one or more parameter. In the simplest case all parameter values are approximated by their nearest neighbour so that always exactly one generic light field is used for rendering.

---

[1]The notation used in this project is listed in appendix A.

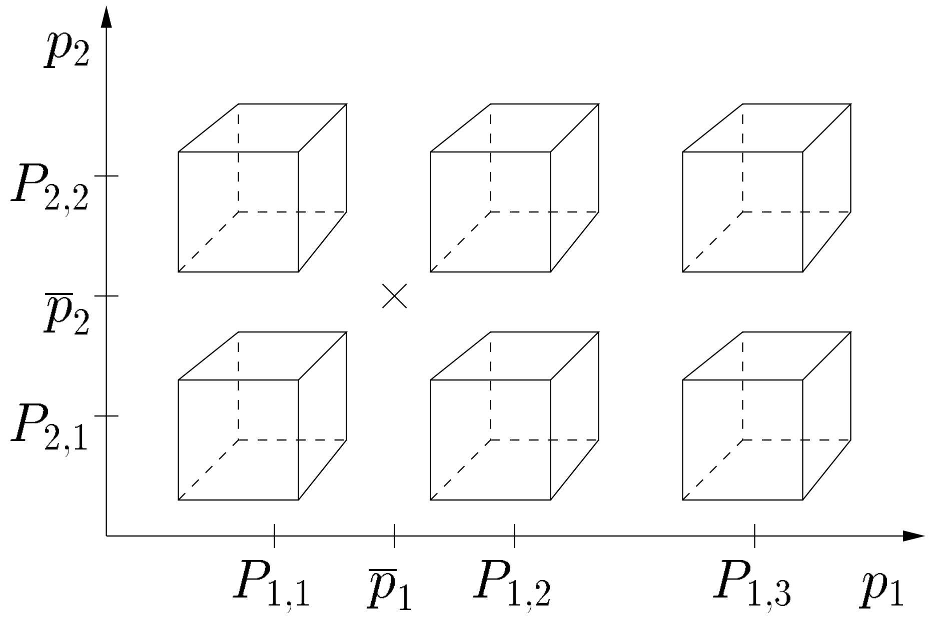


Figure 2.1: Generic light fields within a 2D extended light field.

Obviously, a large number of parameters or a high number of sampling positions within the additional parameters lead to an explosion of storage and CPU requirements. This project follows this approach none the less because of its generality and flexibility. It proposes methods to cope with its basic problem and evaluates for which applications this approach is feasible. *Extended light field* from now on refers to a collection of *generic light fields* that each correspond to a n-tuple of parameter values.

2.2 Efficiency and Implementation Issues

The performance restrictions of a pure software implementation are relevant to the resampling theory, so it is worthwhile to look at implementation issues first.

2.2.1 Rendering Extended Light Fields

The decision to gather several generic light fields into one extended one does not answer how these light fields should be stored or parameterised.

Each generic light field could have its own slabs and therefore (potentially) different u, v, s, t parameterisations. This means that the scan lines have to be calculated separately for each generic light field. If the geometries of the slabs are different, then complete images have to be stored temporarily because generating an image means iterating over the (in this case different) slabs that contribute to it. Clearly this is very memory demanding and could reduce frame rates considerably due to the (compared to CPU performance) relatively slow memory access in modern machines.

This disadvantage definitely outweighs the advantage that different slab configurations might be more adequate for different parameter tuples. This is only the case for objects that change their shape or preferred viewing direction considerably when changing the additional parameters, which is seldom true for volume rendering. In the example data set the outer shape is defined by the skull and the main focus is always the big blood vessel within the brain, so it is appropriate to always place the uv plane within the brain centred on the blood vessel and the st plane where the viewer is likely to look at it.

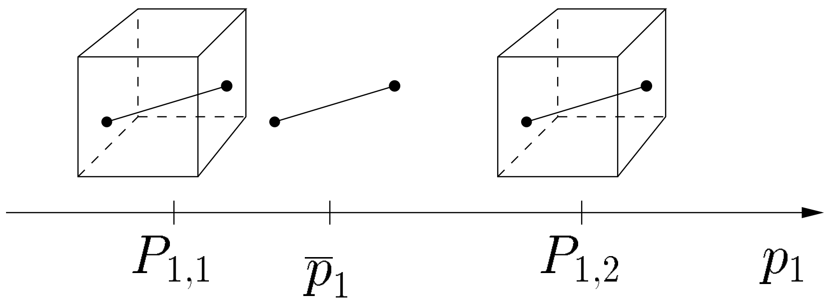


Figure 2.2: Identical slab configurations guarantee identical scan lines.

Figure 2.2 illustrates the advantage of having the same geometry in all generic light fields. The rasterised scan line is the same in both light fields, so the image row for the desired parameter $\overline{p}_1$ can be calculated by resampling both image rows for $P_{1,1}$ and $P_{1,2}$ and interpolating between them. This might be of advantage if there is hardware support for interpolating between rows or complete images, but lacking that it is better to do this additional interpolation per pixel in the inner loop, thus increasing register and cache locality and ILP.

2.2.2 Resampling Extended Light Fields

Not only the rasterised scan lines are the same. Resampling the continuous intensity function for a specific tuple of extended parameters can also take advantage of the fact that neither the required generic light fields nor the resampling within the extended parameters changes while rendering a slab. Therefore all the start address calculations to access the correct generic light fields and the calculations of weights for the reconstruction can be done in advance outside the rasterisation function, *i.e.* they do not affect rendering performance at all.

In contrast to sampling the uv and st plane where neighbouring samples and interpolation weights have to be calculated for every pixel, the highly regular sampling of the extended parameters mentioned before is not necessary to achieve high performance. Instead, it is viable to sample at arbitrary positions in the additional parameter space and to use advanced resampling kernels and not only the computational efficient linear one. However, the number of samples and arithmetic calculations to actually perform the resampling is still crucial.

2.2.3 Vector Compression and Memory Organisation

The regular sampling of the additional parameters leads to highly similar generic light fields, because in large areas changing a parameter does not change the resulting pictures much, as in the head example (figure B.2) when the ramp starts at a position that only leaves the bone opaque: then changing the steepness has hardly any effect. This redundancy could be used to reduce storage requirements by extending the vector compression to the additional parameters.

For regular sampling the implementation was straight-forward: the tiles now extend into the $4+n$ dimensional space and a vector may consist of intensity values of rays from adjacent generic light fields. However, this increases the vector length exponentially. Training and compression time increase dramatically with larger vectors ([GG92]) and although this is elevated in the hierarchical table-lookup vector quantisation used by the `lightpack` source ([CCG96]), this implementation still does not handle the required vector lengths correctly.

Even if the compression would work properly, this approach would still require the huge amount of pictures to be calculated in the first place. It is much more efficient to avoid that by choosing sample positions more efficiently. In this case redundancy is removed by producing generic light fields that are as unsimilar to each other as possible. Vector compression cannot be extended to an irregular sampling pattern because the sampled positions cannot be covered by regular tiles and irregular tiles would produce vectors with little correlation between their entries respectively would lack entries completely (figure 2.3). Vector compression can still be applied to the generic light fields, though.

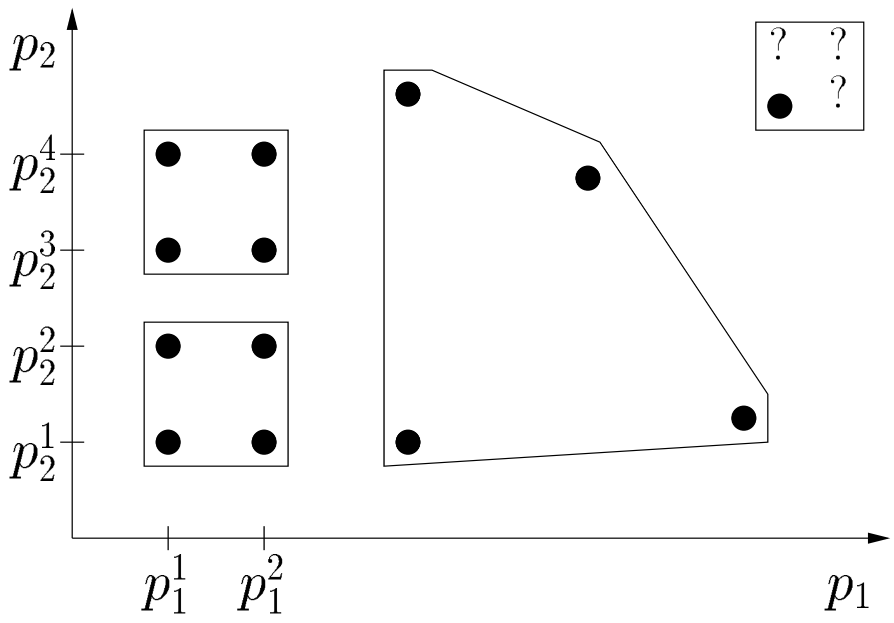


Figure 2.3: Vector compression tiles with regular and irregular sampling.

This figure also shows that it is no longer useful to specify coordinates for sampling points independently. It is more useful to simply enumerate all sampling points and to explicitly save the parameter tuple they belong to.

Then either the generic light fields could be indexed by their number (assuming column major ordering of entries, *i.e.* C-style)

```
Entry_t ExtLightField[num_gen_lfs][nu][nv][ns][nt]
```

or the entries of all generic light fields could be stored interleaved

```
Entry_t ExtLightField[nu][nv][ns][nt][num_gen_lfs].
```

The first approach has been preferred because it allows images to be fed into the data flow one after the other and parameters do not change their values with each new image. This might require additional calculations[2]. Cache usage can be expected to be better during rendering, too, especially if only one generic light field is used.

[2]VolPack for example can generate images of the same classified volume from different view points quickly, but takes quite some time when reclassifying it with new parameters.

Chapter 3

Sampling of Parameters

The generation of a light field is a sampling process that extracts the value of a continuous function at discrete positions and for discrete colour channels, which are in this chapter indexed with another discrete parameter $c = 1, 2, 3$: $L(s, u, v, s, t, p_0, p_1, \ldots, p_n)$. Rendering is the reconstruction of this function based on the sample values and the desired function arguments.

After dropping the uv parameters, each tuple in the additional parameter space maps to colour images that all show the scene from the same point of view. For volume rendering this array of images was shown in figure B.2 and the effect of changing the additional parameters on the whole pictures was analysed.

Dropping the st and c parameters also is the same as picking the colour channel of one pixel within all these images. This chapter deals with sampling and reconstruction of one such function with one parameter first before the results are extended to a collection of several functions and more parameters.

Each of these functions defines a surface in multidimensional space and the problem of reconstructing them is similar to surface reconstruction in multidimensional space. [Bur88] presents a method to reconstruct a grey scale image based on random sample points. He calculates a sequence of *moment images* for each sample point that capture areas around the point. They grow exponentially in size. Polynomial surfaces are fitted to the surrounding area of each sample point. This results in a set of low pass filtered versions of the original image. This multi resolution representation can be combined into a reconstruction of the original image.

Such an approach for the extended light fields would require calculations based on all available samples. It does not take advantage of the fact that the sample points can be chosen to suit the reconstruction. It also does not extend to reconstruction of several functions at once.

Therefore a much simpler approach is pursued: starting from a dense regular sampling of the continuous function, sample points are omitted so that the error introduced by sampling irregularly at a lower density remains as small as possible. While omitting samples the most suitable reconstruction kernels are calculated and stored together with the positions they belong to.

This is an optimisation problem with two equivalent formulations:

- Omit as much sample points as possible without exceeding a threshold for some kind of error metric.
- Remove a fixed number of samples so that the error is as small as possible.

It is not possible to start with no samples and increase their number until the reconstructed function is close enough to the original, because nothing is known about it without sampling it, *i.e.* an error metric can only be defined based on the sampled values.

3.1 Reconstruction with Arbitrary Sampling

The reconstruction of the intensity value within 3D space is done with a generic light field technique, so it is valid to limit the scope of this chapter to reconstruction within the domain of the additional parameters. Therefore the parameters and indices u, v, s, t are omitted unless required in the current context. The parameter c and the subscripts to distinguish different additional parameters are also omitted to simplify the notation in cases where they are irrelevant.

In theory the intensity function is reconstructed by iterating over the total number of samples, multiplying each sampled value with its associated reconstruction kernel evaluated for the current parameter tuple and summing up these products:

$$\tilde{L}(p_0, \ldots, p_n) = \sum_{j=0}^{m} x_j * B_j(p_0, \ldots, p_n) \tag{3.1}$$

In practice however the basis functions only have a limited area where they are non-zero, so that the summation can be restricted to a much smaller number of sample positions.

This is essential to achieve interactive frame rates with extended light fields. Extraction of a ray intensity in 3D space already involves several memory accesses and interpolations (see section 2.2.1) and a complicated reconstruction in parameter space based on these extracted intensities would aggravate this multiplicatively.

These practical issues justify imposing the following restrictions on the form of the basis functions:

- The samples to obtain a value for a tuple should be only those that are directly adjacent to it, *i.e.* left and right neighbour in 1D, four corners in 2D, *etc.* If one or more parameters of this tuple have been sampled exactly, the number of samples should decrease accordingly.
- Reconstruction should not change with the current function that is reconstructed, *i.e.* should be the same for all colour channels and pixels within the images. Otherwise both rasterisation and resampling would become much too time consuming because reconstruction parameters cannot be precalculated any more and resampling would have to use different weights.

3.1.1 One Parameter, one function

The sampling theorem states that a signal can be reconstructed from its samples if it is sampled above the *Nyquist rate*, which is twice the highest frequency component of the signal ([FvFH90]).

The assumption is made that during analysis the extended parameters can be sampled at a higher rate than this lower limit, so that aliasing can be avoided without filtering the signal before sampling and small artifacts are not missed. This assumption is justified for volume rendering in section 5.3.

The following function will be used as an example:

$$L(p) = \cos((1 - exp(p/100)) * \pi) * 127$$

Figure 3.1 shows this function and its reconstruction using 21 evenly spaced sampling points and linear basis functions. The sampling is not dense enough to reconstruct the right part of the function properly with these simple basis functions.

Smoother basis functions might produce better results. Polynomial basis functions of variable degree d are used for two reasons:

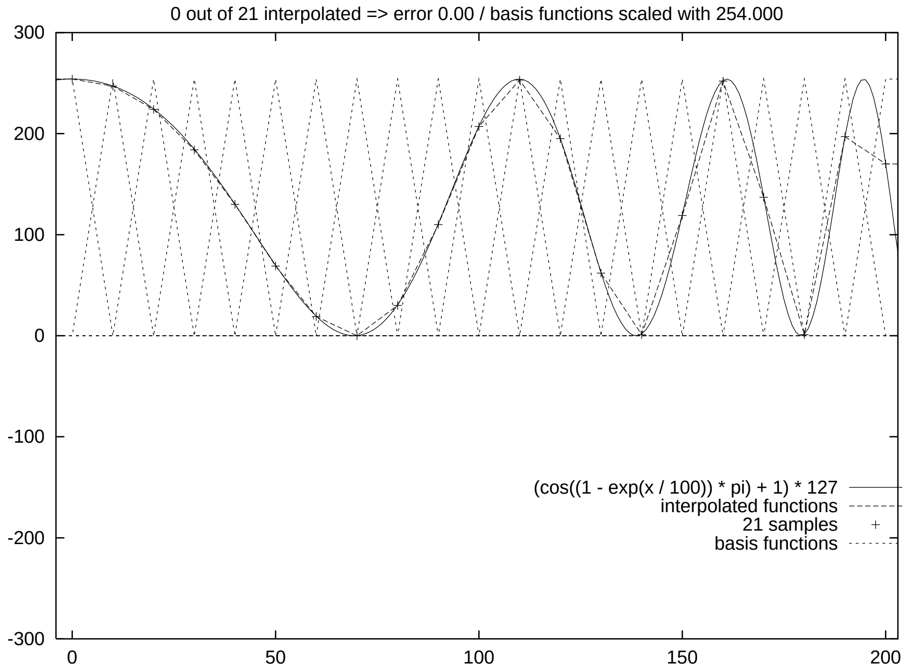


Figure 3.1: Example function with regular sampling and linear basis functions.

