Christian Vogelgsang, Ingo Scholz, Günther Greiner, Heinrich Niemann

lgf3 – A Versatile Framework for Vision and Image-Based Rendering Applications

appeared in:
Vision, Modeling, and Visualization 2002 (VMV 2002)
Erlangen, Germany
pp. 257–264

lgf3 - A Versatile Framework for Vision and Image-Based Rendering Applications

Christian Vogelgsang*, Ingo Scholz†, Günther Greiner‡, Heinrich Niemann§

University of Erlangen, Department of Computer Science

Computer Graphics Group
Am Weichselgarten 9
91058 Erlangen, Germany

Lehrstuhl für Mustererkennung
Martensstr. 3
91058 Erlangen, Germany

Abstract

Applications of computer vision and image-based rendering techniques nowadays often share ideas and algorithms. Thus the components required for implementation show a high degree of similarity. Unfortunately this potential is currently hardly used and investigation of new approaches often means a lot of new but redundant work. In this paper we present our approach to improve software development by describing a versatile and comprehensive implementation framework called lgf3. Our goal is to design components capable of realizing existing and future work in this field easily and quickly. Even advanced concepts such as dynamic light fields were already considered during the analysis phase.

1 Introduction

Image-based rendering techniques grew more and more popular in the last years. Both the computer vision and computer graphics communities regard them as a valuable tool for solving problems in different fields. Starting with basic image operations in space (e.g. warping) and ranging to elaborate 4D scene structures like light fields, all approaches have in common that they build their models basically from a set of input images. Despite the theoretical foundation, these methods often require subtle different structural changes and — in lack of a common environment — often enforce an expensive new implementation in each new project.

We initially encountered the same problems while working on our collaborate research project combining computer vision and rendering approaches. Our research focuses on light fields and we started with the first published methods, the light field [10] and the lumigraph [6]. Beginning with a straight forward implementation of the described algorithms, we soon found out that this platform is not versatile enough for future ideas. This first effort resulted in *lgf1* (named after our first file format: **l**umi**g**raph **f**iles). We worked with our initial system and enhanced it until version 2, but with growing complexity of our problems expansion of the system got more and more costly.

Thus it became necessary to design a new implementation framework, named lgf3, which should be versatile enough to handle all our current algorithms with ease and furthermore be a reusable foundation for future work. We based our work on the light field techniques we are currently investigating, but we tried to include a very broad range of image-based applications in our design considerations.

The creation phase of lgf3 is based on the unified process model and UML as standardized by the Object Management Group[1] and widely used for modern software design. The rest of this paper focuses on these strategies embedded in our context of image-based applications.

The next chapter summarizes the analysis stage of the modeling process. Chapter 3 presents the design essentials of the required components in our model. Chapter 4 gives details on the implementation, followed by examples in chapter 5. We finally close with conclusions and an outlook on future work.

*Vogelgsang@informatik.uni-erlangen.de
†Scholz@informatik.uni-erlangen.de
‡Greiner@informatik.uni-erlangen.de
§Niemann@informatik.uni-erlangen.de

[1] http://www.omg.org/uml

2 Analysis

In the analysis stage of development, the application field of the new system is studied and key components required for a versatile system are extracted. It is very useful to have a common nomenclature as it helps identifying similarities and shared concepts. We investigated the common techniques of light field calibration and image-based rendering methods.

2.1 Light Field Calibration

In order to create a light field from a set of input images the camera parameters of each image have to be known. In most of the pioneering works in this area camera locations have been obtained by using sophisticated setups for positioning the cameras at known coordinates in space [10] or by simplifying calibration using calibration patterns in the scene [6]. In contrast to that we follow a scheme introduced in [8] for using arbitrary image sequences taken by – in the most general case – a hand-held camera.

A calibration typically consists of a three-step processing sequence. Starting with point feature extraction, followed by tracking the features and calibration of the cameras, and finally the 3D reconstruction of the scene. The results of these steps were passed between each other. A feedback from later processing steps, even from the renderer, can be used to refine the calibration. Figure 1 shows the processing steps for calibration.

