NOTE TO USERS

University of Alberta

RENDERING OF MULTI-RESOLUTION GRAPHICS MODELS CAPTURED FROM IMAGES

by

Keith Yerex 

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2005

Your file Votre référence
ISBN: 0-494-09320-X
Our file Notre référence
ISBN: 0-494-09320-X

To Jule King,
I love you lots and stuff.

Abstract

In both computer graphics and computer vision, scenes and objects must be represented mathematically in some way. Representations tailored to ease scene acquisition in vision, may be inefficient to render. Likewise, models designed specifically for efficient rendering may be difficult, if not impossible, to capture from cameras. In this thesis, we explore potential graphical models which are both convenient to acquire from images, and well suited for real-time rendering using modern graphics hardware.

Our representation is a hierarchical hybrid image/geometry based model for graphics objects and scenes. Each of three scales of detail macro, meso, micro is represented differently in an attempt to most efficiently store, render and capture the model. Macro scale is the large scale or overall shape, which we represent using a low-resolution polygonal mesh. Meso scale detail, which is defined loosely as lying between macro and micro, is represented using displacement mapping, for which a novel hardware accelerated rendering algorithm is presented. At the micro scale, light-surface interaction properties are represented using a parameterized texture mapping technique called dynamic texture, where a linear texture basis is precomputed from example images and blended when rendering to reproduce different lighting conditions.

A system was built and used to capture this type of model from real-world objects, which were then rendered, and integrated into virtual scenes. Additionally, experiments were performed which evaluate each subsystem independently.

Acknowledgements

Dr. Martin Jägersand initially developed the dynamic texture idea.

Dr. Dana Cobzas implemented the structure from motion algorithm which we used.

Alias/Wavefront provided Maya licenses free of charge which were used in our system.

Neil Birkbeck implemented some parts of the system including the automatic texture coordinate generation, camera calibration, and user interface.

Table of Contents

List of Figures

Chapter 1

Introduction

The level of detail attainable by real-time rendering engines is quite incredible, and increasing at an astounding rate. Texture mapped triangles have been, and for the most part still are, the basic geometric primitive with which these detailed scenes are modeled. However, it is beginning to become clear that the continued increase in triangle mesh resolution is no longer the most efficient way to increase the realism of such scenes. The main reason for this is that rendering hardware has evolved such that most of the processing power power is available at the pixel level. Simply increasing geometric resolution does not take full advantage of this hardware.

Recently, video game engines have begun to diverge from the philosophy that more triangles necessarily make a more realistic image. The Doom III engine from ID software, for example, is heavily focused on realistic shading using per-pixel lighting and bump-mapping on nearly all surfaces, and extensive dynamic shadowing [28]. While using relatively few polygons, the realism achieved by this game is very impressive.

When graphics models are acquired from real world scenes and objects for the purposes of efficient rendering, the same principles should apply. Although it may be possible to capture an entire object or scene geometrically with 100 micron precision using a laser scanner, it would be both difficult to perform the scan, as well inefficient to render the resulting model. Instead, using image data for smaller scale details can be a more effective use of acquisition time, memory, and rendering time. Image data could be not only in the form of basic texture-mapped images, but also various other image-based representations, such as the parameterized textures that we will describe in Chapter 4.

Pure image-based rendering methods take this even further suggesting that entire scenes or objects can be represented solely by image data, but this requires very many images in the same way that pure geometric models require so many polygons. Clearly, modeling absolutely everything with polygons is not particularly efficient, but it is also problematic to model scenes entirely with images; it seems that some mix of geometric and image-based models would be most efficient.

There are three scales that are often discussed separately in computer graphics, as they can be most efficiently rendered in different ways: Macro, Meso, and Micro. This is similar to how, in physics, problems of a certain scale, like the trajectory of a baseball, can be solved with a simple formula (a mathematical model), but when the same problem is scaled up to the trajectory of a planet, a more complex model is required. We will illustrate each scale, by referring to how one would model a brick wall.

The *macro* scale is the largest scale, and corresponds to the overall shape of the object, such as a single rectangle for our brick wall. This scale is most often represented with polygons, as it will

continue to be in this thesis. Other possible structures for macro scale modeling are parametric and implicit curved surfaces [18].

In the middle, we have the *meso* scale. At this scale, we model the details like the indentations between bricks, imperfections and cracks in each brick etc. This scale is not strictly defined: the size of so called details could vary significantly depending on the application. Most current rendering engines mainly use bump mapping to render details of this scale, which approximates the effect of small bumps using only shading [2]. Some more advanced techniques, which will no doubt be used in future engines, include self shadowing bump mapping [40], parallax mapping [55], and displacement mapping [9].