- solving equations to obtain their coefficients is simple
- increasing their degree provides increasing control over their shape.

It is useful to define them including the position of the sample they belong to. This way the coefficients themselves are independent from the sample position. The basis functions are only non-zero over a limited range, so they have to be specified piece-wise:

$$B_j(p) := \begin{cases} c_{j,k} * (p - P_j)^k + \cdots + c_{j,1} * (p - P_j) + c_{j,0} & \text{for} P_j \leq p < P_{j+1} \\ \text{analogous} & \text{for} P_{j-1} < p \leq P_j \\ 0 & \text{else} \end{cases} \tag{3.2}$$

This ensures that at most two basis functions are non-zero at any position. Furthermore the non-zero basis functions should sum to 1, *i.e.*

$$B_j(p) + B_{j+1}(p) = 1, P_j \leq p \leq P_{j+1} \tag{3.3}$$

This simplifies the evaluation of equation 3.1 by reducing the number of multiplications[1]

$$\begin{aligned} B_j(p) + B_{j+1}(p) = 1 \Leftrightarrow B_{j+1}(p) &= 1 - B_j(p) \\ \Rightarrow \tilde{L}(p) = x_j * B_j(p) + x_{j+1} * \underbrace{B_{j+1}(p)}_{=1-B_j(p)} &= x_{j+1} + (x_j - x_{j+1}) * B_j(p) \end{aligned} \tag{3.4}$$

It also implies that for each basis function only one part has to be specified. This is now always done for the right part. Furthermore, reconstructed values at sample points are equal to the sampled value, because all but the corresponding basis function are zero, so this function has the value one. The polynomials that are exactly determined by these restrictions are those of degree one, *i.e.* the linear ones used in figure 3.1.

If all the basis values were also limited to $0 \leq B_j(p) \leq 1$, then this would guarantee that reconstructed values lie between the sample values they are based on (*convex hull property*). This has the advantage that no clamping to the available range of intensity values (0–255 for a byte) is necessary during reconstruction because the original values are within this range and so is the

[1]However, still $O(n)$ multiplications are required for n relevant samples—see page 14.

result. However, this additional limitation is not desired yet and therefore not enforced.

Smoother polynomials can be specified by restricting the first derivative. A naive approach would simply set them to zero at their joints:

$$B_j'(P_j) = 0 \tag{3.5}$$
$$B_j'(P_{j+1}) = 0 \tag{3.6}$$

This provides two further criteria and cubic polynomials can be used instead, as in figure 3.2. However, zero is not necessarily the best value as the figure shows. It would be necessary to look at neighbouring ranges to find the correct value for the derivative, which was not done in this project.

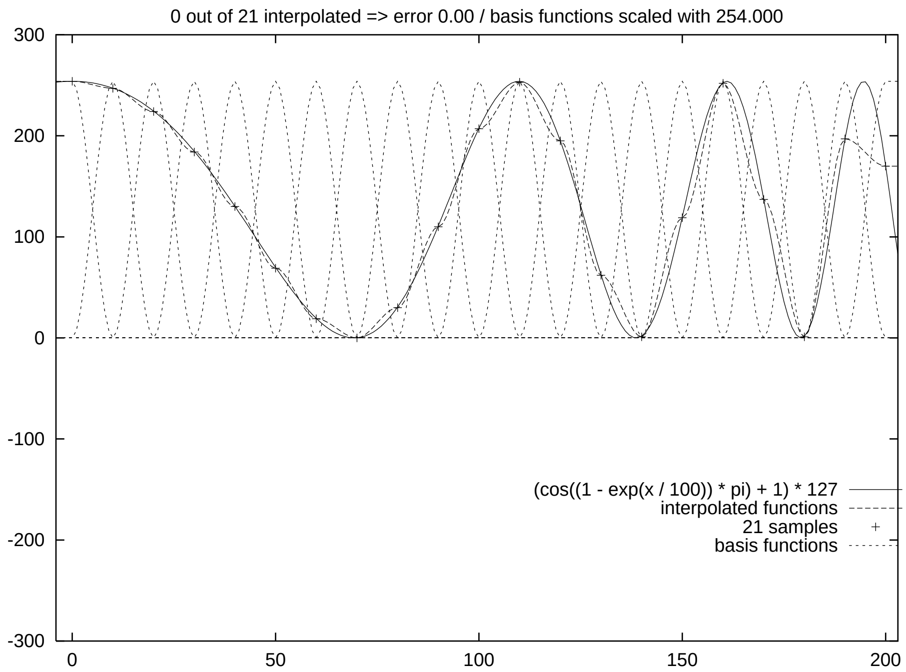


Figure 3.2: Reconstruction with cubic polynomials.

Instead, not using some of the sampled positions provides a better way to increase the degree by ensuring that the interpolated curve passes through the values at those positions. The restrictions for the first derivative are still used in the mathematical derivation to show how the cubic polynomials in figure 3.2 were obtained, but dropping these additional constraints again is simple and was done for the remainder of the work.

Let $B_j^d(p)$ be the basis function of a position where P_{j+d} is the next position that is used for reconstruction and where $P_{j+1}, P_{j+2}, \ldots, P_{j+d-1}$ have been sampled, but are not used for reconstruction. By extending the previous equations regarding the range and demanding that

$$\tilde{L}(P_{j+i}) \overset{(3.4)}{=} x_{j+d} + (x_j - x_{j+d}) * B_j^d(P_{j+i}) \overset{!}{=} x_{j+i}, 0 \leq i \leq d \tag{3.7}$$

$B_j^d(p)$ is exactly specified up to degree $d+2$ as follows:

$$\begin{aligned}
& B_j^{d\prime}(p) = (d+2) * c_{j,d+2} * (p - P_j)^{d+1} + \cdots + c_{j,1} \\
& \text{eq. 3.5: } B_j'(P_j) = 0 \\
& \quad \Rightarrow \quad (d+2) * c_{j,d+2} * (\underbrace{P_j - P_j}_{=0})^{d+1} + \cdots + c_{j,1} = 0 \\
& \quad \Rightarrow \quad c_{j,1} = 0 && (3.8) \\
& \text{eq. 3.7, } i = 0 : \tilde{L}_j(P_{j+0}) = x_{j+0} \\
& \quad \Rightarrow \quad x_{j+d} + (x_j - x_{j+d}) * (c_{j,d+2} * (\underbrace{P_j - P_j}_{=0})^{d+2} + \cdots + c_{j,0}) = x_j \\
& \quad \Rightarrow \quad x_{j+d} + (x_j - x_{j+d}) * c_{j,0} = x_j \\
& \quad \Rightarrow \quad c_{j,0} = 1 && (3.9)
\end{aligned}$$

From the equations in 3.7, d remain with $1 \leq i \leq d$. They can be transformed as follows:

$$\begin{aligned}
x_{j+d} + \underbrace{(x_j - x_{j+d})}_{=:\Delta x} * [c_{j,d+2} * \underbrace{(P_{j+i} - P_j)}_{=:\Delta P_{j,i}}{}^{d+2} + \\
+ \cdots + \\
c_{j,2} * (P_{j+i} - P_j)^2 + \\
\underbrace{c_{j,1}}_{=0 (\text{eq. } 3.8)} * (P_{j+i} - P_j)^1 + \\
\underbrace{c_{j,0}}_{=1 (\text{eq. } 3.9)} * \underbrace{(P_{j+i} - P_j)^0]}_{=1} &= x_{j+i} \\
\Leftrightarrow \Delta x * [c_{j,d+2} * \Delta P_{j,i}^{d+2} + \cdots + c_{j,2} * \Delta P_{j,i}^2] \\
+ x_{j+d} + x_j - x_{j+d} &= x_{j+i} \\
\Leftrightarrow c_{j,d+2} * \Delta P_{j,i}^{d+2} + \cdots + c_{j,2} * \Delta P_{j,i}^2 &= (x_{j+i} - x_j)/\Delta x && (3.10)
\end{aligned}$$

With equation 3.6 this yields $d+1$ equations. Writing them with the known terms as factors in front of the unknown coefficients $c_{j,d+2}$, $c_{j,d+1}$, $\ldots$, $c_{j,2}$

$$
\begin{aligned}
(d+2) * \Delta P_{j,d}^{d+1} * c_{j,d+2} + \cdots + 2 * \Delta P_{j,d}^{1} * c_{j,2} &= 0 \\
\Delta P_{j,1}^{d+2} * c_{j,d+2} + \cdots + \quad \Delta P_{j,1}^{2} * c_{j,2} &= (x_{j+1} - x_j)/\Delta x \\
\Delta P_{j,2}^{d+2} * c_{j,d+2} + \cdots + \quad \Delta P_{j,2}^{2} * c_{j,2} &= (x_{j+2} - x_j)/\Delta x \\
&\vdots \\
\Delta P_{j,d}^{d+2} * c_{j,d+2} + \cdots + \quad \Delta P_{j,d}^{2} * c_{j,2} &= (x_{j+d} - x_j)/\Delta x
\end{aligned}
$$

makes obvious that this could be written as the matrix-vector multiplication

$$
\underbrace{\begin{pmatrix} (d+2) * \Delta P_{j,d}^{d+1} & \ldots & 2 * \Delta P_{j,d} \\ \Delta P_{j,1}^{d+2} & \ldots & \Delta P_{j,1}^{2} \\ \vdots & & \\ \Delta P_{j,d}^{d+2} & \ldots & \Delta P_{j,d}^{2} \end{pmatrix}}_{=:M_j^d}
\underbrace{\begin{pmatrix} c_{j,d+2} \\ \vdots \\ c_{j,2} \end{pmatrix}}_{=:C_j^d}
=
\underbrace{\begin{pmatrix} 0 \\ (x_{j+1} - x_j)/\Delta x \\ \vdots \\ (x_{j+d} - x_j)/\Delta x \end{pmatrix}}_{X_j^d}
\tag{3.11}
$$

and solved by multiplying X_j^d with the inverse of M_j^d. This matrix is invertible, because $P_i \neq P_j$ for $i \neq j$. The coefficients could be derived directly, too, but calculating the inverse of M_j^d will prove to be useful later.

If first order continuity of the polynomial basis functions is not desired, it is sufficient to drop the first line and row in matrix M_j^d and vector X_j^d and to reduce the remaining exponents in M_j^d by one. C_j^d then contains the coefficients $c_{j,d}, \ldots, c_{j,1}$ and the "distance" d between sampling positions is equal to the degree of the basis polynomial.

3.1.2 Dropping Samples

Dropped samples can be exactly interpolated in all but one case: if the remaining samples have exactly the same value, then Δx is zero and X_j^d is undefined. The best guess for the basis function is a linear interpolation in this case, but as both values are the same, all positions between them will adopt this value and simply using a box basis has the same effect. In all other cases values can be extrapolated because the convex hull property is not enforced and the basis functions are constructed this way. Figures 3.3 and 3.4 show both cases.

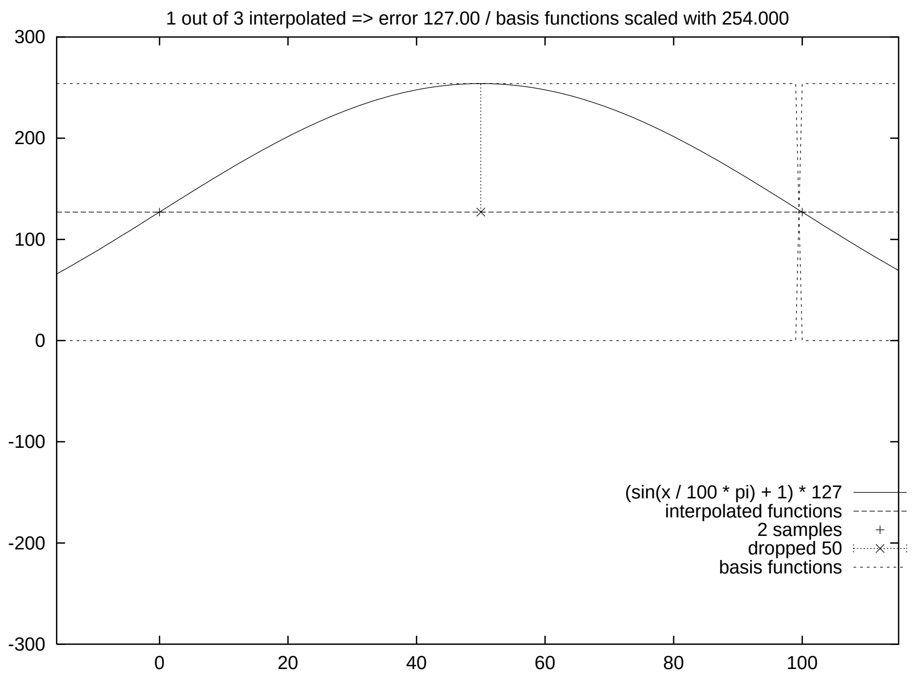


Figure 3.3: No reconstruction possible.

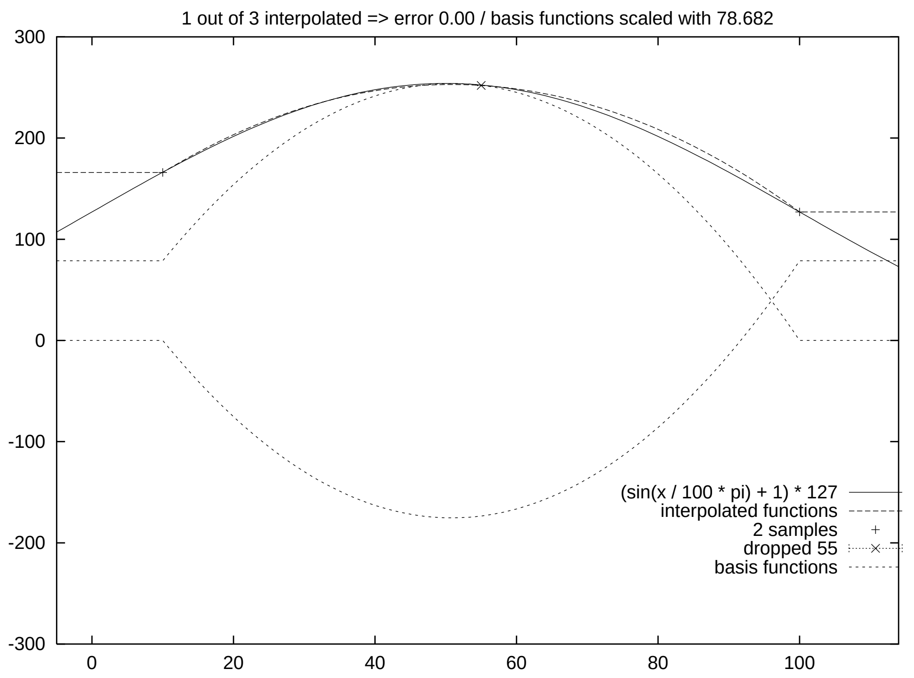


Figure 3.4: Extrapolating the middle value.

