Accelerated Volumetric Reconstruction From Uncalibrated Camera Views

Felicia Brisc, M.S.
Ph. D. Thesis
Dublin City University

Prof. Paul Whelan
School of Electronic Engineering

July 2006

I hereby certify that this material, which I now submit for assessment on the programme of study leading to the award of Ph.D. is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: ___________________________

(Candidate) ID No.: 51173719

Date: ______________________________

Abstract

While both work with images, computer graphics and computer vision are inverse problems. Computer graphics starts traditionally with input geometric models and produces image sequences. Computer vision starts with input image sequences and produces geometric models. In the last few years, there has been a convergence of research to bridge the gap between the two fields.

This convergence has produced a new field called *Image-based Rendering and Modeling* (IBMR). IBMR represents the effort of using the geometric information recovered from real images to generate new images with the hope that the synthesized ones appear photorealistic, as well as reducing the time spent on model creation.

In this dissertation, the capturing, geometric and photometric aspects of an IBMR system are studied. A versatile framework was developed that enables the reconstruction of scenes from images acquired with a handheld digital camera. The proposed system targets applications in areas such as Computer Gaming and Virtual Reality, from a low-cost perspective. In the spirit of IBMR, the human operator is allowed to provide the high-level information, while underlying algorithms are used to perform low-level computational work. Conforming to the latest architecture trends, we propose a streaming voxel carving method, allowing a fast GPU-based processing on commodity hardware.

Acknowledgements

This dissertation would never have been possible without the support of many people.

Thanks to my advisor, Prof. Paul Whelan, who gave me the opportunity to work in his research group. He provided an exciting working environment with many opportunities to work on promising applications and develop new ideas.

Thanks to Mr. Jim Dowling who provided me endless amounts of help and support. The financial support of Enterprise Ireland is also gratefully acknowledged.

This thesis is a testament to my wonderful family, loved ones and friends. They have always provided perspective, advice and encouragement throughout my time at DCU. To them I am most grateful. Special thanks to my mother - if it were not for her inspiration, I would not have come half as far.

Contents

List of Figures

List of Tables

Acronyms

IB	Image Buffer
IBMR	Image Based Modeling and Rendering
GPU	Graphics Processing Unit
GPGPU	General Processing on the Graphics Processing Unit
GVC	Generalized Voxel Coloring
LM	Levenberg-Marquardt
SVD	Singular Value Decomposition
SVL	Surface Voxel List

Chapter 1 Overview

1. 1 Introduction

The quest for visual fidelity has been the ultimate drive of computer graphics, ever since its beginnings. In the ever changing landscape of computer graphics systems, the last decade has seen the most significant transformation. Previously, dedicated hardware for computer graphics was only available in expensive workstations. Today, the vast majority of personal computers include high-performance graphics hardware as a standard component.

Consequently, the ubiquity of high performance hardware has spawn an impressive growth of fields like computer games, special effects, virtual reality, which in turn has triggered an insatiable demand for visual realism.

Image-based modeling and rendering (IBMR) has emerged as a field about half a decade ago, as an alternative to traditional geometry-based techniques. Its main purpose is to bridge the gap between computer graphics and computer vision in an effort to use real world images to create visually compelling photo-realistic images, while diminishing the time and effort needed to achieve this goal. The technology became mature enough to support successful commercial ventures, such as REALVIZ or 2D3 [Web1, Web2]. However, IBMR is still primarily a privilege of research laboratories and high-end studios.

There is an increasing demand for flexibility in IBMR tools, to allow their use under less restrictive conditions and minimize expert guidance. Novel methods for acquiring, reconstruction, and representing geometries and images are necessary in addition to new algorithms to efficiently analyze and process the input data.

In this context, the proposed research is focused on developing new techniques that will get us closer to the ultimate goal of a low-cost, interactive tool allowing non-expert users to build their own models and utilize the authored data for applications that simulate physical interaction with the real world. The system we designed to accomplish this goal presents the IBMR characteristic computer vision and computer graphics elements.

The computer vision component consists of a camera self-calibration method that relies only on information from an extended sequence of images acquired with a single digital camera.

The computer graphics component relies on a voxel carving method for achieving 3D reconstruction. Voxel carving techniques have become very popular in the IBMR field, as they provide a powerful tool for computing the volumetric model of the scene. However, due to their high computational costs they are traditionally the main bottleneck in IBMR pipelines, leading to a trade-off between performance and accuracy.

These computational costs are tackled in this work from two perspectives, both of which embody the interactive character of IBMR. Our first approach addresses the extended computational cost by locally adjusting the level of detail. More specifically, our approach introduces the novel feature of user-driven interactive refinement, resulting in a model reconstructed at varying resolution -and hence level of detail- across the voxel structure. Besides processing speed, the development of this feature is motivated by a second factor: scalability over various environments.

For the second technique, we focused exclusively on the processing speed. Here we have developed a GPU-based voxel carving method, motivated by the rapid increase in the performance of graphics hardware compared to the CPU, coupled with their recently exposed programmability. Both have made graphics hardware arguably today's most powerful commodity computational platform. As such, the computational power of GPUs has been harnessed for demanding tasks like ray tracing and photon-mapping, performed traditionally off-line on the CPU. Moreover, GPUs have transcended the boundaries of computer graphics and have been employed for *general-purpose* computing in a wide variety of domains ranging from physically-based simulations to sparse matrix multiplications techniques.

With our GPU-based work, we describe a method that provides interactive user involvement possibilities, and delivers high performance and flexibility, one that can be adapted for future graphics hardware.

1.2 3D Reconstruction Pipeline

The input to our system is a set of uncalibrated images of a scene acquired with a single moving digital camera, so that we need to perform self-calibration prior to the 3D reconstruction in order to recover the camera intrinsic and extrinsic parameters. The reconstruction pipeline is outlined in Figure 1.1. A number of relevant points are selected manually in a reference view, and then their corresponding points are tracked throughout the sequence. These identified correspondences are the only information from images needed to recover the position and orientation of the camera views. Within the camera self-calibration process, first the 3D structure of the tracked points and the camera parameters are retrieved in a projective frame through a rank-4 iterative factorization, followed by an upgrade to Euclidean structure by imposing metric constraints on the intrinsic camera parameters. Self-calibration is concluded by a sparse Levenberg-Marquardt optimization, providing a maximum likelihood estimation that minimizes the reprojection error with respect to all 3D points and camera parameters.

The voxel-based 3D model building is achieved through a *Space Carving* method, also called *voxel carving* in the literature. Space Carving approaches represent the space in which the scene occurs through a discretized volume of voxels and make occupancy decisions about whether voxels belong to the objects in the scene. The decision mechanism consists of a color similarity check of the pixels a visible voxel projects onto. The resulting 3D shape is the *photo hull*, the union of all possible photo-consistent scene reconstructions. We have developed two voxel carving formulations: a multi-resolution software (CPU-based) implementation and a GPU based carving engine.

1.3 Contributions

This dissertation makes several contributions to the areas of computer vision and computer graphics.

The thesis makes the following theoretical contribution:

- **Quaternion-parameterized optimization of the metric solution**

Structure from motion (SFM) methods process images over time, observing spatial and temporal changes that are caused by relative motion between camera and scene. We have employed robust digital image processing and computer vision techniques that allow the use of low-end acquisition systems such as standard photo or video cameras. Also, no information about the camera nor the scene is known *a priori* and the only requirement regarding the scene is that it will be assumed to be rigid.

Each step of the SFM analysis creates a more abstract and thus flexible representation, but each of these steps often introduces large errors and biases. The different solutions are computationally expensive and noise sensitive, and one of the goals of this work was to obtain more reliable methods based on a combination of linear techniques and non-linear bundle adjustment methods.

As such, we applied a linear stratified approach to compute the parameters of the camera and achieve Euclidean camera self-calibration. We followed up this work by implementing a sparse Levenberg-Marquardt optimization method with a quaternion-based parameterization of the camera rotations. This final non-linear optimization process is required in order to reduce the reprojection error accounting for all the non-linearities not recovered in the metric solution.

Moreover, if a more complete camera intrinsic parameters description is required (e.g. adding the principal point) it can be incorporated into the optimization process as well.

This method brings computational and memory usage benefits over the general variant of the Levenberg-Marquardt algorithm, by exploiting the sparse nature of the problem and reducing the number of overall parameters, respectively.

This dissertation also makes the following practical contributions:

Voxel carving:

The common characteristic for these approaches is that they carve a piece of voxelized virtual material that contains the object, similar to an artist sculpting a raw block of marble. The voxel carving process is based on the classification of thousands of discrete elements in scene space according to photo-consistency within scene images, leading to a typical trade-off between performance and accuracy.

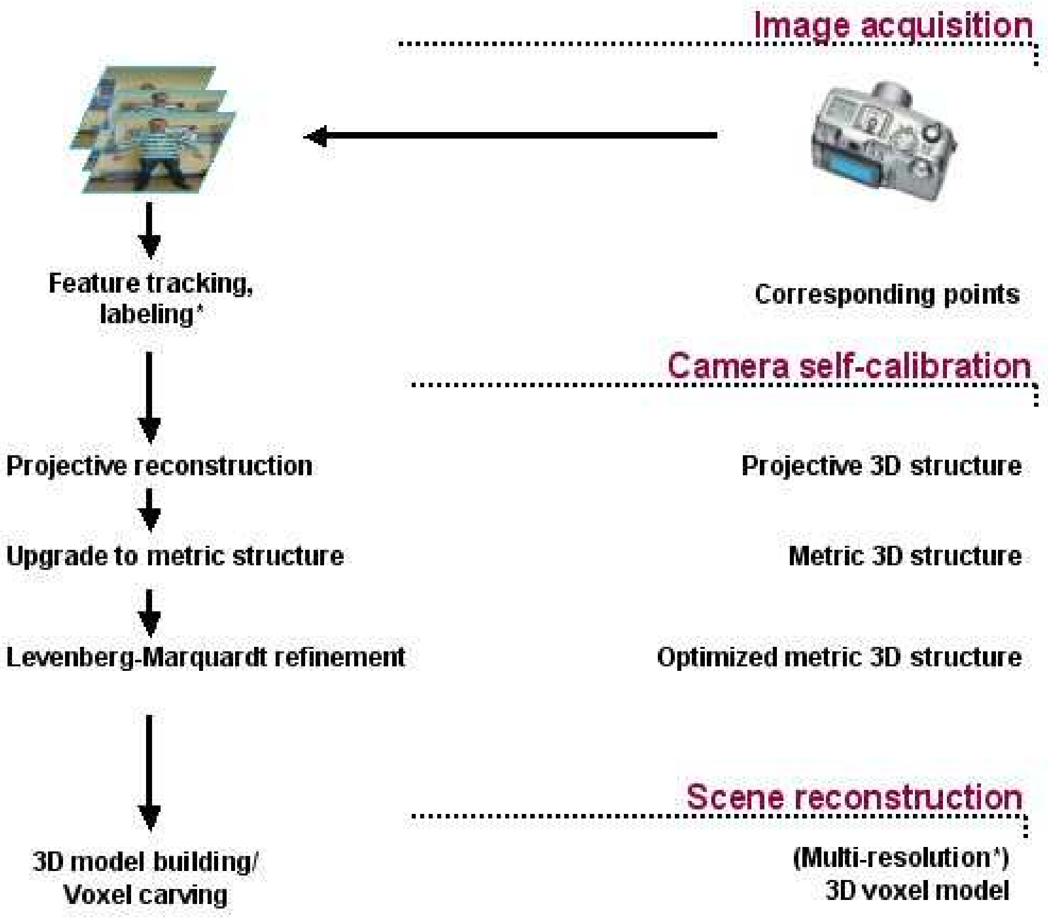


Image acquisition
Feature tracking, labeling*
Corresponding points
Camera self-calibration
Projective reconstruction
Projective 3D structure
Upgrade to metric structure
Metric 3D structure
Levenberg-Marquardt refinement
Optimized metric 3D structure
Scene reconstruction
3D model building/ Voxel carving
(Multi-resolution*) 3D voxel model

Figure 1.1 3D reconstruction pipeline.
**** Labeling and multi-resolution reconstruction pertain to the CPU-based implementation***

It is important to note here that this work focuses on uniformly lit scenes, therefore operates under the Lambertian assumption. The reconstruction of non-Lambertian scenes is an exciting area of future research described in more detail in Chapter 8, § 8.2.

- **Multi-resolution voxel carving**

Our initial approach addressed the extended computational cost by locally adjusting the level of detail [Bri04a, Bri04b]. Since perceptual importance is ultimately determined by the human factor, we have developed a multi-resolution approach that allows users to selectively control the complexity of different surface regions, while requiring only common image editing operations. An initial reconstruction at coarse resolution is followed by an iterative refining of the surface areas corresponding to the selected regions.

- **Voxel carving on the graphics processing unit**

We have developed a streaming, GPU-based voxel carving method, tackling the aforementioned computational costs in the context of the latest graphics hardware trends. The bandwidth efficient *carving engine* highlighted the importance of both the CPU and GPU to work in concert to carry out the given task. The philosophy behind this research was to partition the problem domain based on which computational components were better suited to each processor type while being mindful of the cost of inter-processor communication. Unlike previous approaches [LiMS04, WoeKoch04, ZacKar04] our method creates an explicit volume that can be modified interactively and eliminates the 3D texture inherent drawbacks by employing only surface voxels in a two-dimensional data representation that matches the two-dimensional data layout on the GPU.

- **A complete system for acquisition of metric 3D surface models from uncalibrated image sequences**

The self-calibration and reconstruction techniques mentioned earlier were incorporated into this system allowing for great flexibility in the acquisition of 3D models from images [Bri03, BriWhe04, BBS*04]. To our knowledge, this is the first system to integrate unconstrained structure from motion, self-calibration and GPU-based voxel reconstruction algorithms. This combination results in highly realistic 3D surface models obtained from images taken with an uncalibrated hand-held camera, without restriction on zoom or focus, and confers real-time characteristics to the 3D reconstruction process.

1.4 A Note on the CPU-based, Hardware Accelerated and GPU-based Paradigms

We provide in the following a brief consideration of the above concepts, in the order corresponding to the evolvement of computer graphics.

“CPU-based” applications, also called “software-based” in the literature, are processed entirely on the CPU, and therefore do not employ graphics hardware acceleration or computation.

It is especially important to emphasize the distinction between the “hardware accelerated” and “GPU-based” concepts. The former refers to employing graphics hardware exclusively for graphics processing purposes, while other generic operations are performed on the CPU, in a sequential manner. “GPU-based” on the other hand, refers to employing graphics hardware to process such general operations, in addition to processing graphics primitives.

1.5 Outline

We begin in Chapter 2 with a background discussion on structure from motion, volumetric reconstruction, and modern programmable graphics hardware.

Chapter 3 deals with the self-calibration of a single moving camera. After introducing several theoretical derivations, a flexible calibration method is presented that can deal with unknown motion and varying intrinsic camera parameters. Then, we present the formulation of the Levenberg-Marquardt optimization technique we have developed.

We examine the theoretical foundations of Space Carving and describe our CPU-based implementation of a voxel carving algorithm in Chapter 4. Chapter 5 is concerned with programmable graphics hardware concepts and characteristics. We describe the stream programming model and then present the abstraction of the programmable graphics processor as a stream processor.

We continue with the description of our streaming voxel carving method on the graphics processing unit in Chapter 6. In Chapter 7 results and applications of the system are presented. The flexibility and the potential of our approach is shown in several examples. Finally, we suggest areas of future research, and present the conclusions of our work in Chapter 8.

Throughout this dissertation the words ‘metric’ and ‘Euclidean’ will be used interchangeably.

Chapter 2 Related Work

Introduction

Previous work related to the work in this dissertation falls into three categories. First, structures from motion approaches have provided the basis for our camera calibration method. Second, our 3D reconstruction approach belongs to the generic framework of volumetric techniques, and more specifically to voxel carving techniques. Finally, previous work involving programmability in graphics hardware has inspired the GPU-based features of the implementation described in this dissertation.

2.1 Structure from Motion

Structure from motion methods seek to determine the relative motion of a moving camera from the acquired image sequence, as well as the shape, or *structure* of the observed objects. No information about the camera or the scene is known *a priori* and the only requirement regarding the scene is that it will be assumed to be rigid. Quite an impressive amount of research has appeared in the literature on structure from motion. We focus here on methods designed for full perspective (projective) cameras, arbitrary motion (equivalent to arbitrarily placed cameras), static scene, small baseline of the views and 3D model synthesis.

2.1.1 Camera Self-calibration

Self-calibration is the computation of metric properties of the cameras and/or scene from a set of uncalibrated images. Unlike conventional calibration, where these properties are determined from the image of a known calibration grid, self-calibration computes them directly from constraints on the intrinsic/extrinsic parameters.

Earlier reconstruction methods either worked only for the minimal number of views (typically two), or singled out a few 'privileged' views for initialization before being extended to the multi-view case [Hartley93, McLauMur95]. Shashua was the first to extend the two-image epipolar constraint to a trilinear constraint between matching points in three images [Shashua95]. Hartley [Hartley94a] showed that this constraint also applies to lines in three images, and Triggs [Triggs95a, 95b] studied the constraints for lines and points in any number of images.

For robustness and accuracy, there was a need for methods that uniformly take into account all the data in all the images, without relying on privileged features or images for initialization. The early factorization methods, developed by Tomasi and Kanade [TomKan92] for orthographic views and extended by Poelman and Kanade [PoeKan97] to weak perspective views partially fulfill these requirements, but they only apply when cameras are viewing small, distant scenes, which is seldom the case in practice. Triggs presented a key aspect in [Triggs96], namely that projective reconstruction is essentially a matter of recovering a coherent set of *projective depths* - projective scale factors representing the depth information lost during image projection.

The projective factorization method proposed by Triggs presents two key attractions:

- No initialization is required
- All of the data in all of the images is treated uniformly - there is no need to single out ‘privileged’ features or images

When nothing is known about the camera intrinsic parameters, the extrinsic parameters or the object, it is only possible to compute a reconstruction up to an unknown projective transformation (projective ambiguity) [Hartley94b]. The upgrade to Euclidean reconstruction requires some additional information about either camera or object, in order to be mathematically tractable. Since such information is not available, some assumptions need to be made, translating into constraints imposed on the camera views.

The earlier studies of self-calibration assumed unknown, but constant camera parameters [HeyAst96]. This has the disadvantage that zooming/focusing is prohibited. However, in the last decade there has been significant progress in the case of varying intrinsic parameters. Pollefeys and Van Gool [PolGol97] proposed a stratified approach for the case of varying focal length, which requires a pure translation motion for initialization. Triggs [Triggs97] introduced the concept of *absolute quadric*, and proposed a self-calibration method which relies on its invariant properties. Pollefeys et al. [PKG98] have shown that the absence of skew alone is sufficient for self-calibration and proposed a flexible method based on Trigg’s concept, which can deal with various constraints, but needs an initialization and is biased towards the first view in the image sequence. Heyden and Astrom [HeyAst99] proved that self-calibration can be achieved when only the aspect ratio was known and no skew was present (i.e. the sensor pixels have rectangular form).

The work presented in this dissertation is similar to the method presented in [HanKan00], which is computationally equivalent to recovering the absolute quadric. Their representation is explicit in the motion parameters (rotation axes and translation vectors) and enables the geometric constraints to be naturally enforced.

2.1.2 Bundle Adjustment

Bundle adjustment was employed initially in photogrammetry estimation problems [Sla80] and became gradually the technique of choice for structure and motion refinement in computer vision.

The groundwork for this transition was laid by Hartley [Hartley93]. Earlier structure from motion approaches required extreme computational accuracy and were difficult to work with for more than three or four views, because the number of solutions presented an exponential growth in the number of views [Luong92, MayFau92]. Hartley proposed an efficient solution based on a variant of the Levenberg-Marquardt algorithm, applicable to a large number of views. Building on the sparse block structure of the normal equations, Hartley presented his method in two flavors: as a direct Euclidean reconstruction iterative method, and as a bundle adjustment intermediary refinement step for projective reconstruction followed by Euclidean upgrade. Since then, variations of Hartley`s sparse bundle adjustment approach were frequently employed both as an intermediary and as a final optimization step in the literature.

Fitzgibbon and Zisserman [FitZis98] developed a system that employs a hierarchical strategy starting with image triplets, registered to sub-sequences and eventually to long open or closed sequences, with bundle adjustment applied after each of these processing stages.

Zhang and Shan [ZhaSha01] also employed triplet views, but in a sliding window format and formulated the refinement problem as a series of *local* bundle adjustments in such a way that the estimated parameters are consistent across the whole sequence.

Sainz [Sainz03] proposed a system that processed a large number of views simultaneously through projective factorization [HanKan00] and refined the structure prior to the Euclidean upgrade.

Pollefeys et al. [PGV*04] presented a reconstruction system from a sequence of uncalibrated images where the 3D structure retrieval is initiated with two views in a projective framework. The remaining views are incorporated sequentially in the process and the obtained structure and motion are then refined through bundle adjustment.

Lourakis provided in [LouArg04] an implementation with a detailed design description of Hartley’s sparse bundle adjustment method [Hartley93].

Our work is similar to the approach presented in [Sainz03], in that it follows the projective factorization method in [HanKan00] and applies a sparse LM optimization; however, our method differs by using a quaternion-based parameterization in order to recover the camera rotation matrices.

Among the various ways to represent rotation, we mention here Euler angles, orthonormal matrices, and Hamilton's quaternions. Of these representations, orthonormal matrices have been used most often in photogrammetry and vision. The quaternion formulation, however, presents a number of advantages. Besides the reduced number of necessary parameters, it is much simpler to enforce the unit magnitude constraint for quaternions than it is to ensure that a matrix is orthonormal.

The application of quaternions in stereo photogrammetry was pioneered by Schut [Schut59] and Thompson [Thompson59], who recovered the relative orientation of two coordinate systems with the help of three given common points. Horn developed a closed-form solution for more than three points in [Horn87] and later introduced quaternions to vision applications presenting an iterative scheme for recovering the relative orientation of two calibrated cameras [Horn90, 91].

Further research work explored most notably stereo systems with either pre-calibrated cameras, or calibrated through traditional methods during processing [Chou94, BacKam97, ZPA03]. Relative to these quaternion-based approaches, our method differs by using uncalibrated cameras and solving simultaneously for a large number of views.

2.2 Volumetric Reconstruction

All volumetric reconstruction algorithms assume a discrete and bounded 3D space containing the scene to be reconstructed. Typically, the initial reconstruction volume is divided into voxels and the task is to correctly classify the set of voxels that represent the different objects contained in the scene.

All these algorithms require a set of calibrated input images, and some of the approaches require additional classification of the pixels in background/foreground. A common assumption is that the objects contained in the scene are Lambertian or nearly Lambertian, so they reflect light equally in all directions. The following subsections present a review of some of the most significant methods based on volumetric reconstruction.

2.2.1 Volumetric Intersection

Volumetric intersection algorithms reconstruct the surface and interior space of an object using its silhouettes from the different reference views. The process is performed by tracing rays from the center of projection of each camera through the contour of the object projection in the corresponding image plane. The resulting bounding volume is the reconstructed scene.

The earliest attempts at volumetric model reconstruction from images were approximating the *visual hull* of the objects [Laurentini 94]. Such techniques are also referred to as *shape-from-silhouette* in the literature. The intersection of the generalized cones associated with a set of cameras defines a volume of space in which the object is guaranteed to lie. The visual hull is guaranteed to enclose the actual object. However, the volume only approximates the true 3D shape, depending on the number of views and the complexity of the object. Consequently, the accuracy of the reconstruction increases monotonically with the number of views.

Matusik *et al.* [MBR*00] describe an efficient real-time image-based approach to compute and shade visual hulls from silhouette image data. They use an ingenious traversing of pixels between camera images to reconstruct models. Taking advantage of epipolar geometry and incremental computation they achieve a constant rendering cost per rendered pixel. In a later work [MBM01], the same authors present new algorithms for creating and rendering visual hulls in real-time where an exact polyhedral representation for the visual hull is computed directly from the silhouettes.

Other recent real-time systems employing hardware-accelerated techniques: Lok presents [Lok 01] a system that renders a set of planes to generate novel views of visual hulls, and Li et al [LiMS03a] rasterize generalized cones with projective texturing to achieve real-time rendering frame rates.

Volumetric intersection methods are fast and simple algorithms but effective at reconstructing multi-view scenes. However, their inherent limitation is that they fail to recover concave regions that are not visible in the silhouette of the reference images.

2.2.2 Voxel Carving

Voxel carving methods have proven to be a strong alternative to traditional correspondence-based methods due to their flexible visibility models and explicit handling of occlusions.

Traditional reconstruction methods are using image matching techniques, such as multi-view stereo methods that compute correspondence across images and then recover 3D structure by triangulation and surface fitting. These approaches are especially effective with short video sequences, where tracking techniques simplify the correspondence problem. Some of the shortcomings of these methods are:

- small baseline (i.e. views must be close together) so that correspondence techniques are effective
- many partial models must often be computed with respect to a set of base viewpoints, and these surface patches must then be fused into a single, consistent model
- if sparse features are used, a parameterized surface model must be fitted to the 3D points to obtain the final dense surface reconstruction
- there is no explicit handling of occlusion differences between views

Volumetric methods avoid the listed disadvantages by replacing the image-based search problem used in the above approaches with a three-dimensional space-based search.

We can distinguish three main voxel carving implementation types, corresponding historically to the processing trends in computer graphics: CPU-based, hardware accelerated and GPU-based (please view note 1.4 at page 9).

Our voxel carving related work consists of two techniques that fall into the CPU-based and GPU-based categories, respectively. More specifically, we have developed a CPU-based multi-resolution voxel carving method and a GPU-based carving engine. In this context, we first relate our multi-resolution approach to other CPU-based methods, and then we position the carving engine relative to previous GPU-based research.

CPU-based Voxel Carving

Seitz and Dyer [SeiDye97] demonstrated that a colorful scene (assuming Lambertian illumination) could be reconstructed using full color-based consistency alone, without volume intersection. They introduced with the *Voxel Coloring* algorithm the *color consistency criterion* to distinguish points belonging to the object surface from other points in a scene. The Voxel Coloring algorithm begins with a reconstruction volume of initially opaque voxels that contains the scene to be reconstructed. As the algorithm runs, opaque voxels are tested for color consistency and those that are found to be inconsistent are carved, i.e. made transparent. The algorithm stops when all the remaining opaque voxels are color consistent.

The voxels need to be traversed in a monotonic order for a correct visibility handling. To simplify the voxel visibility computation and to allow reconstruction in a single scan of the voxels, Seitz and Dyer imposed an *ordinal visibility constraint* on the camera locations. The constraint implies however a limitation : since the voxels have to be visited in a single scan in near-to-far order relative to every camera, the cameras cannot surround the scene, so that surfaces that are not visible in any image cannot be reconstructed.

The *Space Carving* algorithm developed by Kutulakos and Seitz [KutSei99] achieves the goal of allowing arbitrary camera placement. Unlike Voxel Coloring, Space Carving evaluates one plane of voxels at a time, using multiple scans, typically along the positive and negative directions of each of the three axes. The scans are performed in near-to-far order relative to the cameras, by using only views behind the scanning plane (Figure 2.1).

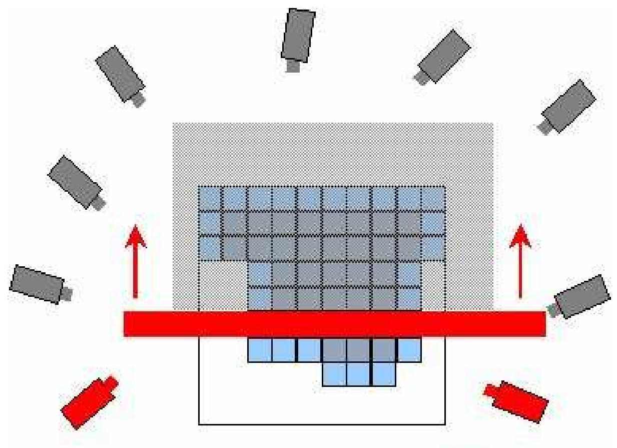


Figure 2.1 Only cameras behind the sweeping plane are used for photo-consistency check

Thus, when a voxel is evaluated, its visibility is already known relative to other voxels that might occlude it from the current camera. Space Carving never carves voxels it shouldn’t, but it is likely to produce a model that includes some color-inconsistent voxels. This is because cameras that are ahead of the scanning plane are not used for consistency checking, even when the voxels being checked are visible from those cameras. Hence, the color consistency of a voxel is, in general, never checked over the entire set of images from which it is visible. (A later paper, [Kutulakos 00b], describes additional book keeping that eliminates this shortcoming).

Culbertson et al. [CMS99] developed the Generalized Voxel Coloring (GVC), that obtains a color consistent model by computing visibility exactly. They provide experimental results that show that exact visibility, when compared with the approximate visibility computed by Space Carving, can result in better looking reconstructions that are numerically more consistent with the input images. Two variants of the algorithm, called

GVC-IB and GVC-LDI, have been developed. They use different data structures, called item buffers (IBs) and layered depth images (LDIs), to compute the visibility of voxels. An item buffer records for every pixel in an image, the surface voxel that is visible from the pixel. An LDI records for every pixel in an image, a depth-sorted list of all surface voxels that project to the pixel. The information in an LDI is a superset of the information in an item buffer and generally consumes considerably more memory.