The analysis phase resulted in the extraction of the following key concepts:

- An image sequence is all the data available in the beginning, with each *image* representing a certain *view* of the scene.
- On this image sequence features are extracted and tracked over the sequence, each leaving a so-called *trail* through the images.
- Using these trails the image sequence is calibrated. Each image and the associated view is thus assigned a set of *camera* parameters, extrinsic and intrinsic.
- Each of the trails now represents a *3D point* in the scene, and these points can be used to calculate e. g. *depth maps* for each view using stereo matching.

Further elaboration of this initial structure leads to the following base layout for our component

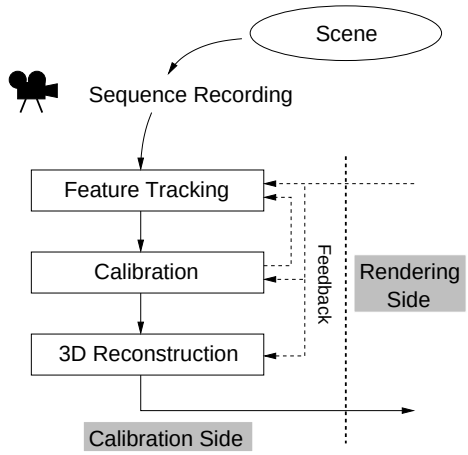

Figure 1: Calibration steps with main data path and possible feedback loops.

model:

- A light field is a set of *views*.
- Each view contains *camera* parameters describing the recording setup for a single image.
- The captured *image* is also contained in the view.
- Additional *depth maps*, usually stored as an image, may be added to a view.
- 3-D *scene points* describe common scene information and should be stored globally.
- A *trail* describes the point correspondences used for reconstructing a scene point and should remember the views where it was tracked in.

This component model for light field calibration will be tested against different rendering applications in the following section and extended to a complete component model in Section 2.3.

2.2 Light Field Rendering

Algorithms and methods for image-based rendering are numerous. We have chosen some and analyzed their requirements and their suitability for our component model.

2.2.1 Warping

Ranging from classical image morphing [12] to various different image warping techniques [3, 4], all these methods have in common that they require a

set of source input images with associated depth information and generate images for novel views out of these values. They reproject each pixel with the depth information into space and then sample this grid in the target image.

For these methods, we can directly share the components of the vision model. The main components required are the *view* storing camera parameters (consisting of intrinsic and extrinsic parameters), the attached image color information and associated depth maps. These techniques fit easily in our current design.

2.2.2 Two Plane Light Fields

The light field as a discrete sample set of the plenoptic function was first introduced with a very restrictive two plane parameterization [10]. It requires a fixed setup of cameras placed onto a plane sharing the same image plane. The lumigraph [6] adds a coarse geometric representation to improve the reconstruction quality.

The main requirement is a strict set of cameras (views) with sheared frustum sharing a single image plane. For rendering the connectivity of the cameras is required and stored in a mesh. In this case the mesh is a simple planar grid. For speed-up techniques [13] and less restricted planar rendering, general planar meshes are required. We need a new component in our model to reflect this information and added the *view mesh* as a container for views and their topology.

We identified the *light field* itself as a special kind of view mesh. It is enriched by adding extra information on the chosen parameterization and will hint rendering algorithms. Each sheared camera is added as a view and stores the image information. Lumigraphs also add a geometric model (e. g. a triangle mesh) or require implicit depth information in per-view depth maps. Geometric models are placed as new global components in our scene model. The depth maps can already be stored in each view, but additionally a *confidence map* can be added to each view to provide information about the reliability of the depth information.

2.2.3 Other Light Field Techniques

Free form light fields [11] and unstructured lumigraphs [1] both relax the strict assumptions posed on the camera layout by the two plane parameterization. They allow an almost free placement of the views. Here, besides the image streams, the depth information and the camera setup, the view mesh for the camera connectivity is essential. By allowing a broader range of view mesh types these models were integrated into our scene model, too.

There also exist some other forms of parameterization for light fields. Spherical [9] or uniformly sampled light fields [2] both use a parameter space which has no immediate mapping onto a 2D image. Other approaches use local parameters on a geometric representation of the scene. The surface light fields [14] and light maps [5] are relevant contributions in this field.

All these techniques use parameterizations which are initially not suitable for storage in 2D images. But all of them try to devise methods that are capable of using rectangular 2D parameters. The main reason for this is the need of high rendering speeds. They are achievable mainly by relying on hardware support and current hardware only supports 2D or 3D rectangular image data. This restriction allows the immediate inclusion of these techniques into our current framework and allows us to focus on 2D images. However our design is not limited to support only this type of image.