So far this is the only restriction on dropping samples. The other one is that extrapolation is not done for positions with only one contributing sample. This would only work for a single function where the function value can be scaled to the dropped values, but would not work well for multiple functions, because this scale factor could be quite different. Without extrapolation the remaining left- and rightmost sample values are simply taken over for all dropped positions to their left and right. Taking this into account introduces the tendency not to drop those samples so that interpolation is possible over a larger range.

In practice arithmetic inaccuracies cause slight errors, but this is irrelevant compared to the errors introduced with multiple functions in the next section. After scaling the parameter p for each basis function so that $P_{j+d} - P_j = 1$, the powers in M_j^d are kept smaller and many samples can be dropped before the first value is not exactly reconstructed (figure 3.5).

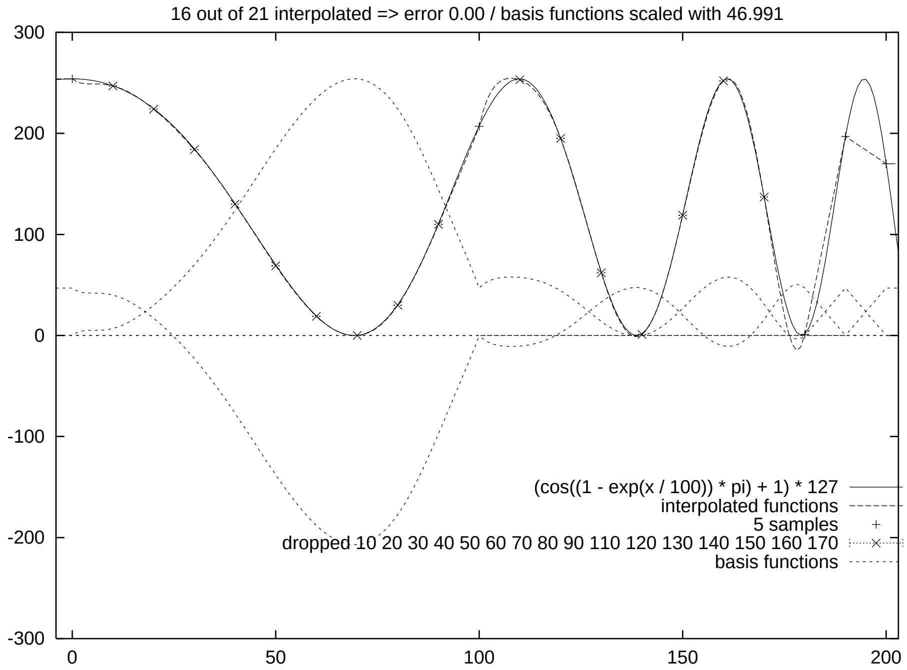


Figure 3.5: Complete reconstruction.

3.1.3 Multiple Functions

Most of all, errors are incurred as soon as the basis functions have to fit not only one function, but several at once, as required to achieve interactive frame rates.

Let k denote an index for these n_k functions. This is just a simplification to avoid the discrete indices c, u, v, s, t of the real application.

For each function $L_k(p)$ at position j and next sample $j+d$ we can calculate coefficients $c_{j,i,k}$ for basis functions $B_{j,d,k}$ as before. Now it is useful that M_j^d is independent from function values, so solving for the coefficients only involves one matrix inversion and several multiplications with different vectors $X_{j,k}^d$.

What we want though are coefficients $c_{j,i}$ of one basis function that exactly interpolates at least all function values at the end points and that produces interpolated values in the middle that are as close as possible to the dropped samples.

Taking the average of the coefficients for the different basis functions provides at least the first part:

$$c_{j,i} := \frac{1}{n_k} * \sum_{k=1}^{n_k} c_{j,i,k} \tag{3.12}$$

With this definition the general basis function is the average of all single ones:

$$\begin{aligned}
B_{j,d}(p) &= \sum_i c_{j,i} * (p - P_j)^i \\
&\overset{\text{def. } 3.12}{=} \sum_i \left(\frac{1}{n_k} * \sum_{k=1}^{n_k} c_{j,i,k}\right) * (p - P_j)^i \\
&= \frac{1}{n_k} * \sum_{k=1}^{n_k} \sum_i c_{j,i,k} * (p - P_j)^i \\
&= \frac{1}{n_k} * \sum_{k=1}^{n_k} B_{j,d,k}(p)
\end{aligned}$$

and start and end points are exactly interpolated:

$$\forall k : B_{j,d,k}(P_j) = 1, \; B_{j,d,k}(P_j + d) = 0$$

$$\begin{aligned}
\Rightarrow B_{j,d}(P_j) &= \frac{1}{n_k} * \sum_{k=1}^{n_k} B_{j,d,k}(P_j) \\
&= \frac{1}{n_k} * \sum_{k=1}^{n_k} 1 \\
&= 1 \\
B_{j,d}(P_{j+d}) &= \frac{1}{n_k} * \sum_{k=1}^{n_k} 0 \\
&= 0
\end{aligned}$$

All sample points in between might not be reached any more, as in figure 3.6. A weighted average could be used to interpolate some functions better than others, but no such bias was applied for the different pixels and colours in this case.

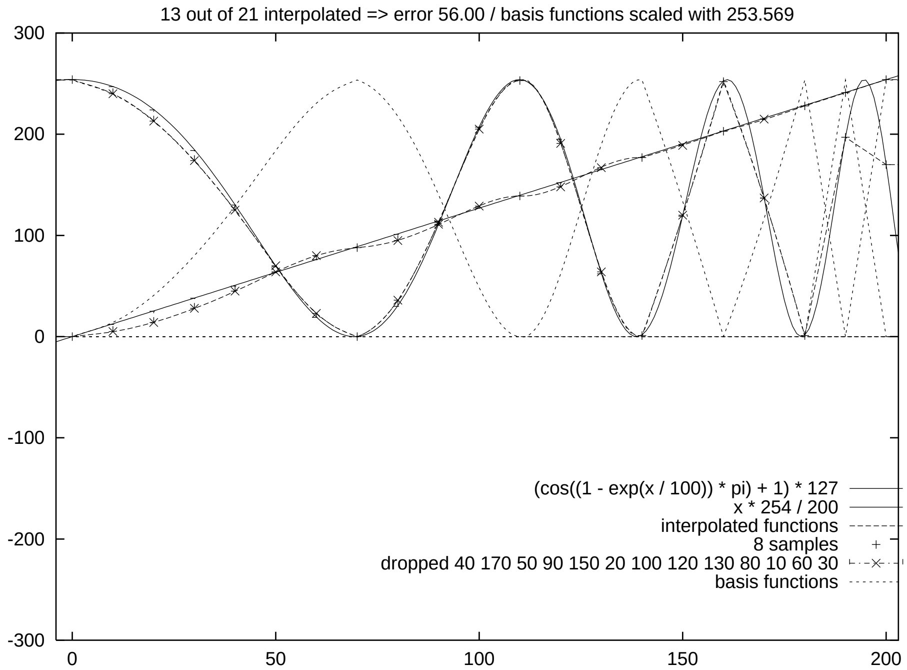


Figure 3.6: Errors incurred because of averaging different basis functions.

Simply using the average coefficients completely ignores the error metric and the effect of multiplying the difference in equation 3.4 by a wrong value. No matter what error metric is used, multiplying a large difference with a wrong value will cause a larger error than multiplying a smaller difference. In the extremest case of a difference of zero, this difference could be multiplied with anything without changing the error. Therefore it is better to use the difference $x_{j+d} - x_j$ as weight for the corresponding coefficient vector C_j^d.

This approach is still able to provide good interpolation if the basis functions are similar, as is the case when changing the parameter has a similar effect on the different functions. In figure 3.7 two functions have completely different function values, but the way they depend on the parameter is the same, so their basis functions are exactly the same and reconstruction is done as accurately as for the single function in figure 3.5.

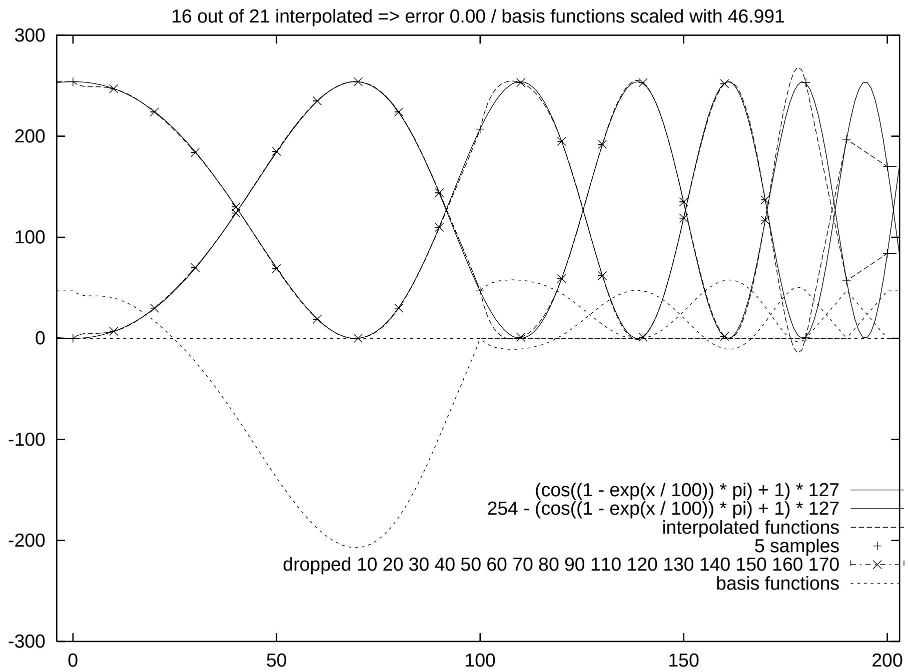


Figure 3.7: Complete reconstruction of samples of two functions.

3.1.4 Multiple Dimensions

Interpolating with only one parameter was simple because the relevant sample points are exactly identified, at least if one assumes that the closest sample points are always the best. There is no way to check this assumption without further sampling, but after dropping sample points the nearest neighbours could be skipped to test if sample points farther away are more appropriate to interpolate those samples that have been dropped. This would increase the complexity of choosing basis functions and sample positions considerably, and as the assumption holds for functions that have been sampled densely enough this extended search is not applied.

Reconstruction is already complicated by the fact that more than one parameter has to be interpolated. Each new parameter doubles the sample positions that are allowed to contribute during the reconstruction.

The basis functions now depend on more parameters and could have any shape within the area their sample supports, but in order to keep the complexity of finding them small a hierarchical approach is used. This also suits the implementation better. It is based on the one dimensional case discussed before.

Associated with each sample position are now n basis functions that determine the influence of the sample position along each direction. Another index is used to indicate this direction and therefore the parameter they take:

$$B^i_{j_1,\ldots,j_n,d}(p_i) \tag{3.13}$$

Reconstruction is done from the bottom up, *i.e.* first samples are reduced within one direction, then within the next until one value is reached. With two parameters one can think of reducing first in columns to obtain values that lie in the same row and then to reduce within this row, or vice versa (figure 3.8). The order in which the parameters are interpolated can make a difference because both the supporting samples and the basis functions can change.

Imagine sample points in a regular grid where the one in the middle is being dropped (point D). Before, it was exactly defined which samples were the nearest neighbours for position P in the plane and the direction of the first interpolation did not matter, but now there are two possibilities (jumping over existing samples has been excluded already).

Basically it is a choice between interpolating first in rows or first in columns. The relevant existing sample values and basis functions are then found by looking for remaining samples in the closest rows respectively columns. This decision can be made for P by looking at how the dropped position D is reconstructed. If its reconstruction from the values within the same column is better than the one from values in the same row, then it is likely that interpolation for A in columns first is better, too.

This would be the case if changing parameter p_1 introduces larger changes

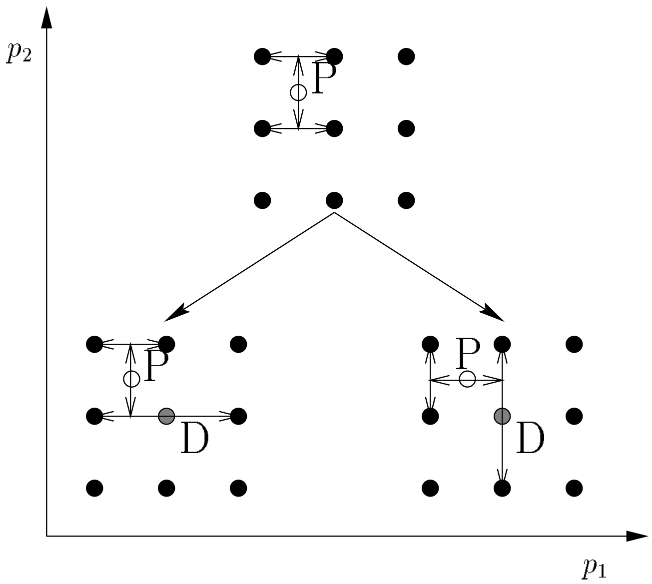


Figure 3.8: Two possibilities to reconstruct a value at position P.

in the images than changing p_2. Larger changes also imply that the error due to the averaged basis function is larger so that it is better to keep the distance between samples short when interpolating along p_1, *i.e.* to do the interpolation over the larger range within the columns.

This criteria does not extend well to higher domains. In a 3D space a dropped sample can be interpolated within three directions, but this would deliver only one criteria whereas a position that is not aligned with any existing sample could be reconstructed in three different directions first. One would have to look at the rate of change in the relevant area to reach a decision or simply prefer one direction.

3.2 Choice of Sampling Positions

So far sampling points have been dropped at random. To achieve the goal of finding the most efficient or at least a good sampling pattern and reconstruction basis it is necessary to define an error metric first.

3.2.1 Error Metrics

The error metric is applied to a set of functions that form an image in this application. It compares either images with different additional parameters or a sampled image with the reconstructed one at the same position.

The simplest metrics are mean-squared error and average absolute difference:

$$MSE(L_1, L_2) = \sqrt{1/k * \sum_k (L_1(k) - L_2(k))^2} \tag{3.14}$$

$$ABS(L_1, L_2) = 1/k * \sum_k |L_1(k) - L_2(k)| \tag{3.15}$$

The first one is biased towards avoiding large differences in favour of several smaller ones. Both are independent from the actual image size. The metric should represent the subjective perception of the differences between the images and therefore should take into account how the human eye perceives colour and details in a scene.

A difference in the blue channel might be less offending than a similar change in the red one, because the eye is less sensitive to blue. Changes are also not perceived linearly, *i.e.* the same absolute difference between two low intensities is more obvious than between two high intensities. Therefore different weights for different colour channels and a logarithmic correction of the difference based on the compared intensity levels might be applied.

Finally not all pixels within the image are as important as others. Some fine details in a limited area might be more important than the large rest. This could be taken into account by using predefined weights for each pixel or assigning such weights dynamically, *e.g.* by emphasising areas of larger change.

None of this has been done though in this project. Mean-squared errors were used for all of the examples unless noted otherwise.

3.2.2 Optimising the Selection of Sample Positions

Optimisation works on a rectilinear grid of sampled positions. This is required so that interpolation can be done along each parameter, but spacing between samples does not necessarily have to be the same. The region covered by the original samples should only be convex so that no holes have to be taken into account. In the head example very wide ramps that already start beyond the part where brain is still visible are not really interesting, so it would be reasonable to restrict the parameter area that is covered by the extended light field to the lower left triangle instead of a square area.