The GVC-IB variant of the voxel coloring algorithm lies at the core of our multi-resolution carving method. However, our approach introduces the novel feature of user-driven interactive refinement, resulting in a model reconstructed at varying resolution - and hence level of detail- across the voxel structure. The development of this feature is motivated by two factors: processing speed and scalability over various environments.

First, voxel carving is a computationally expensive procedure, which typically requires at least tens of seconds up to tens of minutes to compute the reconstruction, implying a trade-off between processing speed and accuracy. Prock and Dyer [ProDye98] utilize a hierachical octree representation to speed up voxel coloring. Their method starts with a low resolution voxel structure, refined further to higher resolutions. Their method needs approximately 15 s to generate a reconstruction with 256^3 voxels. Unlike Prock and Dyer’s approach, our method does not process the entire model at uniform resolution, but introduces a perceptual saliency component in order to represent the information in a hierarchical order similar to that the human perceives.

Second, the delivery and rendering of 3D content over different types of connections to clients with various graphics capabilities requires scalable 3D models that can be approximated through representations of varying complexity. However, automatic simplification algorithms generate approximations that do not preserve the visual appearance of the original model in certain cases. For example, features such as eyes in a face are semantically crucial, but geometrically small. Kho and Garland [KhoGar03] developed a human-guided simplification method where the user can guide the vertex placement of a 3D model by directly interacting with the underlying algorithm.

The multi-resolution method developed for our framework relies similarly on the human factor to assign perceptual significance to selected features. Only simple 2D image editing operations are required to manipulate the complexity of different surface regions. Seitz and Kutulakos [SeiKut98] presented an image editing approach for multiple images of a scene. However, their method focuses on modifying and propagating changes to input images, rather than the voxel structure. The voxel model is used only as a proxy for these modifications, without its structure being altered.

In the following we mention several extensions and improvements that have been investigated in recent years. For a detailed review we suggest [SCM*01].

Eisert et. al [ESG99] proposed the *multi-hypothesis voxel coloring* technique. A *hypothesis* is a possible coloring of a voxel. Their approach begins with a *hypothesis assignment* step that identifies a set of hypotheses for each voxel. The algorithm then narrows down the hypotheses during a *hypothesis removal* step, which carves inconsistent voxels. Slabaugh [Slabaugh00b] presents a volumetric optimization using greedy and simulated annealing methods to refine the reconstruction. While the previous algorithms assumed opaque object voxels, the *Roxels* algorithm [DebVio99] attempts to reconstruct semi-transparent voxels.

All Space Carving approaches listed above need the input of accurately calibrated cameras. The *Approximate Space Carving* was defined later by Kutulakos [Kutulakos00] as an extension to the original algorithm which is capable of handling calibration errors.

Slabaugh [SMC00a] developed a method that warps the voxel space so that large scale scenes can be modeled without an excessive number of voxels (e.g. outdoors scenes). Vedula [VBS*00] presents a voxel coloring method that reconstructs a time-varying scene by linking two time-consecutive 3D voxel spaces together, forming a 6D space. The *Cell Carving* algorithm developed by Ziegler et al. [ZMP*03] uses correspondence between arbitrary image regions to enable the reconstruction of concavities that are difficult or impossible to reconstruct with other methods.

GPU-based Voxel Carving

While we focus in the following on GPU-based research, we also mention here briefly the preceding and rapidly superseded hardware-accelerated work.

Culbertson has pointed out in [Culbertson99] the possibility to perform hardware acceleration on voxel carving approaches. Sainz et al. [SBS02] present a fast hardware accelerated Space Carving method that uses texture mapping features of the graphic card. Their design is optimized further by the use of an octree structure and adaptive subdivision methods to keep track of the set of consistent voxels throughout the carving process. The authors do not provide a CPU/hardware acceleration speedup comparison, but they did not report interactive or real-time framerates.

With the advent of programmable graphics hardware [MGA*03], the research efforts shifted towards GPU-based processing. Li et al. proposed initially a GPU-based method to render visual hulls in [LMS03b, LMS03a], followed by a voxel-based approach that retrieves the photo-hull of a shape [LiMS04]. By adopting a view-dependent plane-sweeping strategy, they achieve rendering frame rates of 2-3 frames per second. A drawback of this approach is that since no branching support was available at that time, photo-consistency check is performed on each fragment, regardless whether it was rejected or not by the silhouette/background test. Also, their method produces no explicit volume, since rendering and reconstruction are combined into a single step.

Woetzel and Koch proposed a live system for image capturing and dense depth estimation in [WoeKoch04]. Their plane-sweep algorithm runs almost entirely on the GPU, leaving the main CPU free for other tasks such as image capture and higher level recognition. Dense depth maps are computed with up to 20 frames per second; however, their system is limited to only four camera views.

Zach and Karner [Zach04] describe hardware accelerated techniques for two scenarios: Voxel Coloring and Space Carving. Their implementation runs entirely on the GPU, with the exception of the latter, when they are performing independent sweeps and no prior information is used in the current sweep. In this specific case, the intersection of the obtained voxel models is performed on the CPU. The first one doesn`t produce an explicit volume, the second one does but at the expense of using two 3D textures simultaneously – one for the current, one for the precedent model – that are interesected at each main iteration. Besides the doubled memory consumption, the performance of the implementation is affected by the continuous access of a 3D texture. The authors report interactive framerates for 256 x 256 x 128 scene voxel resolution.

We have developed a streaming GPU-based voxel carving method, tackling the aforementioned computational costs of voxel carving in the context of the latest graphics hardware trends. Unlike previous approaches [LiMS04, WoeKoch04, ZacKar04] our method doesn't sacrifice interactivity for speed, creating an explicit volume that can be modified interactively and avoids the 3D texture inherent drawbacks (continuous access, using two 3D textures simultaneously) by employing only surface voxels in a GPU-optimal two-dimensional data representation. Moreover, our bandwidth efficient method is the first to minimize the GPU-CPU data transfer, employing a form of effective load balancing and combining the optimal features of both CPU and GPU while being mindful of the cost of inter-processor communication.

2.3 Graphics Hardware and Programmability

The graphics pipeline was historically a fixed-function pipeline, where the limited number of operations available at each stage of the graphics pipeline was hardwired for specific tasks.

One of the earliest efforts toward formalizing a programmable framework for graphics was Rob Cook's seminal work on shade trees [Cook84], which generalized the wide variety of shading and texturing models at the time into an abstract model. He provided a set of basic operations which could be combined into shaders of arbitrary complexity. His work was the basis of today's shading languages, which in turn contributed ideas to the widely-used RenderMan shading language [Upstill90].

RenderMan's success demonstrated the benefit of more flexible operations, particularly in the areas of lighting and shading. Instead of limiting lighting and shading operations to a few fixed functions, RenderMan evaluated a user-defined shader program on each primitive, with impressive visual results.

Over the past few years, graphic cards vendors have transformed the fixed-function pipeline into a more flexible programmable pipeline. This effort has been primarily concentrated on two stages of the graphics pipeline: the vertex stage and the fragment stage. In the programmable pipeline, the fixed-function operations are replaced with a user-defined *vertex program* and a user-defined *fragment program*, respectively. Each new generation of GPUs has exposed additional levels of programmability, precision and functionality of these two programmable stages.

The vital step for enabling not only graphics-specific, but also *general-purpose* computation on the graphics processing unit (GPGPU) was the introduction of fully programmable hardware and an assembly language for specifying programs to run on each vertex [LKM01] or fragment. The raw speed resulting from an abundant parallelism and rapidly expanding programmability of the graphics hardware make it an attractive platform for general-purpose computation. However, harnessing the power of the GPU goes well beyond simply "porting" applications from the CPU, due to its dissimilar programming model (the GPU-characteristic streaming programming model is detailed in Chapter 5).

A significant boost was gained from the appearance of high level languages to support the new programmability of the vertex and pixel pipelines [MGA*03, BFH*04]. An active, vibrant community of GPGPU developers has emerged [Web3] and quite an impressive amount of research has appeared already in the literature.

GPGPU applications range from computer graphics processes such as ray tracing [Purcell04], photon-mapping [PDC*03, LarChr04], collision detection [GLM05] to numeric computing operations such as dense and sparse matrix multiplications [KruWes03], physically-based simulations [Harris04, LFW*05] and computer vision [FunMan04], to name only a few. For an excellent review we suggest [OLG*05].

Kipfer, Segal and Westermann presented a versatile GPU-based particle system engine [KSW04]. They have efficiently implemented on the GPU algorithms used for particle manipulations, i.e. inter-particle collisions and visibility sorting algorithms. The analogy between the constant update and handling of a large number of primitives in a particle system and in a voxel carving application has inspired the GPU-based work presented in this dissertation. Also, our implementation capitalizes on recent GPU features that allow graphics memory objects to be treated as vertex data, texture or render target. [ARB03, NV04].

Conclusion

The surveyed structure from motion approaches depend on, one way or the other, recovering some kind of geometric structure of the scene. Original attempts of self-calibration have yielded successful examples only for special cases. For general cases, new algorithms needed to be developed that output explicit Euclidean structures from uncalibrated images.

Also, a review of the different volumetric methods for scene reconstruction from *N* views has been presented, with a focus on the family of methods based on space carving and color consistency. Finally, we presented a graphics hardware evolvement timeline, from the fixed to the modern flexible programmable pipeline.

Chapter 3 Camera Calibration

Introduction

Generally speaking, 3D reconstruction can be defined as the problem of using 2D measurements arising from a set of images of a scene, aiming to derive information related to the 3D scene geometry as well as the relative motion and the characteristics of the cameras employed to acquire these images.

Original results on this area come from researchers in the computer vision field, however, recent interest on the problem was raised because of the implicit applications in building 3D models for virtual and augmented reality or other interactive applications. Also, in the film and multimedia field, there has been an increased demand for computer-graphics based special effects consisting of the combination of 2D digital image sequences with 3D computer graphics that require a perfect synchronization only obtained with a calibration procedure. The common denominator of all these applications is that they require an accurate 3D scene reconstruction from the 2D source images.

The traditional way of performing such a calibration process is to use special setups and hardware devices in a controlled environment to mount and move the cameras around. Moreover, expert knowledge is required in order to operate such systems.

Since we aim to achieve a flexible, low-cost solution, operated by non-expert users, we will focus on the use of off-the-shelf devices, namely single digital still cameras. Additionally, we will assume that no prior knowledge of the camera or its relative motion in the scene is known. Therefore, the calibration process is based entirely on measurements taken from the input images.

We conclude this chapter with the description of a sparse variant of the Levenberg-Marquardt algorithm we have implemented to efficiently minimize the reprojection error between the observed and predicted image points, with the purpose of producing optimal estimates with respect to both 3D structure and viewing parameters.

Structure from motion

Structure From Motion (SFM) refers to the problem of recovering the 3D structure and motion of a scene from its two-dimensional projection onto the image plane of a moving camera. No information about the camera or the scene is known a-priori and the only assumption made is that the scene is required to be rigid.

The SFM analysis is based on preprocessing the set of reference views to consistently extract and label 2D salient points in the scene. These points can be detected automatically or manually on each image and then associated with their correspondents in the other images.

We can distinguish two main correspondence methods, depending on the number of points tracked along the image sequence. Thus, *sparse* correspondence methods evaluate a small set of points, while *dense* correspondence methods evaluate all the pixels in the sequence. The latter methods are based on determining the optical flow between frames, which limits the baseline or distance between each reference image. Moreover, it is argued in [BFA98] that the determination of the optical flow is an ill-posed problem due to inherent differences between 2-D motion field and intensity variations. It is reflected in [Chen00] that none of the optical flow based techniques produce low error and high density correspondences in all testing cases.

The presented work and the reviewed literature focuses on sparse correspondence methods such as the ones presented in [HanKan00], [Chen00], [PKG99], [Triggs96].

The fundamentals of epipolar and projective geometry, as well as related notions are covered in [Pollefeys00a] and [HarZis00]. Furthermore, a good review of projective and Euclidean reconstruction can be found in [Triggs96] and [Fusiello00], respectively.

3.1 Camera Geometry

This section briefly describes the *pinhole camera* model, perhaps the most widely used in computer vision to model the imaging process (Figure 3.1). In general, a pinhole camera projects a 3D world point with the homogeneous coordinates $\mathbf{M} = [X, Y, Z, 1]^T$ onto an image point $\mathbf{m} = [x, y, 1]^T$, where a line joining the point M to the centre of projection intersects the image plane. The world coordinates of the 3D point and its image coordinates are related by:

$$\lambda \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = P \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$ (3.1), where λ is an arbitrary scale factor, and P is a 3x4 matrix, called the *projective camera matrix*, or simply *camera matrix.*

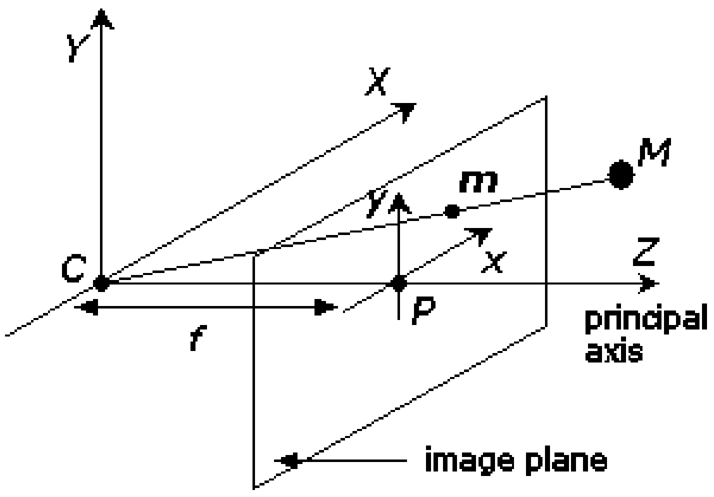


Figure 3.1 The pinhole camera model [HarZis00]

The projective matrix P is effectively modelling the camera, containing both its intrinsic and extrinsic parameters. P can be decomposed as [HarZis00]:

$$P = K[R \mid t] \quad (3.2),$$

where:

- R, t - are the rotation and translation from the world coordinate system to the camera coordinate system, representing the *extrinsic camera parameters*

- K is the calibration matrix, encoding the *intrinsic camera parameters*:

$$K = \begin{pmatrix} f_x & s & u_0 \\ 0 & \alpha f_y & v_0 \\ 0 & 0 & 1 \end{pmatrix} \quad (3.3)$$

where:

$f_x = -fk_x$, $f_y = -fk_y$ are the focal lengths in horizontal and vertical pixels, respectively (f is the focal length in mm, while k_x and k_y are the effective number of pixels/mm along the x and y axes)

s is the *skew parameter* , which is considered 0 for most cameras

α is the aspect ratio, which is considered 1 for most cameras

u_0, v_0 are the coordinates of the *principal point*, given by the intersection of the optical axes with the image plane (fig. 3.1) and the world reference frame.

3.2 Conics and Quadrics

The self-calibration method employed in our work is based on the recovery of the a*bsolute quadric*. In the following, we present the geometric entities and derive the equations underlying this processing step.

The *absolute conic,* Ω_∞ is a pure imaginary point conic situated on the plane at infinity π_∞ (the plane at infinity has in a metric frame the canonical form $\pi_\infty = (0,0,0,1)^T$ [Pollefeys00a]). A key property of the absolute conic is that it is fixed under any Euclidean transformation [Triggs97, HarZis00].

The absolute conic projects in the camera views to the *image of the absolute conic*, which is also an imaginary point conic (Figure 3.3), depending only on the intrinsic parameters of the camera:

$$\varpi = (KK^T)^{-1} \quad (3.4)$$

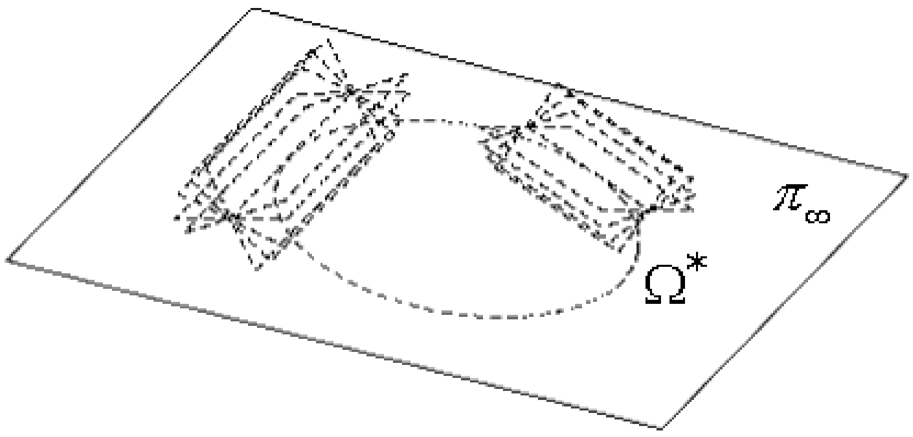


Figure 3.2 The absolute conic Ω_∞ and the absolute dual quadric Ω^* situated on the plane at infinity π_∞ in 3D space [Pollefeys00a]

We also define the *dual image of the absolute conic (DIAC),* which is a line conic (i.e. consisting of the lines tangent to ϖ) as [HarZis00]:

$$\varpi^* = \varpi^{-1} = KK^T \quad (3.5)$$

The equation above is one of the most important in self-calibration and it shows that once ϖ or ϖ^* is identified, then K can be also determined by decomposition (e.g. Cholesky factorization).

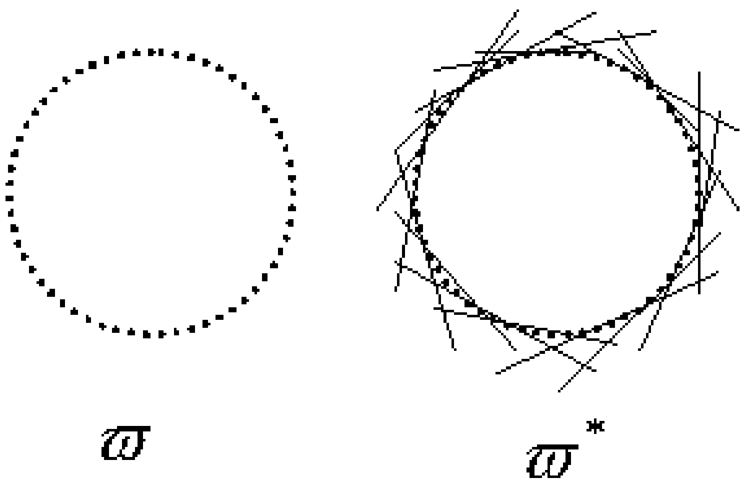


Figure 3.3. The image of the absolute conic (left) and the dual image of the absolute conic (right) [Pollefeys00a]

The *absolute dual quadric* Ω^* (or shortly the *absolute quadric*) is the dual of the absolute conic and is a degenerate dual quadric in 3-space [HarZis00]. Geometrically, it consists of the planes tangent to Ω_∞ (Figure 3.2) and its projection in the image plane is ω^*. Ω^* gives a concise way to compute the calibration parameters , since it's encoding both the plane at infinity π_∞ and the absolute conic Ω_∞, and it projects to the dual image of the absolute conic, so that :

$$\omega^* = P\Omega^* P^T \quad (3.6)$$

The absolute quadric is invariant under all Euclidean transformations, so that its relative position to a moving camera is constant (in a similar way we perceive the very distant objects as being fixed, for example a person driving on a road and observing the moon, will have the impression that the moon is following him).

If a projective reconstruction P^i was retrieved for n camera views, the next step is to determine the intrinsic camera parameters through self-calibration and to achieve a metric reconstruction.

In a metric frame, the absolute quadric's canonical form is a 4x4 symmetric matrix of rank 3:

$$\Omega^* = \tilde{I} = \begin{pmatrix} I_{3x3} & 0 \\ 0 & 0 \end{pmatrix} \quad (3.7)$$

where I_{3x3} is the 3x3 identity matrix.

In a projective frame, the absolute quadric is altered by a projective transformation H :

$$\Omega^* \to H\Omega^* H^T \quad (3.8)$$

Therefore, once Ω^* has been determined, the rectifying transformation can be easily computed by decomposing it as:

$$\Omega^* = H\tilde{I}H^T \qquad (3.9)$$

H upgrades the projective matrices to metric ones :

$P_i^M = P_i H$, such that a 3D point X^M from the Euclidean world frame is projected to the image points $x_i = P_i^M X^M$ in each view.

3.3 Camera Self-calibration

We now tackle the 3D motion and structure determination using a two-step stratified progression. As a pre-processing step we divide the complete sequence into sub-sequences, enforcing a common frame for consecutive fragments in order to increase the robustness of reconstruction [Sainz03]. In each of these subsequences we will use the aforementioned stratified approach to recover both camera and scene structure similar to methods presented in [Pollefeys99, HanKan00].

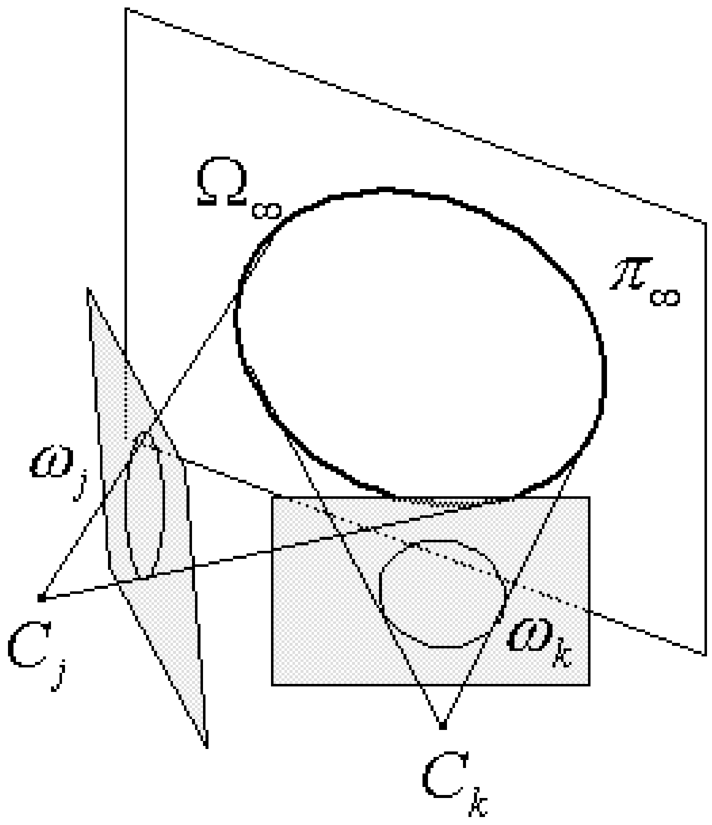


Figure 3.4: The absolute conic and its projection in the images [Pollefeys00a]

First, an initial projective reconstruction is obtained, which is computed from the set of correspondences. Then, depending on assumptions translating to constraints an upgrade to metric structure is computed. One advantage of the presented approach is that it allows recovery of a Euclidean reconstruction of the scene without relying on any initial solution, which is one of the drawbacks of most existing methods. Another important feature is that the entire calibration process relies on solving linear systems using Singular Value Decomposition.

When the different subsequences have been successfully calibrated, a merging process groups them into a single set of cameras and reconstructed features of the scene.

3.3.1 Projective Reconstruction

Given n distinct camera views of m object points represented by homogeneous coordinates $\mathbf{x}_j$, $j = 1...m$, the task is to compute their 3D their projective structure.

Under the pinhole camera model assumption (§ 3.1), the projective mapping between a 3D world point $\mathbf{x}_j$ and its 2D projection in images (u_{ij}, v_{ij}) is given by:

$$\begin{bmatrix} u_{ij} \\ v_{ij} \\ 1 \end{bmatrix} \sim P_i \mathbf{x}_j$$, which holds only up to a constant factor.

Writing this factor explicitely, we have:

$$\lambda_{ij} \begin{bmatrix} u_{ij} \\ v_{ij} \\ 1 \end{bmatrix} = P_i \mathbf{x}_j$$ (3.10), where λ_{ij} are non-zero scale factors called *projective depths*.

We may stack the above equation for n perspective cameras and m object points, obtaining the equivalent matrix:

$$\mathbf{W}_s = \begin{bmatrix} \lambda_{11} \begin{bmatrix} u_{11} \\ v_{11} \\ 1 \end{bmatrix} & \cdots & \lambda_{1m} \begin{bmatrix} u_{1m} \\ v_{1m} \\ 1 \end{bmatrix} \\ \vdots & \vdots & \vdots \\ \lambda_{n1} \begin{bmatrix} u_{n1} \\ v_{n1} \\ 1 \end{bmatrix} & \cdots & \lambda_{nm} \begin{bmatrix} u_{nm} \\ v_{nm} \\ 1 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} P_1 \\ \vdots \\ P_n \end{bmatrix} [\mathbf{x}_1 \ldots \mathbf{x}_m] = \mathbf{PX} \quad (3.11)$$

where:

$\mathbf{W}_s$ is the $3n \times m$ *scaled measurement matrix*

$\mathbf{P}$ is the $3n \times 4$ *perspective matrix*

$\mathbf{X}$ is the $4 \times m$ shape matrix.

$\mathbf{W}_s$ should have rank-4 matrix (since it's the product of two matrices with 4 columns and rows, respectively), so that a rank-4 factorization of it produces a projective reconstruction of the points. However, in reality, due to noise and measurement errors its rank will be different and the rank-4 constraint has to be enforced.

On the other hand, equation (3.11) holds only if the correct scale factors λ_{ij} are applied to each of the measured points $\mathbf{x}_{ij}$. In order to fulfil both requirements, a rank-4 factorization needs to be applied on $\mathbf{W}_s$ until the recovered projective depths make equation (3.11) consistent.

There exist different approaches [StuTri96, Triggs96, HanKan00] to construct an iterative algorithm that converges to a rank-4 decomposition of the measurement matrix $\mathbf{W}_s$. A popular example of the factorization strategy is outlined below:

1. Initialize $\lambda_{ij} = 1$ for $i = 1 \ldots n$ and $j = 1 \ldots m$
2. Compute the current scaled measurement matrix W_s by equation (3.11)
3. Perform rank-4 factorization on $\mathbf{W}_s$, generate the projective motion and shape
4. Reset $\lambda_{ij} = P_i^{(3)} \mathbf{x}_j$ where $P_i^{(3)}$ denotes the third row of the projection matrix P_i
5. If λ_{ij}'s are the same as the previous iteration, stop; else go to step 2.

In order to avoid the trivial solution, a 'balancing' step is required for each iteration [HarZis00] that brings all matrix rows and columns to the same order of magnitude. Unfortunately, because of this step there is no guarantee that the above algorithm will converge, even to a local minimum [MahHeb00, MHO*01, Oliensis99].

We are performing factorization using an iterative approach similar to the one proposed in [Chen00], where the projective depths are rescaled at each iteration to give a closer rank-4 approximation of $\mathbf{W}_s$. While there is no theoretical proof of convergence, [Oliensis99] has shown that the algorithm minimizes an error function that measures the size of the non-rank-4 fraction of $\mathbf{W}_s$. Also, [Oliensis99] and [Sainz03] have reported excellent results both with accurate and noisy data.

The overall sequence of processing steps of the employed Iterative Factorization Algorithm (IFA) is the following:

1. Initialize $\lambda_{ij} = 1$ for $i = 1 \ldots n$ and $j = 1 \ldots m$

2. The current scaled measurement matrix W_s^k (with $W_s^0 = W_s$) is determined by equation (3.11). An initial Singular Value Decomposition of W_s^k is computed:

$$W_s^k = UDV^T \ ,$$

where :

U	is a $3m \times n$ matrix with orthogonal columns
V	is a $n \times n$ orthogonal matrix
D	is a $n \times n$ diagonal matrix, its elements σ_i are the singular values of W_s^k

3. We update the measurement matrix W_s^k with its rank-4 approximation $\tilde{W}_s^k$:

We denote $P^k = U_4$, where U_4 is the submatrix obtained from U by truncating only the 4 first columns (associated to the 4 largest singular values $\sigma_{1,\ldots,4}$).

Similarly, $X^k = D_4 V_4$, and from that we estimate :

$$\tilde{W}_s^k = P^k X^k \ .$$

This solution guarantees [Golub96] that we get the best rank-4 approximation of the measurement matrix W_s^k, and the spectral distance (using $\| \|_2$) from the subspace of the rank-4 is exactly σ_5, the 5th largest singular value.

4. We scale the matrix W_s^k by the λ_{ij}^k coefficients of $\tilde{W}_s^k$ in order to bring each depth factor as close as possible to the ideal rank-4 decomposition. In order to maintain the projection to the image points x_{ij}, we only need to scale λ_{ij}^k along the ray from the centre of projection through x_{ij}. Hence, the new depth λ_{ij}^{k+1} will coincide with the projection of X_j^k into the projection ray. The projective update formula [Chen00] is:

$$\lambda_{ij}^{k+1} = \lambda_{ij}^k \frac{(\tilde{W}_{s\,ij}^k)^T \cdot W_{s\,ij}^k}{(W_{s\,ij}^k)^T \cdot W_{s\,ij}^k},$$

where $\tilde{W}_{s\,ij}^k$ and $W_{s\,ij}^k$ are 3-vectors corresponding the the *ij*-th element of the respective matrices.