2.2.4 Extension of the Component Model

The component model we built up in Section 2.1 can now be extended by the additional components required for rendering:

- *Confidence maps* are added to the views.
- Views are combined to *view meshes* describing the connectivity of the views.
- The *light field* is introduced as a view mesh extended with special parameterization information.

2.3 Component Fusion

Following the component identification we studied a number of possible applications of light fields with regard to the suitability of our current model for their needs. Such applications are for example the use of light fields as models for object recognition [7], as virtual objects in augmented reality or the future use of light fields for modeling dynamic scenes with multiple static light fields.

The conclusion was that the current components are applicable for a wide range of computer vision and graphics applications, but need to be extended by one further component for handling arbitrary numbers of light fields in a scene and linking them together. This *scene database* stores the light field information and global scene data such as trails and scene points which several light fields may have in common.

This last component together with the ones identified in Sections 2.1 and 2.2 constitutes the complete component model the design of our framework is based on.

3 Design

In the design stage each entity will be defined as a class. There the essential functionality is constituted and the interface to other components is defined. Also the relationship and the usage of each object is described.

3.1 Maps

We combine images, depth maps and other per-pixel data structures into the common term *maps*. A map stores a discrete grid of values and has fixed dimension $w \times h$. A single 2D location is called a *cell*. Each cell holds a value of the same type. The type of data in each cell defines the type of the map:

- A *radiance map* stores illumination information. Commonly referred to as a visual image. Either real physical entities (radiance) or values of a color model like RGB, HSV or CMYK are saved.
- A *depth map* stores implicit geometric information in depth values. Either orthographic or ray depth values are commonly used.
- A *confidence map* stores per-pixel weights or quantifiers. These values are useful to judge the validity of depth values in the calibration process and allows the rejection of outliers in the rendering stage.

3.2 Camera

We use the pinhole-camera model which is widely used. Since we try to unify the application base for vision and rendering applications, we have to propose a single *camera* component usable for both parties. We devised a camera object with two representations. The camera is initially defined with one set of parameters and the other format is only created on demand.

In computer vision a camera is usually defined by a 4×3 matrix projecting a point in world space directly onto a pixel coordinate. On request a split set of rotation matrix with translation vector and a projection matrix is provided.

Rendering applications usually describe a camera by a local coordinate system consisting of right, up and direction vector. In this space a frustum window is defined at fixed distance 1 along the z-axis. The frustum window is defined by specifying the bottom left and top right position. For rendering purposes in OpenGL applications the modelview and projection matrix can be calculated quickly from a camera setup.

3.3 View and Layers

A *view* combines a camera with its associated maps. A view can hold a number of maps with different types. We introduced the term *layer* to generalize the concept of maps. A layer also represents a 2D grid of cells but has no knowledge about the underlying storage mechanisms. A map layer always stores all cells but other layer types may store sparse maps for example in a list.

A view holds an arbitrary number of layers of different types. We require every view in a light field to have the same layer arrangement. This simplifies the usage of many views and makes access more efficient. To enforce this rule, we introduced *layer factory* objects that control the creation and destruction of layers for a set of views. Every time a view is added to a light field the factory will automatically add the current layout setup. If you add a new layer to a set of views then the layer factory will create a new layer for each of the existing views.

Different layers of the same type in a view can be used to store hierarchical or multi-resolution data. E. g. multiple depth layers can cope with LDIs (layered depth images).

3.4 View Meshes and Light Fields

As we have observed in the previous section, the different light field rendering approaches need some connectivity information for the views. We introduce the *view mesh* object that interconnects the

views by edges. More topology information can be provided by also specifying triangle information which often helps in rendering algorithms.

View meshes have a wide range of application. Every time some views with connectivity information are required, a local view mesh is created. E. g. view meshes are used in the calibration stage to describe a trail and connect all the views of a tracked feature. The rendering stage creates view meshes — often in real-time — for reconstructing the light field at a given view point.

We define a *light field* object as a view mesh which also controls all contained resources. It manages associated data of the views with all referenced layers. Thus a layer factory is always embedded into the light field. Any other view mesh only references some views for local processing but it cannot control the associated resources.