The *micro* scale is the smallest scale, and represents how light interacts with the surface at a microscopic level. This means, given an input light direction, color and intensity, what light color and intensity is emitted toward the camera. This is a function called the Bidirectional Reflectance Distribution Function (BRDF). Methods for simulating this complex physical process vary widely. The simplest, and thus most common, real time methods use some parameterized lighting equations such as Phong or Lambertian [48]. Although efficient, these equations are approximate, and it can be difficult or impossible to tune the parameters and achieve a desired result. It is possible to compute an approximate BRDF from a set of images, but it is quite difficult, since this is a function of five dimensions: incoming and outgoing light direction, each of which have two degrees of freedom, and light wavelength (which is generally ignored). Acquired BRDF representations may be either approximate, or inefficient to store and render, so the use of such models has remained primarily a research area, and has not yet made it's way into commercial systems.

In this thesis, we will present a tiered model which is designed considering both model acquisition from images and efficient real-time rendering. Our system represents each of the macro, meso, and micro using different data structures, each of which can be acquired reasonably from images or other sources. The method balances well the efficient use of commercial rendering hardware with the ease of automated model construction.

We represent the macro scale using a triangular mesh. Two methods are described in Chapter 3 that we have used for acquiring low resolution meshes from images. Structure from motion (SFM) uses feature correspondences over a sequence of video frames to extract scene geometry with no camera calibration required. Shape from silhouette (SFS) is a very robust algorithm which uses an object's silhouette in multiple views combined with calibration information for each view to compute the *visual hull*, an approximation to the object's shape.

Meso scale structure is represented in one of two ways. In our first system, we use a texture pa-

rameterized by viewing direction. *Dynamic Textures* are used to store and render the parameterized texture efficiently as shown in Chapter 4. Although this method works well for very small geometric details, it becomes less efficient with more reasonable meso scales, and large ranges of viewing angle. The second system stores geometric details in a displacement map: a texture image where each element encodes the distance from the macro scale approximation to the true object along the surface normal at that point. The displacement mapped surfaces are rendered using a novel algorithm for per-pixel hardware accelerated displacement mapping which is described in Chapter 5.

Finally, the micro scale or surface reflectance model is represented using *Dynamic Textures*. Here, we parameterize a texture in terms of lighting conditions. This type of texture variation can be represented very efficiently by modulating a basis of images. We sample large range of lighting conditions, but compress the information down into a small texture basis. In Chapter 4, the technique is discussed in detail.

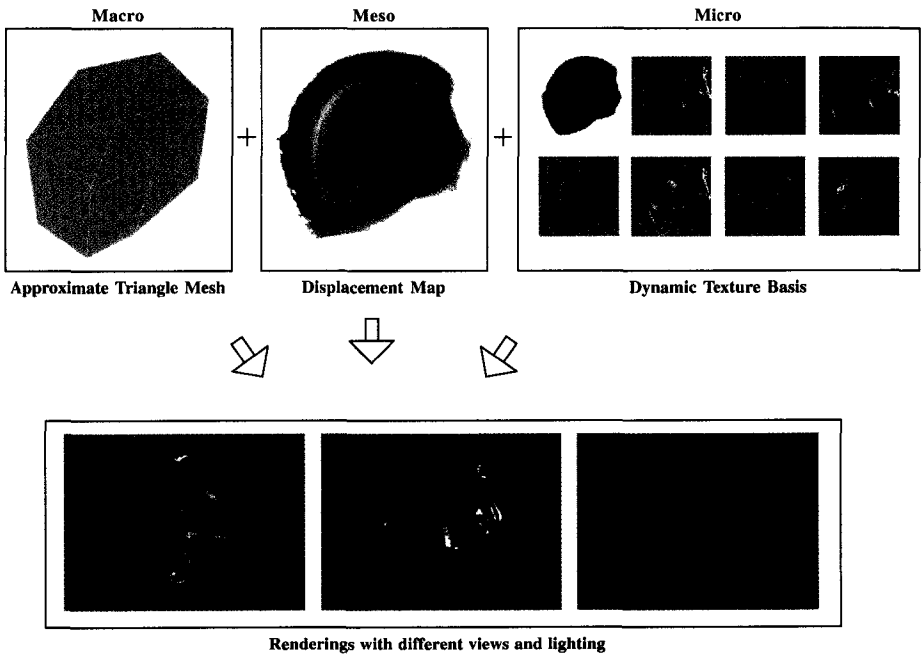

Figure 1.1: Overview of how the Macro, Meso, and Micro scales are represented, and resulting renderings

Two systems were developed. One using a 2 tiered system based on a polygonal mesh combined with dynamic textures. The other has three tiers, combining a polygonal mesh with displacement mapping and dynamic textures. These systems are presented in Chapter 6. In addition, each technique has also been evaluated individually with experiments shown in their respective chapters.

Thesis Contributions

Parts of this thesis were published in seven papers and presentations in SIGGRAPH [58], EuroGraphics [56, 27, 7], IEEE Virtual Reality [4, 8], and the IEEE International Conference on Robotics and Automation (ICRA) [57].

- Several hardware accelerated dynamic texture rendering algorithms were developed and implemented on various examples of consumer graphics hardware. (Section 4.5).
- Techniques for the acquisition and storage of dynamic textures were developed. (Chapter 4 and Section 6.1).
- An easy-to-use system was built for acquiring dynamic textured models from only 2D images (Section 6.1).
- A plug-in was built for AliasWavefront's Maya 3D modeling package. This allows users to render and animate our dynamic textured models alongside conventional models. (Section 4.6)
- A novel displacement mapping algorithm was designed for modern graphics accelerators and implemented in hardware as a fragment program (Chapter 5).
- A hierarchical graphical model was designed as the combination of geometry, displacement mapping, and dynamic texture. (Chapter 1)
- A system was built to acquire hierarchical models of objects and efficiently render them (Section 6.2).