5. The measurement matrix is updated with the new depth values: $\tilde{W}_{s\,ij}^{k+1} = x_{ij}\lambda_{ij}^{k+1}$. We repeat the process until the corresponding σ_5^{k+1} value is either small enough or it is stabilized. Of course, due to noise in the image measurements, σ_5 can reach small values, but will always be different from zero.

In our implementation, we are working with pre-conditioned image coordinates and we are balancing the projective depth matrix before each iteration. The pre-conditioning and balancing processes are described in more detail in chapter 7.

3.3.2 Upgrade to Metric Structure

The factorization of Equation (3.11) is not unique, but presents a *projective ambiguity*. That is, we can recover motion and shape only up to an unknown projective transformation:

$$W_s = \hat{P}\hat{X} = \hat{P}HH^{-1}\hat{X} = P^M X^M \qquad (3.12)$$

with $P^M = \hat{P}H$ and $X^M = H^{-1}\hat{X}$,

where $\hat{P}$ and $\hat{X}$ are the projective motion and the projective shape, respectively.

This projective ambiguity refers to the fact that any non-singular 4x4 matrix could be inserted between $\hat{P}$ and $\hat{X}$ leading to another motion and shape pair. The upgrade to metric structure is reduced then to the recovery of the rectifying transformation H , called the *projective distortion matrix* (PDM).

As mentioned previously in (3.2, § 3.1), in a metric frame the camera matrix P_i can be decomposed as:

$$P_i^M \sim K_i[R_i \,|\, t_i], \quad i = 1,\ldots,n \qquad (3.13)$$

where :

$K_i = \begin{bmatrix} f_i & 0 & u_{0i} \\ 0 & \alpha_i f_i & v_{0i} \\ 0 & 0 & 1 \end{bmatrix}$, the calibration matrix (see also Eq. 3.3, § 3.1) encoding the intrinsic parameters of the *i*-th camera:

f_i represents the focal length, (u_{0i}, v_{0i}) are the image coordinates of the principal point, α_i is the aspect ratio,

while R_i and t_i encode the extrinsic camera parameters :

$R_i = \begin{bmatrix} i_i^T \\ j_i^T \\ k_i^T \end{bmatrix}$ is the *i*-th rotation matrix with i_i, j_i, k_i denoting the rotation axes,

$t_i = \begin{bmatrix} t_{xi} \\ t_{yi} \\ t_{zi} \end{bmatrix}$ is the *i*-th translation vector.

We choose the world coordinate frame to coincide with the first camera, since we are not concerned here with the absolute scaling, rotation and translation of the scene.

Therefore, the rotation and translation for the first camera become $R_1 = I_{3x3}$ and $t_1 = 0$, while Equation (3.13) will have the simplified form: $P_1^M = K_1[I \mid 0]$. The same similarity components can be factored out from the projective reconstruction, with $P_1 = [I \mid 0]$. The condition $P_1^M = P_1 H$ becomes $[K_1 \mid 0] = [I \mid 0]$ and we can write H as:

$$H = \begin{bmatrix} K_1 & 0 \\ v^T & 1 \end{bmatrix}$$

The submatrix formed by the first column, i.e. vectors v together with K_1, specifies the plane at infinity in the projective space. Since the coordinates of π_∞ in the metric space are $\pi_\infty = (0,0,0,1)^T$, in the projective space they will be transformed by H, so that we can recover π_∞ as :

$$\pi_\infty = H^{-T}\begin{pmatrix}0\\0\\0\\1\end{pmatrix} = \begin{bmatrix}(K_1)^T & -(K_1)^{-T}v\\ 0 & 1\end{bmatrix}\begin{pmatrix}0\\0\\0\\1\end{pmatrix} = \begin{pmatrix}-(K_1)^{-T}v\\ 1\end{pmatrix}$$

Writing the plane at infinity as $\pi_\infty = (p^T,1)^T$, where $p = -(K_1)^{-T}v$, the projective distortion matrix H can be written as :

$$H = \begin{bmatrix}K_1 & 0\\ -p^T K_1 & 1\end{bmatrix} \qquad (3.14).$$

From Equation 3.14 results that the projective to metric upgrade involves the recovery of eight parameters: three parameters for p and five for K_1, respectively. This corresponds to a counting argument: the plane at infinity and the absolute conic have 3 and 5 degrees of freedom, respectively.

To recover these parameters, we start by identifying the self-calibration equations. We partition the camera matrices of the projective reconstruction into $P_i = [S_i \mid s_i]$, distinguishing between the first three and last columns. From $P_i^M = P_i H$ and from Equation (3.14) we obtain:

$$K_i R_i = (S_i - s_i p^T)K_1, \quad i = 2,...,m \qquad (3.15)$$

which can be rearranged as :

$$R_i = (K_i)^{-1}(S_i - s_i p^T)K_1, \quad i = 2,...,m$$

and considering that $RR^T = I$ (since rotation matrices are orthogonal) , we obtain :

$$K_i K_i^T = (S_i - s_i p^T)K_1 K_1^T (S_i - s_i p^T), \quad i = 2,....,m \qquad (3.16)$$

From the equation of the dual image of the absolute conic (Eq. 3.5, §3.2): $\varpi_i^* = K_i K_i^T$, and substituting in (3.16) we obtain the basic equation for self-calibration:

$$\varpi_i^* = (S_i - s_i p^T)\varpi_1^*(S_i - s_i p^T)^T , \qquad (3.17)$$

relating the unknown entries of ϖ_i^* and unknown parameters p with the known entries of the projective cameras S_i, s_i.

The dual image of the absolute conic is related to the absolute (dual) quadric by:

$$\varpi_i^* = P_i \Omega^* P_i^T \qquad (3.18)$$

The absolute quadric has in the Euclidean space the canonical form $\Omega^* = \tilde{I} = \begin{bmatrix} I_{3x3} & 0 \\ 0 & 0 \end{bmatrix}$, while in a projective space it will follow the projective transformation rule for dual quadrics, giving :

$$\Omega^* = H \begin{bmatrix} I_{3x3} & 0 \\ 0 & 0 \end{bmatrix} H^T = H\tilde{I}H^T \qquad \text{(Eq. 3.9, §3.2).}$$

Using Equation (3.14) the projective reconstruction gives the relation:

$$\Omega^* = \begin{bmatrix} K_1 K_1^T & -K_1 K_1^T p \\ -p^T K_1 K_1^T & p^T K_1 K_1^T p \end{bmatrix} = \begin{bmatrix} \varpi_1^* & -\varpi_1^* p \\ -p^T \varpi_1^* & p^T \varpi_1^* p \end{bmatrix} \qquad (3.19).$$

If we substitute the above relation in Equation (3.18) we obtain once again the self-calibration equation (3.17). This corresponds to the interpretation provided in § 3.2 of the absolute dual quadric Ω^* as being fixed under the camera motion and each of the dual images of the absolute conic ω_i^* are the respective images of Ω^* for each of the views. The most important consequence is that imposing certain constraints on ω_i^*, we can translate them to Ω^* using Equation (3.18) via the known matrices P_i and solve for Ω^* in projective space, using the resulting matrices from the projective factorization.

A linear system can be obtained making some assumptions on the camera intrinsic parameters:

- if principal point is at the center of the image plane, then $(\varpi_i^*)_{13} = (\varpi_i^*)_{23} = 0$
- zero skew of the pixels implies $(\varpi_i^*)_{12} = 0$
- aspect ratio equal to 1 implies $(\varpi_i^*)_{11} = (\varpi_i^*)_{22}$

These assumptions leave only the focal length as a variable parameter and generate four linear constraints on Ω^* available from each view. The self-calibration equations become an overdetermined linear system of $4\times m$ equations that can be solved by Singular Value Decomposition, with a unique solution for $m \geq 3$. After obtaining Ω^*, H can be easily determined by decomposition and back-substituting it in (3.12) a final metric reconstruction is computed under the above assumptions of known principal points and skew values.

3.4 Non-linear Optimization of the Metric Reconstruction

3.4.1 Bundle Adjustment

Our aim here is to minimize the reprojection error between the observed and predicted image points, with the goal to produce jointly optimal estimates with respect to both 3D structure and viewing parameters (camera pose and/or calibration). This kind of problems can be treated by non-linear least-squares methods, often referred to as *bundle adjustment* in the literature since all of the values of an initial guess of the solution are modified together. Such methods can be summarized as having two distinct phases: initial parameter estimation and then iterative refinement, protecting the refinement process against divergence.

More specifically, we are employing the Levenberg-Marquardt (LM) optimization, a non-linear least-squares technique which has proven to be most successful due to its use of effective damping strategy that confers it the ability to converge quickly from a wide range of initial guesses.

In the general case, least-squares methods are used to solve a set of non-linear equations that have been linearized using a first order Taylor expansion, resulting in a system known as the *normal equations*. The computational stages may become quite expensive, due to the fact that the iterative solving of the normal equations amounts to computing the solution to a dense linear system, with a complexity $O(n^3)$ in the number of unknown parameters. Fortunately, due to the lack of interaction between parameters for different points and cameras, the Jacobian matrix of the objective function has a sparse structure we can exploit in implementing the LM method.

3.4.2 The Levenberg-Marquardt Algorithm

We will provide here a brief description of the LM algorithm, for an extensive analysis we suggest [NocWri99, LawHan95].

As mentioned in the introduction, the non-linear computational model is an iterative process. Let f be an assumed functional relation $\mathbf{X} = f(\mathbf{P})$, where $\mathbf{X} \in \Re^N$ is a measurement vector and $\mathbf{P} \in \Re^M$ is a parameter vector. We start by assigning initial values to the parameter vector, $\mathbf{P}_0$ and to the measurement vector $\hat{\mathbf{X}}$. Our aim is to determine the parameter vector $\hat{\mathbf{P}}$ that best satisfies this functional relation locally. That is, we seek the vector $\hat{\mathbf{P}}$ satisfying $\hat{\mathbf{X}} = f(\hat{\mathbf{P}}) - \varepsilon$ for which the squared distance $\|\varepsilon\|$ is minimized.

We assume that for a parameter shift vector $\Delta\mathbf{P}$, f is approximated by:
$f(\mathbf{P} + \Delta\mathbf{P}) \approx f(\mathbf{P}) + \mathbf{J}\Delta\mathbf{P}$, where $\mathbf{J}$ is the Jacobian matrix of f, $\mathbf{J} = \partial f / \partial \mathbf{P}$.

We set up the normal equations and solve for the shift vector $\mathbf{\Delta P}$:

$$\mathbf{J^T J \Delta P} = \mathbf{J^T}\varepsilon$$

In order to improve the direction of the shift vector, it needs to be rotated so that it point towards the minimum. A way of rotating the shift vector towards the direction of steepest descent was proposed independently by Levenberg [Levenberg44] and later by Marquardt [Marquardt63]. Marquardt introduced a new parameter λ, so that the normal equations become:

$$(\mathbf{J^T J} + \lambda\mathbf{I})\, \Delta\mathbf{P} = \mathbf{J^T}\varepsilon, \qquad (3.20)$$

where λ is a strictly positive scalar called *damping parameter*.

The damping parameter is added to the diagonal elements of the $\mathbf{J^T J}$ matrix, and is then adjusted at each iteration so as to ensure that the error decreases. LM is an adaptive algorithm that controls its own damping: it increases the damping if the step vector $\mathbf{\Delta P}$ fails to reduce the sum of squares $\varepsilon^T\varepsilon$; otherwise it reduces the damping. In this way LM can navigate difficult model nonlinearities, although at low speed, behaving in a steepest descent manner. Yet, it can also rapidly approach a local minimum with nearly quadratic convergence speed, becoming a Gauss-Newton method. The process of repeatedly solving the normal equations for different values of the damping term until an acceptable update $\mathbf{P} + \mathbf{\Delta P}$ to the parameter vector is found corresponds to an iteration of the LM algorithm.

3.4.3 Refinement of the Metric Reconstruction

The camera self-calibration process presented in §3.3 is a closed form least squares constrained approximation of the structure from motion problem. We have extended the self-calibration process by implementing a final non-linear optimization process in order to reduce the reprojection error accounting for all the non-linearities not recovered in the metric solution.

Additionally, if during the preprocessing of the measurement matrix some of the measurements have been left out because they were not present in all the views, we have the possibility to include them in this nonlinear analysis to improve the overall error.

We employ bundle adjustment in order to obtain a maximum likelihood estimation that minimizes the reprojection error with respect to all 3D points and camera parameters, i.e. the mean squared distances between the measurements x_{ij} and the reprojected image points from a new estimation of P_i^M and X_j . The minimization criterion can be expressed as:

$$\min \sum_{i=1}^{m} \sum_{j=1}^{n} d(x_{i,j}, P_i^M X_j)^2 \qquad (3.21)$$

The non-linear functions to minimize in our case are the pinhole camera projection equations (Equation 3.13, §3.3.2) for the different measurements and frames, which can be written in the following form:

$$x = \frac{f(i_x X + i_y Y + i_z Z + t_x) + \beta(j_x X + j_y Y + j_z Z + t_y) + u_0(k_x X + k_y Y + k_z Z + t_z)}{k_x X + k_y Y + k_z Z + t_z}$$

$$y = \frac{\alpha f(j_x X + j_y Y + j_z Z + t_y) + v_0(k_x X + k_y Y + k_z Z + t_z)}{k_x X + k_y Y + k_z Z + t_z}$$

Under the following common assumptions: no skew ($\beta = 0$), the central point perfectly centered on the image plane ($u_0 = v_0 = 0$), and normalized coordinates ($\alpha = 1$), the above equations have a simplified representation:

$$x = \frac{f(i_x X + i_y Y + i_z Z + t_x)}{k_x X + k_y Y + k_z Z + t_z}$$

$$y = \frac{f(j_x X + j_y Y + j_z Z + t_y)}{k_x X + k_y Y + k_z Z + t_z}$$

The general strategy for adjustment of the damping parameter is as follows: We start by initializing λ to 10^{-3}, following Hartley's approach in [Hartley93] and [HarZis00]. A large initial value – for example 1 or 10 – would initially step LM in a more nearly steepest-descent direction, whereas a smaller value, e.g. 10^{-3} or 10^{-2}, will begin in a more nearly quadratic Gauss-Newton solution direction.

If the solution obtained for $\Delta\mathbf{P}_i$ decreases the error, the solution is accepted and the value of λ is divided by 10 before the next iteration. If the case is that the error increases, λ is multiplied by 10 and the normal equations are solved again until an effective value for λ is obtained that decreases the iteration error.

We define our parameter vector $\mathbf{P} \in \Re^M$ by concatenating all parameters describing the m camera projection matrices (see § 3.1 for camera matrix) and the n 3D points in Eq. 3.21.

In order to reduce the overall number of parameters, we have replaced the camera rotation matrices with quaternions of unit length and have imposed the orthonormality of the camera axes. Quaternions are extensions of complex numbers that can be represented as a 4-component vector:

$$q = (q_0, q_1, q_2, q_3),$$

where the first three components are real numbers, and the last one is an imaginary number. Regarding memory usage, quaternions require only four floating point values, compared to the nine floating point values of a 3x3 rotation matrix. Thus, they take up less space but can still be quickly converted to rotation matrices. Additionally, in our parameterization scheme, the fourth component q_3 is fixed under the unit length condition, therefore we can write the camera matrices in the following parameterized form:

$$a_i = (t_x, t_y, t_z, q_0, q_1, q_2, f)^T$$

Also, each 3D point *i* is parametrized by a vector:

$$b_j = (X_x, X_y, X_z)^T$$

We shape the parameter vector P as:

$$P = (a_1^T, ..., a_m^T, b_1^T, ..., b_n^T)^T$$

We accumulate the measured image point coordinates across all cameras in order to construct a vector of measurements $X \in \Re^{\mathbf{N}}$:

$$X = (x_{1,1}, \ldots, x_{1,n}, \ldots, x_{m,1}, \ldots, x_{m,n})^T$$

For each parameter vector, an estimated measurement vector $\hat{\mathbf{X}}$ is generated by our functional relation $\hat{X} = f(P)$:

$$\hat{X} = (\hat{x}_{1,1}, \ldots, \hat{x}_{1,n}, \ldots, \hat{x}_{m,1}, \ldots, \hat{x}_{m,n})^T$$

Since an image point x_{ij} depends only on the parameters of the *j*-th camera, $\partial\hat{x}_{ij} / \partial a_k = 0$, $\forall\ j \neq k$ and $\partial\hat{x}_{ij} / \partial b_k = 0$, $\forall\ i \neq k$

The step vector ΔP and the residual vector ε can be further partitioned into camera and 3D structure blocks as $\Delta P = (\delta_a^T, \delta_b^T)^T$ and $\varepsilon = (\varepsilon_a, \varepsilon_b)^T$ respectively.

Given the above partitioning, the form of the resulting Jacobian matrix $J = \dfrac{\partial X}{\partial P}$ and the normal equation $J^T J \Delta = J^T \varepsilon$ are shown in Figure 3.5 a) and b), respectively.

Figure 3.5 a) illustrates the sparse structure of the Jacobian matrix and of the normal equations $N = J^T J$ (Figure 3.5 b).

The blocks that form these matrices can be written down as follows:

$$U_k = \sum_i \left(\frac{\delta \hat{x}_{ki}}{\delta a_k} \right)^T \left(\frac{\delta \hat{x}_{ki}}{\delta a_k} \right)$$

$$V_i = \sum_k \left(\frac{\delta \hat{x}_{ki}}{\delta b_i} \right)^T \left(\frac{\delta \hat{x}_{ki}}{\delta b_i} \right)$$

$$W_{ki} = \left(\frac{\delta \hat{x}_{ki}}{\delta a_k} \right) \left(\frac{\delta \hat{x}_{ki}}{\delta b_i} \right)$$

$$\varepsilon_{\mathbf{a}_k} = \sum_i \left(\frac{\delta \hat{x}_{ki}}{\delta a_k} \right)^T \varepsilon_{ki}$$

$$\varepsilon_{\mathbf{b}_i} = \sum_i \left(\frac{\delta x_{ki}}{\delta b_i} \right)^T \varepsilon_{ki}$$

We can write the normal equations in the partitioned form:

$$\begin{bmatrix} U^* & W \\ W^T & V^* \end{bmatrix} \begin{bmatrix} \delta_a \\ \delta_b \end{bmatrix} = \begin{bmatrix} \varepsilon_a \\ \varepsilon_b \end{bmatrix} \qquad (3.22)$$

Both ε_a and ε_b are sparse matrices, since measured image points x_{ij} are affected only by the i-th camera and 3D point X_j. Consequently, both U and V are block-diagonal matrices, while W is in general not sparse (Figure 3.5 b).

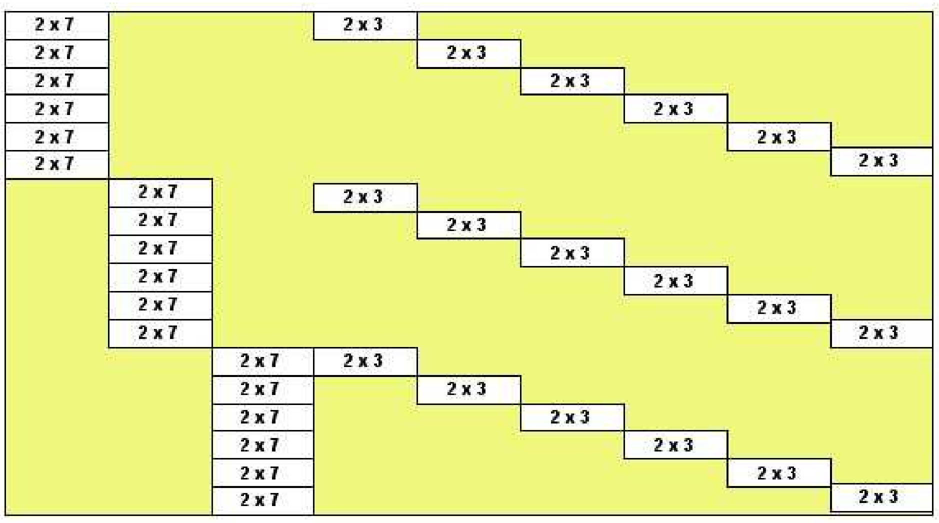
We perform a Gaussian elimination step, by left multiplying both sides of the equation with the block matrix $\begin{bmatrix} I & -WV^{*^{-1}} \\ 0 & I \end{bmatrix}$, assuming that V^* is invertible. This yields a lower triangular block matrix:

$$\begin{bmatrix} U^* - WV^{*^{-1}}W^T & 0 \\ W^T & V^* \end{bmatrix} \begin{bmatrix} \delta_a \\ \delta_b \end{bmatrix} = \begin{bmatrix} \varepsilon_a - WV^{*^{-1}}\varepsilon_b \\ \varepsilon_b \end{bmatrix} \qquad (3.23)$$

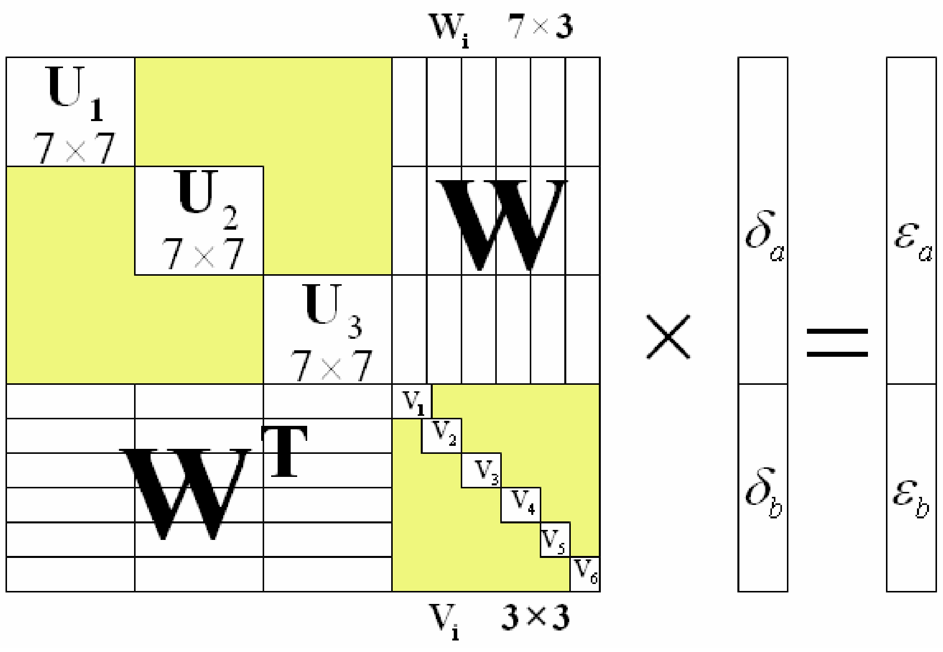
We can determine the motion update vector δ_a by solving the upper block of (3.23):

$$(U^* - WV^{*^{-1}}W^T)\delta_a = \varepsilon_a - WV^{*^{-1}}\varepsilon_b \ , \qquad \text{(eq. 3.24)}$$

Equation 3.24 can be efficiently solved using the Cholesky factorization of $S \equiv U^{*} - WV^{*^{-1}}W^{T}$.



a).



b).

Figure 3.5 Sparse structures of the Jacobian (a) and normal equations (b). This particular example illustrates the matrices for 3 images and 6 points [NocWri99].

Having solved for δ_a , the structure update vector δ_b can then be computed by substitution into the bottom half of Eq. 3.23, obtaining:

$$V^* \delta_b = \varepsilon_b - W^T \delta_a$$

In our calibration problem, V^* is a *3m x 3m* matrix, composed of diagonal blocks of *3 x 3*, making the calculation of $V^{*^{-1}}$ very efficient. The matrix W is a set of blocks of *7 x 3* giving a final matrix of *7m x 3n*. The most expensive operation to solve Equation 3.20 is the inversion of the term $(U^* - WV^{*^{-1}} W^T)$ which turns to be a matrix of *7m x 7m* elements, where m is the number of reference images and n is the number of measured features.

Keeping in mind that usually $n >> m$ in order to obtain statistically stable solutions, the sparse approach we have developed provides a better solution than solving directly the normal equations which require inverting the *3n x 3n* sized $\mathbf{J}^T\mathbf{J}$ matrix.

3.5 Sequence Merging

As mentioned in §3.3, we fragment long sequences into subsequences sharing at least one common frame, and each of them is calibrated into a metric reconstruction. The next step is to merge the information of the individual fragments to recover a complete scene structure (Figure 3.6).

Therefore the merging of two sub-sequences is performed in metric space by determining the set of common points, between the last frame of one sub-sequence and the first of the next one, which by construction corresponds to the same camera and measurements. Any pair of corresponding image points X and X' representing the same 3D point M in two consecutive frames is related by a homography H, according to the equations:

$X \cong HX'$ and $P \cong P'H^{-1}$

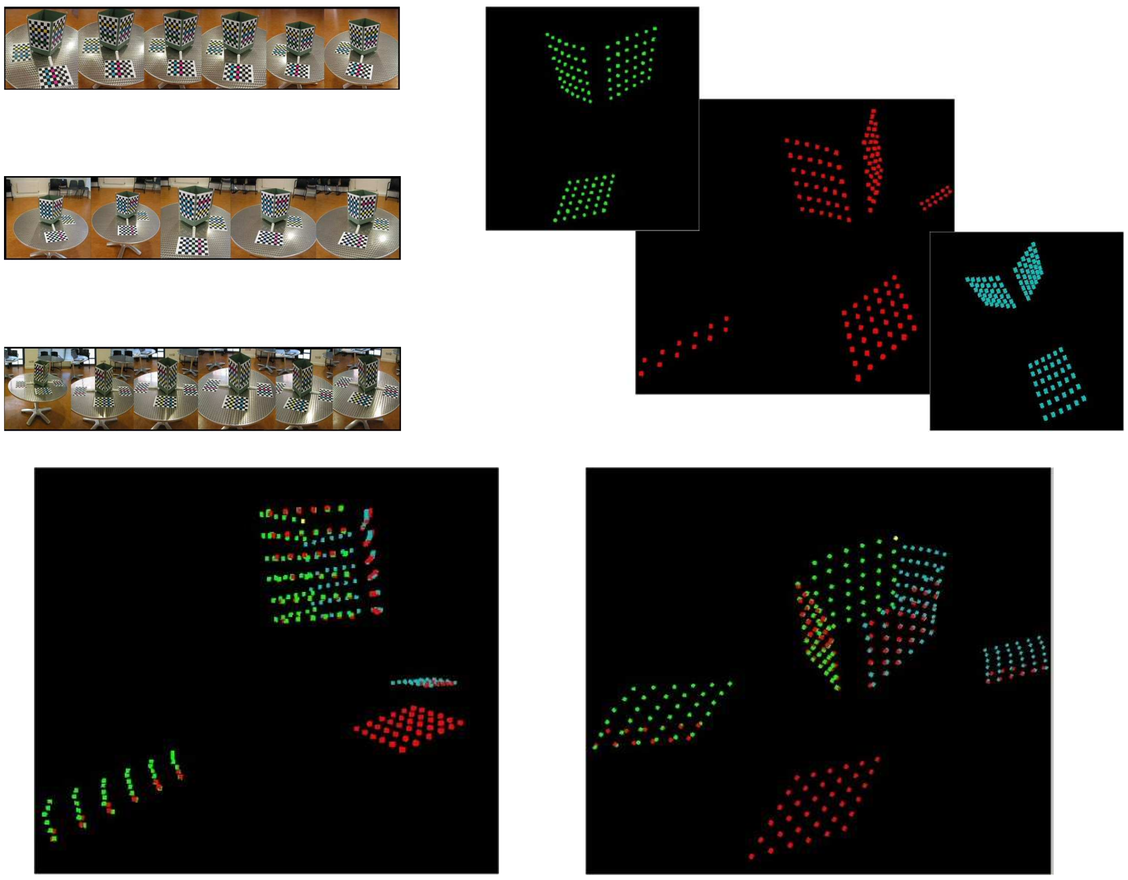


Figure 3.6 An example showing three merged sub-sequences.

where P and $P^{'}$ are the metric camera projection matrices representing the same reference view expressed in different basis. H is a homography that maps the points from one basis to the other one. We want to determine H minimizing the following distance:

$$\sum_i d^2(PHX^{'}, PX)$$

for all common overlapping points M between the two subsequences. The distance function is assumed to be the standard Euclidean distance. Considering that we have two reconstructed metric frames P **,** $P^{'}$, we can bring them to a common reference basis. We can multiply each of the camera matrices by its inverse as follows:

$$PX = PP^{-1}PX = [I\,|\,0]PX = [I\,|\,0]\tilde{X}$$

$$P'X' = P'P'^{-1}P'\;X' = [I\,|\,0]P'X' = [I\,|\,0]\tilde{X}'$$

Since the two sets of points are representations of the same set of real points and they share identical 2D projections, we can restrict the homography H to be a uniform scale and a translation, yielding the following expression:

$$H = \begin{bmatrix} s & 0 & 0 & t_x \\ 0 & s & 0 & t_y \\ 0 & 0 & s & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The homography can be recovered by establishing the correspondence between four known points. An overdetermined equation system can be built and a least squares solution can be obtained.