3.5 Geometric Scene Description

Geometric information about the objects captured in a scene has quite diverse locality of data. Local implicit depth information is stored as per-pixel values in the depth layers. All other global geometry is stored in the scene directly.

An initial geometric data set is created while tracking. Each tracked point is located in 3D space after calibration and we define a *point set* model of the object. This coarse point set can be enriched or exchanged with depth information found in the layers.

Mesh creation techniques are then used to build a topology for the unconnected point set and this usually results in a *triangle mesh*. Our component model also provides triangle meshes with hierarchical setup and supports other types of geometry as well. It is also possible to import geometry from external sources. In some IBR approaches (e. g. [14, 5]) an exact model of the scene is required and acquired externally through spatial measurement like 3D scanning.

3.6 Scene Database

All the described components are stored in a scene database. There a list of light fields and global geometry is found. This is the common scene structure that is available to all algorithms working in the system. Figure 2 summarizes the major entities of a scene.

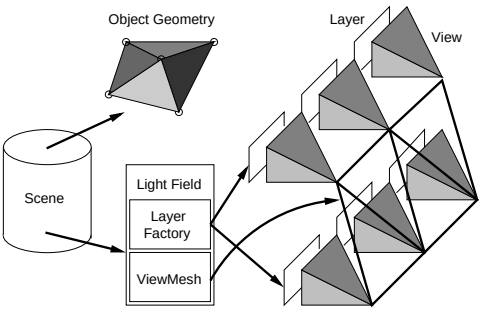

Figure 2: Common components of a scene model. Starting from the scene database, the arrows represent the object interconnections.

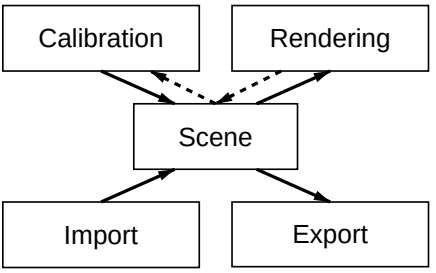

Figure 3: Modules of lgf3. The solid lines denote the main data paths. The dashed lines mark the feedback loop.

The calibration or rendering parts of an application extend the scene and store additional information in it. E. g. the calibration stage adds trail components and interlinks them with existing views and layers.

3.7 Modules

Currently we have only discussed the central data base where all scene information is stored, but not the organization of algorithms creating and working with this data. All methods are sorted by field of application, placed into components and grouped into *modules*. Besides the central module *scene*, four other modules are currently found in our design (cf. Figure 3):

- The *calibration module* combines computer vision methods working on light fields. They are responsible for initial data generation. Their main task is to calibrate a video stream:

An image sequence is fed in and placed into an uncalibrated light field component. The calibration algorithm then supplements each view of the light field with camera parameters and adds an initial point set to the scene.

- The *rendering module* uses the stored scene information to reconstruct novel views of the scene. It may also extend the scene database with render specific information (e. g. render view meshes). A rendering framework is located here and allows easy implementation for a wide range of display methods. Every information available in the scene database can be rendered with purely software based or hardware accelerated OpenGL methods.
- An *import/export module* is used to store and retrieve the scene database to and from external persistent media. Starting with file I/O in different formats this concept is also capable of handling inter process data exchange and supports various network transfer strategies.

When creating or extending modules one must be very careful where to place functionality. In general it is reasonable to place methods of general use and working only on a single or only a few objects directly into the scene components. All other algorithms combining very diverse scene information to solve a problem should be placed into an external module. E. g. a method for depth estimation in a map is placed directly into the depth map layer but a tracking algorithm is stored in the calibration module.

4 Implementation

One of our major concerns while implementing the library was efficient handling of all resources. In image-based applications usually large amounts of per pixel data are required and thus careful management of the maps is very important.

We started with the implementation of a map pool. This structure can store large amounts of maps with the same properties very efficiently in physical memory. With map pools it is still necessary for the application programmer to explicitly allocate and fill these buffers on demand and later store and free maps if memory is becoming scarce. This is very cumbersome to implement and in algorithm development these mechanisms blur the algorithm structure by overcrowding the code with pure management instructions.