Chapter 2

Review: Graphics Modeling

In computer graphics, mathematical models of the world are required in order to generate synthetic images of it. Traditionally, scenes have been represented by a variety of geometric primitives such as polygons or curved surface patches. Building these geometric models manually can be time consuming, even for experienced artists. More recently, building geometric models automatically from images, or even working with models defined by images, with no explicit associated geometry has become a popular area of research known as image-based modeling and rendering (IBMR). Although commercial graphics software is still largely focused on geometric models, different forms of image data, such as texture maps, has been slowly making its way into the traditional graphics pipeline.

In this chapter, we very briefly describe conventional methods used for modeling graphics scenes. Although we attempt to provide some context in the area of computer graphics modeling, we do not go into extreme detail. For a more complete overview of image-based and geometric models, respectively, see [18] and [5].

2.1 Conventional Models

Traditional graphics models represent the world as some set of geometric constructs. The geometry is rendered by simulating or approximating the physical processes that occur when a real photo taken.

2.1.1 Polygonal Models

The most commonly used geometric model is the polygonal mesh. Surfaces are represented by a set of connected polygons. 3D graphics accelerators and ray-tracing software generally focus on efficiently rendering triangles, so any higher order polygons in a model are usually triangulated before rendering. For the same reason, curved surfaces are often sampled discretely and also rendered as triangular meshes.

Polygonal meshes are often used as piecewise planar approximations to smooth surfaces. For lighting purposes, it is assumed that the normal to the true surface is computed by interpolation (either spherical or linear) of the surface normals at polygon vertexes. Vertex normals can be derived from the function defining a curved surface, but if such a function is unavailable, they are approximated by a weighted average of all polygons containing that vertex.

2.1.2 Curved Surface Models

Curved surfaces can be represented as implicit or parametric functions.

Implicit surfaces are of the form $S(x, y, z) = 0$. It is simple to represent a sphere, torus, cylinder, or other geometric primitives in this way. More complex shapes are usually generated by CSG (Constructive Solid Geometry) operations between primitives such as union, intersection, etc. CSG operations are simple to implement with implicit surface models, making them popular with some modeling software. However, rendering these surfaces is not so simple. With ray-tracing, it is possible to simply intersect rays directly with an implicit surface by solving a system of equations, but it is not necessarily efficient. Hardware accelerators only render triangles, so implicit surfaces must be converted to triangular meshes before rendering, which can be costly.

Parametric surfaces are formulated as $[x, y, z] = P(p, q)$, where $p, q \epsilon [0, 1]$ parameterize its surface. This formulation is popular because of convenience in modeling, as well as simplified rendering. Commonly used parametric curves are mostly cubic polynomials: bezier, hermite, NURBS, etc. Parametric surfaces form patches which are used together to build an object, similar to how polygons are used in polygonal models. It is much easier to convert parametric curved surfaces into triangular meshes, making them more useful for real-time use. Most recent graphics hardware actually supports tessellation of some kinds of parametric surfaces.

2.1.3 Lighting

In model-based graphics, lighting is usually performed by evaluating a parameterized lighting equation, either at each vertex or each rendered pixel and modulating that result with the surface color.

Ambient Ambient lighting is simply a constant amount of light, used to approximate global illumination effects.

$$L = a \tag{2.1}$$

Here, a is the amount of ambient light.

Diffuse The diffuse or Lambertian lighting equation simulates lighting from rough surface. In this model, light incident to the surface is emitted equally in all directions.

$$L = a + d(n \cdot l) \tag{2.2}$$

Here, a is the amount of ambient light, d is the amount of diffuse light, l is the light direction, and n is the surface normal.

Specular Shiny surfaces can be simulated by having some amount of light reflect specularly (like a mirror). The Phong lighting equation is an example of such a model.

$$r = 2(n \cdot l)n - l \tag{2.3}$$

$$L = a + d(n \cdot l) + s(r \cdot v)^n \tag{2.4}$$

Here, a is the amount of ambient light, d is the amount of diffuse light, l is the light direction, n is the surface normal, v is the view direction, s is the amount of specularity, r is the reflection vector, and n controls the size and intensity of specular highlights.

2.2 Image-Based Models

IBMR (image-based modeling and rendering) research seeks to simplify the modeling process, at least for scenes and objects that exist in the real world, which we may simply want to capture for re-rendering. There are two categories of IBMR algorithms: methods which simply use images to generate a conventional 3D model that is then rendered with standard methods, and methods which acquire some other sort of model which requires specialized code for rendering.