In the examples a regularly spaced sampling of a square has been used to simplify the implementation and because it demonstrates that large areas can be interpolated with little error. It is also simpler to use because parameters can be specified and selected within their ranges independently from each other.

Starting with this predefined grid all positions have to be sampled in advance. Then gradually positions are not used for interpolation any more, but instead interpolated themselves with the remaining ones. The original sample values at each position are retained and compared with the currently best interpolation for this position. An error metric has been chosen and is used to measure this error. Each single error contributes to the global error measure for the current set of dropped samples.

Again, different ways to compute this global error are possible. Simply summing them up takes the complete covered parameter area into account equally. More emphasis could be put onto optimising a certain area of interest by using weights. This global error is not independent from the number of dropped samples, which makes it difficult to judge the quality of a set of positions. Therefore the average is used instead of the sum.

Let d be a vector of length m with binary decision variables for each position in the parameter grid, *i.e.* $d \in \{0,1\}^m$. 0 represents a dropped and 1 a sampled position.

Then the global error metric is a function $E : \{0,1\}^m \rightarrow R$ and the two variations of the optimisation problem can be formalised to finding a $d \in \{0,1\}^m$

- with $\min \underbrace{\sum_i d[i]}_{\text{number of samples}}$ such that $E(d) \leq \hat{E}$
- or with $\min E(d)$ such that $\sum_i d[i] = s$

for an error limit $\hat{E}$ or a desired number of remaining positions s.

Solving these problems without making any assumptions is hard. In the first version 2^m vectors would have to be tested as no predictions from the error of one vector to the errors of others could be made. For the second version a search tree of depth s and branching factors $m, m-1, \ldots, m-s+1$ has to be searched. Without checking for duplicate ways to nodes this yields $m!/(m-s)!$ tests, but even with avoiding duplicate tests there are still $m!/[(m-s)!s!]$ combinations.

Clearly this is not feasible even for modestly sampled parameter areas with *e.g.* 16x16 sample positions. $!m/!(m-s)!s$ might be small enough for very small values of s, but usually still around 10 remaining positions are required. A good value for s is hardly ever known in advance, so it would still be necessary to calculate the minimum error for several other and probably larger values.

Fortunately assumptions can be made about the effect of dropping samples: usually it is always better to interpolate a position from positions closer to it. Then the error for a dropped position can only increase when the position it was based on so far are dropped, too. One can also assume that a position that was not relevant to support its surrounding area and that could be interpolated easily will not become important later either.

These observations support the decision to choose one position for being dropped and to sample it *never* again, even if further positions around it have been dropped meanwhile. This is the underlying concept of a *greedy* algorithm. In this case it is a good heuristic to guide the search for a good solution, but not necessarily optimal, because dropping further samples may affect the error incurred by positions that were dropped earlier.

The greedy algorithm requires a prediction of what effect removing each sample would have in the current situation. This is one of the entries that is maintained for each position within the original grid:

- the original image
- the error for interpolating it with the remaining samples (even if it has not been dropped yet): now called *local error*
- the increase in the global error if this position was dropped (*accumulated error*)
- whether the position has been dropped already (*dropped flag*)
- basis function coefficients for interpolation along each direction and for different distances
- how this positions is or could be interpolated: primary direction and source samples

Basis functions are calculated on demand. They only depend on the original images and the desired distance, so once coefficients have been calculated they can be saved and stored for later reuse.

Therefore initialisation is done as follows:

1. Render or load original images.
2. Mark all positions as being not dropped and having no coefficients yet.
3. Mark each position as being dropped, interpolate it, set its local error, and finally reset the “dropped flag” to “sampled”.
4. Set all accumulated errors to the local error: dropping a position would have no other effect yet as all other positions are still being sampled themselves.

When dropping a sample, this state has to be kept consistent. Simply dropping a sample and recalculating everything would increase the complexity of this basic operation dramatically, so the recalculations have to be restricted to those that are really necessary.

The effect of a sampled position is limited by its sampled neighbours in row and column. Beyond them these neighbours will be used for interpolation, so dropping the current position only affects the neighbours themselves and the positions before them, as shown in figure 3.9. The positions within the border have to be reinterpolated.

This is not necessarily the actual *area of influence*. The rectangle also includes positions that are not affected at all because of other sample positions that are still available. In figure 3.10 the area of influence could have been truncated at the dotted line because of the other remaining sample within the rectangular area of influence.

This can be done by keeping records of which positions were or could have been interpolated with the sample that is now being dropped. This information is obtained during interpolation of a position anyway and also saving it for each position increases the memory demand only by a small amount per position.

When dropping the sample all positions within the area of influence are reinterpolated to find their new local errors. Now the accumulated errors have to be recalculated, too. This is done by looking at all non-dropped affected samples and simulating their removal. Those affected are all positions that have an area of influence that overlaps with the one from the dropped position.

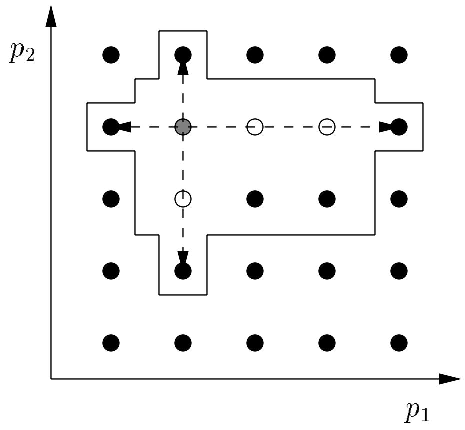


Figure 3.9: Area of influence.

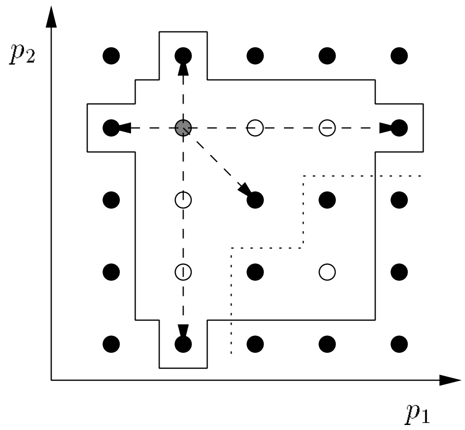


Figure 3.10: Overestimated area of influence.

To recalculate an accumulated error of one position (S in figure 3.11), its own area of influence is calculated. Then the dropping of S is simulated. This includes marking it as dropped and extending the area of influence for D, if they are adjacent in column or row. Then all interpolated samples within both areas are temporarily interpolated. For each of these positions the difference between the current local error and the local error after the simulated removal of S is calculated and added to the accumulated error of S, because if S was removed, the global error would change by this amount.

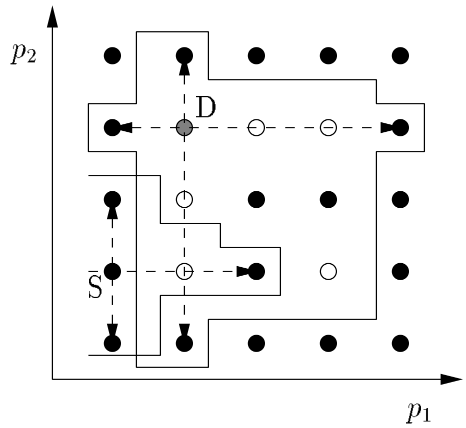


Figure 3.11: Recalculating accumulated errors.

The error accumulated from positions outside D's area of influence does not change because D did not affect their interpolation, *i.e.* dropping S still has the same effect on them as it had before. Their contribution to the global error in S has been calculated before and could have been stored, so that a simple lookup could replace the simulated reinterpolation.

3.2.3 Choosing the Starting Grid

[Whi80] introduces the idea of *adaptive supersampling* to raytracing. The same idea can be used to improve the grid the optimisation algorithm starts with. A regular grid is adequate if nothing is known about the effect of the parameters, but after calculating the corresponding images it can become obvious that some parts change more rapidly than others.

Therefore it is advisable to add rows and columns where adjacent images change more than a certain threshold. Now the limitation of the greedy algorithm to a rectangular grid is a disadvantage, because even if only two adjacent images change, a complete line has to be inserted and the remaining positions have to be dealt with somehow.

There are three possibilities:

Calculate the correct images. This might be a waste of time and space if the other pairs of images are very similar.

Interpolate the images. This saves computing time when obtaining the images, but still requires main memory to store and increases the running time of the optimisation.

Extend optimisation. The algorithm would have to be extended to deal with holes in the sampling while it optimises the set of dropped samples, *e.g.* by using a default interpolation for these positions and not taking them into account for the global error.

The first approach was pursued because in practice a row or column where one image changes rapidly usually also features sufficient changes at other positions to justify the additional calculations.

With adaptive sampling, it is possible to start with a sparser grid and to sample only necessary areas more densely, thus avoiding useless calculation for areas that provide little information and would be removed anyway.

This approach has a bias to optimise interpolation in areas of high change, because more samples are used there and contribute to the global error when being dropped. This bias is desired and corresponds to the aims of the project.

If predictions can be made about the effect of changing a parameter without actually rendering an image, then the starting grid can be chosen so that it already has more equally spaced differences between rows and columns. This is possible for volume rendering by looking at the average opacity.

For the average opacity in the complete head sampling area from figure B.2, the average and maximum difference between adjacent rows and columns was calculated. The results are functions that depend on either position or steepness. They are shown in figures 3.12 and 3.13.

Now sample positions can be chosen so that the average or maximum difference in the opacity between adjacent rows or columns is equal. This is equivalent to summing[2] up the area below the curves and dividing it into

[2]Both parameters are integers.

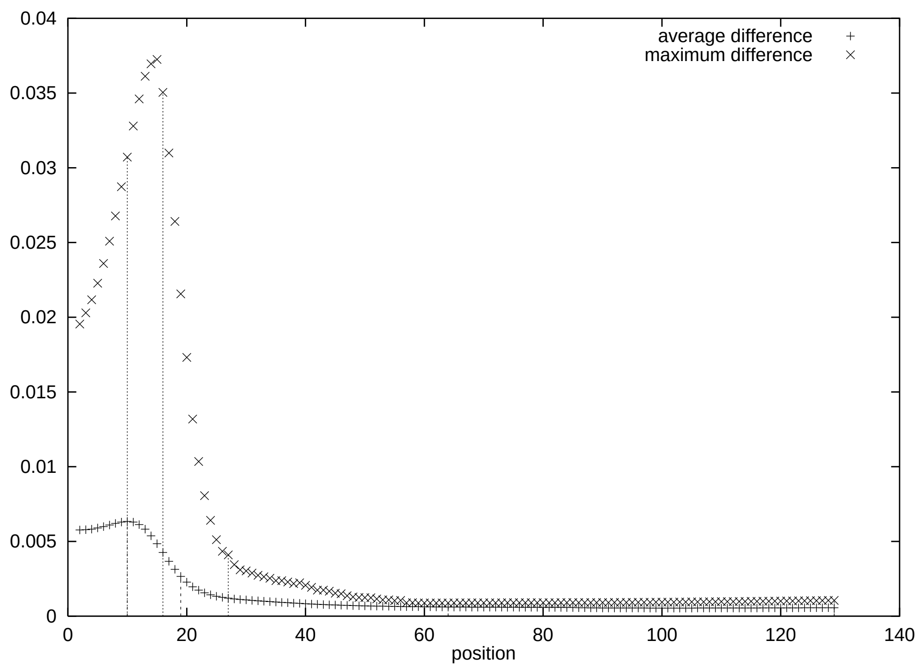


Figure 3.12: Opacity differences between adjacent position values.

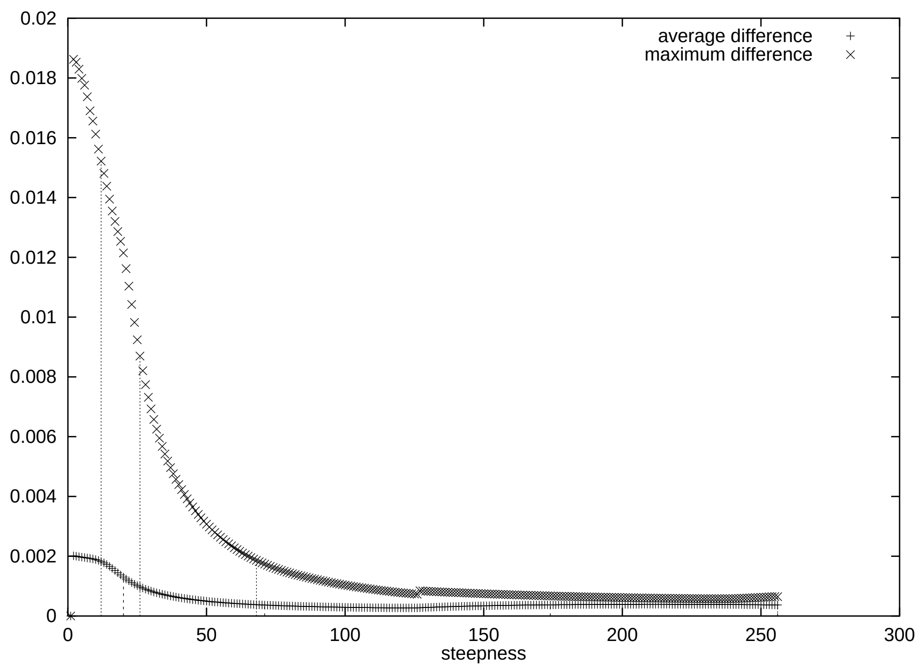


Figure 3.13: Opacity differences between adjacent steepness values.

equal parts. As with the adaptive sampling, the maximum difference is more relevant, so it is better to use the curve for the maximum difference to choose sampling positions.

This does not guarantee a smooth coverage of the parameters, because the prediction based on the opacity may be wrong. This happens when different colour is used for different intensities (then little changes in the opacity can cause a larger change in the colour) or when dense material hides less dense material (then changing the opacity of this material is invisible in the images). Therefore adaptive sampling should still be used to refine the grid.

Chapter 4

Implementation

The implementation of extended light fields has been realised as an extension to the existing light field implementation `lightpack 1.0a`[1]. It has been ported to Linux, replacing the original SGI specific libraries with standard implementations. Then rendering was added as a new feature to the library. Previously, it was necessary to generate all required images in advance, which requires huge storage space and is difficult because the viewing transformation had to be correct for each image. Rendering within the light field library `liblight` also has the advantage that the light field viewer `lifview` can show both original rendering and light field rendering.

The implementation of extended light fields is only a prototype that was used to evaluate this extension, just as the original `liblight` is still "alpha". The supported platforms are Solaris, Linux and SGI. It requires `libtiff` and takes advantage of `flex` if available instead of `lex`.

The advanced reconstruction and sampling optimisation has been implemented and studied with the program `panalysis` (parameter analysis). It features interactive control of the optimisation and gathers the source data for the analysis of parameter effects.

4.1 Port to Linux PC and OpenGL + GLUT

The original source depended on some libraries and features unique to SGI workstations:

[1]http://www-graphics.stanford.edu/projects/lightfield/

IrisGL was used for dumping the rasterised image onto the screen in `liblight` and by `lifview` for the window management. This has been replaced by OpenGL (`opengl_driver.c`) and GLUT. For the latter, the complete input event handling had to be rewritten because GLUT calls registered functions whereas IrisGL is asked for events.

SGI RGB images were the only supported file format for images and were loaded with a SGI library. This library has been replaced by code that is now part of `liblight` and in addition, loading and saving can also be done with the more portable `libtiff`. Reading binary `.ppm` images is also implemented.

Byte order was relevant for the vector quantisation code. It only worked with little endian machines after fixing the way how codeword indices were written into memory (`compression.c:lfVQCompress()`).