To solve this problem we created a map input/output interface that can export and import local map data to and from external storage media in different common formats. This interface is connected to a map pool so that the pool itself can manage the map data. The concept is further enriched by the use of smart pointers to the maps. Now the pointer can automatically query the pool whether a map is available and load it on demand, completely freeing the user from this task.

Map pools are embedded into light fields and control the resource of one set of layers in all views.

All other resources in the scene structure are negligible in size compared to the amount of image data and thus are kept permanently in physical memory. Only the number of geometric data may grow very fast, but in this case we prepared the same techniques as applied for maps.

5 Examples

In this section we will present a selection of components we have created with our framework. We will show how difficult tasks are simplified and how especially a cooperate project like ours can profit from a common base framework.

5.1 Example 1: Tracking Features

As described in Section 2 the primary task for calibrating an image stream is the generation of point correspondences between frames by conducting feature tracking. A typical tracking system in lgf3 consists of the components shown in Figure 4 and is integrated in the calibration module.

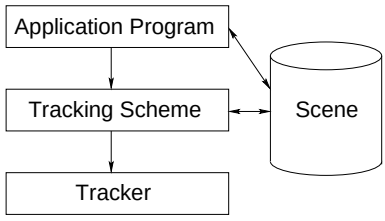

Figure 4: Setup of a tracking application.

The bottom of the application hierarchy consists of the tracker itself, which is responsible for extracting new features from an image and tracking

them from one image to another. For these tasks the tracker has access to an extensive library containing structure-from-motion algorithms. Although it has no knowledge about the scene itself, it can reference the set of trails stored in the scene object, and is given the view object of each image, from which it can extract all necessary data like the image itself.

The next hierarchy level is a control layer which handles the scene object and the access to its data. Different schemes of tracking over a sequence can be implemented as different aspects of this control layer. Since the tracker itself only tracks from one image to another, the tracking scheme decides on the sequence of tracking. Three different types are currently implemented. Linear tracking which only tracks once from the beginning to the end of an image sequence, bilinear tracking which also tracks backwards in a sequence, and finally a tracker which is given a known view mesh and tracks features from one view to all neighbouring views.

5.2 Example 2: Rendering

In the following we will have a look on the inner workings of the rendering module by describing the integration of a free-form light field renderer in our system with the provided renderer interface.

Every renderer module has access to a global scene object. This is either constructed by a connected calibration module or restored from external storage with the import module. Now the rendering environment is created by opening a render window on a GUI or by using the direct-to-memory image renderer interface with helper classes available in the rendering module. The renderer module is then initialized. The free-form light fields are selected from the scene and the associated view meshes are transformed into a render view mesh.

The renderer is then ready for processing. The controlling application sends two important signals to notify the renderer. One requests a new reconstruction at a novel camera position and another one signals a change in the scene database. The first one triggers the hardware based rendering algorithm using OpenGL and 2D texturing or a software based renderer drawing only pixels. The second signal forces a rebuild of the render view mesh.

The actual rendering requires the image information from the view layers and geometric object information for depth correction. Now the powerful resource handling concept comes in handy. While

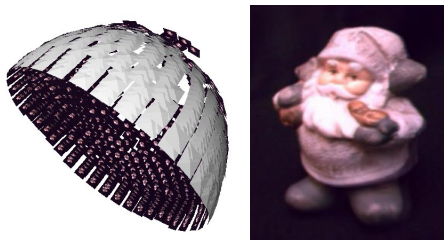

Figure 5: An example scene reconstructed and rendered with lgf3. All used views with their frustums are depicted on the left. A novel view from a free-form light field model of the scene is on the right.

the radiance maps are accessed the underlying map pools handle memory allocation and resource import and export. The performance of these operations can be improved with map caches.

For hardware accelerated rendering using OpenGL, a texture manager is available. This tool shuffles the radiance map information onto the graphics card and keeps track of the texture memory usage. The renderer only selects maps as textures and the manager loads and uses the right map in an efficient way.

5.3 Module Coupling

The use of a common component model and furthermore the same code base make it very easy for computer vision and computer graphics applications to cooperate. Extra flexibility is provided by allowing the user to choose the degree of connectivity. Loose coupling is established by using the same library in two rather distinct projects and by exchanging all data using shared import and export modules. This is a very powerful approach while testing and debugging parts of a larger system. In a later stage of development you can strengthen the coupling of the modules by simply compiling the code into a single application and by replacing the file based import/export with in-memory data transfer strategies.