Conclusion

In order to extract the 3D information of a scene from a sequence of images, we have to completely recover the camera external and intrinsic parameters, i.e. position and orientation as well as at least the focal length that were used during the acquisition process. In this context, camera calibration is a critical problem in the absence of prior geometric information. We described a robust stratified linear based algorithm that calibrates each of the subsequences to a metric structure. Further, in order to deal with the errors accounting for all the non-linearities not recovered in the self-calibration solution, we described a maximum likelihood optimization implementation that minimizes the reprojection error between the observed and predicted image points.

Chapter 4 Voxel Carving

Introduction

Volumetric models are a natural choice for scene reconstruction, the task of generating a 3D model of a scene from multiple 2D images. Three broad classes of volumetric reconstruction techniques have been developed based on geometric intersections, color-consistency, and stereo matching. Some of these techniques have spawn a number of variations and undergone considerable refinement.

The focus of this work lies in the second class of techniques, that obtain shape from color-consistency, which have the generic name of Space Carving or simply voxel carving. As noted in the introductory chapter, we operate under the Lambertian assumption. In this chapter, we present the theoretical foundations of Space Carving as introduced by Kutulakos and Seitz [KutSei99] followed by a description of our multi-resolution solution that tackles the extended computational costs from a human perception point of view.

4.1 Background

4.1.1 Theoretical Foundations of Space Carving

Kutulakos and Seitz have proposed in [KutSei99] a novel approach for reconstructing 3D scenes from a set of N camera views that gracefully handles shape recovery when no constraints are placed upon the shape of the scene or the placement of the cameras.

The *Space Carving* theory addresses the problem of reconstructing scenes from a set of *N* views for the case when:

- there are no constraints on the scene geometry,
- also, there are no constraints on the position of the input views,
- no information is available of any type of salient features in the input photographs (i.e. edges, lines, points, etc)
- there is no prior correspondence information.

The authors note in [KutSei99] that a first requirement is that the viewpoint of each image is known (i.e. calibrated reference views are needed). A second requirement is that the radiance of the scene is *locally computable*, that is, the scene has a parameterized radiance model (e.g. Lambertian, Phong).

There are many advantages of this family of scene reconstruction:

- The Space Carving algorithm is the only provably correct method that handles shape reconstruction from arbitrarily placed cameras.
- The solution volume provides the tightest possible bound of the scene that can be extracted from the set of *N* given views, regardless of the specific algorithm employed to obtain it.
- Since no constraints on the camera positions are imposed, the solution is a global reconstruction, eliminating the need of partial reconstructions and merges.

- Since the recovered shape is guaranteed to be photo-consistent with the reference views, visually accurate reprojections can be obtained.

In the following, we present a summary of the Space Carving theory introduced in [KutSei99].

We assume a volume in space with an unknown shape υ defined by a closed and opaque set of points. We also assume that there exists a set of N perspective projection views $I_1,\dots,I_N$ taken respectively from a set of known locations $C_1,\dots,C_N$.

The points on the surface of the shape are contained in $Surf(\upsilon)$ and the radiance of a point $p \in Surf(\upsilon)$ in this surface is described by a function $Rad_P(r)$ that maps every oriented ray r passing through the point to the color of light reflected from p along that direction. The set of radiance functions $Rad(r,p)$ for every point $p \in Surf(\upsilon)$ and υ form the *shape-radiance scene description*, which can reproduce uniquely any image from any viewpoint.

The set of all possible shape-radiance descriptions can be partitioned in two sets, based on whether they reproduce or not the input images. This constraint is defined for a given shape and a given scene radiance as *photo-consistency* and was formalized by Kutulakos and Seitz in the following set of definitions:

Point Photo-Consistency:

Let S be an arbitrary subset of R^3. A point $p \in S$ visible from C_i is photo-consistent with the image I_i if it does not project to a background pixel and the color of the projection of p is equal to $Rad_P(r)$. If p is not visible from C_i it will be conservatively considered as photo-consistent.

Shape-Radiance Photo-Consistency:

A shape-radiance scene description is photo-consistent with a reference image I_i obtained from C_i, if all points visible from C_i are photo-consistent and every non-background pixel is the projection of a point in υ.

Shape Photo-Consistency:

A shape υ is photo-consistent with a set of N reference views if there is an assignment of radiance functions $Rad(r, p)$ to the visible points in υ that produce a photo-consistent shape-radiance description with all the reference images.

The constraints that photo-consistency imposes on the shape of a scene in order to obtain a valid reconstruction are the following:

Background constraint:

Since photo-consistency requires that no point of υ projects to a background pixel, in the case an image I obtained from C has identifiable background pixels, υ is restricted to the cone defined by C and the non-background pixels of I.

This constraint exploits the information about the background pixels and is very powerful when such information is available. Given a set of N such images, the scene is then restricted to a useful volume obtained by intersecting their respective cones, known as the the *visual hull* [Laur94]. However, this constraint becomes useless when we have no information on background pixels, and the visual hull degenerates to R^3.

The main drawback of the visual hull is that it does not model the shape concavities. The following constraints are needed in order to enable the reconstruction of shape concavities:

Radiance constraints

To model these restrictions let's define the following consistency criteria [KutSei99]:

- A method $consistency_K$ is available that takes as input $k \leq N$ colors $col_1, \ldots, col_k$, k vectors $r_1, \ldots, r_k$ and the known light source positions (if non-Lambertian models are used) and determines if it is possible for a surface point $p \in Surf(\upsilon)$ to reflect light of col_i in direction r_i for all $i = 1, \ldots, k$ at the same time.

- $consistency_K$ is monotonic, i.e. if $consistency_K(col_1,\dots col_j, r_1,\dots r_j)$ is true, then $consistency_K(col_1,\dots col_{j-1}, r_1,\dots r_{j-1})$ for every permutation set of $1,\dots, j$ is also true.

These criteria define a *locally computable* class of radiance models, that is, they present a locality property: the radiance at any point is independent of the radiance of all other points in the scene. Given a locally computable radiance model and a $consistency_K$ function, the photo-consistency status of every point $p \in Surf(\upsilon)$ of the scene from a set of N images can be fully determined, and more importantly, the non-photo consistency, which conveys significant information about the underlying shape of the scene, is known too.

With the following lemmas Kutulakos and Seitz, describe how the non-photo-consistency of a shape υ affects the photo-consistency of its sub-sets.

Visibility lemma

Let $p \in Surf(\upsilon)$, and let $Vis_\upsilon(p)$ be the set of reference views in which p is not occluded by υ. If $\upsilon' \subset \upsilon$ is a shape that contains p, then $Vis_\upsilon(p) \subset Vis_{\upsilon'}(p)$

Non-photo consistency lemma

If $p \in Surf(\upsilon)$ is not photo-consistent with a subset of $Vis_\upsilon(p)$, it is not photo-consistent with $Vis_\upsilon(p)$.

These lemmas (illustrated in Figure 4.1) show the underlying monotonic tendency exhibited by the family of Space Carving algorithms: the set of reference views from which a given point $p \in Surf(\upsilon)$ is visible, strictly expands as υ gets smaller.

Also, if more reference images are available, new constraints are added, which means that once a point fails to be photo-consistent, there is no additional reference view that can re-establish photo-consistency.

This is stated in the following theorem [KutSei99]:

Subset theorem

If $p \in Surf(\upsilon)$ is not photo-consistent, no photo-consistent subset of υ contains *p*.

These concepts can be further developed to lead to the following theorem [KutSei99], which states that for any shape υ there is a unique photo-consistent shape that contains any other photo-consistent within its volume, giving a least commitment reconstruction.

Photo hull theorem

Let υ be an arbitrary subset of R^3. If υ^* is the union of all photo-consistent shapes in υ, every point on the surface of υ^* is photo-consistent. The set υ^* is called the photo hull.

When we have a finite scene that can be contained in a discretized volume, these set of properties can be used to define a generic algorithm that will compute the photo-hull by iteratively removing elements of the initial volume υ until it converges to the photo-hull. The decision mechanism is the photo-consistency criterion, where voxel evaluation is done with the help of a sweeping plane moving along preset directions (usually the *XYZ* axes). For each position of the plane voxels are evaluated by projecting them onto camera views that are *behind* the sweeping plane (Figure 2.1, §2.2.2, Chapter 2). This is a convenient method of keeping track of voxel visibility, i.e. occluders are visited before the voxels that they occlude.

The space carving algorithm requires a number of photo-consistency tests that is upper bounded by *n x m*, where *n* is the number of reference views and *m* is the initial number of voxels in the uncarved volume.

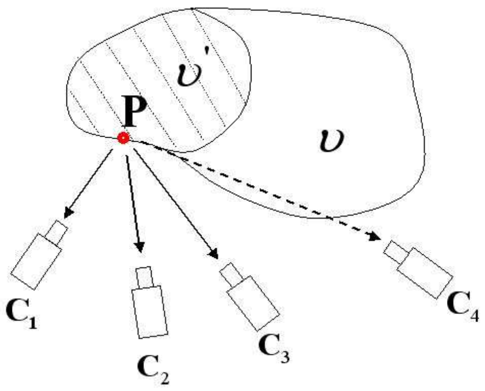


Figure 4.1 Illustration of the visibility and non-photo-consistency lemmas. If P is non-photo-consistent with the views at C_1, C_2, C_3, it is also non-photo-consistent with the entire set $\mathrm{Vis}_{\upsilon'}(P) = \{C_1, C_2, C_3, C_4\}$

The different implementations that follow this procedure belong to the family of space carving algorithms. In order to specify a useful algorithm we need to specify the following issues:

- How can we select the initial volume υ that contains the scene
- What representation of that volume facilitates carving.
- How the carving process is carried in each iteration to guarantee the convergence to the photo-hull.
- What conditions are needed to terminate the carving process.

4.2 Multi-resolution Voxel Carving

Space Carving [KutSei99] is conservative, it never carves voxels it shouldn't, but it may produce a model that includes some color-inconsistent voxels. This is because cameras that are ahead of the scanning plane are not used for consistency checking, even when the voxels being checked are visible from those cameras. Hence, the color consistency of a voxel is, in general, never checked over the entire set of images from which it is visible.

Our work builds on an alternative approach proposed in [CMS99]. The *Generalized Voxel Coloring* (GVC) provides an efficient implementation of the space carving algorithm that computes visibility exactly in order to obtain a color-consistent model. The authors provide experimental results that show that exact visibility, when compared with the approximate visibility computed by Space Carving, can result in visually more accurate reconstructions that are numerically more consistent with the input images.

The algorithm operates only on surface voxels, providing a two-way mapping between border voxels and image pixels. We exploit this bidirectional relationship to propose a user-guided method for creating multi-resolution 3D models, with varying level of detail across the surface.

Voxel carving approaches imply the classification of a large number of discrete elements, implying a trade-off between performance and accuracy. Moreover, the delivery and rendering of 3D digital content over different types of connections to clients with various graphics capabilities requires scalable 3D models that can be approximated through representations of varying complexity.

The automatic simplification algorithms developed in the last decade generate an approximation of fewer polygons from complex models. However, their approximations do not preserve the visual appearance of the original model in certain cases. For example, features such as eyes in a face are semantically crucial, but geometrically small. Kho and Garland [KhoGar03] developed a human-guided simplification method where the user can guide the vertex placement of a 3D model by directly interacting with the underlying algorithm.

The multi-resolution method developed for our framework relies similarly on the human factor to assign perceptual importance to selected features. However, by exploiting the two-way voxel-pixel mapping provided by the surface voxel list and the image buffer, only simple 2D image editing operations are required to control the complexity of different surface regions.

The size of the initial volume of voxels containing the 3D scene is determined by upscaling the spatial bounds of the recovered 3D points during self-calibration (Figure 4.2). First, we assign every voxel a unique ID. We also assume that initially all voxels are opaque, i.e. uncarved. We are considering a point voxel projection, so that only the voxel center is projected to the input images, leading to a single pixel in each view.

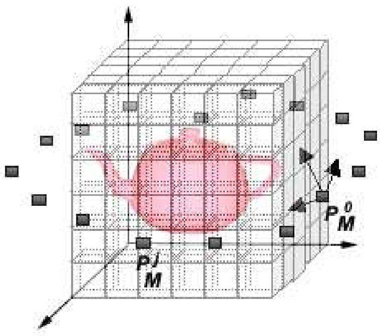


Figure 4.2 The initial bounding box of voxels is containing the 3D scene. $P_M^0 \ldots P_M^j$ denote the camera projection matrices

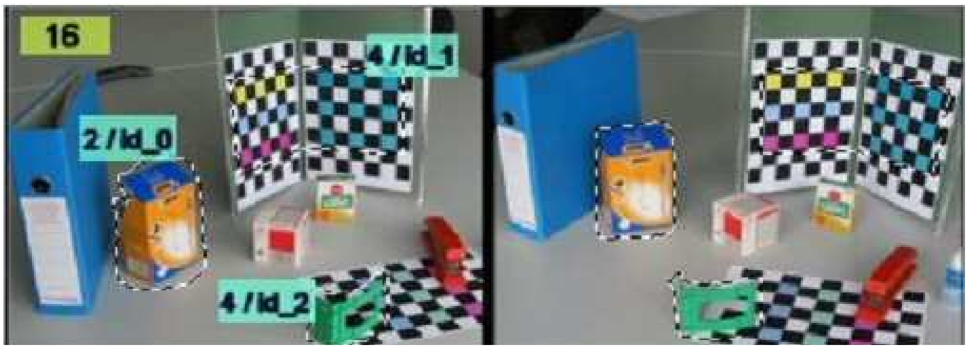


Figure 4.3 a. An example showing two images (out of a sequence of seven) with the regions corresponding to a box, a checkerboard area and a plastic object selected and labeled with the final resolution (*r*=6). The resolution of the initial reconstruction is the upper left corner (*r*=16).

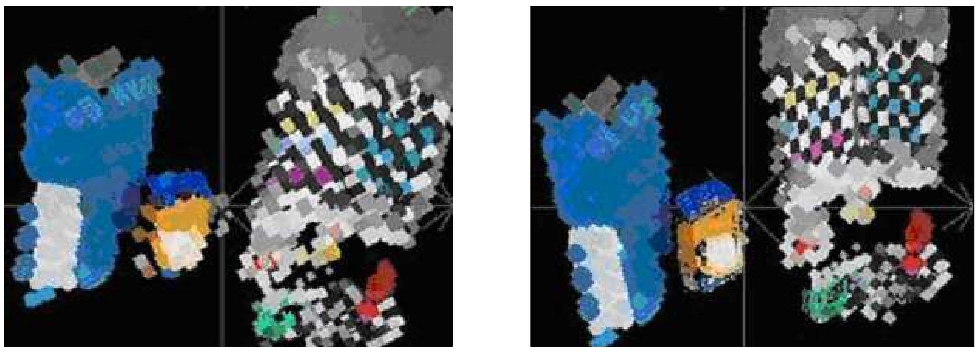


Figure 4.3 b. Left: The coarse 3D reconstruction of the scene at r=16. Right: The multi-resolution reconstruction with the selected regions refined to r=6, respectively.

- **User Input**

The user selects polygonal regions (e.g. corresponding to salient features) in one or more images using common selection tools, such as polylines and scissors, and assigns them a label ID corresponding to the chosen resolution (Figure 4.3a). Adjacent polygons must intersect along a set of common edges.

- **Visibility**

A correct visibility handling is required to compute photo-consistency. As shown in Figure 4.4, a voxel that does model a scene surface could erroneously be declared inconsistent if visibility is not taken into consideration. The voxel is not visible to the rightmost camera, which observes a blue color resulting from occluding geometry in the scene. Only the viewpoints where a voxel is visible should be taken into account during the photo-consistency check.

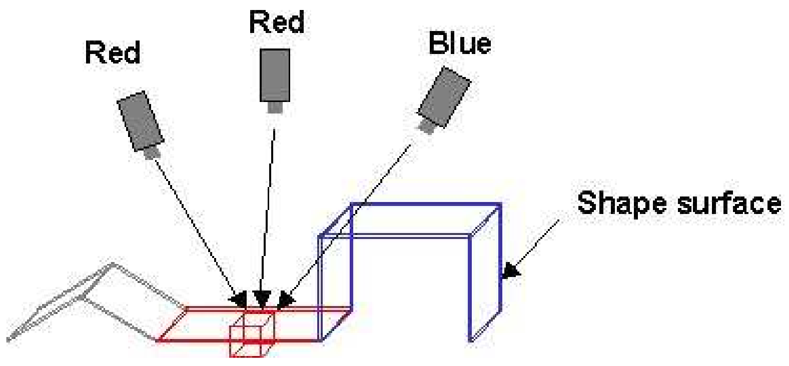


Figure 4.4 Correct visibility determination is required to compute photo-consistency

As mentioned previously, we operate only on surface voxels that are embedded in a *surface voxel list* (SVL). The SVL is initialized with the outside layer of voxels of the bounding box and is then updated at each iteration: carved voxels are removed from the SVL, while adjacent uncarved voxels which become visible are added to the SVL (Figure 4.5).

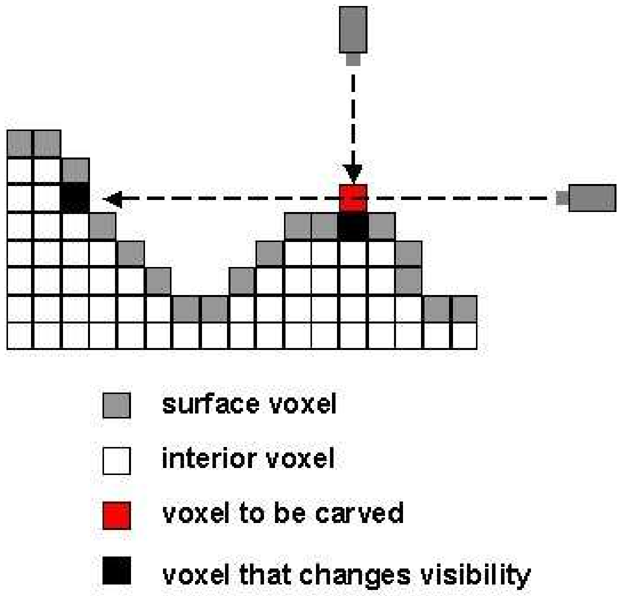


Figure 4.5 Voxels that change visibility [CMS99]

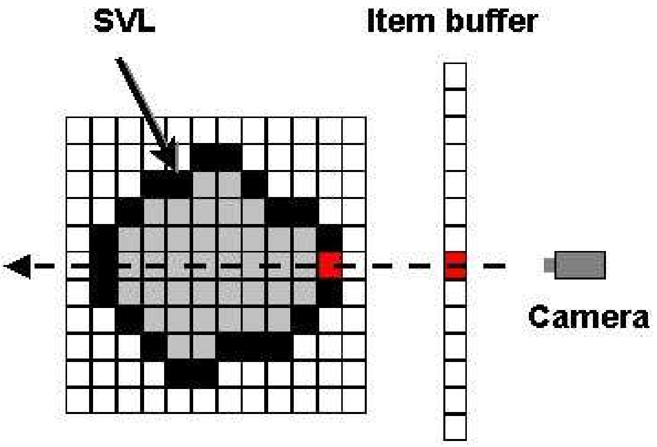


Figure 4.6 The item buffer records for each pixel the ID of the closest visible voxel that projects onto it [CMS99]

In order to determine the visibility set $vis(\mathbf{V})$ of pixels from which a voxel is visible, an image buffer is computed for each reference view as follows: the SVL is examined sequentially in order to find all the pixels that a voxel $\mathbf{V}$ projects onto and a comparison is done with the depth value already stored at the respective pixels. If $\mathbf{V}$ is closer to the camera than the distance previously recorded for the pixel, then its distance and ID are stored and included in the visibility set, while the previous pixel statistics are discarded.

If the new pixel's depth is greater than the current pixel, the voxel is not visible from that view and the voxel information will not be included in the visibility set (Figure 4.6).

The pseudo-code is outlined in Figure 4.7. The bidirectional voxel-pixel mapping enabled by the SVL and the IB allows the identification of voxels projecting to pixels belonging to labeled regions. First we perform a coarse reconstruction, in order to isolate and differentiate the voxel groups that project to labeled regions. The 3D bounds of each voxel group are computed and a spatial constraint grid is applied, which restricts further refining to labeled voxels. The resolution is increased by tesselating each voxel into eight subvoxels [ProDye98]. Next, voxel carving is performed on the higher resolution voxels. The above steps are repeated iteratively until the required resolution is obtained (Figure 4.7). The algorithm stops when every voxel is found to be color-consistent and no carving occurs.

- **The Photo-consistency Criterion**

The 3D shape of the scene is constructed by carving voxels that are not photo-consistent with the reference views. According to the photo-consistency definitions, in order to be photo-consistent (Figure 4.8), a voxel must not project to background in any reference view, and has to be color-consistent. Although the use of other reflectance models [Chhabra01] is possible, we will assume here that the scene is or nearly is Lambertian.

More specifically, the color-consistency check is done by computing a dissimilarity metric σ of the color components $c_1,...,c_\mathrm{k}$ of the pixels from the set $vis(\mathbf{V})$. The voxel is consistent if $\sigma < \tau$, where τ is a predefined threshold. Voxels found to be consistent are assigned the mean value of the color components, while inconsistent voxels are carved. The photo-consistency metric we have chosen is the true color variance value of corresponding visible voxels in reference views.

```
int $N_{LR}$; //number of labeled regions

int[] resolutions //array of the user-required resolutions

//compute initial low resolution reconstruction
initialize $SVL_{lores}$

loop{
     until no further voxels are carved
          for all images $i \ldots n$
               compute image buffer

          for every voxel $V \in SVL$ {
          compute $Vis_V$
               determine label status $LR_V$
               record color statistics for $V$
          }
     perform photo-consistency check
     delete inconsistent voxels from $SVL_{lores}$
     add uncarved adjacent voxels to $SVL_{lores}$
    }
}

//identify voxels belonging to labeled regions and build
separate SVLs for each region

for  $(\text{int } i = 0$; $i$<resolutions.length; $i{+}{+}$) {
     while $current\_resolution_i < final\_resolution_i$ {
          increase resolution by voxel subdivision
          initialize $SVL^i_{hires}$

          //perform voxel carving at the higher
          resolution
          loop{
               until no further voxels are carved
                    for all images
                    compute image buffer
```

-continued on the next page

```
                for every voxel $V \in SVL^{i}_{hires}$ {
                     compute $Vis_V$
                     record color statistics for $V$
           }
          perform photo-consistency check
          delete inconsistent voxels from $SVL^{i}_{hires}$
          add uncarved adjacent voxels to $SVL^{i}_{hires}$
       }
    }
}
```

Figure 4.7 The multi-resolution voxel carving pseudo-code

The photo-consistency metric we have chosen is the true color variance value of corresponding visible voxels in reference views. We compute the variance σ^2 according to the following equation:

$$\sigma^2 = \left[\sum_{i=1}^{N} (R_i - R_m)^2 + \sum_{i=1}^{n} (G_i - G_m)^2 + \sum_{i=1}^{n} (B_i - B_m)^2 \right] / (N-1)$$

Where N is the number of those active views in which the voxel is visible, (R_i, B_i, G_i) is the sampled pixel color from the i-th view, and (R_m, G_m, B_m) is the mean color of the corresponding pixels in all N views. The photo-consistency can then be expressed as a threshold function:

$$photo-consistency = \begin{cases} 1, & \sigma^2 < \tau \\ 0, & \text{otherwise} \end{cases}$$

where τ is a user-defined threshold. In our current implementation, the variance computation is based on a single sample from each reference view. Therefore, calibration errors and image noise can introduce instabilities to the photo-consistency check process. Incorporating local neighborhood information will provide more robust results.

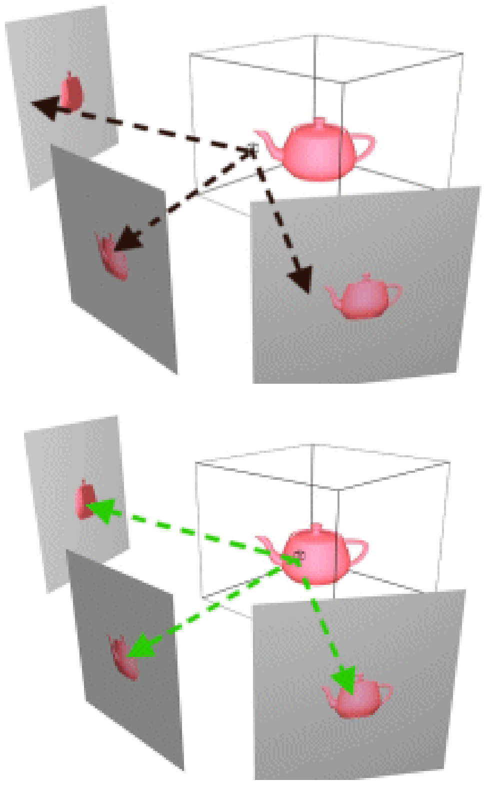


Figure 4.8 Top - the voxel projects in two views to background. Bottom - the voxel projects to the same color in all three views.

Conclusion

In this chapter, we have focused on the family of methods based on space carving and color consistency. We have presented the underlying theory that supports and guarantees a valid space carving algorithm. A multi-resolution voxel carving implementation has been described, that aims to reduce the computational cost of the voxel carving algorithm by enabling the human factor to assign various perceptual importance levels to surface regions of the reconstructed model.

Chapter 5 Programmable Graphics Hardware

Introduction

The classical programming model used in languages like C/C++ has been very successful for the development of non-parallel applications as it provides an efficient mapping to the classical von Neumann architecture. However, this model does not map very well to next generation parallel architectures. For developing efficient applications on such architectures with maximum efficiency, alternative programming paradigms are required. The stream programming model has shown to be a promising approach going in this direction. Furthermore, the stream programming model provided the foundations for the architecture of modern programmable high-performance graphics hardware.

The GPU, just like a CPU, has its own caches and registers to accelerate data access during computation and also its own main memory before beginning program execution. This memory hierarchy, however, is designed for accelerating graphics operations that fit into the stream programming model rather than the general, serial computational model. Moreover, graphics APIs such as OpenGL and Direct3D further limit the use of this graphics memory to graphics-specific primitives such as vertices, textures and frame buffers. This chapter gives an overview of the current memory model on GPUs and how stream-based computation fits into it.

5.1 The Stream Programming Model

The key to using the GPU for purposes other than real-time rendering is to view it as a streaming, data parallel processor. The Stream Programming Model (SPM) is of great importance to the way in which we structure computation and access memory on the GPU. As such, we will give an overview of this model in the following.

Streaming processors are programmed in a fundamentally different way than serial processors like today's CPUs [HHN*02, KDR*03, PAB*05]. In the stream programming model, applications are organized into *streams* and *kernels* (Figure 5.1). *Streams* are defined as ordered arrays of data, while *kernels* are small programs (or specialized functions calls) that perform operations on such streams, loading one or more streams as inputs and writing one or more streams as outputs. Applications are constructed by chaining multiple kernels together. The distinctive characteristic of the SPM – as opposed to the CPU general programming model – is that a kernel operates on *entire* streams and the same kernel is executed on each element of a stream in parallel.

The SPM constrains the way software is written such that parallelism and locality are explicit within a program, enabling compilers to optimize automatically the code to take advantage of the underlying hardware.

- **Parallelism**

Parallelism is ensured by two effective kernel independence constraints, allowing the underlying hardware to exploit parallelism both at task and data level. First, within a kernel, computations on one stream element are independent of computations on another element. Second, kernel outputs are functions of only their kernel inputs.

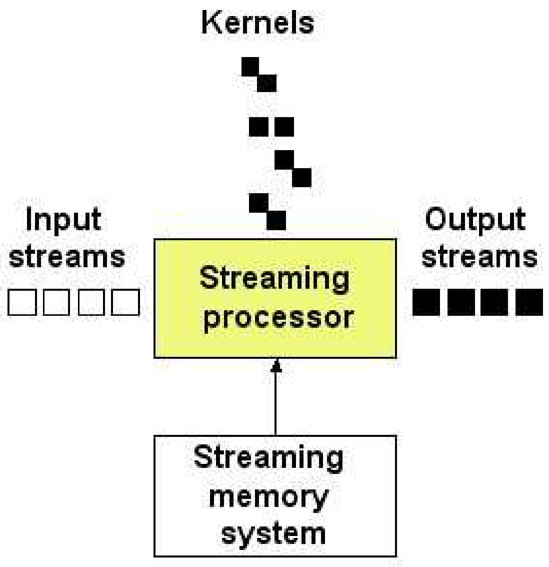


Figure 5.1.The stream programming model

Task-level parallelism is the ability to have multiple stream processors divide the work of a kernel, or to have different kernels run on different stream processors**.** Thus, the first processor in the pipeline executes one or more kernels to generate output stream that is passed to the next processor. As the next processor operates on those streams, the original processor repeats its kernels on the next set of input data. GPUs exploit task parallelism through *processor specialization*, i.e. by mapping kernels to separate processors placed on a single chip allowing efficient communication between kernels, by avoiding off-chip memory access.