Many algorithms exist for modeling conventional geometry from images. Stereo or multi-view stereo imaging computes depth images by finding image point correspondences in two or more images from calibrated cameras [47]. Shape from shading and photometric stereo solves for a depth image using one or multiple lightings of a scene [59]. Shape from focus acquires depth information using multiple images from the same view with different focal settings [43]. Structure from motion (SFM), like stereo, uses multiple image point correspondences to build 3D models. However, many image frames from a moving camera are used, and camera calibration is not required. A basic SFM method is shown in Section 3.1.

Shape from silhouette (SFS) uses the occluding contour of an object in multiple images to construct a volume approximating that object: the *visual hull*. We will discuss SFS in more detail in Section 3.2. Photo-consistency methods, use multiple images and the property that a particular point on an object should be the same color when viewed from any direction. With this property the *photo-hull* of the object can be "carved" from a volumetric structure [29].

Although the research in modeling from images is extensive, most methods are neither computationally efficient, nor particularly robust. This is part of the motivation behind pure image-based rendering, where the intermediate geometric model is eliminated, and novel images are generated more directly from sample images.

Most image-based models can be thought of as storing a large database of light rays which are acquired from some set of sample images. New images can be generated by finding a ray in the database that corresponds to each pixel in the desired image. The set of all rays through a scene with a given origin and direction is called the plenoptic function [41]. Pure image-based methods usually take samples of the plenoptic function for a scene or object using many images, where each image pixel samples a single ray. Models differ in the way that the large amount of data is organized and how the function is approximated between samples. Early methods of this type include lumigraph [20] and light-fields [31], which have led to many different varieties. Rendering new views using this type of model can require relatively little computation, but the memory requirements for approximating the 5D plenoptic function are very large. To reduce this effect achieving equivalent photo-realism, image-based data is often combined with approximate geometric models.

2.3 Hybrid Models

In real systems, pure geometric models are rarely used since it would be difficult, or near impossible, to model small scale details such as surface texture using polygons. However, due to their incredible storage and acquisition requirements, pure Image-Based Models are not very practical either. Some combination is usually used, giving rise to hybrid methods.

Although most conventional modeling and rendering software is geometry-based, texture mapping is always supported. Texture mapping is where images are simply pasted onto geometric models. This allows small and sometimes repetitive details to be represented much more efficiently than with polygons. Texture maps can be thought of as storing spacially varying albedo (or diffuse reflectance color). Texture mapping has been extended to vary other parameters in the lighting equation, such as surface normal (as in bump maps), specularity, or glow.

On the opposite side of the spectrum, some image-based rendering algorithms are based on reprojection of images with depth. This means that at each pixel we know the depth of that part of the scene along the camera's viewing direction, and thus we know some geometric information about the scene.

More in the middle lies view-dependent texturing, where different textures are applied to a surface depending on the viewing direction. This reduces the visual effect of slight inaccuracies in the geometric model [12]. The bidirectional texture function (BTF) takes this one step further by sampling textures for variation in both the viewing direction and the lighting direction. This takes quite a bit of memory and acquisition time, but allows a real-world texture to be captured and applied to an arbitrary geometric model [11]. In Chapter 4 we describe our own approach to this type of parameterized texture rendering and we detail the storage and rendering optimizations that were made.

2.4 Graphics Hardware

The need for specialized graphics hardware grew primarily out of the complexity of a few particularly computationally expensive problems such as perspective correct texture-mapping, and visible surface determination. Early graphics chips, such as the 3Dfx Voodoo, solved only these problems related to the rasterization of triangles. These include texture-mapping, color interpolation, alpha blending, and depth buffering.

In the second generation of graphics processors (now called GPUs) vertex transformations, lighting and triangle clipping were performed in hardware. In the third generation, programmability was added to the pipeline. *Vertex programs*, which allow precise control of how vertexes are transformed, and *fragment programs* which compute the color of each pixel were added. The most recent iterations of the GPU have seen primarily an increase in the complexity of fragment and vertex programs.

The GPU is an incredibly parallel architecture. Since both vertexes and pixels are not allowed to rely on any particular ordering of computation, many vertexes and pixels can be processed at once. For example, the NVidia GeForce 7800GTX can process 24 pixels and 8 vertexes simultaneously.

A slightly simplified flow chart of the hardware graphics pipeline is shown in Figure 2.1. The following is a brief description of that hardware model. For a more detailed examination of graphics hardware see [1].

Vertex data including position, and optionally other attributes such as texture coordinates and color, is stored in a vertex buffer. A second buffer called an index buffer stores indexes into the vertex buffer indicating which vertexes are connected to form triangles.