RGB[A] vs. [A]BGR: This differs between IrisGL and OpenGL raster images. The original implementation and light field file (.lif) format both use [A]BGR with an optional alpha byte. Datatypes `LFPixelRGBA8` and `LFPixelRGB8` are defined, but not used everywhere. As only RGBA can be passed to OpenGL without byte swapping, the internal byte order is now RGBA. This can still be switched back to ABGR by defining `ABGR_LF` and both works: reading and writing .lif files swaps the bytes as required (`fileIO.c`), inserting the redundant alpha byte (which is done for rendering to improve cache performance) either inserts at the beginning or end (`util.c:lfPadData()`), and elsewhere the datatypes are used consistently.

4.2 Extensions to `liblight`

`liblight` keeps the information about a light field in a `struct LFField`. However, the dataflow functions are only called with the collection of `struct LFSlabs` of this light field. This will have to be changed, because the rendering and parameter extensions belong to the light field and not to one or more slabs. A light field without slabs, but with additional parameters, is possible. In this case the built-in renderer would be used to produce an image.

The routines have not been rewritten, though. `liblight` already had a global `struct LFContext` with a pointer to the current light field, so whenever a routine needs information about parameters or the renderer, it uses the global variable to find them. Defines have been added to do this: `__LF_GET_PARAMS()`, `__LF_GET_RENDERER()`.

4.2.1 Parameters

The additional parameters are simply a fixed sized array of `struct LFParam` that are combined with the used number of parameters in a `struct LFParams` for an extended light field. Their number can be set at compile time with the define `LF_MAXPARAMS`.

For each parameter, its minimum, maximum, default and current integer values are saved plus a text string that serves as a description of these parameters. Integer values are used, but could be replaced by floating point values should the need arise.

Parameters are addressed simply by their index in this array. This corresponds to the order in which they appear in the light field description file (.lid):

```
PARAMETER min max default 'name of parameter'
...
```

The lex and yacc descriptions of this file format (`desc_file_[yacc.y|lex.l]`) have been extended to parse these additional entries and to copy their values to the parameter array. The .lif format now stores the same information. Reading files is downward compatible: the additional information is set to defaults if not available. Saving in a .lif file does not write them if they have their default values, so generic light fields generated with the extended `liblight` can still be read by old viewers.

4.2.2 Rendering

A modular concept is used for rendering. Different types of renderers have been defined (`enum LF_RendererType, LF_RENDERER_[NONE|VOLPACK|VOLVIS|RENDERMAN|DUMMY]`). The corresponding functions are never called directly, instead `lfRD*()` functions in `renderer.c` are called which in turn call the correct function.

All renderers use the same data structure (`LFRenderer`) that tells them what, with which arguments, and with which quality to render. What to render is specified by a simple string that all currently implemented renderers interpret as the filename of a data file. A variable number of arguments can be picked from the array of available parameters in a configurable order, since the order of parameters should be independent from the order in which a renderer expects its arguments.

This is specified in the .lid file as:

```
RENDERER               VOLPACK
                       head.den
                       1    # position is taken from parameter #1
                       0    # and steepness from #0
```

with `VOLPACK, VOLVIS, RENDERMAN, DUMMY` for the different types. In this example, the first parameter in the parameter array is interpreted as the steepness and the second one is the position.

Different levels of quality are possible. The renderer tells the application which levels are supported by setting a bit field in the renderer data structure. The application can then present the user with a choice to allow the selection of a certain level for rendering.

Each renderer provides functions `lf*Alloc()`, `lf*Render()`, `lf*Done()`, `lf*RenderParams()`, with * representing a short token for each renderer: `VV, VP, RIB`. Allocation, rendering and freeing resources are done by the first three functions. Rendering is called with model and projection matrices as specification of the viewing transformation, but in order to simplify the implementation the global `struct LFView` also contains a different representation which specifies the positions of observer and focus, field of view, size and offset (for sheared images) of the image plane plus redundant rotations of the object. This is the viewing model used by the original `lifview`.

`lf*RenderParams()` is used to provide a viewer with the ability to present a visual interpretation of the parameters and/or the data being viewed. Only the renderer knows how to do this, so a viewer has to setup the OpenGL viewing projection so that coordinates 0.0–1.0 correspond to an area of the window where this visual representation is to be placed and to call this function to have it rendered with OpenGL. Even if original rendering of the data is not possible, *e.g.* because it has been disabled or the original data is not available, then the parameter rendering can still be done.

This is used for volume rendering to show the ramp and the histogram of the data. The two volume renderers, `Volpack` and `VolVis`, share this and some other code such as histogram calculation in the source files `volume.[h|c]`.

From the two renderers, `Volpack` always has to be linked to an application because its mathematical support functions are used for the other renderers. `VolVis` support (and the rather large code for it) can be removed by compiling with `NO_VV` being defined.

4.2.2.1 `Volpack`

The usage of `Volpack` by `volpack.c` is rather limited in its possibilities. The configuration string is used as the filename of a .den file which only provides the raw data, so all additional settings such as colour mapping and lightning are determined by the source code at compile time, not with the renderer's configuration. The quality setting is only interpreted during the initialisation and can be used to down-sample x and y direction by 8, 4, or 2.

The first parameter always specifies the starting position of a ramp, the second its steepness.

4.2.2.2 `VolVis`

`volvis.c` is more versatile. It reads the environment files as created by the `VolVis` application and uses all objects, colour mapping, projection methods, *etc.* as contained in this file. The quality directly maps to the different `VolVis` quality settings, with one exception: quality 0 uses `Volpack` instead.

Two different interpretations of the additional parameters are possible:

ISO surface: the only parameter specifies the value of an ISO surface within the volume.

opacity mapping: ramp parameters as before.

Which mode to use defaults to the mode of the first volumetric object in the scene. This can be overridden by prepending "OPACITY" or "ISO" to the configuration string. The remainder after these prefixes still specifies an environment file. This is useful to switch the mode quickly without starting `VolVis` and creating a new environment file.

4.2.2.3 `Renderman`

The Blue Moon Rendering Tools are called by `renderman.c` to create the rendering of a RIB file. This file is composed from a header that specifies the current viewing transformations as calculated in `renderman.c` and the configuration file. This file contains only the world description. The rendered image has to be written to "`/tmp/`" and loaded again, so rendering speed is limited, but for complex scenes no interactive rates are possible anyway.

No parameters control the rendering in the current implementation. Later, scenes within the file could be selected to generate an animation with an extended light field, or text replacement could influence the rendering.

The following qualities are supported:

0. Polygon rendering with `rgl`, pixel sampling 1x1
1. Polygon rendering, default pixel sampling
2. full quality rendering with `rendrib`
3. full quality rendering with artificial aperture

4.2.2.4 Using the Renderers

They are used for two purposes: during creation of a light field, `lfReadSliceFile()` in `io.c` first looks for an .rgb or .tiff file for the current position in the *uv* plane. If neither is available, it calls `RenderUV()`, which calculates the position of the camera according to the current *uv* parameters and then has it rendered by the original renderer.

This behaviour can be controlled by passing some new *description* tags to the `lfRead()` API function:

LF_READ_NO_LIGHT suppresses the creation or loading of a light field when reading a file, *e.g.* if the viewer only wants to use the original rendering.

LF_READ_NO_RENDERER suppresses the full initialisation of the renderer, so that the renderer is limited to rendering only parameters and not the scene.

LF_READ_NO_IMAGES enforces the rendering of the images even if they already exist.

LF_READ_CREATE_IMAGES saves each image as a .tif file after it has been rendered.

LF_READ_QUALITY has one argument which is the quality level used for rendering these images.

LF_READ_UVFILTER enables the approximation of an aperture filter by rendering several images and averaging them.

The other usage is the rendering from the original data when no light field has been loaded or generated, or this is enforced by setting the viewing parameters in the `LFView` context. Then `draw_fb.c:lfDrawFB()` calls the renderer instead of iterating over the slabs in the light field. As each window update in `lifview` calls this function and rendering can be very slow, this function now simply returns the previous image if no relevant parameter has changed meanwhile.

4.3 Extended Light Fields

A simple rectangular sampling of the additional parameters can be selected by specifying the number of sampling positions between the minimum and maximum value for each parameter `pindex` in the .lid file:

```
SAMPLES_P pindex number
...
```

or the sampling positions directly:

```
SAMPLE_POINTS_P pindex sample1 sample2 ...
...
```

These positions are an attribute of a slab just like the sampling of uv and st, so they have been added to the data structure `LFShared`, which may in theory be used by more than one slab but in practice is created separately for each slab.

Wherever `liblight` iterates over the samples for u and v, these nested loops are extended by an outer looping over the additional parameters. As their number can change, no fixed number of loops could be added. Instead, an array of current indices for each parameter is maintained and looping is done by incrementing this array. Ascending or descending one level in the nested loops for the additional parameters is the same as incrementing or decrementing the index within this array.

Irregular sampling has been realised, too. Unless interpolation has been pre-calculated and stored as described below, a strictly hierarchical interpolation between the available samples is derived: first the parameter with the highest index is interpolated, then the second highest until the first parameter can be interpolated. A tree with one level per parameter is used to represent the available samples, as it is well suited to finding the values between which the interpolation can be done for each parameter.

For example, for two parameters the tuples (1,20), (1,40), (1,50), (5,30) would be represented as the tree in figure 4.1. To interpolate an image with parameters (2,30), the reconstruction will interpolate the first parameter with the sampled values at 1 and 5, but has to look at the remaining parameters first. For both sampled values it descends one level. For 1, another interpolation of the second parameter is necessary between values for 20 and 40, whereas 30 was sampled directly with 5 as first parameter value, so no additional interpolation is necessary here. This search is done in `draw_fb.c` by calling `interpolateParam()` recursively.

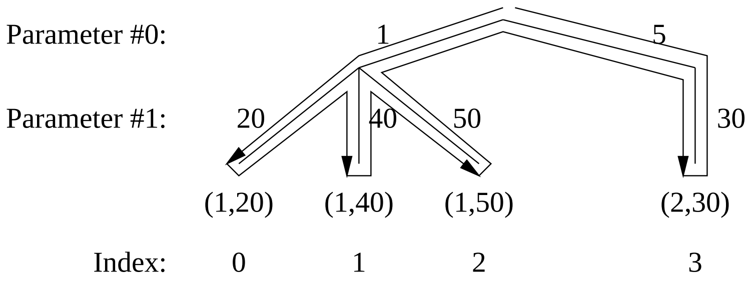


Figure 4.1: Tree representation of arbitrary sampling positions.

The result is an array of indices of generic light fields that are required (`LFShared:p_indices`) and an array of interpolations to reduce the extracted values to the final one (`LFShared:interp`). Together they form a *plan* that describes how to do the reconstruction. They can be saved in simple arrays in the structure `LFInterpolationPlan`, because their maximum number can be derived from `LF_MAXPARAMS` at compile time. Now it is possible to do the address calculations of the source generic light fields for each slab (`LFSlab:lfs`).

Rendering follows the precalculated plan in the resampling loop in `resample.c` to obtain all required ray intensities from the generic light fields and then to interpolate the final values. Defines are used to generate special versions of the inner loop for the common cases: one source + no interpolation, two sources + one interpolation, *etc.* This way the iteration over the arrays for sources and interpolation can be avoided.

Iteration over the generic light fields is now a depth-first iteration of this tree that visits children from left to right. The order in which leaves of this tree are encountered is also the order in which the generic light fields are stored in memory. The iteration is still done as before with an array of indices for each level, but changing levels also has to remember the current position in the tree, so that the available sample positions can be found.

In the .lid file, complete parameter tuples are given with
`SAMPLE_POINT sample1 sample2 ... sample_n`
The regular sampling mentioned before is also represented by a tree, so both methods of specifying a sampling pattern can be mixed.

Optionally, the interpolation can be specified explicitely. `LFShared:plans` points to an array of variable dimensions that contains a `LFInterpolationPlan` for every valid combination of parameter values. This information has to be provided in the .lid file with
`INTERPOLATION plan1 plan2 ... plan2.`
Each plan is a short machine readable representation of a `LFInterpolationPlan` as generated automatically by `panalysis`.

It is hardly feasible to specify a complete interpolation plan manually due to the large number of parameter combinations. It is still small enough to be stored completely, though, therefore this general approach was chosen although a representation as in `panalysis` itself (*i.e.* with basis functions) would be smaller. The advantage is that different analysis tools or a new method in `panalysis` could be used without changing the file format.

4.4 Tools

There are some new options that are supported by all tools. They directly map to the new features of `liblight` for image handling and generation:

`-no_images`	→	`LF_READ_NO_IMAGES`
`-no_light`	→	`LF_READ_NO_LIGHT`
`-no_renderer`	→	`LF_READ_NO_RENDERER`
`-create_images`	→	`LF_READ_CREATE_IMAGES`
`-quality n`	→	`LF_READ_QUALITY`
`-uvfilter`	→	`LF_READ_UVFILTER`

4.4.1 `lifauth`

`lifauth` also supports new options to extend the vector compression tiles to the additional parameters: `-vqtilep pindex size` sets the size of the tile in the direction of parameter `pindex`. It defaults to 1. This maps to the new tag `LF_VQ_PTILESIZE` in the `liblight` API. This has only been implemented for regular sampling of the additional parameters.

4.4.2 `lifview`

`lifview` supports the vector compression options of `lifauth` for testing purposes, too. The most relevant change however was the replacement of IrisGL with OpenGL. `lifview` has also been extended to show information about extended light fields. It maintains an area in the top of the window where the additional parameters are displayed, if available. To the left of these lines the renderer specific parameter visualisation is placed. Its size can be changed with the cursor keys and is kept in `params_height, params_left`.

A parameter is changed by pressing down a mouse button over its line. This sets `dragging_param` to the index of this parameter, so that `viewMouseFunc()` knows that vertical mouse movements map to a change of this parameter and not a rotation or translation. The left mouse button causes faster changes than the middle one.

`lifview` also displays one status line below the parameters that shows the average intensity of the current image or the average frame rate since changing an option that affects performance. The contents of this status line can be changed with the menu item "Intensity" (figure 4.2). This menu is setup anew by `viewSetMenu()` each time an option is changed, because the GLUT menu support is very simple and checkmarks have to be simulated by prefixing an option with a "+" instead of spaces. Entries with no text separate real entries.

New options include the one that turns on interpolation for the additional parameters and the top one that allows switching between original and light field rendering. If only one of these is available, it turns into a fixed display of the current mode. This is determined by looking at the current setting for rendering, the `noLight` argument and the renderer type for the current light field.

Finally `lifview` supports animated parameters in addition to animation of the object. `animate` is set to true if any animation has to be done. Then

Figure 4.2: The menu of `lifview`.

after doing the rendering in `viewDisplay()`, the object is rotated and/or enabled parameters are incremented. An immediate redisplay is then requested from GLUT with `glutPostRedisplay()`. This makes GLUT call the same function again immediately after handling input. Parameters are enabled for animation by pressing the digits "0", "1", *etc.* "a" enables or disables the rotation around the vertical and horizontal axis, but this is now implemented by setting `rotate` accordingly and not by enabling or disabling animation explicitly as before.

4.4.3 `panalysis`

`panalysis` is highly experimental and therefore does not always follow sound programming style. Most of it is a direct implementation of the starting grid calculation and subsequent optimisation, so this section only connects the theory with the implementation without discussing it again.