Data-level parallelism: Since kernels perform the same instructions on each element of a stream, data-level parallelism can be exploited by performing these instructions on many stream elements at the same time. Moreover, due to kernel independence, every stream may be processed by a separate processing unit. Data parallelism is employed effectively by GPUs through the addition of more execution units.

Therefore, task- and data-level parallelism potentially allow the processing pipeline to be arbitrarily wide – in terms of the number of processors executing the same kernels across the data, or arbitrarily deep – in terms of the number of processors in the pipeline.

- **Locality**

There are two main types of locality exposed by the SPM: *kernel locality* and *producer-consumer locality*.

Kernel locality refers to the SPM constraint that intermediate values exist only temporarily and strictly within a kernel. *Producer-consumer locality* refers to streams produced by one kernel and consumed by subsequent kernels without going back to the main memory. This type of locality enables GPUs to fill regions of memory with contiguous data blocks, which is extremely useful when applied to one of the key tasks of the GPU: loading texture data in the memory.

5.2 The GPU as a Stream Processor

The stream processing model maps to a large number of different high performance processing models: multithreaded, pipelined SIMD, distributed and shared memory parallel architectures.

Furthermore, since graphics applications can be expressed as a series of computations performed on streams of data, the SPM stream and kernel paradigms naturally fit the graphics pipeline (Figure 5.2).

The computation involved in each stage of the graphics pipeline is uniform across data primitives, allowing these stages to be mapped to kernels. Similarly to the SPM, data flow between stages in the graphics pipeline is highly localized, with data generated by a stage immediately consumed by the next stage.

In the following we detail the major blocks of the modern programmable graphics pipeline, starting with input arriving from the CPU and finishing with pixels being drawn to the frame buffer (or render target).

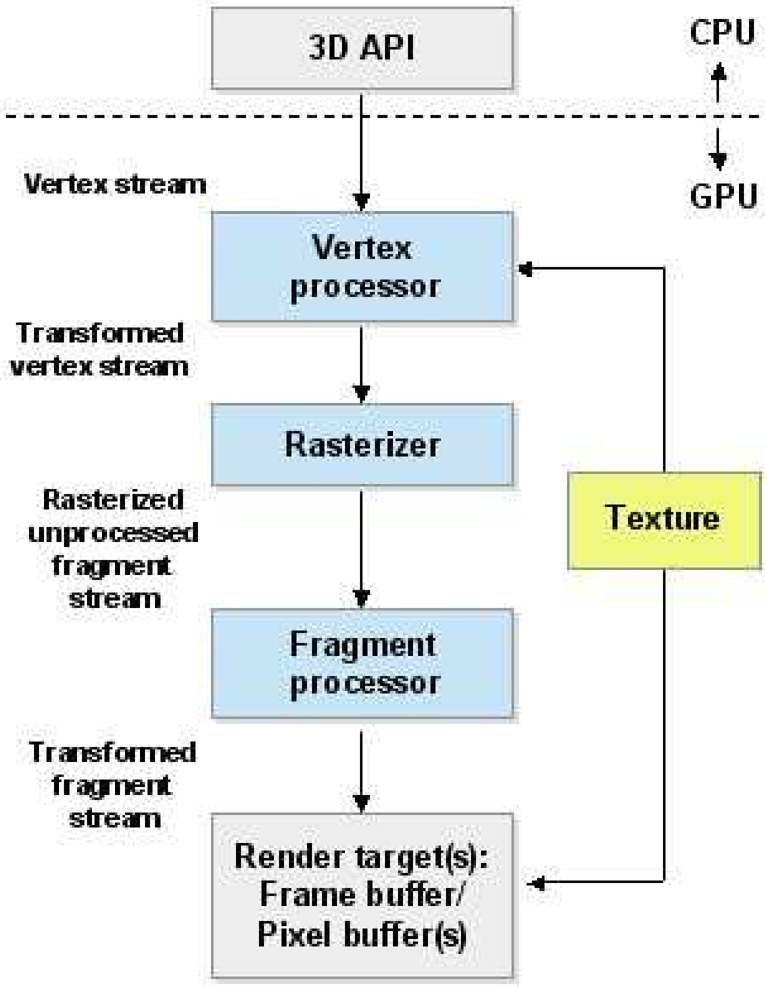

3D API
CPU
GPU
Vertex stream
Vertex processor
Transformed vertex stream
Rasterizer
Rasterized unprocessed fragment stream
Texture
Fragment processor
Transformed fragment stream
Render target(s): Frame buffer/ Pixel buffer(s)

Figure 5.2 The stream formulation of the graphics pipeline: data are expressed as streams (indicated by arrows) and computations are expressed as kernels (indicated by blue boxes).

The Vertex Processor

GPUs support multiple vertex processors that are fully programmable and operate in either SIMD or MIMD fashion on the input vertex stream referenced by the CPU rendering commands. The vertex processors apply a vertex program to *each* vertex in the object, transforming the vertices into a common model space and performing any other user-specified per-vertex operation.

Vertex processors are capable of *scatter*, i.e. they can control where in memory data will be written [LBC*05]. Thus, vertex processors are capable of changing the position of input vertices, deciding ultimately where the image pixels will be drawn. Traditionally, vertex processors were able to fetch information strictly from the current input stream and no other memory location, therefore they were not able to *gather*. However, the latest GPUs (i.e. with Shader Model 3.0 support), have a limited gather feature, called *vertex texture fetch* [GFG04], enabling vertex processors to perform texture memory reads for up to four textures.

GPU-writable streams are another recent hardware feature addition. Previously, vertex streams could be updated only on the CPU, requiring a bandwidth consuming GPU-CPU data transfer. This functionality, called "*render-to-vertex-array*" is of vital importance for GPGPU computations and will be detailed in a separate section (§ 5.6).

The Rasterizer

The transformed vertex stream produced by the vertex processor is converted by the rasterizer into *fragments*. Typically, three vertices groups are used to compute triangles (the triangle is the basic primitive for 3D representations), transformed then into fragments. From a GPGPU point of view, the rasterizer may act as a data amplifier, since it generates an increased output of elements from only a few input elements.

At this stage, each fragment can be considered a "proto-pixel" [Harris05] that encapsulates all information needed to generate a shaded pixel in the final image, including color, coordinates and possibly depth.

The Fragment Processor

Modern GPUs support a scalable number of fragment processors. Fragment processors are also fully programmable and work in SIMD fashion on input elements [LBC*05]. The fragment processors apply a *fragment program* (i.e. *pixel shader*) to each fragment in the stream to compute the final characteristics of each pixel.

Fragment processors have the ability to fetch data from textures, therefore they are capable of gather. They are however not capable of scatter (i.e. change the output location of a pixel) - since the output address of a fragment is determined prior to reaching the fragment processor.

GPUs have typically more fragment processors than vertex processors, in order to handle the amplified data load generated by the rasterizer. Consequently, GPGPU applications put a heavier emphasis on the usage of fragment processors compared to vertex processors.

Render Targets

Pixels are typically rendered on-screen to the frame-buffer or alternatively to an off-screen render target called a *pixel buffer* or a *framebuffer object*. Due to recent API features, the pixel buffer can consist of frame buffers, vertex buffers or textures. All of these three types of data can be associated with a texture (*render-to-texture*) enabling them to be bound either as render targets or input textures for further processing.

Therefore, while the traditional use of pixel streams is to hold pixels for display to the screen, GPGPU computations rather use these pixel buffers to hold the results of intermediate computational stages.

Texture Units

GPUs support a number of texture units, which determines how many textures may be simultaneously applied in the same render pass. Textures are bridging the random access into vertex, fragment, or pixel streams, since all these stream types need to be converted to textures to allow random indexing. Textures can be read from and written to by either the CPU or the GPU. On the GPU side, textures are the only memory that is randomly accessible by fragment programs and also vertex programs, though the latter refers to the limited access provided by the *vertex texture fetch* functionality.

5.3 GPU Computations

In order to activate kernels, we simply draw geometry so that the vertex and fragment processors will operate on the input primitives and output the result as pixels. Besides specialized computations on primitives as dictated by program requirements, the most generic invocation in GPGPU programming is a rectangle, typically processing every element of a stream of fragments representing a grid [Harris04].

The GPU equivalent of CPU array data structures consists of streams, therefore, anywhere we would use an array of data on the CPU, we can use one of these streams on the GPU.

On the CPU, in order to perform serial processing, we would use a loop to iterate over the elements of an array. In the GPU case, the instructions inside the loop are the kernel, while the streams replace the array structures. That is, on the GPU, we write similar instructions inside a fragment or vertex program, which are applied to all elements of the stream.

Conforming to the SPMs task-level parallelism, a kernel must process an entire stream to generate output for the next kernel in the pipeline, therefore, each step depends on the output of the previous step. The feedback needed to proceed with each step is trivial to implement on the CPU, where memory can be accessed anywhere in a program. As we discussed previously, on the GPU we need a texture to bridge our access, i.e. we must use the *render-to-texture* technique to write the results of a fragment program to memory so they can then be used as input to future programs. Texture coordinates are stored for each vertex and are used in GPGPU as indices for texture fetches (see next paragraph for more details).

5.4 Dependent Texturing

Modern graphics processors have the ability to perform dependent texture lookups. Purcell [PDC*03] has introduced the abstraction of texture memory that enables us to load a complex data structure into memory and use fragment programs to navigate through it via dependent texture lookups.

Algorithms may involve complex data structures and lookup of elements within these structures. Dependent texture fetching allows the address being fetched from texture memory to be computed by the fragment program. It also allows the results of a memory lookup to be used to compute another memory address (Figure 5.3).

More importantly, it allows us to think about texture memory on the GPU as general read-only memory. Rather than worry about texture management and texture coordinates, etc. we can think about memory and addresses.

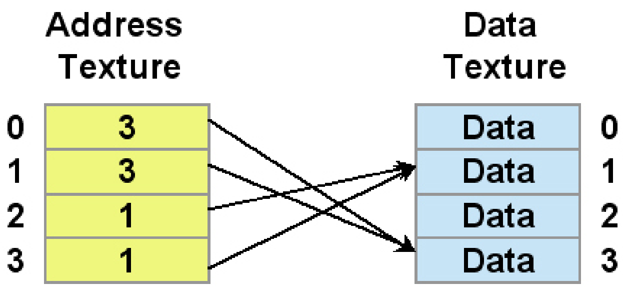


Figure 5.3 Dependent texturing [PDC*03]

5.5 Render-to-vertex-array

General processing on the GPU relies on computing intermediate stream results on the GPU, saving them in graphics memory (off-screen targets), and then feeding them again as vertex data textures to the geometry engine to render images in the frame buffer. This process requires application control over the allocation and use of the graphics memory.

Recent features of graphics hardware blur the boundaries between textures, vertex data and render targets, allowing graphics memory to be treated as any of such objects. These features are exposed to the application through a set of supporting OpenGL extensions: Vertex Buffer Objects and Pixel Buffer Objects [ARB03, NV04].

The Vertex Buffer Objects (VBO) interface enables the application to allocate high-performance graphics memory on the GPU, and to specify how that memory is to be used. Buffer objects can be used as either data sources or sinks for any graphics API command that takes a pointer as an argument. The VBO extension binds buffer objects to given targets and then instead of treating the argument as a pointer to memory, it is used as an offset into the buffer object's data store.

Previously to the PBO addition, the VBO targets consisted of buffers containing either vertex attributes, such as vertex coordinates, texture coordinates data, per vertex-color data, and normals, or only indices of elements, respectively. The recent Pixel Buffer Objects extension (PBO) expands the VBO functionality with two new read/write targets, permitting buffer objects to be used not only with vertex array data, but also with pixel data.

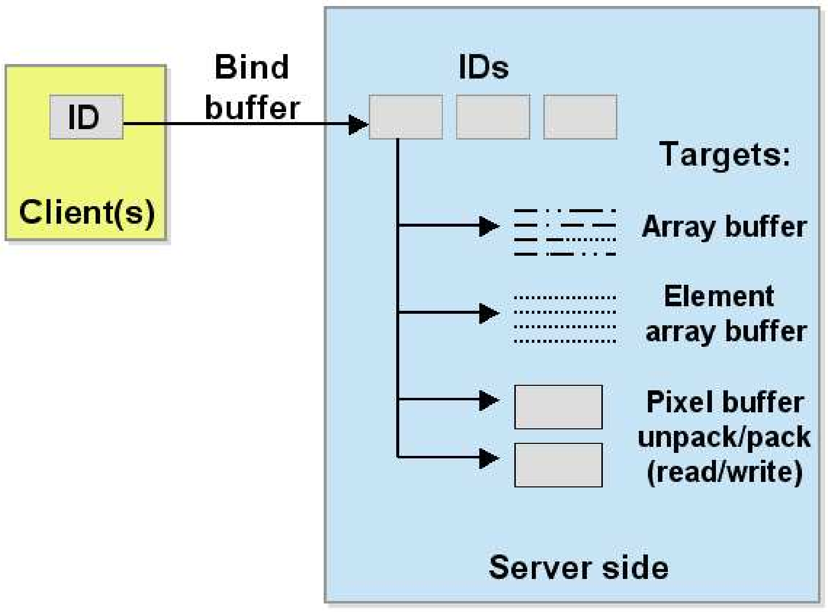


Figure 5.4 VBO targets [Nvidia03]

"Render-to-vertex-array" is one of the most interesting optimizations provided by the VBO/ PBO combination. Buffer objects are viewed at application level simply as arrays of bytes, differentiated only by the targets they are bound to. Therefore, a vertex buffer can be bound to a pixel buffer target and then use this buffer as a source of pixel data and vice versa (Figure 5.4). From a GPGPU point of view, the VBO/PBO combination enable GPUs, for the first time to loop streams results from the end to the top of the pipeline (Figure 5.5).

Moreover, VBO/PBO avoids the GPU-CPU bandwidth taxing transfer, which has traditionally been a bottleneck for many applications: the VBO/PBO mechanism keeps all the data flow inside the server, avoiding copying the pixel buffer on the client's side and putting it back on the server's side as an input for a vertex program.

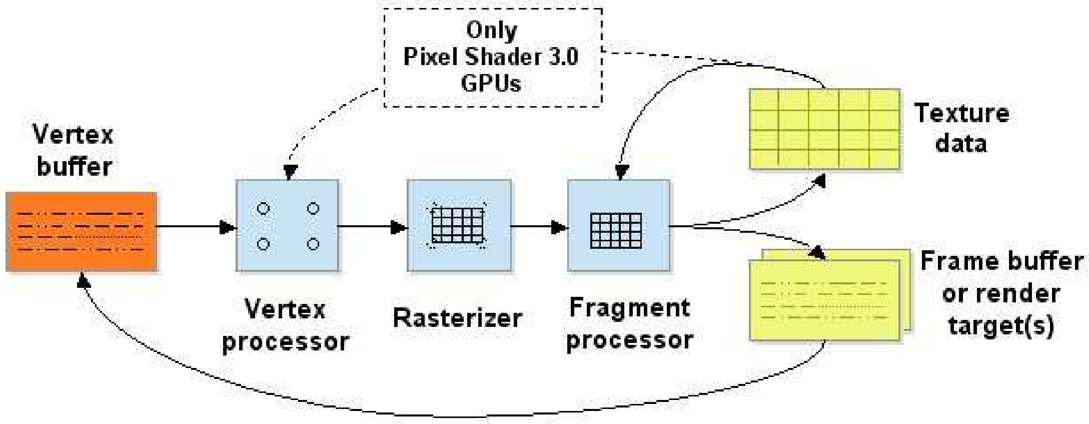


Figure 5.5 Render-to-Vertex-Array: writing rendering results to vertex array allows the GPU to loop back to the top of the pipeline [LBC*05]

5.6 Shading Languages

Originally, GPUs could only be programmed using assembly languages. Early work at Stanford University [PMT*01] provided valuable research, producing the Real Time Shading Language (RTSL). RTSL provided an abstraction layer over the (at the time) fixed-function hardware that would compile to GPU assembly, reducing the difficulty of GPU programming.

GPU and 3D API vendors soon followed suit, releasing their own languages and compilers. Nvidia was the first to expose a more general programming model for GPUs, starting with the vertex unit [LKM01]. The fragment unit was also made gradually programmable, having eventually a full programming model very similar to that initially supported on the vertex unit.

Varying degrees of programmability had been appearing in OpenGL as vendor-specific extensions. These finally converged in the `ARB_vertex_program` and `ARB_fragment_program` extensions [ARBa]. The assembly-level shaders were naturally followed by higher level shading languages, having a C-like syntax with minor differences. Nvidia and Microsoft worked closely to develop Cg and HLSL [MGA*03, BFH*04]. After an almost two year refinement by the the OpenGL Architecture review board, the OpenGL Shading Language, was released as part of OpenGL 2.0 core at SIGGRAPH 2004.

All the above languages require thorough understanding of the features and limitations of latest graphics hardware and graphics APIs, making the general-purpose GPU computing accessible to only the most advanced graphics developers. Two general purpose shading languages have been developed, trying to overcome these drawbacks by hiding the details of the runtime: Brook at Stanford University [BFR*04] and Sh at Waterloo University [McD04].

Following the development of the Imagine stream processor [KDR*02] Brook exploits the stream aspect of GPUs explicitly. Brook targets scientific applications, comes with an associated compiler (brcc) and is implemented as a preprocessor that maps programs to a C++ and Cg implementation.

Sh is also supporting the stream programming model, however it targets not only scientific but also graphics computation. Sh is embedded inside C++, so that no compile tools are necessary and parameter passing into streams is seamless eliminating the need for parameter binding code.

At the time of writing both Sh and Brook are under heavy development. Brook has performances close to hand-written code [BFR*04], however many special GPU features cannot be expressed cleanly and it lags in the timely updates.

Sh lacks certain features that affect its efficiency and are intended to be added in the future. For example, downloading the data between the host program and the processor running the Sh kernel can be a very expensive operation. [MD04]

Therefore, low-level shaders are still needed to extract the best performance from the GPU. RTSL is no longer active and under development, having evolved into Cg practically, HLSL is DirectX specific and GLSL is OpenGL specific.

The Cg language is multiplatform, API neutral and independent of the generation of GPU that it is running on. Moreover, it has the important benefit of constant updates according to the rapidly evolving GPU technologies. These features have made Cg the language of choice for the GPU shaders developed for our work.

Conclusion

In this chapter, we have touched concepts of modern programmable graphics hardware and the main differences between the CPU and GPU memory and programming model. The GPU memory model is based on a streaming computational model that supports a high degree of parallelism and memory locality. This implies a number of restrictions on when, how and where memory can be used. Many of these restrictions exist to guarantee parallelism, but some exist because GPUs are designed and optimized for real-time rendering rather than general high-performance computing. Nonetheless, many of these constraints are likely to be relaxed in the future. We have also reviewed several recent hardware features that broaden the use of GPUs within the general processing realm.

Chapter 6 Voxel Carving on the GPU

Introduction

The primary goal in developing the carving engine is to design a method that allows voxel carving to occur at real-time rates. GPUs power real-time systems with a peak performance about two orders of magnitude greater than that of the CPU, however this performance implies the constraints of the streaming programming model.

One major challenge in developing GPGPU algorithms is to design appropriate data representations and develop techniques that fully utilize the graphics pipeline, multiple vertex and fragment processors and high inner memory bandwidth.

Another issue to overcome is the GPU-CPU transfer rate, the traditional bottleneck for many applications – due to the asymmetric interconnecting AGP bus that delivers performant bandwidth only from the CPU to the GPU. The advent of the new bidirectional-transfer capable PCI Express bus standard may make sharing memory between the CPU and GPU a more feasible possibility in the future. Nevertheless, attaining low bandwidth consumption is of great importance.

Our approach is designed for a high utilization of the graphics pipeline parallelization and employs an efficient external and inner bandwidth strategy. More specifically, the output format is designed to return as little data as necessary, limiting itself to surface voxels rather than the entire volume, while the input format corresponds to the optimal two-dimensional data layout on the GPU.

If carving progresses at more than 20 frames per second, we claim that real-time performance is achieved. Interactive frame rates are considered to be between 2 and 20 frames per second. Depending on the voxel resolution and the number of reference views, the algorithm that we will present runs at least interactively, and in several cases achieves real-time frame rates. Compared to a purely CPU-based implementation, the performance is approximately 3-8 times faster.

6.1 The Carving Engine

This section outlines the encapsulation of the voxel carving process into a carving engine. We structured the carving engine to perform image sampling, background/silhouette test, visibility test and photo-consistency check on the GPU and let the CPU host manage the resulting output and organize the dynamic update of the voxel structure. Since the bulk of the computational resources are spent on the former operations, the management of results and of voxel state is a relatively small overhead for the CPU, certainly smaller than performing the entire voxel carving on the CPU.

We determine the original size of the volume of voxels containing the 3D scene by upscaling the spatial bounds of the recovered 3D points during self-calibration. Voxels are assigned a unique ID and only voxels that belong to the surface are processed. We are considering a point voxel projection, i.e. only the voxel centre is projected to the input images, leading to a single pixel in each view.

The pseudocode for the algorithm presented in Figure 6.1. reveals that the algorithm is compliant with the generic space carving method. Given a set of voxels in the surface, each voxel is examined and tested for consistency. During each iteration of the algorithm, we perform an outer loop where the surface voxel structure is rasterized against each camera view, followed by voxel sorting, as a precalculating step towards determining $vis(\mathbf{V})$ of each voxel. This results in a set of image buffers for each of the views for every loop iteration. The calculation of $vis(\mathbf{V})$ is embedded within the photo-consistency check. Once $vis(\mathbf{V})$ has been determined, the color consistency function computes the color statistics and decides the consistency status of the voxel. If it is consistent it will be kept and re-examined in further iterations until it is rejected or it remains as part of the reconstructed object. Following the conservative approach of this family of methods, in case of uncertainty, the voxel is left unprocessed, expecting that later on its situation will become better defined.

The employed data representations, kernels and computational stages will be detailed in the following paragraphs.

6.2 Memory Layout

The voxel structure and its attributes correspond to 1D and 3D data arrays. However, we will pack this data in two-dimensional textures, which is the optimal layout to utilize the high memory bandwidth available in GPUs. That is, GPUs provide only 2D rasterization and 2D frame buffers, meaning the bidimensionality of such textures ensures a maximum efficiency update during processing.

Graphics processors currently offer only scarce support for simple 1D texturing. Many data structures will overflow the maximum size of a 1D texture, since current GPUs do not support textures with more than 4096 elements. However, the use of 2D textures requires address translation to convert an *n*-D array address into a 2D texture address, similar to a virtual to physical memory translation.

That is, each time this packed array is accessed from a fragment or vertex program, the 1D address must be converted to a 2D texture coordinate. It is important to note that these conversions are performed very efficiently, because the GPU's texture-addressing hardware actually minimizes the cost of address translations to look up values in the underlying 1D array

```
loop {
  until no further voxels are carved {
      update voxel state map and generate SVL
          for each voxel {
              delete rejected voxels from SVL
              add adjacent uncarved voxels
              mark modified voxel state
          }
       enable depth test
       transfer voxel state map to GPU
       transfer SVL map to GPU

       for all images $i$ … $n$ {
            //projective texture mapping
            bind image buffer $IB_i$ as the target
            bind camera image $C_i$ as input texture
            render SVL to image buffer
                 load camera parameters
                 run projective texturing kernels
                     transform object-space coordinates
                     to texture coordinates $TEX_{xy}$ (VP);
                    query $C_i$ at $TEX_{xy}$ and draw vertex
                     with the found color (FP);

            //sort by pixel routing
            set viewport to routing buffer dimensions
            transfer image buffer to vertex array
            bind routing buffer as the target
            render image buffer to routing buffer
                 run pixel routing kernels
                     compute new vertex address(VP)
                     draw vertex at the new position(FP)
       }
       //perform photo-consistency check
       bind routing buffer as input texture
       bind photo-consistency buffer as the target
       run photo-consistency kernel
            determine Vis(V)
            compute color statics for V
            output photo-consistency status
```

-continued on the next page

```
        read-back to CPU photo-consistency buffer
        //display consistent voxels
        render SVL to frame-buffer
            set photo-consistency buffer as the color
            attribute
            discard inconsistent/background voxels
    }
}
```

Figure 6.1 Carving engine pseudo-code

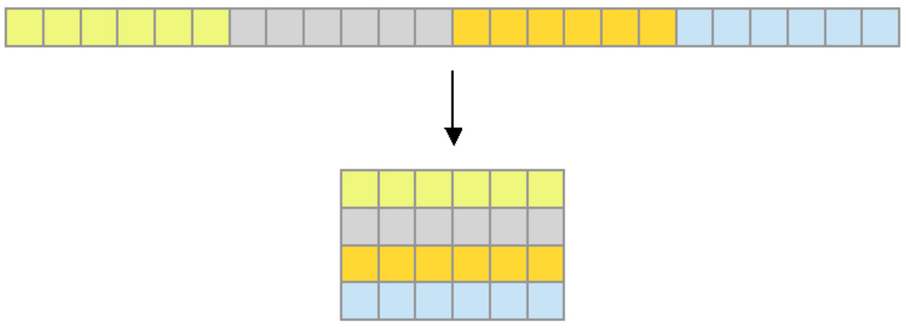


Figure 6.2 1D array packed into a 2D texture

3D texture maps are the easiest way to store 3D arrays, however, they present several drawbacks. 3D textures tend to take up a large amount of texture memory, and they grow rapidly in size with increases in resolution. For example, the memory cost of a 32-bit 256^3 texture is 64 MB representing a considerable burden on most current graphics systems. As a consequence, 3D textures are expensive to change dynamically which can affect multipass algorithms requiring multiple passes with different textures, as in our specific case.

Alternatively, each volume slice can be stored separately in a 2D texture [HCT*02], or the entire volume can be packed in a single 2D texture [HBS*03]. We have employed the latter method, which unlike the 2D slice layout, allows the entire array to be updated in a single render pass and eliminates the need of a "render to slice" functionality. This may allow a significant performance improvement, since it implies processing large streams that use more efficiently the GPU parallelism. Also, such “flat 3D textures” provide a performance and scalability advantage over true 3D textures on current hardware [HBS*03].

Moreover, the entire 3D array can be randomly accessed from within a kernel. The procedure is identical to the one used with 1D arrays, with the 3D address being converted to a large 1D address space, previous to packing the 1D space into a 2D texture [BuckPur04].

During address conversions and look-ups, precision issues need to be treated carefully: current GPUs do not have integer data types, therefore we have to avoid poor address calculations caused by the limitations of floating-point addressing. Additionally, the number of bits dedicated to floating point mantissa that limits the size of our 1D virtual address space varies from architecture to architecture.

	VBO/PBO state			**CPU data**	**GPU data**
	Render target	**Input texture**	**Vertex array**		
1	-	Yes	-	3D array	Voxel state map
2	-	-	Yes	1D array	SVL map
3	Yes	-	Yes	-	Image buffer
4	Yes	Yes	-	-	Routing map
5	Yes	-	Yes (color attribute)	1D array	Photo-consistency map

Table 6.1 Data layout for the carving engine

Table 6.1 shows the data storage layout we have employed for the carving engine. As mentioned above, the voxel attributes and processing data are stored in 32-bit floating point textures. Conforming to the stream programming model (chapter 5, § 5.2), textures represent either the input or the output data stream. Several of these textures will be treated alternatively as render targets, input textures or vertex arrays via the VBO/PBO interface.

In the following, we detail the memory objects employed by the carving engine.

- **Voxel state map**

The discretized voxel cube corresponds to a 3D array of voxel coordinates which is stored in a three-component floating point texture reflecting the state of each voxel and containing its unique voxel ID.

Due to the serial nature of this process, the voxel state map is updated on the CPU. The voxel state map will be bound as an input texture and will be fetched by most of the kernels during processing, for voxel ID and position/voxel ID conversions. However, for readability reasons it is not represented in Figure 6.1.

During the carving process, each voxel can be found in one of the following three states:

- active: voxel has been added to the surface set of voxels and was found consistent at each evaluation
- undefined: it is surrounded by uncarved voxels, so it is visible from no images and its consistency is undefined (wasn't added to the surface voxel set)
- carved: it has been found to be inconsistent and has been carved.

- **Surface Voxel List map (SVL map)**

The SVL map stores the XYZ position coordinates of all currently active voxels and will be found in a single VBO/PBO state on the GPU, namely as a vertex array that will be used to replicate each camera view. This occurs by rasterizing the SVL to a pixel buffer and applying projective texturing with corresponding camera-based rendering parameters. Similarly to the voxel state texture, the SVL map will be updated on the CPU, due to the serial nature of this process.

- **Image buffer**

The image buffer is a pixel buffer with the same dimensions as the reference images bound as a render target for the SVL texture. As mentioned above, the SVL texture is rendered as a vertex array for each reference view by a kernel that loads the current camera parameters and performs projective texturing to sample the respective view. The image buffer will eventually store voxels that survive a visibility, i.e. a depth test. During the next processing step – pixel routing – the image buffer will undergo a VBO/PBO transfer to a vertex array and will be rendered to the routing map.