The GPU reads vertexes from the vertex buffer and passes them to a *vertex program.* The vertex program usually transforms each vertex from its local object coordinate system into the global world

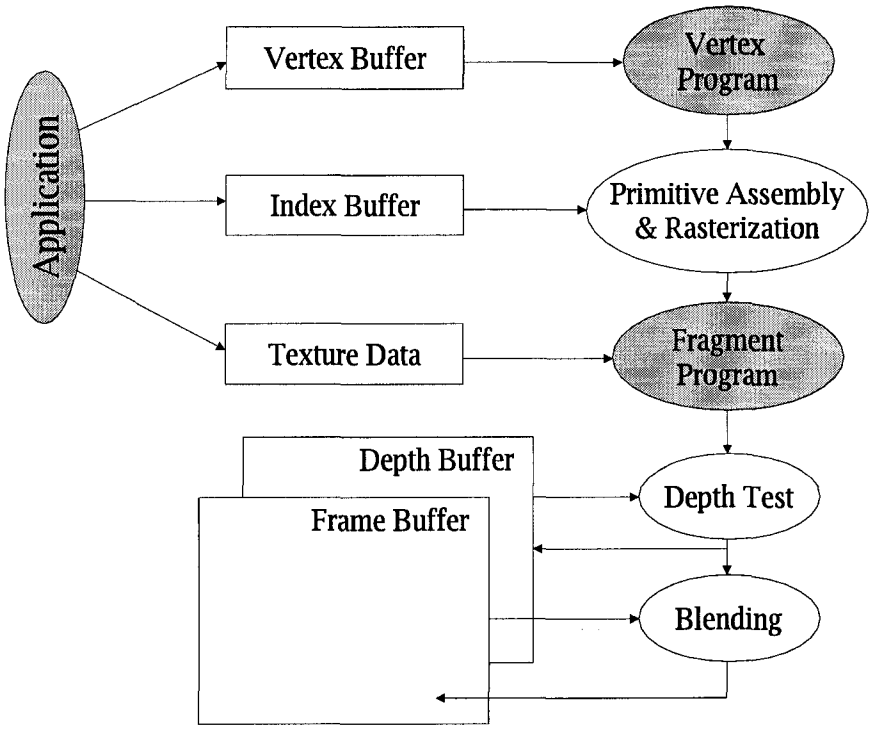

Figure 2.1: Simplified view of the graphics pipeline. Programmable components are shaded in gray

coordinate system, and then projects the vertex onto the virtual camera. However, the vertex program is written by the user, and can perform any operation desired. The vertex position in camera space must be output from the vertex program; a number of additional 4D attribute vectors can be output which will be interpolated between triangle vertexes during rasterization.

Vertexes output from the vertex program are then assembled into triangles which are clipped to the screen and rasterized. The rasterizer finds each pixel inside a triangle, computes interpolated attributes for that pixel, and passes them as parameters to a *fragment program*. The fragment program computes a color - and optionally depth - for each pixel.

The depth of the current pixel is compared against the current value in the depth buffer to see if it is visible. If the depth test fails, the pixel is not rendered. If it passes, the depth is written to the frame buffer, and the color is sent on to blending. The blending stage reads the color at the current pixel location from the framebuffer, combines it with the rendered color, and writes it back to the framebuffer. Usually the 4th component of the color vector, called the alpha, is used as an opacity to render translucent surfaces, but other blending functions are available.

Chapter 3

Review: Building Geometric Models from Images

With the rapidly decreasing prices, and increasing quality of consumer grade digital imaging devices, geometric modeling from images has become a very active research area. Suddenly, a large number of consumers own digital still cameras, video cameras, and web-cams making applications of passive vision technology reasonable consumer products. Applications such as personal captured custom character models in games, automatic modeling of household objects for use in home design software and many others would be commercially viable if they were robust and easy to use. Now that the required hardware is affordable, research into how to accomplish these tasks on available hardware has suddenly become very interesting.

Here, we describe two algorithms for acquiring geometric structure from images, and show results from each method. Structure from motion (SFM) uses corresponding feature points in a video sequence to compute geometric information, while shape from silhouette (SFS) uses the silhouette of an object in several views to approximate its shape. Some models acquired using each algorithm will be shown, and the two methods will be compared and contrasted.

The geometric models acquired by these methods will be used as the first level of detail in our hierarchical model, representing the macro scale features. Following chapters will describe how micro and meso scale information is represented and rendered.

3.1 Structure from Motion

Structure from motion (SFM) is a method for acquiring geometric information from a sequence of images. SFM is similar to stereo methods, but uses multiple images from a single camera rather than simultaneous images from multiple cameras. The main advantage of SFM methods is that they require little or no camera calibration, which can be tedious. A disadvantage of this method is that, due to the difficult task of feature correlation or tracking, the result is a quite sparse geometry. Since we only use this as the lowest detail level in our hierarchical model, this is not much of a problem in our case.

Uncalibrated SFM solves for both camera calibration and geometric structure in one step. Such a problem is very difficult using an accurate model of the imaging process, but with some approximate camera models, the solution is simple.

The simplest camera models are parallel projection models. This type of camera does not model the convergence of rays to a focal point in a true perspective camera and is therefore only reasonable in certain situations. In particular, parallel projection models are appropriate for long telephoto lenses or when the object of interest is always near the central axis, which is true in our experi-

ments. Although it is possible to perform SFM using more accurate perspective projection models, the parallel projection model is used here for simplicity [24, 45]. We will describe in detail the implementation of SFM using the *orthographic* camera model [51]. In this type of camera, after rotation and translation into the camera's coordinate system, a 3D point is projected onto the camera by simply removing it's Z coordinate.