4.4.3.1 Obtaining the Starting Grid

The internal routines of `liblight` are used to read .lif or .lid files. As there is no public API to do so, the relevant steps of the dataflow pipeline are reproduced: `lfBeginReadSliceFile()` or `lfBeginReadLif()` and `lfRDAlloc()`. Then the creation of a light field is simulated so that the dataflow function to get one image can be used to either load or render the required image (`InitEntry()`).

These images are stored in the linear array `Positions` of varying size `Columns * Rows`. The define `POSITION(i,e)` maps the row i and column e to an entry in this array. This array is the grid that the optimisation will work on. Images are only referenced with a pointer in each entry of type `struct Position`, so the array itself is rather small and can easily be resized.

`InitArray()` sets up the complete array. This is where adaptive sampling has been implemented. Each time a new image has been obtained, it is compared to its left and bottom neighbour. If the difference is larger than the chosen `Threshold`, `InsertRow()` respectively `InsertColumn()` increase the size of the array and restart the iteration over the complete area to fill in the missing positions.

The average and maximum opacity differences have been calculated before by `InitOpacity()` and stored in `OpacityPos/Steep`, so this information is optionally (`UseOpac`) used to setup the initial sample positions (`DivideArray()`) and to choose a new value between two positions that need refinement (`DivideRange()`).

4.4.3.2 Optimisation

`optError()` is the function that repeatedly drops positions until a certain abort criteria (`MaxError, MaxAvgError, MaxInterp`) is reached. It uses `InterpolatePosition()` to find the interpolation of one position, but restricts this recalculation as described before by checking if the last dropping of a position affects another position at all (`InflInterp()`). Similarly, accumulated errors are only recalculated if there is an overlap in the area of influence (`InflAccum()`).

Simulating the dropping of a sample is simplified by the fact that all the relevant entries for the position are kept in one `struct interp`. `DropSample()` drops a sample and takes care that the invariants (like neighbouring remaining samples, remaining samples within columns and rows) are kept valid. It can be called again to restore the previous state quickly.

`RecalcAll()` is the brute-force method of establishing these invariants: it only assumes that the images are available and that the dropped/sampled flag is set. Based on that it recalculates all errors and invariants.

The optimisation can be called for different subsets or grids of the complete image to compare the effect of using less pixels. This is implemented by specifying how many and which indices within the original images are relevant for comparing the images and the basis function calculation (`GSize, Grid`).

Regular subsets, random and Poisson grids can be generated (`RandomGrid()`).

4.4.3.3 Interaction

An interactive mode with display of the original and interpolated images and the difference between them has been implemented with OpenGL. GLUT never gets full control, instead `optError()` is registered as display function and starts its work as soon as it is called for the first time. Then control never returns to GLUT.

`DoInteraction()` is called for several different levels of interaction and the user can choose to automatically continue for some of them (`Interactive`). Each time this function is called it updates the window by dumping the complete array of images (`Redisplay()`). The sources of the best interpolation for each position are marked by placing small squares in the appropiate direction(s).

4.4.3.4 Browsing

From interactive optimisation one can switch to "browsing" the parameter area with the mouse. The coordinates of mouse movements with pressed mouse button and of clicks in the display of the grid of images are translated to a pair of parameter values. A second window compares the interpolated image with the original rendering for these parameters.

`InitBrowsing()` opens the second window and installs the callback functions (`MW*f()`) necessary to handle the input. Then `optError()` returns control to GLUT and browsing starts. The display function `BWdisplay()` calls `InterpolateParameters()` to obtain the interpolated image as derived from the current set of remaining samples and—if the mouse button has been released—also the renderer for the original image.

For each source that contributes to the interpolated image, its influence as percentage is also reported by `InterpolateParameters()` and the functions it calls (`InterpolateDirection()`, `InterpolateLine()`). By calling this function for all parameter combinations, an influence map for each remaining position is calculated and can be displayed instead of the original or interpolated images in the main window. Figure 4.1 lists all keys and their function. The interpolation can still be changed, but the set of remaining samples itself is fixed now.

key	action
o	display original images
i	display interpolated images
e	display difference between both
f	display influence of one remaining sample (selected with cursor keys)
v	restriction of interpolation to first vertical on/off
h	restriction of interpolation to first horizontal on/off
l	restriction of interpolation to linear basis functions on/off
d	display of interpolation direction on/off
b	display of borders around images on/off
s	save window contents

Table 4.1: Key actions for the main window during browsing.

4.4.3.5 Results

`panalysis` generates a file that contains a lot of information that is only used to create figures in this report. All of those lines are comments that are ignored when using the file for its main purposes:

With "-dummy n" `panalysis` uses hard-coded functions (as the one in figure 3.1) as source of the images and outputs `gnuplot` instructions on how to plot them, so the output file can be fed into `gnuplot` directly.

When analysing real images, the output can be included into a .lid file and specifies the irregular sampling of the parameter space and the interpolation plan as calculated by `panalysis`. The statement "`include`" has been added to the .lid file parser so that the .lid file itself can be left unchanged when calculating a new sampling.

Chapter 5

Evaluation

The evaluation of the proposed techniques uses the head data set presented before as a case study. The larger parameter area with position 1–129 and steepness 0–256 from the analysis in figure B.2 serves as a test case for optimisation and different starting grids in general.

The final aim is to capture the area with positions 1–30 and steepness 30–128 in an extended light field. This area is the most interesting one: in the lower left corner the brain is very opaque, but starting with a less steep ramp avoids the extremly shiny rendering of the brain. To represent this would require many sample positions, but provides little information about the head data set. In the upper right corner the blood vessel is visible nicely without occluding brain. Making the blood vessel more transparent provides no further information and is therefore omitted, too.

All tests were done on a Linux PC with Pentium Pro II processor running at 233MHz, 64MB main memory, an ATI Rage Pro graphic card in 1152x900x24, Mesa 3.0 beta and XFree86 as X server. `VolPack` was used to generate the original images from the head data set.

5.1 Performance

To measure rendering performance, the light field viewer renders images continously and changes some attributes for each image. For view point animation, the object is rotated in steps of 1 degree around the vertical and horizontal axis in each new frame. The direction is reversed after reaching 10 degrees, so the head is always completely covered by the slab configuration. For parameter animation, every enabled parameter is increased by one and wrapped around to its minimum as soon as it has reached its maximum. Frame rates are calculated by counting frames and measuring the time to draw them. All measures have been done with an original image size of 150x150 pixels that were zoomed to 300x300. This is a reasonable size for a light field with an image plane of 256x256 samples.

The performance measures in table 5.1 have been done with an extended light field of the head data set, so that comparison with the original volume rendering systems is possible. Generic light fields have been generated at ramp position/steepness 1/30, 1/128, 30/30, and 30/128, *i.e.* the four corners of the interesting area. The light field has been generated with 256x256 sampling of the image plane and 16x16 sampling of the camera plane. Vector quantisation was used with a tile size of 2x2x2x2 for u, v, s, t. This results in an extended light field of ≈10MB.

In practice, one would use interpolation of u, v and the additional parameters. Interpolation within s, t is not necessary because sampling was sufficiently dense. Simple linear interpolation is used for the additional parameters, but as explained before, a non-linear basis would not affect the performance. With this kind of interpolation a frame rate of 7.2 frames per second is reached. As the complete extended light field fits into main memory, no page swapping is necessary and random access to all generic light fields is guaranteed at the same speed. Therefore animating the additional parameters is as fast as animating the view point or both.

`Volpack` as the fastest volume renderer available for tests with this data set still manages 1.7 frames per second, but the data set has been down-sampled from 256x256x55 to 128x128x55 already for that and the full possibilities of precalulation have to be applied. Reclassification slows down `Volpack` to 0.7 frames per second as soon as not only the view point but also the additional parameters are changed. It would also be considerably slower for larger data sets, which would not affect the light field rendering at all. Even for this example were `Volpack` is able to perform very well, extended light fields are still ten times faster. Their true advantage become completely apparent when comparing them to `VolVis`: they could provide rendering with the same quality and effects as `VolVis`, but at interactive rates, whereas the frame rate for `VolVis` itself could not even be measured. It takes minutes to render an image.

Performance of extended light fields largely depend on how many generic light fields are required. Table 5.2 compares the rendering of a dummy ex-

Rendering method/Interpolation	frames/second
Rotations	
Extended Light Field:	
uv or st	12.5
uv and st	9.2
uv and either position or steepness	10.10
uv and both	7.2
uv, st, and both	3.9
`Volpack`	1.7
`VolVis`	< 0.01
Changing Ramp and Rotations	
Extended Light Field:	
uv and both	*as before*
`Volpack`	*0.7*

Table 5.1: Performance of rendering images of size 150x150, zoomed to 300x300.

tended light field with three additional parameters that simply control the colour with which the image plane was filled. It was generated without vector quantisation and lower sampling to keep the memory requirements low: 8x8 and 128x128. Performance tests with vector quantisation and this dummy example are pointless because the image planes do not change at all within each generic light field and would required only a minimal number of codewords.

Interpolation	number of generic light fields	frames/second
uv	1	13.5
uv and one parameter	2	10.4
uv and two parameters	4	6.9
uv and three parameters	8	4.0
uv, st, and three parameters	8	2.0

Table 5.2: Performance with increasing number of additional interpolations.

Because of caching effects of the codebook a real application with three parameters might reach a better performance than the 4.0 frames per second. The processor caches cannot help to speed up access to the light fields, as the number of rays is too large. They seem to help for the codebook itself, though, as rendering a compressed generic light field is hardly slower than rendering an uncompressed one (12.5 versus 13.5). It even becomes faster as soon as memory pressure is increased when more generic light fields and therefore more memory accesses are required. The codebook was quite large with $16384*2*2*2*2*4$[1] $=$ 1MB, but caching still helps as not all codewords are required at once. So rendering performance would allow the usage of one or even two more parameters.

More important than the processor caches is virtual memory and the storing of main memory on disk. So far all test examples fitted into main memory. This does not have to be the case. There might be extended light fields that capture larger parameter areas, require a finer sampling, or simply have more parameters. In this case rendering of extended light fields slows down as soon as virtual memory management has to bring so far unused parts of the extended light field into main memory. This happens if parameters are changed or if a completely new point of view is chosen.

[1]The three intensity entries per ray in each codeword are extended by a dummy alpha byte to improve cache performance.

Table 5.3 gives some rough measures for this effect. It is based on the same dummy example as before, but this time one parameter has been sampled at eight times as many positions, so more than 100MB are required for the uncompressed extended light field now. Changing both view point and the parameters continously is the worst case as it sweeps across the whole extended light field and requires constant page swapping.

Animation of	frames/second
view point and all parameters	1.3
view point (after changing parameters)	1.5 – 4.0 (after $\approx 20s$)
parameters (after changing view point)	1.5 – 4.0 (after an even shorter time)

Table 5.3: Performance with interpolation between 8 generic light fields. Slowed down by page swapping of the extended light field with > 100MB.

Virtual memory is quite efficient in bringing the relevant part of the data into main memory. In practice usually either view point or parameters are changed (the user interface of the current viewer does not even support changing both at once), and then the frame rates quickly reach their previous maximum for the current number of interpolations because the required data is available in main memory. It is only at the beginning after switching to a different part of the extended light field that animation is a bit jerky. Therefore extended light fields are still useful even if their size exceeds the size of the available main memory. Due to the dataflow architecture of the implementation they can even be generated on such a machine.

One final remark has to be made about the size of the images that are rendered. As any image based rendering technique, rendering of light fields is linear in the number of pixels. For larger images, the direct comparison is more favourable for `Volpack`, because it is less dependent on the image size and more on the original data size: it still reaches 1.5 frames per second at 300x300 whereas extended light fields drop to 2.4 frames per second with interpolation of uv and both parameters. This emphasizes the need of hardware support for the huge amount of interpolations. They are simple enough to be realised either with MMX commands in the inner loop for all interpolations or with texture mapping hardware at least for the additional parameters outside the inner loop.

5.2 Quality

5.2.1 Interpolating Images

The discussion of using polynomial basis functions and their derivation ignored the fact that real images are going to provide the functions that are being interpolated. This affects both the function values and the effect that changing parameters has.

The range of the functions are the integer values from 0–255. If this range is exceeded by a reconstructed value, it has to be clamped to this range. Otherwise a value of *e.g.* 256 would map to the value of its lower eight bits, *i.e.* 0. A very light pixel would suddenly turn black.

Because the convex hull property is not enforced, in theory weights could occur where this could happen. This can be tested outside the inner loops. If the weight is larger than 1.0, then clamping has to applied to each calculated value before storing it, otherwise it can be avoided.

In practice however, averaging the basis functions of every pixel leads to well-behaved and smooth general basis functions that do not exceed 1.0, so in general performance is not affected. If hardware support is available for doing the interpolation[2], then it does not matter at all.

Figure B.3 shows how dropped positions in the head example would be interpolated with linear interpolation and with higher order polynomials calculated with weights as discussed before. In this example the position was fixed at 10 and the steepness varied from 30–256. Using the polynomial basis provides a considerable advantage over the linear basis: the influence of the left picture is too large with the linar basis so that the green brain matter is much too visible in the first few interpolated images on the left.

The advantage is smaller if there is no general tendency for the effect of changing one parameter, for example if some pixels tend to be more similar to one source and some more to the other, or if the effect of the change is really linear. The only disadvantage of the polynomial basis is that there derivation and evaculation become numerical unstable for degrees larger than approximately 10. This prevents the optimisation from dropping samples that are too far apart from their neighbours.

However, the error for dropping such samples is usually large anyway unless the original grid was sampled very densely, so this does not affect optimisation

[2]For example MMX commands do arithmetic with clamping.

in practice. If necessary, this problem could be solved with more sophisticated numerical methods or simply by limiting the degree of the basis polynomials and solving an over specified set of equations. Using larger degrees does not provide much benefit anyway, because dropped positions of images can only be approximated and not interpolated exactly.

In the real application on the smaller area, the non-linear basis functions reduce the average error from 5.467 to 5.127. Slightly different positions are chosen by the optimisation then, too. When enforcing that the same set of samples is used, the linear basis functions become even worse with an error of 5.547. The difference is smaller than in the previous example, but some positions differ visible as shown in figure B.4. Again, the influence of the steepness on the opacity of the brain is captured much better in the marked area.

5.2.2 Sampling Optimisation

5.2.2.1 Performance Analysis

The complexity of the greedy algorithm to optimise the selection is determined by the step in which the effect of dropping a position is simulated. As the sampling becomes more sparse, accumulated errors of more positions have to be recalculated which in turn requires the reinterpolation of an increasing number of positions. However, the time spent on recalculation after dropping another sample starts to rise rather late so that the optimisation does not have to go on for much longer. Figure 5.1 shows the total running time for dropping $17^2 = 289$ samples in the large area and gives the average error for each number of dropped samples. Dropping further samples is useful up to the point where the average error starts to rise faster.

So in practice, the limiting factor is not so much the complexity of the optimisation but rather the time and memory required to acquire and to store the original images. The optimisation was fast enough to drop 284 positions out of 289 in less than 12 minutes. Generating the original images can take considerably longer. After dropping one more sample the average error exceeded 10, so this is a reasonable upper limit. It is even larger than usually desired in practice.