- **Routing map**

The routing map represents a render target for the voxel sorting rendering pass**.** Sorting is performed by a pixel routing kernel that outputs fragments to the 4-component routing map in a tightly packed format, according to their ID and camera view. The routing map will be bound as an input texture during the photo-consistency check process.

- **Photo-consistency map**

The photo-consistency map forms a render target for a kernel that computes the consistency status of each voxel on the current SVL, the mean average color for consistent voxels and a marking value for voxels that have been found inconsistent or belonging to the background. While the currently consistent voxels are displayed on-screen, the photo-consistency map will be employed as a color attribute for the corresponding vertex array.

6.3 Computational Stages and Kernels

The vertex and fragment processors run computational kernels producing output for all rendered pixels to the currently active memory surface of the render target. Among the techniques used by the carving engine are the following:

- Bind two-dimensional textures, forming the input for the kernel.
- Set the target surface for rendering. This surface forms the output of the kernel.
- Activate a vertex or fragment program, i.e. set up the vertex or fragment pipeline to perform the kernel computation on every vertex or fragment, respectively.

We have implemented our carving engine on a GPU supporting the Shader Model 3.0 standard that supports conditional branching and looping, allowing for more flexible kernels.

As mentioned previously, the lack of integer operations needs to be treated carefully. The carving engine has to access specific texture addresses, and we need to compensate the floating point arithmetic units whenever integer data types are needed (all of our data structures use integer addresses) That is, we need to simulate integer operations with floating point operations.

We can compute most integer operations by taking the floor of the result of a floating point operation. Integer modulus operations, however, require a few more operations including *frac* which returns the fractional part of a floating point number:

$$X \bmod Y = \mathrm{floor}(\mathrm{frac}(X/Y) * Y)$$

In the following, we will describe the three main computational blocks (Figure 6.3) and their corresponding kernels, where this applies.

6.3.1 Process Voxel Birth and Death

We have employed a particle system paradigm to describe the computational stage of updating voxel state, i.e. activating and carving voxels in order to generate the surface structure.

As mentioned previously, each voxel can be found in one of three states during the carving process: active, undefined and carved. In order to activate a voxel we need to associate new data with an available index in the voxel state texture.

Due to the serial nature of our problem, this cannot be done efficiently with a data-parallel algorithm on the GPU. Therefore, the voxel emitter module, responsible for determining an available index, is placed on the CPU.

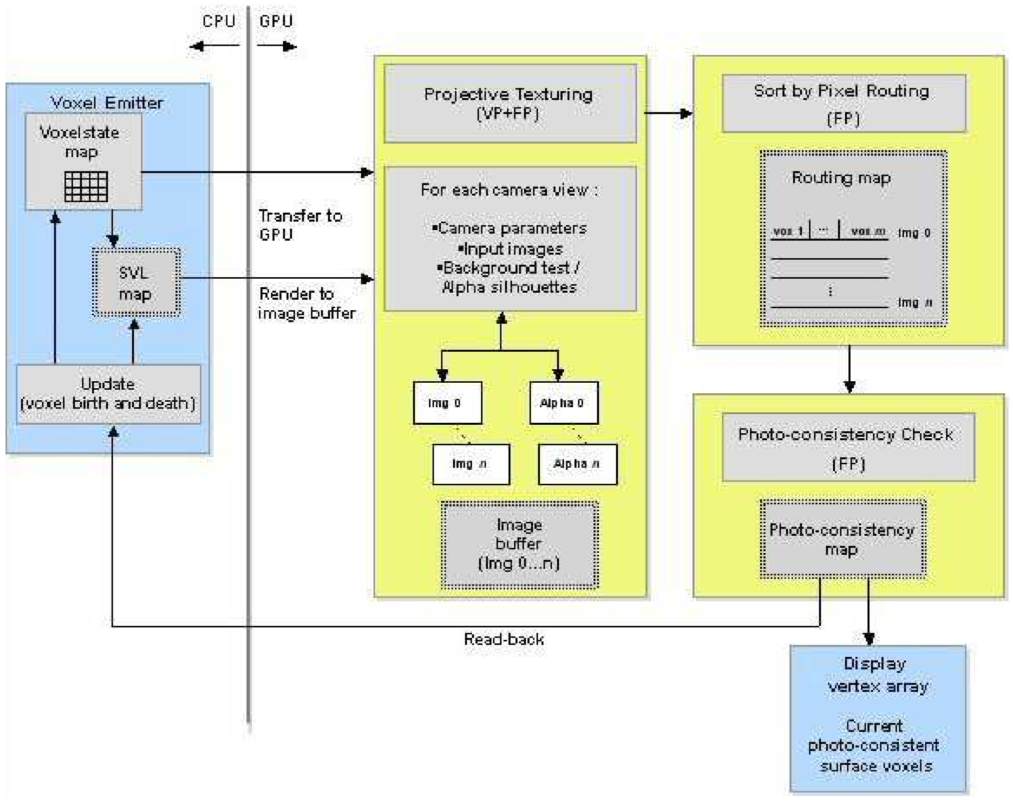


Figure 6.3 Carving engine computational stages

We initiate the SVL with the outer layer of the discretized voxel volume, and the activated voxels are marked on the voxel state map. In order to perform a voxel state update, we read-back the photo-consistency buffer, containing the information of the current SVL and perform the necessary modifications on the voxel state map, as well as add/delete operations on the SVL array. More specifically, carved voxels are deleted from the SVL, while their adjacent voxels are activated and added to the SVL.

Voxels are registered for deactivation independently on the CPU and GPU:
The CPU registers the deactivation of a voxel and adds the freed index to the allocator, while the GPU discards deactivated voxels with an early z-kill during rendering the current SVL to the on-screen framebuffer.

6.3.2 Projective Texturing

During projective texture mapping we render the SVL vertex array to the image pixel buffer, for each input view. A vertex and a fragment program are needed to perform projective texturing. Camera reference images are loaded as textures, and their corresponding camera matrix is set as the projective texture matrix.

The vertex program works by applying a sequence of transformations, that map object-space coordinates into the 2D space of a texture, i.e. the loaded camera image. This computed position is assigned as the texture coordinate for the vertex, and then the appropriate sampled color from the texture is applied by the fragment program. In order to account for voxel visibility, we enable depth testing in the supporting OpenGL API. The built-in *z*-test is used so that the voxels will overwrite the value stored in the *z*-buffer if the new value is smaller, i.e. they are closer to the camera.

Projective texturing also serves as a background or silhouette test step, performed in the fragment program. For background testing, the alpha value assigned to the output fragments is set to 1 for foreground objects and 0 for the background.

In case we are employing silhouettes, the procedure is identical to the one described above: the silhouette images are loaded as textures and their corresponding texture matrix is set from the calibration data associated with that view. Similar to background testing, the alpha value of the texture is set to 1 for foreground objects and 0 for the background.

These values will be considered during the photo-consistency check, as pixels that don’t survive the background/alpha test will be eliminated.

6.3.3 Sort by Pixel Routing

In order to perform a coherent photo-consistency check, we need to sort the voxels contained in the camera pixel buffer and arrange them in a tightly packed texture (Figure 6.4), according to their identifier and camera view.

Several authors have proposed implementations of sorting algorithms on graphic processors [PDC*03, KipWes05]. However, sorting algorithms require a high number of iterations, resulting in a high number of rendering passes on the GPU. For example, bitonic merge sort needs O*(log2 n)* rendering passes and *O(n log2 n)* bandwidth [PDC*03].

Since we strive to achieve real-time framerates, we need to avoid the latency of several hundred rendering passes when generating the photo-consistency map. We would also prefer an algorithm with less bandwidth consumption. To address these problems, we have employed an alternate algorithm for constructing a photo-consistency map that runs in a single pass and only requires *O(n)* bandwidth.

Although fragment programs cannot change the address to which they are writing, vertex programs have the ability to write to a computed destination address, i.e. to perform scatter (Chapter 5, §5.2).

That is, if we know the exact destination address for each voxel, we could route them all into the buffer in a single rendering pass by drawing each of them as a point. Essentially, drawing points allows us to solve a one-to-one routing problem in a single rendering pass.

While we render the image buffer as a vertex array, the application issues points (glPoint for the OpenGL API) to render and the vertex program computes the scatter address based on the voxelID and assigns it to the point's destination address with the appropriate scatter data.

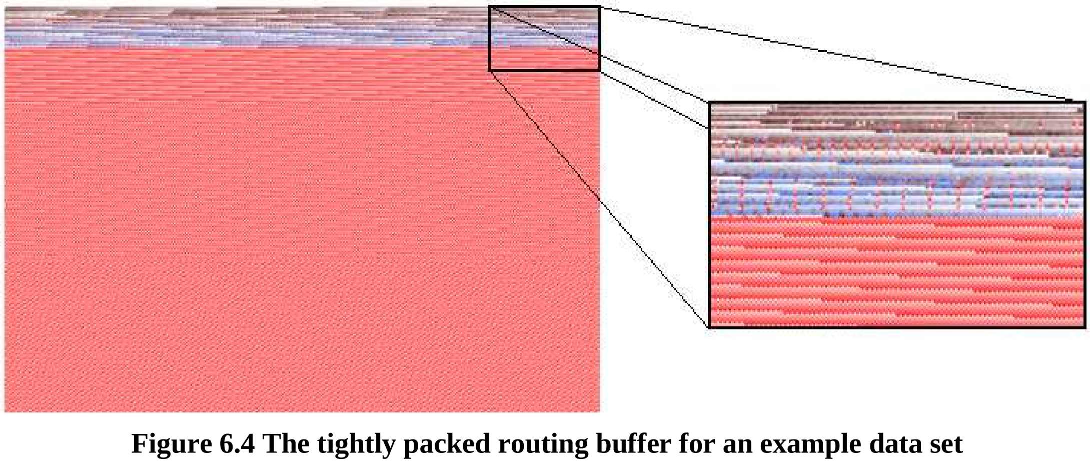


Figure 6.4 The tightly packed routing buffer for an example data set (empty pointers are shown in red)

In the OpenGL API we have adopted, we set the viewport to the buffer's rectangular dimensions and disable depth testing. We generally use a routing buffer with the same dimensions as the image buffers, in order to ensure the necessary number of available positions. The idea is to draw each voxel (i.e. vertex) as a glPoint over the entire footprint of its destination cell, so we draw with glPointSize set to 1 which when transformed by the vertex program will cause the voxel to cover the grid cell. The vertex program computes the new vertex address based on the routing texture width and height, pixel-texel ratio, SVL index and size, and also camera view index. The depth component of the output fragments is set uniformly to 0, in order to avoid collisions.

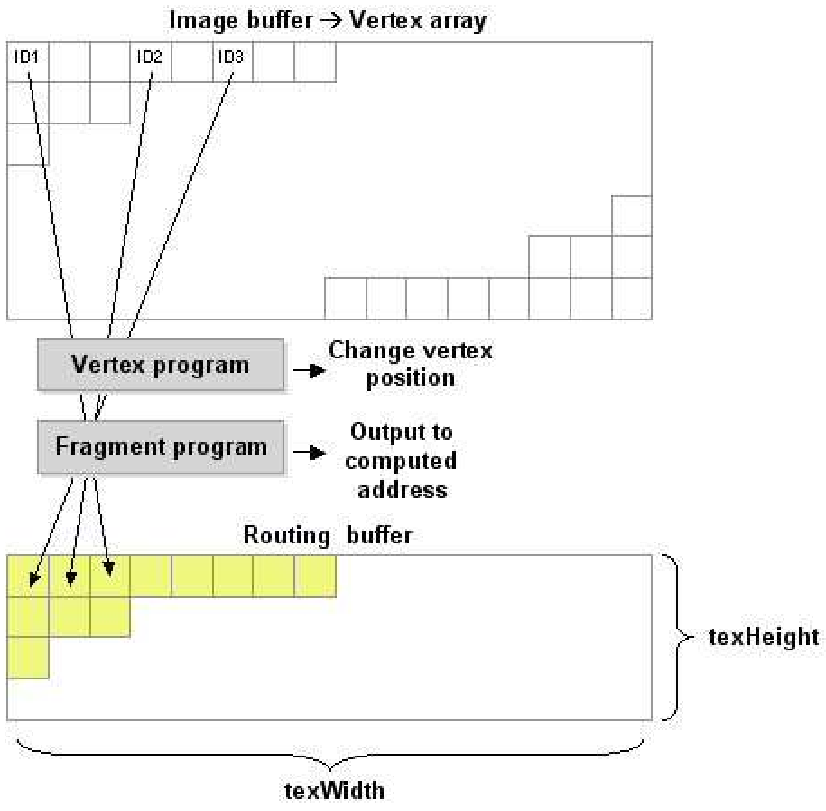


Figure 6.5 Pixel routing

6.3.4 Photo-consistency Check

We perform the photo-consistency check in a single rendering pass. The routing pixel buffer containing the sorted voxels is fetched by a fragment program that computes the variance of corresponding visible pixel samples in reference views, which we chose as the photo-consistency metric.

We mentioned the photo-consistency computation previously in chapter 4, however for readability reasons we will present it here in the context of GPU-based processing.

The fragment program computes the variance σ^2 according to the following equation:

$$\sigma^2 = \left[\sum_{i=1}^{N} (R_i - R_m)^2 + \sum_{i=1}^{n} (G_i - G_m)^2 + \sum_{i=1}^{n} (B_i - B_m)^2 \right] / (N-1)$$

where N is the number of those active views in which the 3D point associated with the fragment is visible , (R_i, B_i, G_i) is the sampled pixel color from the i-th view, and (R_m, G_m, B_m) is the mean color of the corresponding pixels in all N views.

The photo-consistency can then be expressed as a threshold function:

$$photo - consistency = \begin{cases} 1, & \sigma^2 < \tau \\ 0, & \text{otherwise} \end{cases}$$

where τ is a user-defined threshold. In our current implementation, the variance computation is based on a single sample from each reference view. Therefore, calibration errors and image noise can introduce instabilities to the photo-consistency check process. Just like in the CPU-based case, incorporating local neighborhood information will provide more robust reconstruction results. The mipmapping technique utilized in [YanPol03] could be adopted in this context.

Finally, if a fragment passes both the background/silhouette (performed during projective texturing) and the photo- consistency check, color values are assigned to the fragment by computing the mean average of the sampled colors.

Table 6.2 illustrates the instructions count of the principal computational kernels.

Process	**Kernel**	**Number of instructions**
Projective texturing	Vertex program	72
	Fragment program	36
Photo-consistency check	Fragment program	148
Pixel routing	Vertex program	25
	Fragment program	10

Table 6.2 Instruction count for the main kernels of the voxel carving engine

6.3.5 Display Consistent Voxels

Optionally, the SVL can be rendered to the display during processing. In order to reduce the workload of the fragment unit, voxels are rendered as point sprites. The photo-consistency buffer will be set as a color attribute for the vertex array, and a fragment program will discard fragments corresponding to voxels that were marked for rejection.

Conclusion

We presented in this chapter the carving engine, a GPU-based algorithm that extracts a voxelized representation of a scene from a set of images depicting that scene. The bandwidth efficient carving engine produces an explicit volume at frame-rates ranging from interactive to real-time.

Our approach employs a form of effective load balancing that allows the GPU to do what it does best (perform the same computation on arrays of data), and lets the CPU do what the GPU does worst (reorganize the data into efficient structures). By partitioning computation between the CPU and GPU, we combined the optimal features of both.

Chapter 7 Experimental Results

Introduction

In the following we will present results obtained by the system described in the previous chapters. Several results on self-calibration from photographs are given, including the Levenberg-Marquardt refinement of the initial estimates of the 3D Euclidean structure and camera motions. We present voxel carving results, both in CPU and GPU context, with a focus on performance. The flexibility of our approach is shown by reconstructing a 3D model from an extended sequence of camera views.

7.1 Self-calibration

7.1.1 Conditioning and Balancing

The scaled measurement matrix W_s (chapter3, §3.3.1, Equation 3.11) is poorly conditioned, mainly because of the lack of homogeneity in the image coordinates. To ensure good numerical conditioning, we work with normalized image coordinates, as described in [Hartley95]. This normalization consists of applying a similarity transformation (translation and uniform scaling) T_i to each image, so that the transformed points are centred at the origin and the mean distance from the origin is $\sqrt{2}$. The projective motion and shape are computed for the transformed image points $T_i x_{ij}$, $P_i X_{ij} = \lambda_{ij} T_i x_{ij} \sim T_i x_{ij}$, therefore the resulting projective estimates P_i must be corrected : $P^{'}_i = T_i^{-1} P_i$. The matrices $P^{'}_i$ and X_{ij} then represent projective motion and shape corresponding to the measured image points x_{ij} . Figure 7.1 illustrates the reconstructed correspondences of a checker board without/with pre-conditioning (bottom row, left and right image, respectively).

Another technique applied to ensure good numerical conditioning was balancing, i.e. rescaling the projective depth matrix [StuTri96] so that all matrix rows and columns have on average the same order of magnitude. We achieved this by the following scheme:

1. Rescale each column *l* so that $\sum_{r=1}^{3m} (\lambda_{rl})^2 = 1$
2. Rescale each triplet of rows (3*k*-2, 3*k*-1, 3*k*) so that $\sum_{l=1}^{n} \sum_{i=3k-2}^{3k} \lambda_{il}^2 = 1$

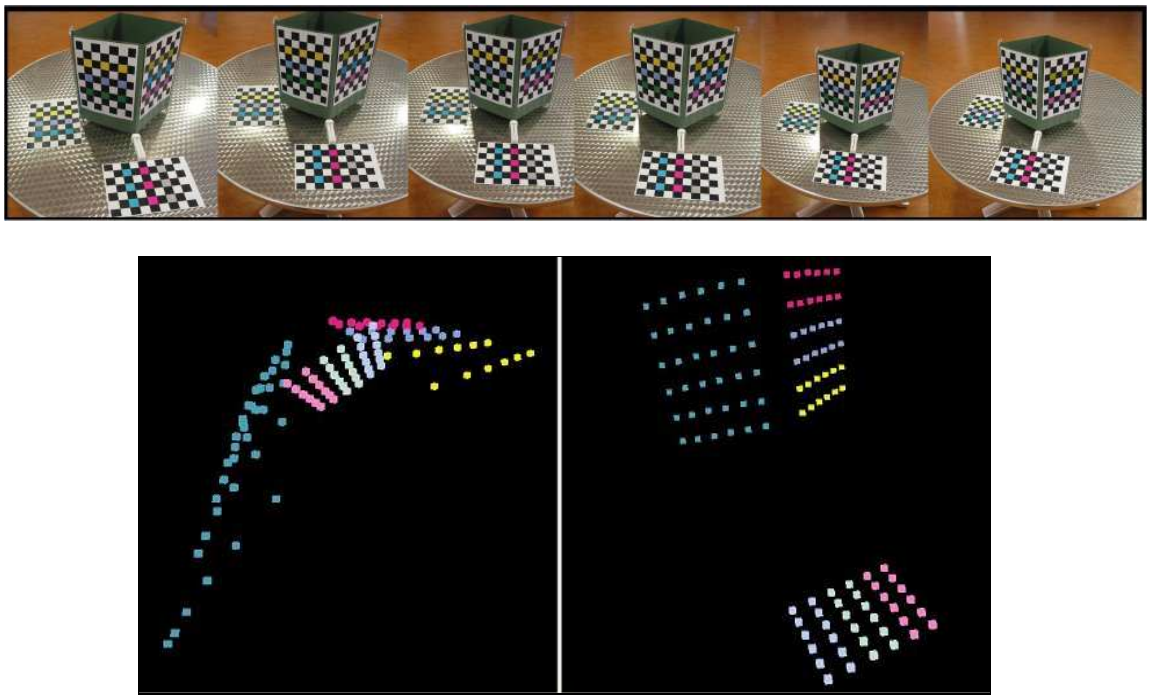


Figure 7.1 Upper row: the 6-image sequence of a checker board . Lower row: the reconstructed structure of the corresponding features (the colored corners of the pattern) without/with pre-conditioning (left and right image, respectively)

7.1.2 Iterative Factorization Algorithm

Several experiments have been carried out to observe the convergence of the Iterative Factorization Algorithm (IFA).

A set of experiments were conducted on the *CIL–0001* dataset [Web7] provided by the Calibrated Imaging Laboratory of Carnegie Mellon University (Figure 7.2 left). The *CIL-0001* sequence consists of 11 views, 28 corresponding points, the mean calibration error is within 0.1 pixels.

The 2D coordinates of the image points were perturbed with Gaussian noise of zero mean and standard deviation ranging from $\sigma = 0.5\ldots3$. The number of iterations and 2D reprojection error vs. noise level are shown in Figure 7.3. The 2D reprojection errors result from projecting the recovered points using the recovered camera geometry and parameters and are measured in pixels.

The convergence of IFA is illustrated in Figure 7.4 where the residual of Equation (3.11) is plotted against number of iterations.

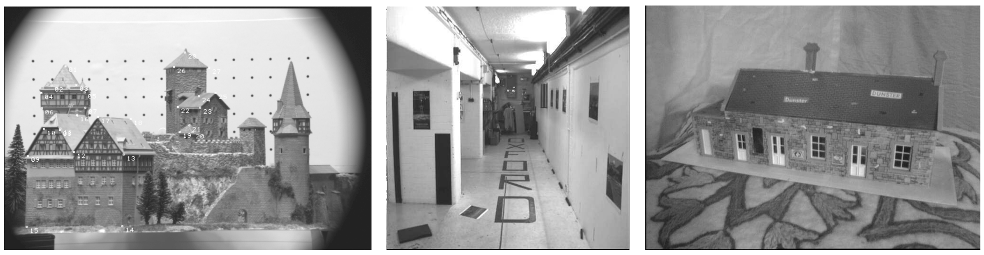


Figure 7.2 Images belonging to the *CIL–0001* (left), *corridor* (middle) and *model house* (right) datasets

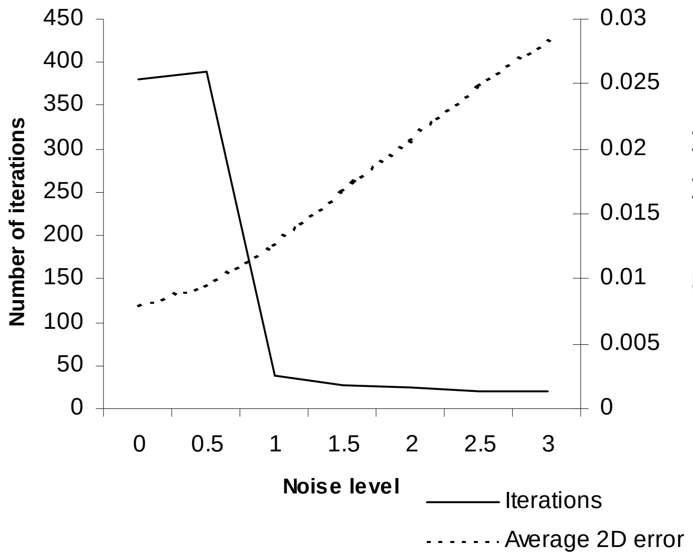


Figure 7.3 Number of iterations and 2D error vs. noise level

We have also performed a number of tests on image sequences acquired with a Canon G2 camera with varying focal length. The main features of the data sets are described in table 7.1. The convergence of IFA is illustrated in Figure 7.5 where the residual of Equation (3.11) is plotted against the number of iterations.

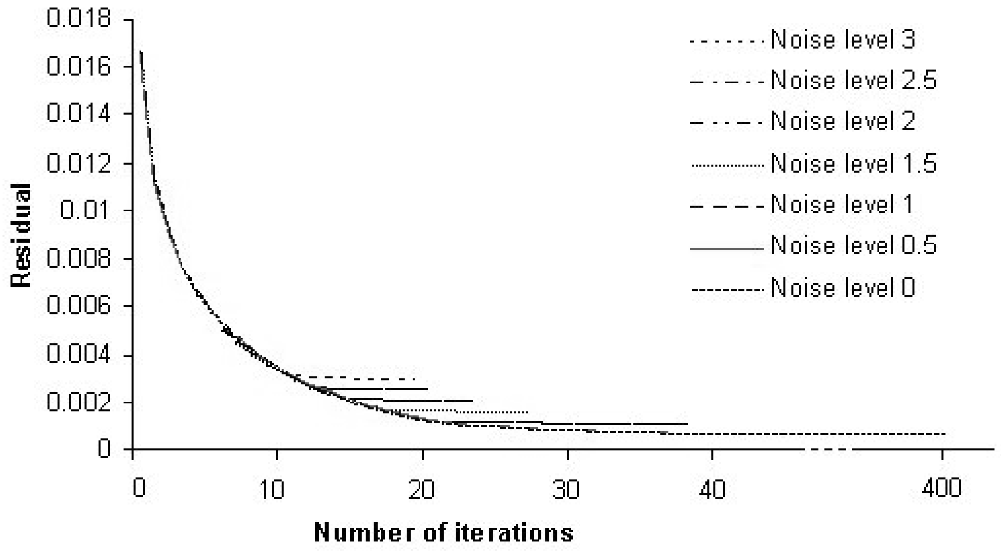


Figure 7.4 Residual vs. number of iterations

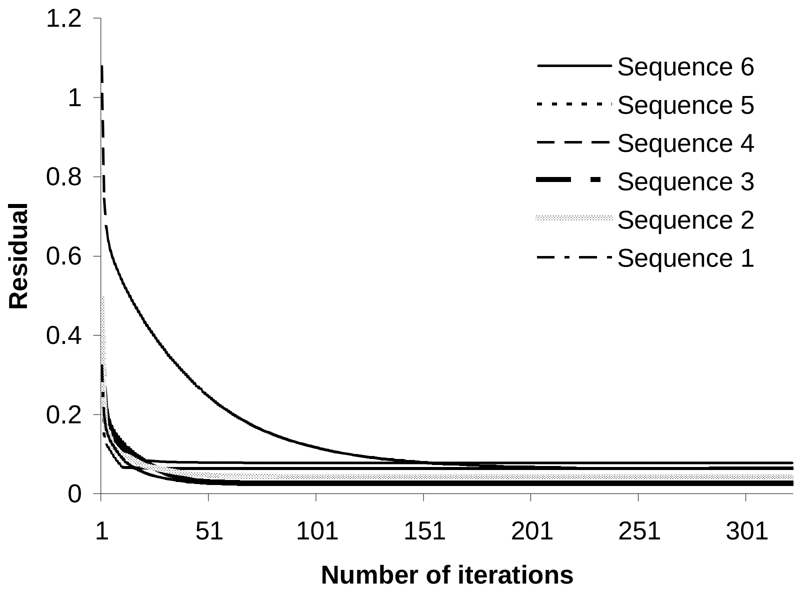


Figure 7.5 Residual vs. number of iterations

Data set	**Description**	**Images**	**Tracked points**	**Iterations**	**2D error (pixels)**
Sequence 1	objects, checker board	7	41	73	0.55
Sequence 2	human subject with markers, checker board	6	56	173	1.28
Sequence 3	human subject with markers, checker board	7	51	88	0.91
Sequence 4	checker board	5	108	290	0.49
Sequence 5	checker board	6	108	158	0.46
Sequence 6	checker board	6	108	321	0.48

Table 7.1 Experimental data sets

We have observed that with high accuracy data the IFA method requires a large number of iterations, and when noise is added, it stabilizes with much less iterations. This is because when data is accurate a very accurate solution can be achieved, taking more processing time. We consider this to be a good behavior of our system.

7.2 Bundle Adjustment

In order to investigate the performance of the quaternion-parameterized sparse LM algorithm we have conducted comparison experiments with a dense, general version as well as a sparse version of the LM algorithm, available at [Web4]. We have employed the c*orridor* and *model house* datasets (Figure 7.2 middle and right, respectively) of the Oxford's Visual Geometry Group [Web6], frequently used for benchmarking in the vision literature. Since we operate under the assumption that the tracked features are visible in all views, we have restricted the sequences to the number of points and frames shown in table 7.2.

Table 7.2 illustrates several statistics: the average reprojection error of the initial reconstruction and the average reprojection error after sparse LM refinement, the number of iterations as well as the processing time.

The corresponding processing times using dense bundle adjustment were 89.58 and 112.1 seconds, respectively. The processing times for the general sparse bundle adjustment were 0.42 and 0.65 seconds, respectively. Compared to the processing times needed by our method, these results show performances close to the general sparse LM implementation and also the computational benefits achieved by the exploiting the sparsity of the problem.

Data set	**Images**	**Tracked points**	**Initial 2D error (pixels)**	**Final 2D error (pixels)**	**Time (s)**
Corridor	6	100	0.87	0.41	0.63
Model house	7	76	1.76	0.23	0.94

Table 7.2 Sparse Levenberg-Marquardt optimization statistics for the benchmark sequences

Data set	**Images**	**Tracked points**	**Initial 2D error (pixels)**	**Final 2D error (pixels)**	**Time (s)**
Sequence 1	7	41	0.55	0.24	0.20
Sequence 2	6	56	1.28	0.59	0.31
Sequence 3	7	51	0.91	0.33	0.23
Sequence 4	5	108	0.49	0.19	0.44
Sequence 5	6	108	0.52	0.17	0.58
Sequence 6	6	108	0.48	0.17	0.52

Table 7.3 Sparse Levenberg-Marquardt optimization statistics

Table 7.3 presents experimental results gathered from the application of the sparse Levenberg-Marquardt optimization to the initial 3D structure estimates of our test sequences (i.e. sequences 1-6). The benchmark sequences experiments were conducted on a Microsoft Windows XP, 1.66 GHz Intel Dual Core T5500 platform. The sequences 1-6 experiments were conducted on a Microsoft Windows XP, 3.2 GHz Intel P4 platform.