First, we must choose a set of feature points on our object or scene which are all visible in every sample image. Then we need to find the 2D positions in each image where these features project. This can be done in various ways; we have used visual tracking. Using the XVision2 tracking libraries, we track several manually selected feature points in real time during image acquisition [30].

We compute the average feature position in each image and subtract it from all the feature points in that image so that translations can be ignored. After removing translation, a single point can be projected onto image i by multiplying by a 2x3 matrix P_i which is simply a 3D rotation matrix with the third row removed.

$$x_{i,j} = \begin{bmatrix} P_{i,1,1} & P_{i,1,2} & P_{i,1,3} \\ P_{i,2,1} & P_{i,2,2} & P_{i,2,3} \end{bmatrix} X_j \tag{3.1}$$

A set of m points can be projected onto image i with a single matrix multiplication:

$$\begin{bmatrix} x_{i,1} & x_{i,2} & \cdots & x_{i,m} \end{bmatrix} = P_i \begin{bmatrix} \hat{X}_1 & \hat{X}_2 & \cdots & \hat{X}_m \end{bmatrix} \tag{3.2}$$

Finally, all m points in n images can also be projected using a single matrix multiplication:

$$\begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,m} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n,1} & x_{n,2} & \cdots & x_{n,m} \end{bmatrix} = \begin{bmatrix} \hat{P}_1 \\ \hat{P}_2 \\ \vdots \\ \hat{P}_n \end{bmatrix} \begin{bmatrix} \hat{X}_1 & \hat{X}_2 & \cdots & \hat{X}_m \end{bmatrix} \tag{3.3}$$

$$x = \hat{P}\hat{X} \tag{3.4}$$

Note that the $\hat{X}_j$ are 3x1 vectors, the $x_{i,j}$ are 2x1 vectors and the $\hat{P}_i$ are 3x2 projection matrices.

So, the x matrix contains our known feature positions in each image after removing translations. The rest of the equation is unknown. In order to compute the 3D structure X and the camera orientations P, x must be factored. Using SVD, we decompose x into USV^T. Ideally, using an actual orthographic camera, x would be rank 3 and thus have only 3 non-zero singular values. Due to noise and the approximate camera model, this may not be the case, but we still expect the 3

largest eigenvalue-eigenvector pairs to capture the geometric structure. So we drop all but the first 3 columns of US and all but the first 3 rows of V^T. Then $\hat{X} = V^T$, and $\hat{P} = US$.

This first step is simple, but we are not finished. As stated earlier, each P_i is 2 rows of a 3x3 rotation matrix, meaning that those two rows should be orthonormal. The solution found with SVD is an *affine* structure, with no constraints on the $\hat{P}$ matrix. However, we can enforce Euclidean constraints by using the SVD solution as a starting point for a non-linear optimization algorithm.

First we must convert each matrix $\hat{P}_i$ into a 3x3 matrix $\tilde{P}_i$ by computing the third row as the cross product of the first two. Then we solve for a single $3x3$ matrix Q which, when multiplied by any $\tilde{P}_i$, results in an orthonormal matrix $P_i = \tilde{P}_i Q$. Then the X matrix is multiplied by the inverse of Q, $X = Q^{-1}\hat{X}$, so that $P_i X = \tilde{P}_i Q Q^{-1}\hat{X}$ for all i. To compute Q, we use a non-linear solver with an error metric which should be zero when all P_i are orthonormal.

This algorithm gives us a set of Euclidean 3D points. To form a polygonal model, we have often simply performed a delaunay triangulation on the average projected positions of all points. If the results are not exactly as desired, the user can manually modify the triangulation. Since the number of polygons on this type of model is typically very low, this is not usually a difficult task.

3.1.1 Results

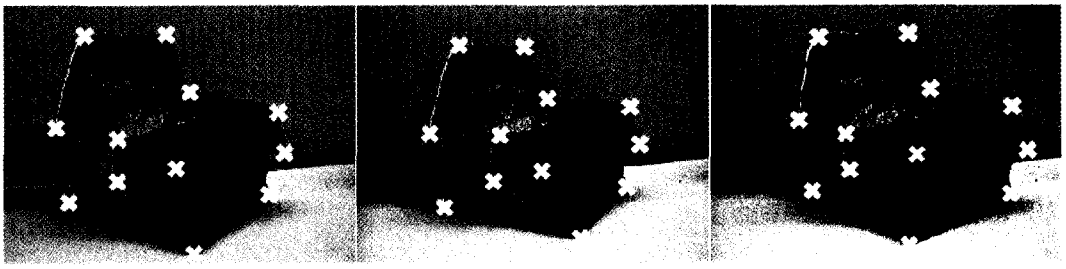

Figure 3.1: A few sample inputs to the SFM algorithm with tracked feature positions

Here is an example of structure captured using this technique. Figure 3.1 shows three of many input images with the tracked features marked. A rendering of this house after capturing its geometry using SFM is shown in Figure 3.2 along with the triangulation which was generated manually to make a 3D surface out of the set of points.