Figure 5 illustrates the coupling process with a sample scene. First the calibration module reads the input images and creates the initial scene components. The result after calibration is depicted on the left: each view has a valid camera location and orientation. The scene data base is then accessed by a free-form light field renderer: a novel view point

is chosen and the new view is rendered interactively (image on the right).

6 Conclusions and Outlook

We think it is important to investigate the algorithmic infrastructure of commonly used image-based vision and rendering approaches since the process of unifying the environment has many advantages. It helps generalizing and merging different approaches and is a very valuable common framework for future development. Our platform allows fast and easy testing of novel ideas but also enables the implementation of robust applications.

We have presented the core ideas of our library called lgf3 and showed the steps we have taken to build a versatile structure. The different stages in development, namely analysis, design, implementation and example usages were discussed.

For the future there are two main directions of work. One goal is to integrate more calibration and rendering techniques in our framework, and to extend the system by exploiting the possible feedback paths and adding new approaches such as the dynamic light field. By including current and novel techniques we can study if our framework is versatile enough or needs adjustment. In addition to that we want to promote the usage of the library and see if the framework is as useful to a wide range of applications as it is to our work.

Acknowledgements

This work was supported by the Deutsche Forschungsgemeinschaft under grant SFB 603/C2. Only the authors are responsible for the content.

References

[1] C. Buehler, M. Bosse, L. McMillan, S. Gortler, and M. Cohen. Unstructured lumigraph rendering. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 425–432. ACM Press, 2001.

[2] E. Camahort, A. Lerios, and D. Fussell. Uniformly sampled light fields. In *Rendering Techniques '98 (Proceedings of Eurographics Rendering Workshop '98)*, pages 117–130, New York, NY, 1998. Springer Wien.

[3] S. E. Chen. QuickTime VR — an image-based approach to virtual environment navigation. *Computer Graphics*, 29:29–38, 1995.

[4] S. E. Chen and L. Williams. View interpolation for image synthesis. *Computer Graphics*, 27:279–288, 1993.

[5] W.-C. Chen, J.-Y. Bouget, M. H. Chu, and R. Grzeszczuk. Light field mapping. In *Proceedings of SIGGRAPH 2002*, 2002. to appear.

[6] S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen. The lumigraph. In *Proceedings of SIGGRAPH '96*, pages 43–54, 1996.

[7] B. Heigl, J. Denzler, and H. Niemann. On the application of lightfield reconstruction for statistical object recognition. In *European Signal Processing Conference*, pages 1101–1105, 1998.

[8] B. Heigl, R. Koch, M. Pollefeys, J. Denzler, and L. Van Gool. Plenoptic modeling and rendering from image sequences taken by a hand–held camera. In W. Förstner, J.M. Buhmann, A. Faber, and P. Faber, editors, *Mustererkennung 1999*, pages 94–101, Heidelberg, September 1999. Springer.

[9] I. Ihm, S. Park, and R. K. Lee. Rendering of spherical light fields. In *Proceedings of Pacific Graphics '97*, pages 59–68, 1997.

[10] M. Levoy and P. Hanrahan. Light field rendering. In *Proceedings of SIGGRAPH '96*, pages 31–42, 1996.

[11] H. Schirmacher, C.Vogelgsang, H.-P. Seidel, and G. Greiner. Efficient free form light field rendering. In T. Ertl, B. Girod, G. Greiner, H. Niemann, and H.-P. Seidel, editors, *Workshop Vision, Modeling and Visualization*, pages 249–256,528, Saarbrücken, Germany, Nov. 2001.

[12] S. M. Seitz and C. R. Dyer. View morphing. *Computer Graphics*, 30:21–30, 1996.

[13] P.-P. Sloan, M. F. Cohen, and S. J. Gortler. Time-critical lumigraph rendering. In *1997 Symposium on Interactive 3D Graphics*, pages 17–24, 181. ACM SIGGRAPH, April 1997.

[14] D. N. Wood, D. I. Azuma, K. Aldinger, B. Curless, T. Duchamp, D. H. Salesin, and W. Stuetzle. Surface light fields for 3d photography. *Proceedings of SIGGRAPH 2000*, pages 287–296, July 2000.