In all cases, the sparse LM algorithm terminated due to the magnitude of the computed step Δ being very small.

7.3 Voxel Carving

The following voxel carving-related experiments are partitioned in two main subsets, corresponding to the CPU-based and GPU-based aspects, respectively.

At the time of writing, GPGPU researchers - including voxel-carving related work [LiMS04, WoeKoch04, ZacKar04] - provide strictly GPU vs. CPU performance comparisons of own implementations. Besides the extremely fast-paced hardware features changes, the major reason behind this is the lack of disclosed manufacturer details and of a unified framework for the existing graphic cards and shading languages that would make GPU vs. GPU comparisons of related approaches meaningful. We will provide accordingly a GPU vs. CPU performance comparison.

7.3.1 Multi-resolution 3D Reconstruction

We will present in the following a multi-resolution 3D reconstruction using a data set of five images acquired at resolution 1704 x 2272, with a human subject with placed markers for easier point selection. The set of frames used to reconstruct the object are shown in Figure 7.6.

The set of tracked features and their correspondences were set manually. Also, the background of the images is segmented manually to facilitate the reconstruction process. Details of the sequence and preceding self-calibration are provided in Table 7.4. The initial and final 2D errors are the values obtained after the IFA algorithm and after bundle adjustment, respectively. The left image in Figure 7.7 shows the selected corresponding points, while the right image shows the recovered metric structure of the correspondences, as well as the camera positions.

Data set	Description	Images	Tracked points	Initial 2D error (pixels)	Final 2D error (pixels)
Sequence 7	human subject with markers, checker board	5	63	1.02	0.47

Table 7.4 Human subject data set description

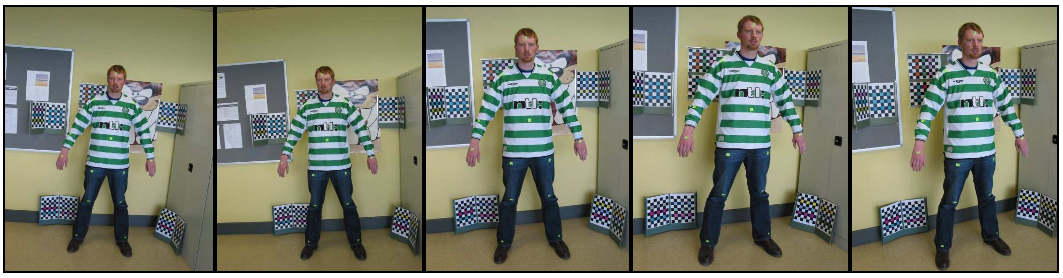


Figure 7.6 The 5-image input sequence

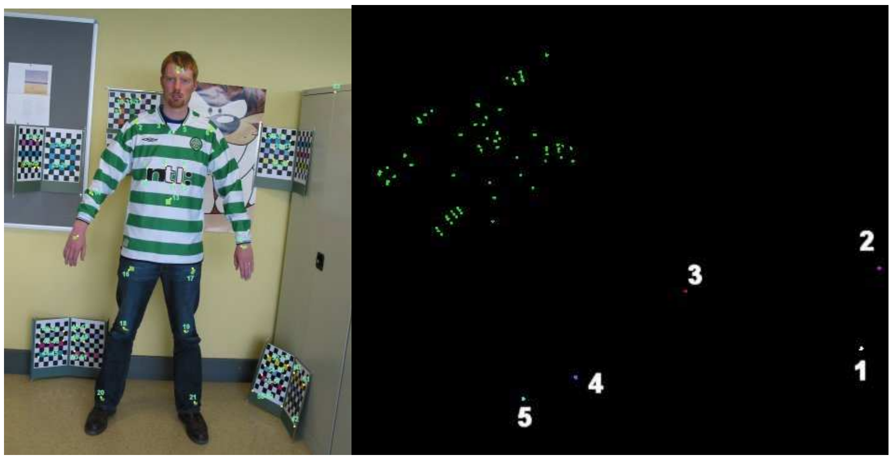


Figure 7.7 Left: A sequence image with the tracked points.
Right: The recovered metric structure of the tracked points and the camera positions

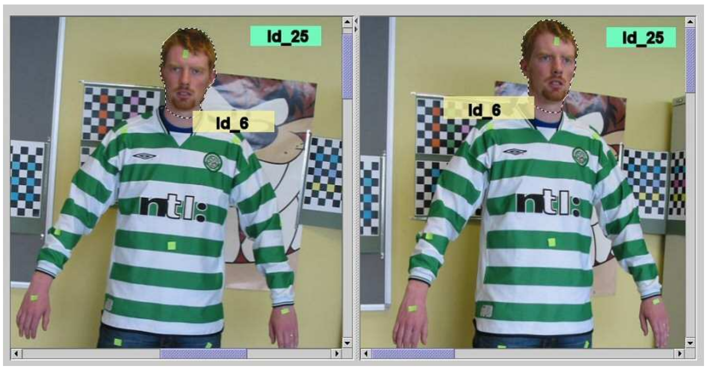


Figure 7.8 Two sequence frames with the user-labeled regions.

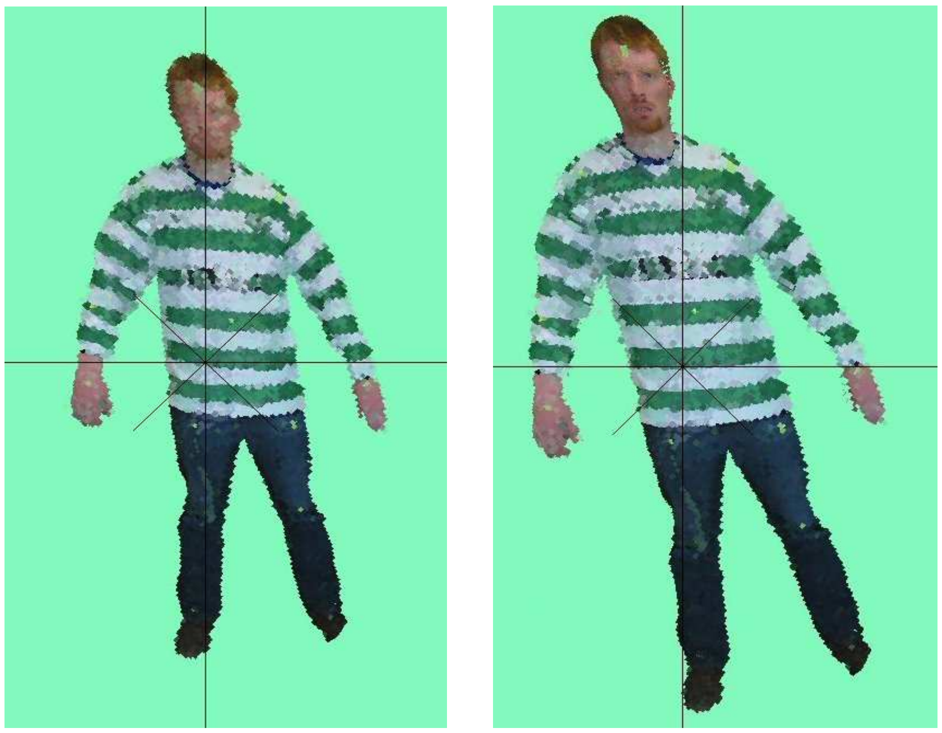


Figure 7.9 Left: the reconstructed human model at resolution *r*=25. Right: same 3D model with the face region refined at resolution *r*=6

Voxel carving was initialized with a bounding box with the volume 168 x 160 x 72 voxels. The left image in figure 7.9 shows the 3D shape reconstructed at resolution r=25. With the face area of the subject selected for refinement in only two frames (Figure 7.8), we performed the algorithm for two resolution increases, resulting in a final resolution r=6. The multi-resolution reconstruction is shown in the right image of Figure 7.9. Figure 7.10 presents detail views of the above reconstructions.

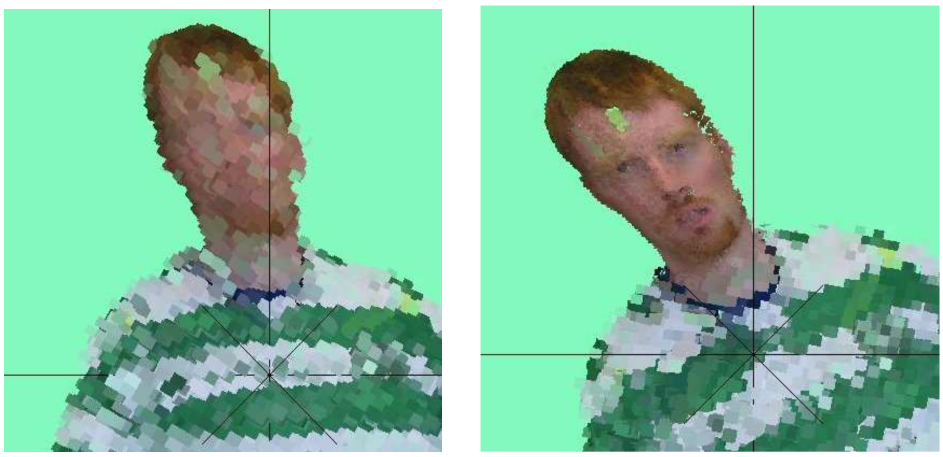


Figure 7.10 Detail views of the above left and right images, respectively.

7.3.2 Reconstruction from Extended Sequences

In order to investigate an extended sequence we have employed a set of frames consisting of 16 images captured at resolution 1704 x 2272, which were divided into four subsequences, illustrated together with the number of tracked points for each of them in Figure 7.11. Point tracking and background segmentation in a few frames is performed manually. In order to increase the stability of sequence merging we have chosen two overlapping frames between sequences 3 and 4, and three overlapping frames between sequences 2 and 3.

Dataset	**Initial 2D error (pixels)**	**Final 2D error (pixels)**
Sequence 1	0.60	0.25
Sequence 2	1.15	0.47
Sequence 3	0.83	0.32
Sequence 4	1.29	0.36

Table 7.5 Calibration and bundle adjustment statistics for the subsequences of the extended sequence

Table 7.5 shows the calibration and LM optimization results for all subsequences.We have performed reconstructions with 4 different voxel resolutions in order to observe the relation between the model size and processing times for the CPU- and GPU-based algorithms. The statistics illustrated in Table 7.6 show that the computation times achieved by the carving engine are approximately three to eight times faster than the software-based algorithm, depending on the model complexity.

The experiments were conducted on a Microsoft Windows XP, 3.2 GHz Intel P4 platform and a Nvidia Quadro FX 3400 graphics unit. Figure 7.12 shows the final reconstructed object in novel rendering positions.

Voxel resolution	CPU time (s)	GPU time (s)
30^3	125.49	16.24
40^3	187.26	29.82
50^3	240.06	73.42
80^3	523.22	168.10

Table 7.6 Reconstruction CPU and GPU statistics for voxel carving

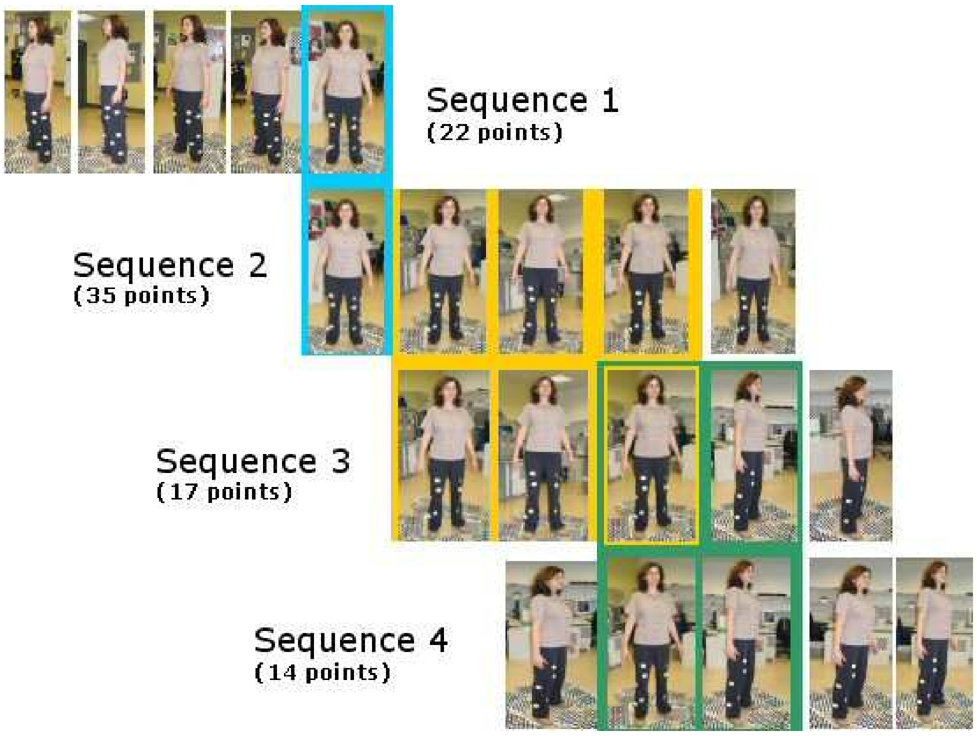


Figure 7.11 The division scheme of the extended sequence

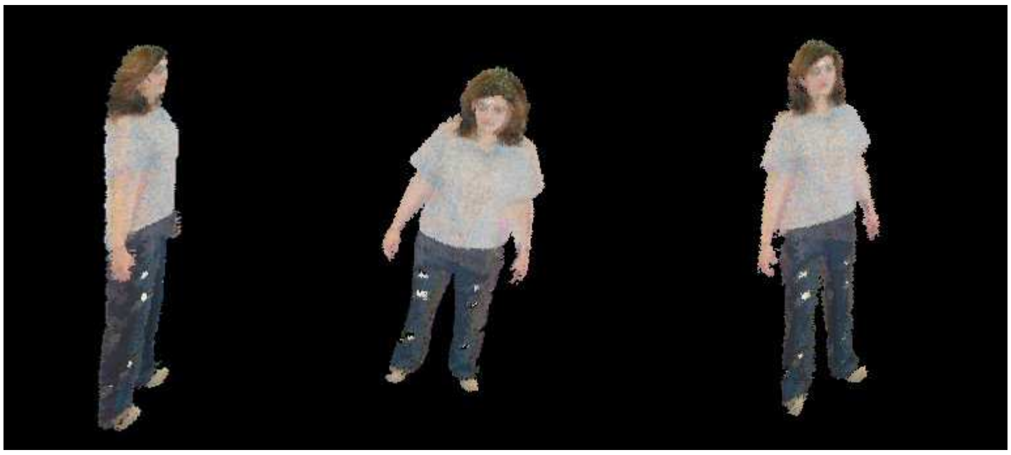
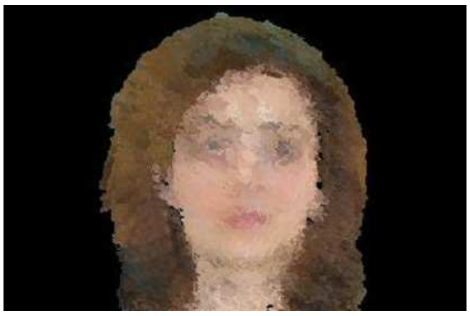



Figure 7.12 Novel rendering positions and a detail view of the human subject

7.3.3 Carving Engine Analysis

In order to observe exhaustively the performance parameters of the carving engine, we have used a calibrated dataset available at [Web5]. The Millie dataset consists of a set of 10 images, obtained by placing the object on a turn table and rotating the platform with angle increments from the starting position. The sequence presents a 720 x 480 resolution. Background segmentation occurs with the help of alpha-map silhouettes. For performance tuning, we have disabled during the experiments the render to display function, that is, voxels will be rendered on screen only after processing terminates.

Table 7.7 shows the global reconstruction statistics, namely the iterations number, processing time and the highest framerate value, since the framerates are varying during reconstruction with the number of surface voxels. The obtained results show that the carving engine is capable of achieving real-time or at least interactive framerates.

Voxel resolution	Time (s)	Max framerate (frames/s)
15^3	8.34	23
25^3	13.09	21
35^3	34.63	16
45^3	65.23	7

Table 7.7 Carving engine performance on a Quadro FX 3400

Furthemore, our method is adaptable to the rapidly evolving hardware features. Pending or very recently added graphics hardware functions may enhance further the performance of the carving engine.

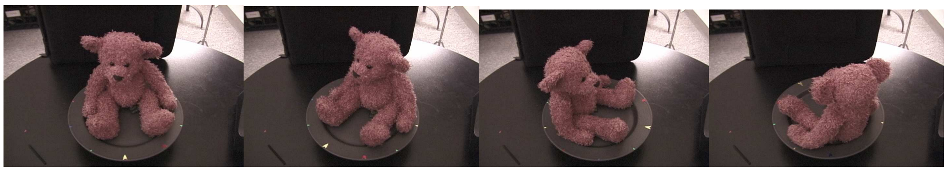


Figure 7.13 Several images of the *Millie* dataset

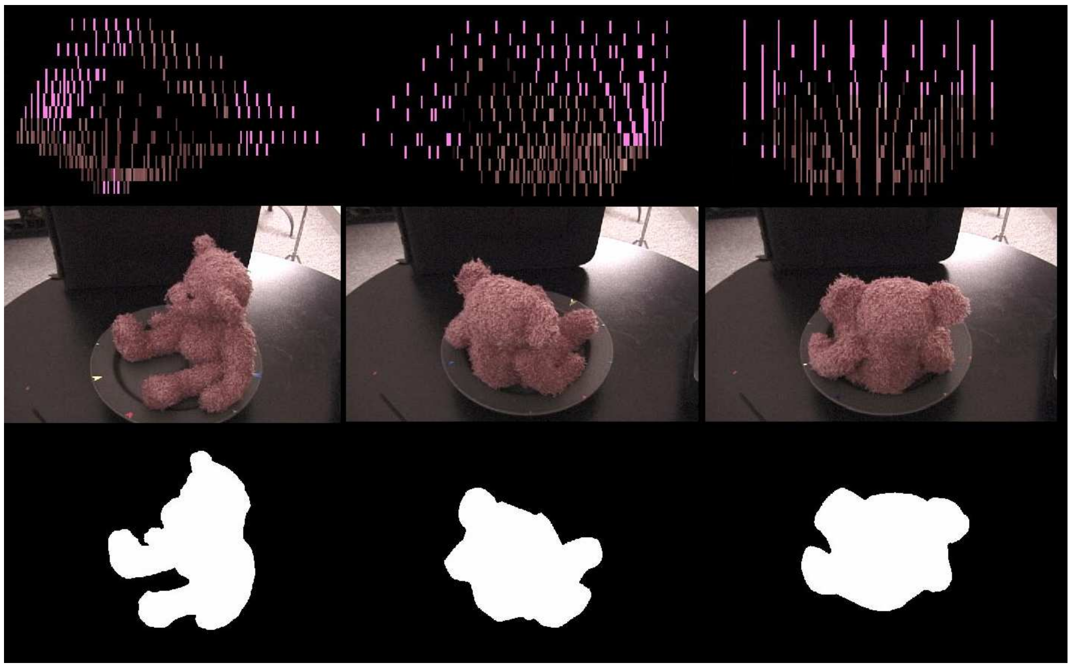


Figure 7.14 Visualization of image buffers with their corresponding camera views and silhouette alpha maps during the projective texturing stage.

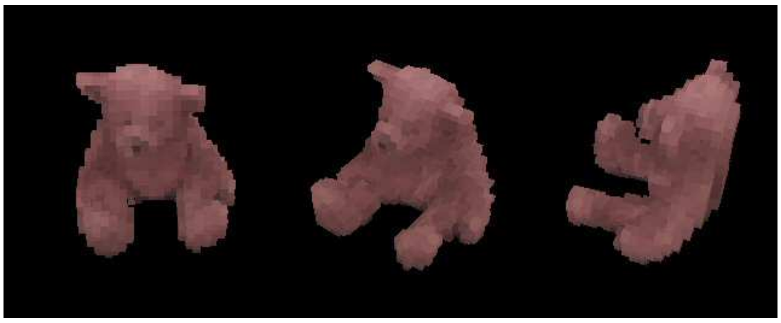


Figure 7.15 Novel rendering positions of the *Millie* dataset

Figure 7.14 represents a visualization of the projective texture mapping stage (Chapter 4, § 6.3.2). The upper row shows three image buffers (bound as render targets during this step), while the middle the bottom row show their corresponding camera views and silhouette alpha maps (bound as input textures). Magenta pixels represent voxels that didn't survive the alpha test.

Figure 7.15 shows the final reconstructed object in novel rendering positions.

The carving engine was written using Cg [MGA*03], OpenGL, OpenGL extensions and graphic card vendor specific extensions. The carving results were measured on a Microsoft Windows XP, 3.2 GHz Intel P4 machine with 1 GB RAM, with an Nvidia Quadro FX 3400 graphic card.

In the following, we analyze the carving engine in terms of memory bandwidth and computational complexity.

Bandwidth considerations

In order to investigate the potential bandwidth limitations for our method, we first distinguish between the two bandwidth types of modern GPUs. The *external* bandwidth is the rate at which data may be transferred between the GPU and the main system memory. Conversely, the *internal* bandwidth is the rate at which the GPU may read and write from its own internal memory. The external bandwidth of the GPU presents importance for our application mainly during the read-back of the photo-consistency computation results from the card to the CPU, into main memory. As discussed previously, the carving engine output format is designed to return minimal data to the main memory. For the *Millie* example, we have measured transfer times between 38-61 milliseconds, amounting to about 8-11% of the total processing time. Thus, external bandwidth transfer is a significant, but fairly small fraction of the total time.

Concerning the internal bandwidth transfers, we have considered the main processing stages of the carving engine. In our algorithm, every memory operation transfers 4 bytes of data. The projective texturing step requires three texture fetches and writes one value per fragment. The pixel routing step requires only one write per fragment. The photo-consistency step performs an inner loop over the number of camera views, and requires two texture fetches per each iteration and then one write per fragment.

Table 7.8 summarizes these results and shows the total internal bandwidth in bytes transferred by each method, which is the product of the number of passes, the fragments per pass and the bytes per fragment.

The parameters are as follows: *s* is the current SVL size, *n* is the number of reference images, *v* represents the size of the image buffer (i.e. texture width x height).

Process	**Fragments**	**Passes**	**Bytes/ fragment**	**Total bytes**
Projective texturing	s	n	16	$16sn$
Photo-consistency	s	1	$8n+4$	$8sn+4$
Pixel routing	v	n	4	$4vn$
			Total :	$4n(6s+v)+4$

Table 7.8 Bytes transferred internally by the rendering passes

Process	**Arithmetic operations/ fragment**	**Fragments**	**Passes**	**Total operations**
Projective texturing	96	s	n	$96sn$
Photo-consistency	$52n+58$	s	1	$52sn+58s$
Pixel routing	27	v	n	$27vn$
			Total:	$148sn+27vn+58s$

Table 7.9 Floating point operations required by each main rendering pass

Arithmetic complexity

The GPU uses the fact that the same instructions are being executed on a large number of fragments simultaneously. As no communication between executions of the kernels is needed, an abundant amount of parallelism is available. This parallelism is used to hide the latency of memory operations and other bottleneck causes.

As a result, when enough fragments are available - as in our case - the running time of a kernel is approximately linear in the number of instructions executed. Therefore, we have summarized the number of instructions required by each computational stage of our algorithm in Table 7.8

Similarly to Table 7.7, the parameters are as follows: *s* is the current SVL size, *n* is the number of reference images, *v* represents the size of the image buffer (i.e. texture width x height). The projective texturing and pixel routing perform 96 and 27 arithmetic instructions per rendering pass. The photo-consistency check performs 52 arithmetic instructions within the loop over camera views mentioned above, and 58 instruction outside the loop, amounting to 52*n*+58 operations per fragment .

Chapter 8 Conclusions and Future Work

Introduction

The previous chapters have introduced the theoretical considerations of a complete pipeline for reconstruction of objects from images. We described the implementation details and presented the achieved results. In this chapter we point out the advantages as well as the limitations of the system. Additionally, as a conclusion, some reflections on the work, its limitations, its applicability and future work are discussed.

8.1 Conclusions

The principal objective of this work is to develop a software pipeline, based on IBMR techniques, that allows the reconstruction of real objects with their shape and color properties recovered.

The first stage of the proposed system requires a set of features tracked across a sequence of images. Keeping in mind that we target non-expert users, we have used different techniques to achieve a reliable calibration from a set of manually selected features in sequences which usually contain less frames. However, the proposed solution equally allows the use of automatically tracked video sequences, entailing an extended number of frames.

The complete sequence is divided into subsequences and, in each of them, a set of keyframes is selected and calibrated, recovering both camera parameters and structure of the scene. A Levenberg-Marquardt non-linear optimization is performed in order to reduce the overall reprojection error. When the different subsequences have been successfully calibrated a merging process groups them into a single set of cameras and reconstructed 3D features of the scene.

The camera calibration process is a critical problem in our application. One advantage of the presented calibration approach is that it allows to recover an Euclidean reconstruction of the scene without any initial solution or prior information and it amounts to solving only linear systems. The knowledge of the geometric meaning and rank properties of the different transformations represented by the matrices allows to enforce a valid Euclidean reconstruction. The presented solution is designed to be flexible with respect to the input data allowing the use of varying focal length throughout the sequence.

There are however several directions of vast investigation in this stage of the pipeline, a couple of them concerning the analysis of critical camera configurations [Pollefeys00b, CPV02, CVG04] and the sensitivity of bundle adjustment to false matches. Related to this, the spatial distribution of the image feature points represents a further examination direction. For example, situations where points are chosen too close to each other, or are biased towards an image region should be avoided because the estimation of the epipolar and projective geometry becomes highly unstable [Zhang98].

Another important aspect is that the "perspective effect" present in many of our experiments reveals the necessity of modeling lens distortion. Unmodeled camera lens distortions cause a warp- or bend-like error in the recovered structure and motion since the self-calibration pipeline expects the camera to comply to a purely perspective projection model. While the bundle adjustment stage performs a minimization of the reprojection error, it cannot remove the effect of lens distortion [CPV02]. Therefore, the camera model needs to be extended with at least one parameter for radial distortion in order to improve the recovered metric structure [PVV*04].

Also, further analysis should be conducted on the sequence merging. When two consecutive subsequences present very different focal length settings, this process becomes extremely difficult, even impossible.

The second stage of the pipeline, the scene reconstruction, has the objective of extracting a voxelized reconstruction based on the reference views and the calibration information. As one can imagine this is tedious task, because reconstruction from images is an ill-posed problem unless a large number of images is provided, covering all possible features of the model, or additional information is introduced in the pipeline. Carving algorithms proved to be a decent approach, however they are highly dependent on the implementation and on the quality of the input images. A further direction to explore could be a hybrid approach that integrates space carving and long baseline multi-view reconstruction, in such a way that the methods complement each other introducing constraints on the final shape.

The voxel carving process requires the analysis of a large number of discrete elements. The main reason we introduced the multi-resolution calculation was to address this extended computational cost by restricting locally the level of detail, with the help of common image editing operations. Moreover, as we have seen in the previous examples, the complete sequence does not need to be edited, but only a few frames. Therefore, the user can manually process 2 or 3 frames and use those as a starting point for a refinement process.

Conversely, the voxel carving engine, tackles the aforementioned computational costs from a different angle, capitalizing on the abundant parallelism offered by modern graphics hardware. Our approach eliminates the 3D texture restrictions and efficiently uses the GPU-CPU bandwidth as well as the GPU inner bandwidth by returning only compact data and employing a two-dimensional data representation that fits the two-dimensional data layout on the GPU.

The performance of the carving engine would benefit from the recently introduced *framebuffer object* (FBO) extension [EXT05], an enhanced and simplified method of doing render-to-texture. The frequent pixel buffer swaps during the carving process imply an equal number of expensive context switches, since pixel buffers require their own rendering context within the graphics API. One of the main advantages of FBOs is that they only require a *single* graphics API context, so that switching between framebuffers is at least twice as fast as switching between pixel buffers, depending on the employed technique.

The principal theoretical contribution of this body of work is a quaternion parameterized Levenberg-Marquardt optimization technique. Furthermore, we made the following practical contributions:

- A multi-resolution, user-guided voxel carving method
- A GPU-based voxel carving engine
- A complete system for flexible retrieval of metric 3D surface models from uncalibrated image sequences

This work is relevant for the fields of structure from motion, voxel-based 3D reconstruction, and also for general processing on the graphics processing unit. The presented tailored sparse optimization and GPU-based voxel carving methods bring significant computational gains compared to dense and software-based techniques, respectively. Additionally, our system would scale well and benefit from the graphics hardware trend of expanding the number of fragment and vertex processors, and texture units, as well as other future enhancements. Moreover, both developed voxel carving approaches present potential for the field of human-computer interaction due to the interactive user involvement possibilities they provide.