The methods to reduce the recalculations are essential to achieve this performance. Figure 5.2 shows how long it would take to drop all samples when all positions are recalculated after dropping one. It takes over 16 hours to

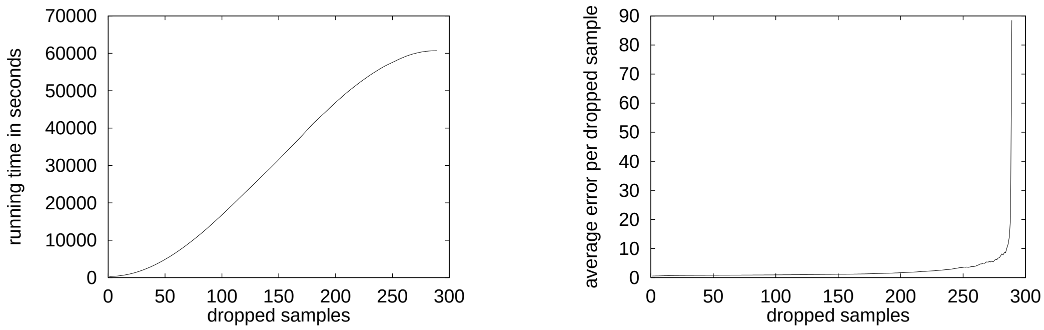


Figure 5.1: Running time of optimisation and average errors.

reach the same number of dropped positions.

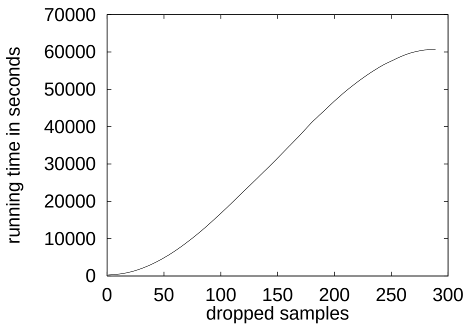


Figure 5.2: Performance without reduced recalculations.

Optimisation also linearly depends on the number of pixels because interpolating and comparing two images are the basic operations. Profiling shows that most of the time is spent on them, so any book keeping to reduce their number is justified. Rendering is usually also linear in the number of pixels. Optimisation requires random access to all images, so if they do not fit into main memory performance will suffer.

These dependencies make it worthwhile to reduce the number of pixels obtained for each image. The next section investigates this.

5.2.2.2 Chosing Pixels for Optimisation

So far a set of fixed sized complete pictures was assumed to be the basis for the optimisation. This set is already only a subset of the images that will be generated for the generic light fields, as they represent only one single view. This is justified if there is one main viewing direction that captures the relevant effects of the additional parameters. If this was not true, several relevant views could be used.

The other relevant factor is the size of the images. Less than the number of pixels stored in the light field could be looked at as long as no details are lost. This can be done either by reducing the size of the images or chosing a fixed subset of the pixels.

The effect of using less pixels is evaluated by applying the optimisation to the complete set of pixels first, thus obtain the best results the algorithm could deliver for each number of dropped samples. Then a subset is chosen and the algorithm is applied to this subset. For each set of dropped samples that was derived from the reduced set of pixels the global error for the complete images is calculated and compared to the optimal result.

Using a regular grid as subset is equivalent to simply reducing the size of the images, unless the renderer uses advanced methods to compensate for the small resolution at which the images are created. The problem of chosing a subset of pixels for optimisation is exactly the same problem as sampling a continous illumination model to produce an image, *e.g.* by ray tracing. Advanced techniques to avoid aliasing or to improve the image quality include *supersampling* and *adaptive sampling*.

Supersampling simply increases the number of samples per pixel and applies a filter to them to obtain one value. Adaptive sampling takes the difference between samples into account to increase the sampling rate only in areas with high changes. These techniques might be enabled within a renderer that generates the images to improve the quality of smaller images, but they also increase the rendering time. They are not available in `VolPack`.

However, using less pixels does not increase the error that much: figure 5.3 compares the error when using images of size 8x8 and 32x32 to the error for the original size 128x128. The averaged error metrics are very insensitive to a loss of detail in the original and interpolated images. As long as the general trend is represented by the images, similar positions are dropped. The calculated basis functions are still very similar, too.

Images of size 32x32 (as in figure B.6) still represent the general effect of

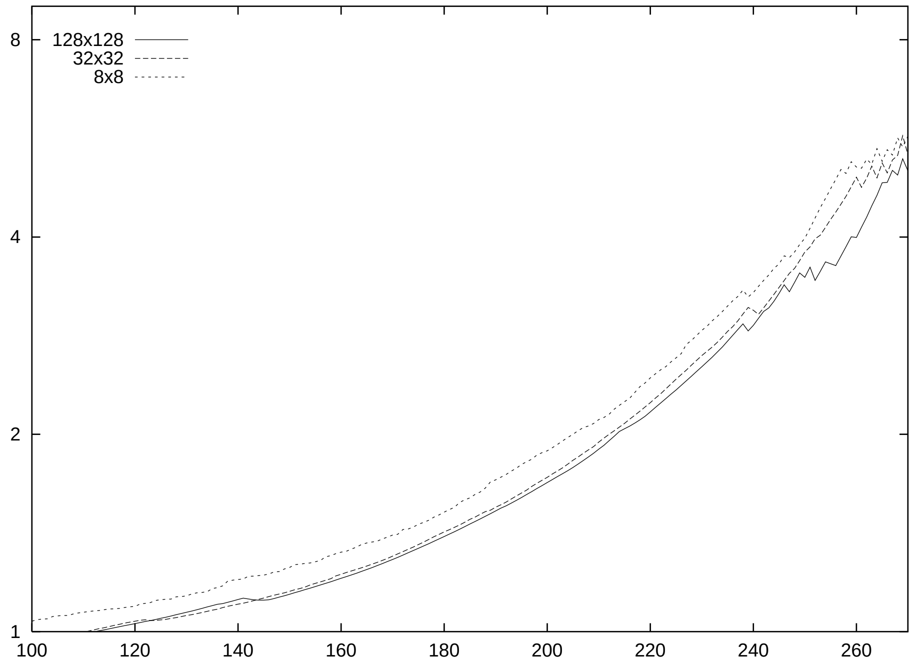


Figure 5.3: Optimising ramp position 1–129 and steepness 0–256 with smaller images.

the parameters, so in the end very small images are sufficient during the optimisation.

If aliasing in the images would affect optimisation in a negative way, *stochastic sampling* ([Coo86]) could be used to avoid aliasing by using a random sampling pattern. This replaces aliasing with an increased level of noise without increasing the number of samples. This can be applied to chosing a subset on top of the renderer: the subset is now a set of points distributed with the *Poisson disk distribution*. This is effectively a random distribution where all points have a minimum distance. It can be generated by choosing a position randomly and discarding it if it is too close to existing points until the desired number of points has been reached. This is not necessary, though, as a regular pixel grid is perfectly suitable for the optimisation. Random or Poisson grids did not improve the result.

5.2.2.3 Set Optimisation

Despite the attempts to improve interpolation by using a non-linear basis, in the end still the average difference between the supporting images determines the quality of interpolating the images between them. The main effect of the optimisation is not so much that it calculates appropriate basis functions, but rather drops positions that can be interpolated well, usually simply because the changes in this area are small.

In example B.6, the change is generally smaller in horizontal direction and in the upper right part. Therefore positions are dropped in columns first, *e.g.* the upper part of the second rightmost column. After dropping 30 out of 49 samples, the remaining samples in the middle follow the isolines in the average intensities from figure B.2. The major direction is horizontal, because samples in the same row are more similar. After dropping eight further samples the samples in the corners and some in the middle remain because they support large areas. The lower left corner is covered more densely because of the large changes there. After this step dropping more samples starts to increase the error more rapidly and in the end positions in the top row cannot be interpolated any more at all.

The greedy algorithm converges well to a good solution. In contrast to other possible approaches to the problem like only looking at differences between images or even only the original data, it really takes the effect of the available reconstruction into account, as is shown by the fact that the algorithm was able to obtain a better result for the linear basis by choosing different positions (section 5.2.1).

In this real example, the final eight positions are in fact very good and perhaps even optimal: exchanging any of them with their neighbours only increases the error. For example, replacing the remaining sample in the middle with its left, right, top or bottom neighbour increases the error from 5.127 to 6.789, 5.81, 5.14 and 5.645.

5.2.2.4 Starting Grids

After the grid has been initialised, the optimisation will add no further sample positions. Therefore it is important that the grid includes those samples that have a large influence on their surrounding area. More samples are required in areas of high changes between the images.

A regular grid is usually very bad at covering these regions. Instead, many positions are placed in areas with little change. Starting with a lower regular sampling and adding lines where necessary is better, but it still leads to unnecessary positions both from the original regular pattern and from bad subdivision: dividing the range between two parameter values exactly in two halves is likely to miss the point where the new value produces images that are equally similiar to both sides, because in areas of high change the effect of a parameter is usually also not linear. The best results are achieved by employing problem specific knowledge to predict this change.

The aim is to increase the average difference between columns and rows in the starting grid and to decrease the maximum difference. The standard deviation is a good measure of how much the differences vary. It should be as small as possible.

Figure 5.4 compares the average differences between rows and columns without adaptive sampling. Both grids use nine values for position and steepness, therefore the size of eight steps in each direction can be compared. The average difference in both directions is shown in the figures. The regular grid looks very smooth, but only because most of the samples are placed in the area with little change. Consequently the maximum difference in the corner with the lowest ramp position and steepness is large and this area is covered badly. Spreading the sample values according to the maximum changes in the opacity already provides a considerable improvement.

Both grids can be refined by applying adaptive sampling, but this retains the unnecessary positions. Better results are obtained by starting with a lower sampling in the first place. In figure 5.5 only five values were chosen

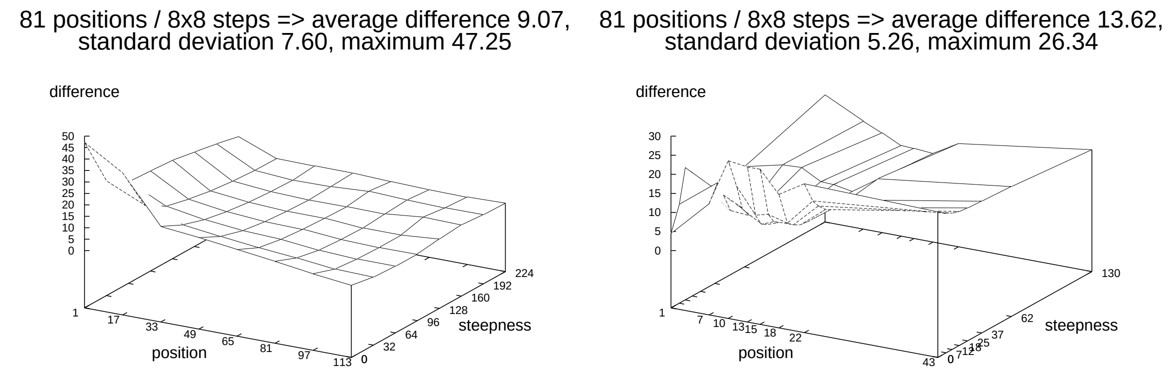


Figure 5.4: Starting grids without adaption: without/with change prediction.

as before and then further values were added wherever two adjacent images differed by more than 32. The prediction of change based on the opacity is already quite accurate, so using both prediction and adaptive sampling only decreases the deviation from the slighly lower average when averaging both directions.

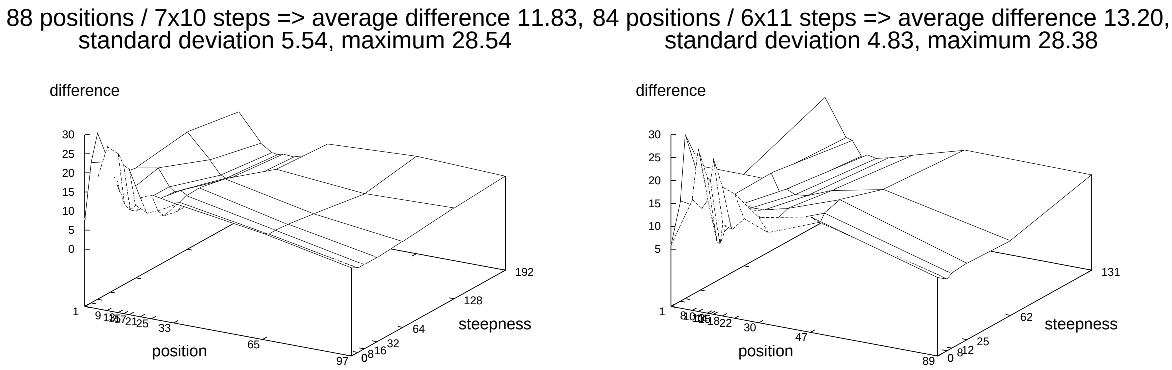


Figure 5.5: Starting grids with adaption: without/with prediction.

For the direction along of the ramp positions alone however, the prediction underestimated the change between position 43 and the highest value 129 (figure 5.6), and so the maximum change is quite large with 41.01 here. With adaptive sampling, a value was placed at 47 (higher because the regular grid was half the size in both directions) and then another value was added at 89 to avoid the large jump.

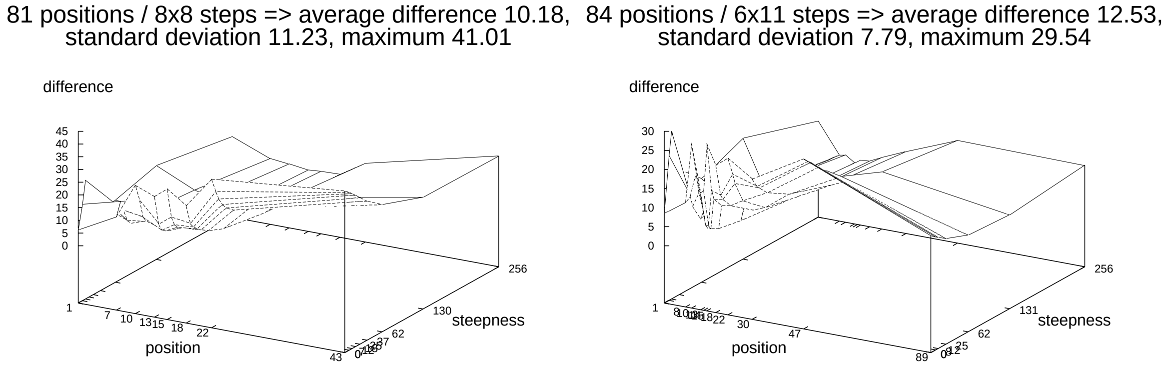


Figure 5.6: Difference along position, with prediction: without/with adaption.

5.2.3 Interpolation with Irregular Sampling

Specifying each complete basis function manually in higher dimensions is difficult, because it is assembled from possibly many different one-dimensional basis functions and therefore piece-wise defined. They can be calculated and visualised with `panalysis`: the influence, *i.e.* the values of basis functions for certain positions, of the top four remaining positions in the real application of the optimisation (B.4) is shown in figure B.5 with four separate images. The grey scale directly maps to the influence. The visible steps in the vertical direction are caused by the fact that the same discrete parameter value for the ramp position is represented by several pixels. The horizontal resolution of these influence maps is higher because the parameter range is much larger (1–128 compared to 1–30).

Where the primary interpolation direction changes, discontinuities are introduced where the partial basis functions change (*e.g.* region B) and in particular where the set of supporting samples itself changes completely, as in region A.

The image for the middle position M also shows the main interpolation directions of all dropped samples with white lines within each square. These lines show where the main interpolation direction changes. The sampled positions were used for interpolation of the position with parameters 28/95. Following the direction of the closest dropped position, this position is interpolated first in columns. The column with M is closer than the first column on the left with positions ML and UL. Therefore the secondary horizontal interpolation interpolates between the column of M and the rightmost column with position R. In both columns M respectively L haven been taken over without interpolation due to the lack of another sampled position above them, so only R and M have any influence. This is indicated with their green[3] borders in contrast to the unused samples with red borders.