Texture patches are tracked in video to identify correspondences using XVision2, a tracking library for C++ [30]. These patches must be sufficiently textured and locally unique in order to track robustly. These types of patches are difficult to identify automatically, and sparse on most objects. For this reason, we have users manually select points of interest. This results in sparse models, but where features lie on important points on the model, such as the corners of the house in Figure 3.1.

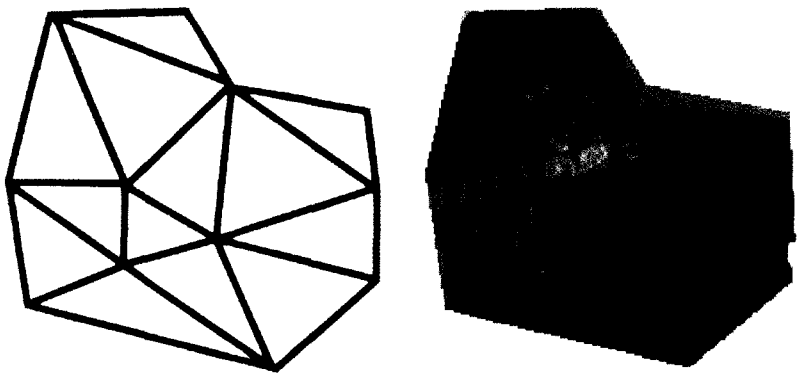

Figure 3.2: SFM results triangulated geometric structure and a rendering (rendering uses view dependent dynamic texture)

3.2 Shape from Silhouette

The silhouette of an object s_v is the set of points where the object projects onto an image plane from a particular view. A single silhouette contains a large amount of information about the shape of the object. Given s_v, and camera parameters v we know that the object is inside the volume that the silhouette cone S_v defined by all rays from the camera center passing through all points in s_v. Since this holds for all views, given a set of views V, we can constrain the object's volume further, to the intersection of all silhouette cones: $\bigcap_{v \in V} S_v$. The limit of this volume as the number of distinct views $|V| \rightarrow \infty$ is known as the *visual hull* of the object, provided that no views in V are are centered inside the convex hull of the object. From a finite number of views, the volume acquired is called the approximate visual hull, but we will simply refer to it as the visual hull.

The process of computing the visual hull from a set of images is known as shape from silhouette (SFS). Intersection of arbitrary shapes can be very expensive, but by taking advantage of some key features of silhouette volumes, and the intended application, the visual hull can be computed efficiently, in some cases in real time. In the following, we classify methods by image-based (2D to 2D), model-based (2D to 3D) and by data structure.

3.2.1 Silhouette Extraction

To extract shape information from silhouettes, the silhouettes must first be extracted from images. This is the problem of segmenting an object from the background. Two methods are commonly used: background subtraction and blue-screening.

We use the following statistical color segmentation method. The user selects some region(s) in some example image(s) to identify a set of pixels in the solid colored background. Principle

components analysis is applied to the selected RGB pixels, giving us a space that is well aligned to the statistical properties of the selected background pixels.

We build a matrix C where every row is formed by subtracting the mean color value $\bar{c}$ from a pixel in the selected set. Then we find the eigenvectors E_C and eigenvalue matrix λ_C of the covariance matrix C^TC. We then use $E_C^T\lambda_C^{-1}$ as our new color space. To perform the segmentation, we apply a threshold τ to each pixel's norm after being transformed into the new color space. Background pixels satisfy equation 3.5.

$$|E_C^T\lambda_C^{-1}(c - \bar{c})| < \tau \tag{3.5}$$

Normally, a small number of pixels are misclassified. SFS is very robust to pixels falsely marked as foreground by color segmentation. Pixels incorrectly labeled as background would cut holes through our results, but other noise is simply removed by silhouette volumes from different views.

3.2.2 Intersection Algorithms

There are three classes of algorithms used in SFS: volumetric methods, polyhedral methods, and image-based methods. Volumetric methods use some kind of discretized volume representation, such as a voxel grid, and perform intersections discretely in that space. Polyhedral methods perform exact intersections of multiple polyhedral meshes. Image-based methods only compute new views of the visual hull given input silhouettes.

Polyhedral Methods

The best possible model of the visual hull that we can compute from a set of images is the exact polyhedral visual hull. This could be computed naively by forming polyhedra from each silhouette and intersecting them all in 3D using a general CSG algorithm. However, due to the projective nature of the silhouette volumes, much more efficient algorithms exist. Matusik et al. [39] has recently proposed a very efficient algorithm which will be briefly described here.

First, each silhouette image is converted into a set of line segments separating the object from the background. These edges form a set of polygons with possible holes, that represent the occluding contour. The resolution of the edges is chosen based on curvature, so that regions of higher curvature in the contour are represented with more edges than regions of lower curvature. The number of polygons in the resulting model can be adjusted by controlling the number of edges in the input silhouettes, as well as the number of input silhouettes.

Each edge generates a 3D polygon of a silhouette cone when it is back-projected with its corresponding camera parameters. It is important to note that every such polygon will generate one or more polygons in the output model. Each polygon is projected onto each image plane, intersected with the silhouette polygon for that image (in 2D). Both polygons are possibly non-convex with holes. The resulting 2D polygon intersection is projected back onto the plane in 3D.