8.2 Future Work

Until now we have focused on the geometric and processing performance aspects of IBMR, leaving the rendering part almost untackled. Rendering together with geometric accuracy and non-Lambertian lighting conditions remain areas to be further explored and developed. In the following, we will outline several future work directions investigating possible improvements which are still needed for an accurate and efficient recovery of the 3D scene.

Robust Self-calibration of Long Baseline Sequences

The self-calibration method presented in Chapter 3 starts from the assumptions that the tracked correspondences are static points present in all views (i.e. all features are valid for calibration). However, due to camera paths around objects, we have to deal with large amounts of frames and the features will not be visible in all of them. Therefore, we need to develop a strategy for dividing long sequences into manageable sub-sequences suitable for self-calibration. Sequence division and self-calibration will be followed by sequence merging in order to recover the complete scene structure. Moreover, often consecutive frames reflect very little changes, so that for computational cost reasons it would be useful to detect the keyframes that introduce significant 3D information.

Also, a crucial aspect of the self-calibration process is the convergence of the projective factorization. Further analysis should be conducted on the projective reconstruction stage that could be enhanced with an algorithm that presents a faster convergence.

Volumetric Reconstruction Using an Evolution Surface

Incomplete surface data can produce reconstructions with missing areas (Figure 8.1), requiring a post-processing step with hole-filling algorithms.

Furthermore, reconstructions produced by space carving can present ragged surfaces with floating voxels, especially for high curvature surfaces. Rather than post-process the reconstructed surface, the level set approach for surface evolution proposed in [SSH02] mitigates the above problems during reconstruction and obtains a smooth, watertight geometry. An initial surface is embedded as the zero level set of a volumetric function that moves along its inwardly pointing normal, with a speed based on a photo-consistency measure of surface points. Level set theory [Sethian99] provides a numerical scheme that solves the partial differential equations that characterize the motion of the surface.

Space carving with an evolution surface employing a function that includes a flow term modeling a non-Lambertian color-consistency measure (discussed below) could represent a direction of future investigations.

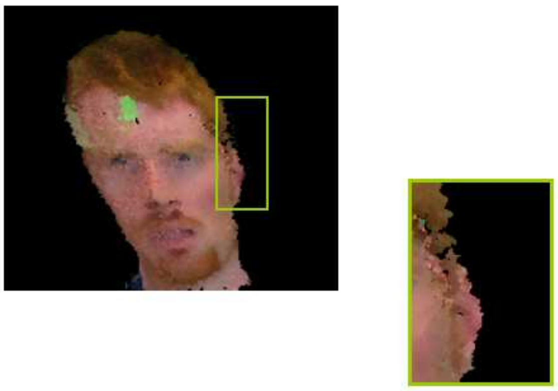


Figure 8.1 Detail of holes in the reconstructed surface, caused by grazing view angles

Reconstruction of Non-Lambertian Scenes

Currently we rely on the Lambertian assumption, commonly made in reconstruction algorithms, that simplifies the problem, but limits the class of scenes that can be reconstructed. However, real surfaces interact with light in complex ways, producing view-dependent effects such as specularities and reflections. Thus, more sophisticated modeling of the bidirectional reflectance distribution function (BRDFs) will be required to improve the flexibility of the reconstruction algorithm. Work on this problem has started to emerge in the literature [CarKut01, Chhabra01, Magda01, JSY03, YPW03, THS04].

Issues that need to be further explored are handling general BRDFs, and possibly employing new cues, like *orientation-consistency* within the voxel coloring framework (the orientation-consistency cue introduced in [HerSei03] states that under orthographic projection and distant lighting, two surface points with the same surface normal and material exhibit the same radiance).

GPU-based IBMR Pipeline

Our IBMR system features mainly a CPU-based component encapsulating computer vision algorithms and a GPU-based component, enclosing computer graphics algorithms. An exciting area of investigation is the prospect of a full GPU-based reconstruction pipeline. Early work on efficiently mapping computer vision algorithms for a stereo pair of images to the GPU has been presented in [FMA04, FM05]. The sparse Levenberg-Marquardt optimization requires solving repeatedly a sparse equation system. Recently, GPUs have been used for linear algebra, including programs for matrix multiplication [JH03], an iterative sparse system solver [BFGS03], and a direct dense system solver [GGHM05].

Our system performs a serial update on the CPU due to the insert/delete operations required by the update of the dynamic surface voxel structure. However, dynamic complex data structures on the GPU are an area of active research, as they have applications in many computer graphics areas. In [LKH*04], [CHL04] the authors describe efficient GPU-based dynamic algorithms that use the CPU only as a memory manager. A system that builds on the work enumerated above would undoubtedly bring benefits to the field of IBMR.

References

[ARB03] OpenGL ARB Shading Language Extension: ARB_vertex_buffer_object, February 2003, http://oss.sgi.com/projects/oglsample/registry/ARB/vertex_buffer_object.txt

[BacKam97] Bacakoglu, H. Kamel, M. "An optimized two-step camera calibration method" Proceedings., 1997 IEEE International Conference on Robotics and Automation, pp. 1347-1352 vol.2, 1997.

[Broadhurst01] Broadhurst, A., "A Probabilistic Framework for Space Carving". Ph.D. thesis, University of Cambridge, 2001

[BTZ96] P. Beardsley, P.H.S. Torr and A. Zisserman. "3D Model Acquisition From Extended Image Sequences". In Proc. of ECCV '96, pp. 683-695, 1996.

[BuckPur04] Buck, I., Purcell T.J. "A Toolkit for Computation on GPUs", Chapter 37, in "GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics", Addison Wesley. pp 621-636, 2004.

[CarKut01] R. Carceroni, and K. Kutulakos, "Multi-View Scene Capture by Surfel Sampling: From Video Streams to Non-Rigid Motion, Shape, and Reflectance," in International Conference on Computer Vision, vol. 2, pp. 60–67, 2001

[Chen00] Q. Chen, "Multi-view Image-Based Rendering and Modeling", Ph.D. thesis, University of Southern California, 2000

[Chhabra01] V. Chhabra, "Reconstructing Specular Objects with Image-based Rendering Using Color Caching," Master's thesis, Worcester Polytechnic Institute, 2001

[CHL04] G. Coombe, M. Harris, A. Lastra, "Radiosity on graphics hardware", in Proceedings of the 2004 Conference on Graphics Interface (May 2004), pp. 161–168.

[CMS99] W. Culbertson, T. Malzbender and G.Slabaugh, "Generalized voxel coloring", *International Workshop on Vision Algorithms*, Corfu, Greece, Springer Verlag Lecture Notes on Computer Science, pp. 100-115, 1999

[CPV02] K. Cornelis, M. Pollefeys, L. Van Gool, ``Lens Distortion Recovery for Accurate Sequential Structure and Motion Recovery'', Lecture Notes on Computer

Science - 2351, Proceedings of the European Conference on Computer Vision, A. Heyden, ed., vol. 2, p. 186-200, May 2002, Springer Verlag

[CVG04] K. Cornelis, F. Verbiest, L. Van Gool, " Drift detection and removal for sequential structure from motion algorithms" IEEE Transactions on Pattern Analysis and Machine Intelligence, Volume: 26, pp. 1249- 1259, Oct. 2004

[DebVio99] J. Debonet and P. Viola, "Roxels: Responsibility Weighted 3D Volume Reconstruction," Proceedings of the IEEE International Conference on Computer Vision, 1999, Vol. 1, pp. 415- 425

[ESG99] P. Eisert, E. Steinbach and B. Girod, "Multi-Hypothesis, Volumetric Reconstruction of 3-D Objects From Multiple Calibrated Camera Views," Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, 1999, pp. 3509-3512.

[Faugeras95] O. Faugeras, "Stratification of three-dimensional vision: projective, affine, and metric representations", Journal of the Optical Society of America A, pp. 465–483, Vol. 12, No.3, March 1995.

[FitZis98] A.W. Fitzgibbon and A. Zisserman."Automatic Camera Recovery for Closed or Open Image Sequences". In Proc. of ECCV'98, pp. 311-326, 1998.

[FunMan04] J. Fung, S. Mann. "Computer vision signal processing on graphics processing units." In Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (May 2004), vol. 5, pp. 93–96.

[Fusiello00] A. Fusiello, "Uncalibrated Euclidean reconstruction: a review", in Image and Vision Computing, 18, pp. 555-563, 2000.

[GFG04] P. Gherasimov, R. Fernando, S. Green, "Shader Model 3.0: Using Vertex Textures", Nvidia whitepaper, June 2004

[GLM05] N. K. Govindaraju., M. C. Lin., D. Manocha. "Quick-CULLIDE: Efficient inter- and intra-object collision culling using graphics hardware." In *Proceedings of IEEE Virtual Reality,* March 2005, pp. 59–66.

[Golub96] G.H. Golub and C.F. Van Loan, "Matrix Computations", John Hopkins University Press, 1996

[HanKan00] M. Han and T. Kanade, "Creating 3D models with uncalibrated cameras", IEEE Computer Society Workshop on the Application of Computer Vision, (WACV2000), 9(2), pp. 137-154, 2000.

[HCT*02] M. J. Harris, G. Coombe, T. Scheuermann, and A. Lastra "Physically-Based Visual Simulation on Graphics Hardware" in Proc. SIGGRAPH / Eurographics Workshop on Graphics Hardware 2002.

[HBS*03] M. Harris, W. Baxter III, T. Scheuermann, A. Lastra, "Simulation of Cloud Dynamics on Graphics Hardware", in Proc. SIGGRAPH / Eurographics Workshop on Graphics Hardware 2003.

[Harris04] M. Harris, "Fast fluid dynamics simulation on the GPU". In GPU Gems, Fernando R., Addison Wesley, Mar. 2004, pp. 637– 665.

[Hartley93] R. Hartley, "Euclidean reconstruction from multiple views", Europe-U.S. Workshop on Invariance, pages 237–56, Ponta Delgada, Azores, October 1993.

[Hartley94a] R. Hartley, "Lines and points in three views – an integrated approach", Image Understanding Workshop, Monterey, California, November 1994.

[Hartley94b] R. Hartley. Euclidean reconstruction from uncalibrated views. Proceedings of CVPR94, 1994, pp. 908-912.

[Hartley95] R. Hartley, "In defence of the 8-point algorithm", Proceedings of International Conference on Computer Vision, pages 1064–1070, 1995.

[Hartley99] R. Hartley, E. Hayman, L.s de Agapito and I. Reid. Camera calibration and the search for infinity. Proceedings of International Conference on Computer Vision, Kerkyra, Greece. IEEE Computer Society Press, 1999.

[HarZis00] R. Hartley and A. Zisserman, "Multiple View Geometry in Computer Vision", Cambridge University Press 2000

[HerSei03] A. Hertzmann, S.M. Seitz, "Shape and materials by example: A photometric stereo approach", Proceedings of International Conference on Computer Vision 2003, pp. 533-540

[HeyAst96] A. Heyden, A. and K. Astrom, "Euclidean Reconstruction from Constant Intrinsic Parameters", Proceeding International Conference on Pattern Recognition, 1996.

[HeyAst99] A. Heyden and K.Astrom, "Flexible calibration: Minimal cases for auto-calibration" Proceedings of International Conference on Computer Vision, Kerkyra, Greece, IEEE Computer Society Press, 1999.

[HeyKah01] A. Heyden and F. Kahl, "Euclidean reconstruction and auto-calibration from continuous motion" Proceedings of International Conference on Computer Vision, 2001.

[HHN*02] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. Kirchner, J. Klosowski. “Chromium: A stream processing framework for interactive rendering on clusters” ACM Transactions on Graphics 21, 3 (July 2002), 693–702.

[Horn87] Berthold K. P. Horn. Closed-form solution of absolute orientation using unit quaternions. Journal of the Optical Society of America, A, Vol. 4:629 – 642, 1987.

[Horn90] B. K. P. Horn. Relative orientation. International Journal of Computer Vision, 4:59 78, 1990.

[Horn91] B. K. P. Horn. Relative orientation revisited. Journal of the Optical Society of America, A, Vol. 8, No. 10:1630 – 1638, 1991.

[JSY03] H. Jin, S. Soatto, A. Yezzi, “ Multi-view stereo beyond Lambert”, Proceedings of International Conference on Computer Vision, 2003

[KDR*03] - “Exploring the VLSI Scalability of Stream Processors”, Brucek Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, J. D. Owens, B Towles - Proc. 9th Int’l Symp. High-Performance Computer Architecture, IEEE CS Press, 2003, pp. 153-164.

[KhoGar03] Y. Kho and M. Garland. User-Guided Simplification. In Proceedings of ACM Symposium on Interactive 3D Graphics, April 2003.

[KSW04] P. Kipfer, M. Segal, R. Westermann.: UberFlow: A GPU-based particle engine. In *Graphics Hardware 2004* (Aug. 2004), pp. 115–122

[KipWes05] P. Kipfer, R. Westermann, “Improved GPU Sorting”, in GPU Gems 2, Addison-Wesley, pp. 733-746, 2005.

[KutSei99] K. Kutulakos and S. Seitz, “A theory of shape by space carving,” Proceedings of International Conference on Computer Vision, pp. 307–314, 1999

[Kutulakos00] K. N. Kutulakos, “Approximate N-View Stereo,” Proceedings of the European Conference on Computer Vision, Springer Lecture Notes in Computer Science 1842, Vol. 1, pp. 67-83,   June/July 2000

[KSW04] P. Kipfer, M. Segal, R. Westermann. “UberFlow: A GPU-based Particle Engine”. In Graphics Hardware 2004 (Aug. 2004), pp. 115–122

[KruWes03] J. Kruger, R. Westermann: “Linear algebra operators for GPU implementation of numerical algorithms”. ACM Transactions on Graphics 22, 3 (July 2003), 908–916.

[LarChr04] B.D. Larsen, N. J.Christensen: “Simulating photon mapping for real-time applications”. In Rendering Techniques 2004: 15th Eurographics Workshop on Rendering (June 2004), pp. 123–132

[LawHan95] Lawson, Charles L., and Richard J. Hanson. *Solving Least Squares Problems.* Society for Industrial and Applied Mathematics, 1995.

[LBC*05] A. E. Lefohn, I. Buck, P. McCormick, J. D. Owens, T. Purcell, R. Strzodka "GPGPU: General-Purpose Computation on Graphics Processors" , Tutorial at IEEE Visualization 2005, Minneapolis, Oct. 2005

[Levenberg44] K. Levenberg, "A Method for the Solution of Certain Non-Linear Problems in Least Squares", Quart. Applied Mathematics, 2, 1944, pp. 164-168

[LFW*05] W. Li, Z. Fan, X. Wei, A. Kaufman: "GPU-based flow simulation with complex boundaries". In GPU Gems 2, Pharr M., (Ed.). AddisonWesley, Mar. 2005, ch. 47, pp. 747– 764.

[LiMS03] M. Li, Magnor M., H.-P. Seidel. Hardware-accelerated visual hull reconstruction and rendering. Graphics Interface 2003, pp. 65-71

[LiMS04] Hardware Accelerated Rendering of Photo Hulls, Eurographics 2004

[LKH*04] A. Lefohn, J. Kniss, C.D. Hansen, R. Whitaker, "A streaming narrow-band algorithm: Interactive computation and visualization of level-set surfaces" IEEE Transactions on Visualization and Computer Graphics 10, 4 (July/Aug. 2004), 422–433.

[LKM01] E. Lindholm, M. Kilgard, H. Moreton. "A user-programmable vertex engine", In Proceedings of ACM SIGGRAPH 2001 (Aug. 2001), Computer Graphics Proceedings, Annual Conference Series, pp. 149–158.

[Lok01] B. Lok: Online model reconstruction for interactive virtual environments. *Symposium on Interactive 3D Graphics* , 2001, pp. 69-72

[LouArg04] M.I.A. Lourakis and A.A. Argyros. "The Design and Implementation of a Generic Sparse Bundle Adjustment Software Package Based on the Levenberg-Marquardt Algorithm", ICS/FORTH Technical Report No. 340, Aug. 2004

[LouArg05] M.I.A. Lourakis and A.A. Argyros. Efficient, Causal Camera Tracking in Unprepared Environments. Computer Vision and Image Understanding Journal, 99(2), pp. 259-290, 2005.

[Luong92] Q.-T Luong. Matrice Fondamentale et Calibration Visuel le sur l'Environnement. PhD thesis, Universite de Paris-Sud, Centre D'Orsay, 1992.

[Magda01] S. Magda, D. Kreigman, T. Zickler and P. Belhumeur "Beyond Lambert: Reconstructing Surfaces with Arbitrary BRDFs," Porceedings of International Conference on Computer Vision, vol. 2, pp. 391–398, 2001

[Marquardt63] D.W. Marquardt, “An Algorithm for Least-Squares Estimation of Non-linear Parameters”, J. Soc. Indust. Appl. Math., No.2, 1963, pp. 431-441

[MBM01] W. Matusik, C. Buehler, L. McMillan : Polyhedral visual hulls for real-time rendering. *12th Eurographics Workshop on Rendering*, 2001, pp. 115-125

[MahHeb00] S. Mahamud, M. Hebert, “Iterative Projective Reconstruction from Multiple Views”, CVPR II 430-437, 2000

[MayFau92] S. J. Maybank and O. D. Faugeras. A theory of self-calibration of a moving camera. International Journal of Computer Vision, 8:2:123 – 151, 1992.

[MBR*00] W. Matusik, C. Buehler, R. Raskar, S. J. Gortler, L. McMillan. “Image-based visual hulls” SIGGRAPH 2000, pp. 369-374.

[McLauMur95] P. F. McLauchlan and D. W. Murray, “A unifying framework for structure and motion recovery from image sequences”. Proceedings of IEEE Int. Conf. Computer Vision, pages 314–20, Cambridge, MA, June 1995

[MGA*03] W.R. Mark, R.S. Glanville, K. Akeley, and M. J. Kilgard. *“Cg: A system for programming graphics hardware in a c-like language.”* ACM Transactions on Graphics, 22(3):896--907, July 2003.

[MHO*01] S. Mahamud, M. Hebert, Y. Omori and J. Ponce, “Provably-convergent iterative methods for projective structure from motion”. CVPR I, pp. 1018-1025, 2001

[Microsoft06] D. Blythe, “The Direct3D 10 System”, International Conference on Computer Graphics and Interactive Techniques ACM SIGGRAPH 2006.

[NocWri99] J. Nocedal, S.J. Wright. Numerical Optimization. Springer, New York, 1999.

[Nvidia03] Using Vertex Buffer Objects (VBOs), Nvidia white paper, October 2003

[NV04] NVIDIA Extension: EXT_pixel_buffer_object, March 2004 http://oss.sgi.com/projects/ogl-sample/registry/EXT/pixel_buffer_object.txt

[OLG*05] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E.Lefohn and T. J. Purcell. “A Survey of General-Purpose Computation on Graphics Hardware.” In Eurographics 2005, State of the Art Reports, August 2005, pp. 21-51.

[Oliensis99] J. Oliensis, “Fast and Accurate Self-Calibration”, ICCV pp. 745-752, 1999

[PAB*05] D. Pham, S. Asano, M. Bolliger, M.N. Day, H.P. Hofstee, C. Johnson, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, K. Yazawa. “The design and implementation of a first generation CELL processor” In Proceedings of the International Solid-State Circuits Conference (Feb. 2005), pp. 184–186.

[PDC*03] T. J. Purcell, C. Donner, M. Cammarano, H. Jensen, P. Hanrahan. “Photon mapping on programmable graphics hardware” In Proceedings ACM SIGGRAPH/ Eurographics Workshop on Graphics Hardware (2003), pp. 41-50.

[PoeKan97] C.Poelman,T. Kanade. “A paraperspective factorization method for shape and motion recovery”. PAMI, 19(3):206-218, 1997.

[PolGol97] M. Pollefeys and L. Van Gool . “A stratified approach to self-calibration.” In Proc. International Conference on ComputerVision and Pattern Recognition, San Juan, Puerto Rico, pp.407-412, 1997

[PKG99] M. Pollefeys, R. Koch and L. van Gool, “Self-Calibration and Metric Reconstruction Inspite of Varying and Unkonwn Intrinsic Camera Parameters,” International Journal of Computer Vision, vol. 32, pp. 7–25, Jan. 1999.

[Pollefeys00a] M. Pollefeys, “Obtaining 3D Models With a Hand-Held Camera”, course Siggraph 2000

[Pollefeys00b] M. Pollefeys and L. van Gool, “Some Issues on Self-Calibration and Critical Motion Sequences”, Proc. Asian Conference on Computer Vision, pp.893-898, 2000

[PGV*04] M. Pollefeys, L. van Gool, M. Vergauwen, F. Verbiest, K. Cornelis, J. Tops, and R. Koch.” Visual Modeling With a Handheld Camera.” IJCV, 59(3), pp. 207-232, 2004.

[PVV*04] M. Pollefeys, L. Van Gool, M. Vergauwen, F. Verbiest, K. Cornelis, J. Tops, R. Koch, *Visual modeling with a hand-held camera*, International Journal of Computer Vision 59(3), 207-232, 2004.

[ProDye98] A.Prock and C. Dyer, “Towards real-time voxel coloring”, Image Understanding Workshop, 1998, pp. 315-321

[Purcell04] T. J. Purcell *“Ray Tracing on a Stream Processor”* PhD thesis, Stanford University, Mar. 2004.

[Sainz03] M. Sainz, “3D modeling from images and video streams”, Ph.D. thesis, University of California, 2003.

[SSB03] Sainz, M. Susin, A. Bagherzadeh, N. “Camera calibration of long image sequences with the presence of occlusions”. Proceedings 2003 International Conference on Image Processing, ICIP 2003 pp. I- 317-20 vol.1

[SBS02] M. Sainz , N. Bagherzadeh and A. Susin, “Hardware Accelerated Voxel Carving”, Proceedings of the 1st Ibero-American Symposium on Computer Graphics, 2002, pp. 289-297, Guimaraes, Portugal

[Salamin79] E. Salamin, "Application of quaternions to computation with rotations" Internal Report, Stanford University, Stanford, California, 1979.

[SeiKut98] S. M. Seitz , K. N. Kutulakos, “Plenoptic image editing“, Proc. Fifth International Conference on Computer Vision, pp. 17-24, 1998

[SeiDye97] S.M. Seitz and C.R. Dyer, “Photorealistic Scene Reconstruction by Voxel Coloring,” Proc. CVPR ’97, pp. 1067-1073, Puerto Rico, 1997.

[Schut59] G. H. Schut, "Construction of orthogonal matrices and their application in analytical photogrammetry" Photogrammetria 15, 149-162, 1959.

[SCM*01] G. Slabaugh, B. Culbertson, T. Malzbender, R. Schafer. “A Survey of Methods for Volumetric Scene Reconstruction from Photographs”, *Volume Graphics 2001*, Proc. of the Joint IEEE TCVG and Eurographics Workshop (Mueller, K. and Kaufman, A., eds.), Springer Computer Science, 2001, pp. 81 - 100.

[Shashua95] A. Shashua, “Algebraic functions for recognition”, IEEE Trans. Pattern Analysis & Machine Intelligence, 1995.

[Slama80] C. Slama, Manual of Photogrammetry, American Society of Photogrammetry, Falls Church, VA, USA, 4th edition, 1980.

[SMC00a] G. Slabaugh, T. Malzbender, and W. B. Culbertson, “Volumetric Warping for Voxel Coloring on an Infinite Domain," Proceedings of the Workshop on 3D Structure from Multiple Images for Large-scale Environments (SMILE), July 2000, pp. 41-50.

[SSH02] G.Slabaugh, R. W. Schafer, M. Hans, “Multi-resolution Space Carving Using Level Set Methods”, Proceedings of the International Conference on Image Processing, 2002, pp.545-548.

[Slabaugh02] G. Slabaugh, “Novel Volumetric Scene Reconstruction Methods for New View Synthesis”, Ph. D. thesis, Georgia Institute of Techology, 2002

[StuTri96] P. Sturm and B. Triggs, "A Factorization Based Algorithm for Multi-Image Projective Structure and Motion," in Proc. European Conference on Computer Vision, pp. 709-720, 1996

[TomKan92] C.Tomasi and T. Kanade, “Shape and motion from image streams under orthography: A factorization method” IJCV, 9(2):137-154, 1992

[THS04] A. Treuille, A. Hertzmann and S. M. Seitz, “Example-Based Stereo with General BRDFs”, ECCV 2004

[Triggs95a] B. Triggs, “The geometry of projective reconstruction I: Matching constraints and the joint image”. IJCV 1995

[Triggs95b] B. Triggs, “Matching constraints and the joint image”. In E. Grimson, editor, IEEE Int. Conf. Computer Vision, pages 338–43,Cambridge, MA, June 1995.

[Triggs96] B.Triggs, “Factorization methods for projective structure and motion”, Proceedings of Computer Vision and Pattern Recognition, pp. 845-851, 1996.

[Triggs97] B. Triggs, “Autocalibration and the absolute quadric”, Proceedings of Computer Vision and Pattern Recognition, 1997.

[Triggs00] B. Triggs, P.F. McLauchlan, R.I. Hartley, and A.W. Fitzgibbon, “Bundle adjustment – a modern synthesis”, In Vision Algorithms: Theory and Practice, Lecture Notes in Computer Science 1883. Springer-Verlag, 2000.

[Thompson59] E. H. Thompson, "On exact linear solution of the problem of absolute orientation," Photogrammetria 15, 163-179 (1959).

[VBS*00] S. Vedula, S. Baker, S. Seitz and T. Kanade , “Shape and Motion Carving in 6D,” IEEE Computer Society Conference on Computer Vision and Pattern Recognition, vol. 2, pp. 592–598, 2000.

[Web1] http://www.realviz.com

[Web2] http://www.2D3.com

[Web3] http://www.gpgpu.org

[Web4] http://www.ics.forth.gr/~lourakis/levmar

[Web5] http://loper.org/~matt/Archimedes/millie.zip

[Web6] http://www.robots.ox.ac.uk/~vgg/data1.html

[Web7] http://www.cs.cmu.edu/afs/cs.cmu.edu/project/cil/ftp/cil-0001

[WoeKoch04] J. Woetzel, R. Koch : “Multi-camera real-time depth estimation with discontinuity handling on PC graphics hardware” International Conference on Pattern Recognition, Cambridge, United Kingdom, 2004.

[YPW03] R.Yang, M. Pollefeys, G. Welch, "Dealing with Textureless Regions and Specular Highlights. A Progressive Space Carving Scheme Using a Novel Photo-consistency Measure", ICCV 2003

[YanPol03] R. Yang, M. Pollefeys "Multi-resolution real-time stereo on commodity graphics hardware. In 2003 Conference on Computer Vision and Pattern Recognition, pp. 211-220. June 2003.

[ZacKar04] C. Zach, K. Karner. "Space Carving on 3D Graphics Hardware" VRVis Technical Report 2004-014, 2004

[Zhang98] Z. Zhang "Determining the Epipolar Geometry and its Uncertainty: A Review" International Journal of Computer Vision, 27(2), 161–198 (1998) 1998 Kluwer Academic Publishers, Boston.

[ZMP*03] R. Ziegler, W. Matusik, H. Pfister and L. McMillan, "3D Reconstruction Using Labeled Image Regions", Eurographics Symposium on Geometry Processing , pp. 1–12, 2003.

[ZFC99] A. Zisserman, A. Fitzgibbon, and G. Cross, "VHS to VRML: 3D Graphical Models from Video Sequences," Proc. International Conference on Multimedia Systems, pp. 51–57, 1999.

[ZPA03] Q. Zhou, J. Park, and J. K. Aggarwal, Quaternion-based tracking multiple objects in synchronized videos, Lecture Notes in Computer Science (LNCS 2869), pp. 430-438, 2003.

List of Publications

[Bri03] Felicia Brisc , "Towards Voxel-based Reconstruction from Uncalibrated Views", Eurographics Ireland Workshop, Ulster, Northern Ireland, April 2003

[Bri04a] Felicia Brisc , "A Framework for User-guided Multi-Resolution 3D Reconstruction" IEEE Virtual Reality 04, VR for Public Consumption Workshop, Chicago, USA, March 2004

[Bri04b] Felicia Brisc "Multi-Resolution Volumetric Reconstruction Using Labeled Regions", poster, IEEE Southwest Symposium on Image Analysis and Interpretation, Lake Tahoe, USA, March 2004

[BriWhe04] Felicia Brisc, Paul Whelan "Creating Virtual Models from Uncalibrated Camera Views" Eurographics Ireland Workshop, Cork, Ireland, September 2004

[BBS*04] "Framework and Applications for Mobile Networks using Synthetic Multimedia" Eamonn Boyle, Felicia Brisc, Saman Cooray, Bartek Uscilowski, Andrew

Brosnan, Robert Sadlier, Carol O’Sullivan. 3G2004, 5th International Conference on 3G Mobile Communication Technologies, London, October 2004