This unexpected influence of M in the top right corner of the parameter area is correct and follows from the limitations of the hierarchical interpolation. The interpolated result is closer to the original image this way than it would be with first horizontal interpolation and UL as source. The error would increase from 14.07 to 17.29 for this way of interpolating 28/95.

The hierarchical approach to interpolation can lead to abrupt changes in the influence of one sampled position. Right next to the dropped position that determines the direction for 28/95 a dropped position is better interpolated with first horizontal interpolation. Again, this is also the correct choice for 28/87, so changing the primary direction of interpolation is important to achieve a good reconstruction.

However, the exact border between vertical and horizontal direction does not quite coincide with the borders of the squares. A finer starting grid would help to determine it more exactly. In addition, the borders as found by the optimisation could be refined by a post-processing step that subdivides only the squares at the borders and compares more rendered images with interpolated ones.

Despite this abrupt change in the influence of UL from full to no influence, the interpolated images themselves are still similar: the sampling pattern and primary directions have been chosen so that the differences between interpolated and original images are small, consequently adjacent interpolated images are also similar to each other, as the changes between the original images are gradual. So although the change between them is not gradual, the step where the primary direction of interpolation changes is not really disturbing. In contrast to artifacts within an image, this step is also only visible when comparing images by changing parameters back and forth.

The effect of the change of interpolation direction is even less obvious in the other areas. The area that was chosen for this detailed comparison is the

[3]Their influence is nearly equal, so both shades of green represent approximately 50% influence and are very dark.

area where the results are worst when compared to the original rendering, because reconstruction can only be done by at least partially taking over values from below or from the left without interpolation. Even in this area the results are reasonable.

5.3 Sampling the Opacity Ramp Parameters

In the previous considerations the assumption was made that only low frequencies constitute the sampled signals, *i.e.* that changing a parameter only causes slight changes in the generated pixel intensities. This assumption can be justified for volume rendering with the ramp as mapping function as long as the volumetric data itself has gradual changes in its density distribution.

This is the case for data that has been gathered from scanning natural objects. They usually consist of different material that are often mixed with each other or lack firm boundaries, so that their histograms are smooth across the range of possible density values.

A ramp with a steepness of zero acts as a binary filter that either leaves a voxel completely opaque or completely translucent. Changing the position of such a ramp produces the sharpest changes in the average opacity of the volume and therefore the sharpest changes in the rendered images. But even with this ramp changes are still gradual if the histogram is smooth.

This is not necessarily the case for artificially constructed objects or volumetric data that is generated by a CAD system. They may have voxels filled completely with one of very few materials of a specific density, so that the histogram is not continuous but rather discrete itself. It makes no sense to apply the volume rendering with a ramp in this case. Volume rendering with extended light fields can still be applied, but the additional parameters should better be used to control other aspects of the rendering.

The example of a medical application of volume rendering shows that the changes are in fact very smooth. The only sharp transition happens outside the range of interest: the bone causes a very sharp transition in the histogram and with a binary ramp adjacent positions could either show the complete bone or a blank image with not opaque matter left at all.

This could easily be avoided by prohibiting such a steep ramp, smoothing the upper and lower edge of the ramp or by avoiding this extreme end of the range for the position. It is more a restriction of the volume rendering than of the extended light fields. The volume rendering itself is an imperfect approximation to the theoretical model because it can only work on discretely sampled volumes.

In this example only the average intensities were given so far. It is still possible that some pixels are much more affected by a change of parameters than the majority. Therefore the absolute differences between colour intensities between samples adjacent in parameter space were accumulated over the sampled range to find the pixel and colour that changes most, *i.e.* the discrete approximation to gradients along position and steepness were calculated.

Figure B.7 identifies those regions of the images by mapping the average differences along position, along steepness and their average to the intensity of the pixel. Again, red, green, blue and their average are shown with a normalised scale.

The green channel of the pixel in column/row 55/74 (marked at the border of the image) has the highest average change for both directions of 129 per step. The same pixel also has the highest change for blue with 125 per step. Its green intensity value still changes quite smoothly when changing the parameters, as illustrated in figure 5.7.

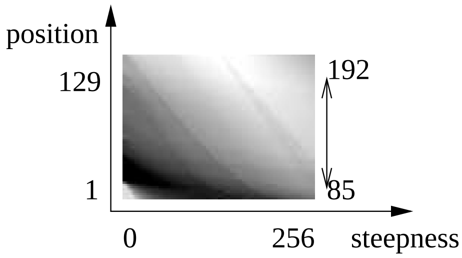


Figure 5.7: Average green intensity of pixel 55/74.

5.4 Application of Extended Light Fields

5.4.1 Volumetric Data

A generic light field that is suitable for volume rendering can be expected to have around 9MB. For the smaller parameter area and the head data set, this could be reduced to 2MB by using a small uv plane. This way a sampling of 16x16 was sufficient to avoid the artifacts mentioned before (page 7), and

256x256 pixels could be used for the *st* plane. Generating a first extended light field with four positions in the corners of the parameter area took about 2 hours with `Volpack`.

The size of an extended light field now depends on the volumetric data, how well it can be interpolated, and which additional parameter area shall be covered. For the smaller parameter area, the optimisation reaches the average error of 5.13 after dropping 64 positions out of 72 in a starting grid that was chosen with change prediction and adaptive sampling (threshold 10). Both the small original threshold and the small error when stopping the reduction guarantee a good interpolation of the correct volume rendering, which can be verified by comparing the original images with the interpolated ones in figure B.4.

With only eight positions left, it is possible to create such an extended light field within approximately 2 hours on a SGI O^2. It is only 17MB large. The result is visually pleasing and close to the original rendering (figure B.8), although fine details are lost due to the blurring which is inherent to light fields and worsened by the additional interpolation in extended light fields. So even though extended light fields should not be used to actually analyse a data set, they are perfectly suitable to chose a view point and parameter values for a high quality rendering of the data set. This is already supported by `liblight` and `lifview`.

5.4.2 Polygon Rendering

The "aging buddha" animation[4] is a good example of another possible application of extended light fields: in this animation, the effect of weathering on a metallic surface ([DH96]) was simulated. Time is the varying parameter that controls the texture of the buddha. As in every animation the view point had to be fixed when generating it. Figure B.9 shows how the shiny metall turns dull first and how green patina spreads from crevices that are not reached when polishing the statue until it is completely covered by patina in the end.

Analysis of these images showed that less than 8 positions would still provide a very good interpolation of the whole time interval. Unfortunately the source data for the animation was not available, so no extended light field could be generated although `liblight` supports the original rendering that was used (RIB). Generic light fields of this buddha have been generated

[4] `http://graphics.stanford.edu/papers/patina/buddha.mpg`

already and depending on the sampling density resulted in .lif files of 2.1–8.8MB gunzipped. An extended light field of a very high quality would require approximately 80MB of main memory, in other words, an interactive animation of the aging process is possible that would allow the observer to choose a view point while the buddha is aging.

Chapter 6

Conclusions

For the first time, light fields have been extended to varying scenes with arbitrary additional parameters. This extension is completely independent from the effect that the additional parameters have on the scene and changing the additional parameters of an extended light field during the interactive viewing is as fast as changing the view point. This is a major advantage especially in cases where changing these parameters would slow down the original rendering.

A pure software implementation is capable of achieving interactive frame rates for two additional parameters. A hardware supported implementation is possible either with MMX commands or texture mapping and is expected to achieve considerable performance improvements.

With these two parameters controlling the opacity mapping of volume rendering, rendering of an average volumetric data set can be speeded up by a factor of ten compared to a volume renderer that is already optimised for speed. The improvement is even higher for larger data sets or for more sophisticated volume rendering. Only extended light fields might make interactive viewing feasible in these cases.

The main problems of the extension have been dealt with by developing algorithms to find a good irregular sampling of the additional parameters and non-linear basis functions. They are especially tailored towards fast reconstruction and do not sacrifice rendering performance to increase quality.

Even though the non-linear basis functions cannot solve the problem that the quality of the reconstruction depends largely on the number and choice of sample positions, in reality the application of advanced basis functions is still an improvement that is definitely worth the minor additional computations during the optimisation.

More important is that the greedy algorithm finds a good set of samples within the parameter area. It relies on some assumptions about the effect of the additional parameters that hold for volume rendering and probably for many other applications, too. Apart from that it is as independent from the effect of the additional parameters as the extended light fields themselves, although problem specific knowledge can be applied to achieve better solutions. It can be applied to different reconstruction methods and works well for both linear and non-linear basis functions. Thus the reconstruction actually influences the choice of sample positions.

Although the greedy algorithm could be applied to any reconstruction, the efficient implementation described in section 3.2.2 relies on the imposed restrictions to cut down the amount of recalculations. These restrictions and the methods relying on them are essential to reduce the running time of the optimisation.

The hierarchical composition of the complete basis functions is sufficient to achieve a good interpolation. The advantage of increased similarity with the original images by keeping the primary direction of interpolation flexible outweighs the disadvantage of non-gradual changes between interpolated images when changing the parameters. A possible extension of this project might smooth the influences of the remaining samples after the optimisation and this way avoid the non-gradual changes. This would not complicate the reconstruction in `liblight`, because `liblight` could simply pick the relevant samples and their influences from a precalculated table for all parameter combinations.

Applied to volume rendering, a good reconstruction can be achieved with a sufficiently small number of sample positions, so that an extended light field can be generated for volumetric data. This approach is definitely feasible and fulfils the requirements established for this project: despite the tradeoff between *quality*, *performance* and *resource requirements*, the result satisfies all three and makes volume rendering using light fields a worthwhile alternative to the original volume rendering. The original rendering is still required, of course, but extended light fields can replace it, *e.g.* as a presentation tool on machines that would otherwise not be capable of interactive volume rendering.

The extension is not restricted to volume rendering. Other applications might be even more interesting, *e.g.* an animation with only one parameter (time) where computing single frames might take considerably longer than rendering an image of a volumetric object and close reconstruction of the original images is not as important as long as the result is visually pleasing. In this case very few sample positions may be sufficient.

Bibliography

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

[AHH+94] R. Avila, T. He, L. Hong, A. Kaufman, H. Pfister, C. Silva, L. Sobierajski, and S. Wang. VolVis: A diversified volume visualization system. In R. Daniel Bergeron and Arie E. Kaufman, editors, *Proceedings of the Conference on Visualization*, pages 31–39, Los Alamitos, CA, USA, October 1994. IEEE Computer Society Press.

[Bur88] P. J. Burt. Moment images, polynomial fit filters, and the problem of surface interpolation. In *CVPR'88 (IEEE Computer Society Conference on Computer Vision and Pattern Recognition, Ann Arbor, MI, June 5–9, 1988)*, pages 144–152, Washington, DC., June 1988. Computer Society Press.

[CCG96] Navin Chaddha, P.A. Chou, and Robert M. Gray. Constrained and recursive hierarchical table-lookup vector quantization. In James A. Storer and Martin Cohn, editors, *Proceedings of the 6th Data Compression Conference (DCC '96)*, pages 220–229. IEEE Computer Society Press, 1996.

[CLF98] Emilio Camahort, Apostolos Lerios, and Donald Fussel. Uniformly sampled light fields. In *9th EUROGRAPHICS Workshop on Rendering*, 1998. held in Vienna, Austria, June 29 – July 1 1998.

[Coo86] Robert L. Cook. Stochastic sampling in computer graphics. *ACM Transactions on Graphics*, 5(1):51–72, January 1986.

[DH96] Julie Dorsey and Pat Hanrahan. Modeling and rendering of metallic patinas. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 387–396. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[FLN90] Thomas A. Foley, David A. Lane, and Gregory M. Nielson. Towards animating raytraced volume visualization. *The Journal of Visualization and Computer Animation*, 1:2–8, 1990.

[FvFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics — Principles and Practice (2nd Ed.)*. Addison-Wesley, 1990.

[GG92] Allen Gersho and Robert M. Gray. *Vector Quantization and Signal Compression*. Communications and Information Theory. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[GGC97] Xianfeng Gu, Steven J. Gortler, and Michael F. Cohen. Polyhedral geometry and the two-plane parameterization. In Julie Dorsey and Philipp Slusallek, editors, *Eurographics Rendering Workshop 1997*, pages 1–12, New York City, NY, June 1997. Eurographics, Springer Wein. ISBN 3-211-83001-4.

[GGSC96] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 43–54. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[Kau96] A. Kaufman. Volume visualization. *ACM Computing Surveys*, 28(1):165–167, 1996.

[LH96] Marc Levoy and Pat Hanrahan. Light field rendering. In Holly Rushmeier, editor, *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 31–42. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[LL94] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear–warp factorization of the viewing transformation. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando,* *Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 451–458. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[SCG97] Peter-Pike Sloan, Michael F. Cohen, and Steven J. Gortler. Time critical lumigraph rendering. In *Symposium on Interactive 3D Graphics Conference Proceedings*, pages 17–23. Microsoft Research, ACM Press, 1997.

[TGFV98] G. Tsang, S. Ghali, E. L. Fiume, and A. N. Venetsanopoulos. A novel parameterization of the light field. In *Tenth Image and Multidimensional Digital Signal Processing Workshop*, 1998. held in Alphbach, Austria.

[Whi80] Turner Whitted. An improved illumination model for shaded display. *CACM, 1980*, 23(6):343–349, 1980.

[WHON97] Tsien-Tsi Wong, Pheng-Ann Heng, Siu-Hang Or, and Wai-Yin Ng. Image-based rendering with controllable illumination. In J. Dorsay and P. Slusallek, editors, *Rendering Techniques '97*, Eurographics Workshop Proceedings, pages 13–22. Eurographics Association, Springer Verlag, Wien, 1997. held in Saint Etienne, France, June 16-18.

Appendix A

Notation

$$
\begin{aligned}
n &= \text{number of additional parameters} \\
p_i &= \text{additional parameter } \#i \\
c, nc &= \text{discrete index of colour channel and their number} \\
u, v, s, t &= \text{plane parameters, either continuous or discrete} \\
&\quad\ \text{depending on the context} \\
nu, nv, ns, nt &= \text{number of sample positions for each plane parameter} \\
L(c, u, v, s, t, p_0, p_1, \ldots, p_n) &= \text{continuous intensity function} \\
\tilde{L}(c, u, v, s, t, p_0, p_1, \ldots, p_n) &= \text{reconstructed intensity function}
\end{aligned}
$$

regular sampling of parameter space:

$$
\begin{aligned}
m_i &= \text{number of sample position for parameter } \#i \\
x_{c,u,v,s,t,j_1,j_2,\ldots,j_n} &= \text{sampled value at a special position,} \\
&\quad\ 1 \leq j_i \leq m_i, 1 \leq i \leq n \\
P_{i,j_i} &= \text{value of parameter } \#i \text{ at sample } \#j_i \\
B_{j_1,\ldots,j_n}(c, u, v, s, t, p_0, p_1, \ldots, p_n) &= \text{basis function (reconstruction kernel)} \\
&= \text{associated with sample position } (j_1, \ldots, j_n)
\end{aligned}
$$

irregular sampling:

$$
\begin{aligned}
m &= \text{number of sample positions for the additional parameters} \\
x_{c,u,v,s,t,j} &= \text{sampled value at a special position, } 1 \leq m \leq m \\
P_{i,j} &= \text{value of parameter } \#i \text{ at sample } \#j \\
B_j(c, u, v, s, t, p_0, p_1, \ldots, p_n) &= \text{basis function associated with sample position } \#j
\end{aligned}
$$