The resulting set of polygons are the polyhedral visual hull. Each polygon in the resulting model has its own copies of all vertices, so, in a final step, duplicate vertices are removed.

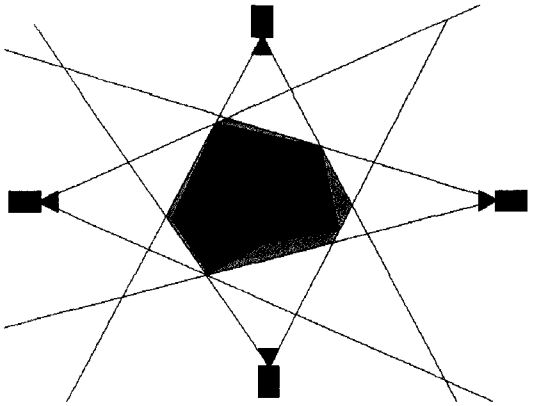

Figure 3.3: Polyhedral Visual Hull: the bounding polygons of the volume are calculated

Although this algorithm is very efficient compared to the naive implementation, when using large numbers of images, as our we have in most of our experiments, performance would be much slower than volumetric methods. Because of this, in addition to the implementation complexity of exact polyhedral algorithms, we have favored volumetric methods.

Volumetric Methods

The easiest way to intersect volumes efficiently is by quantization of the 3D space. By representing the space as a finite number of basic elements, it becomes easier to enumerate which portions of the space are inside or outside the object. The obvious down side is that with quantization comes aliasing. If any of the details of the object are smaller than the size of a single volume unit, they will be lost. However, quantization is also a means for adjusting the speed vs. quality trade-off by changing the quantizing resolution.

This method was first used for SFS by Martin et al. [37] in 1983. They represented the volume as a grid of parallel rays with points of volume entry/exit recorded for each ray. It was implemented with an orthographic camera model, but easily extends to perspective.

The simplest way to quantize a volume is to split it into a 3D grid of equal-sized cubes, called

voxels or volume elements. This is the simple 3D extension of the way images are stored as an array of pixels. We have used a more processing and memory efficient method for quantization called marching intersections [46].

Developed recently by Tarini et al. [46], the marching intersections data structure (MI) was initially used for performing boolean operations, fixing errors, and adjusting polygon count in polyhedral models. It was shown to also be particularly efficient structure for use in SFS [42].

The MI data structure consists of three sets of rays, each parallel to one of the three axes (X Y or Z). For each set, there is an NxN grid of rays, so they combine to form an NxNxN cube. Each ray stores all points along its path where the object being represented is entered or exited. This representation contains all the information of a voxel grid, plus it stores exact surface intersection points along each ray, yet uses only $O(N^2)$ storage (since each ray only passes through the object a small number of times). The reason for the name marching intersections is that this data structure stores the exact information necessary to render the surface properly with marching cubes (no interpolation necessary) [35].

Boolean operations (such as intersection) on the using the MI data structure are reduced to 1D operations between rays. So in order to perform silhouette intersections, each silhouette cone can be converted to MI, and then all the MI cones can be intersected with each other easily by performing intersections on all the rays. In practice, a single MI structure can be updated iteratively to save memory.

Efficient intersections of a MI structure with a silhouette cone are performed as follows. Each ray is projected onto the image. The rays are "drawn" using the Bresenham line algorithm [3], and points where the line crosses from a background pixel to a silhouette pixel, or from a silhouette pixel to a background pixel are noted. Each of these intersection points is then back-projected to form a ray, which is intersected with the current ray being processed, and the intersection point is stored on that ray in the MI data structure.

Further performance increases are achieved by noting that when using a high resolution MI data structure (say 512x512x512), many of the parallel rays project to the exact same 2D line in the image. Therefore, the 2D intersection points of that line with the image don't need to be recalculated (intersection of the back-projected ray with the 3D ray still does need recalculation). In [42] a cache is used to store this data, and the resulting improvement in performance is significant, making the method scale much better to higher resolutions than other algorithms.

One of the problems with this method is that the MI data structure can easily become in conflict with itself. Errors are the result of floating point error, or aliasing problems as a result of intersecting

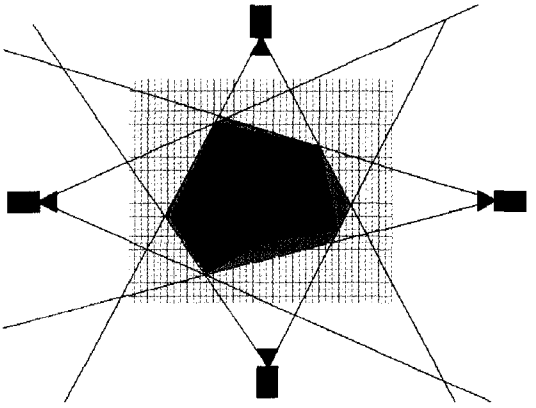

Figure 3.4: Visual hull with marching intersections: ray intersections of a grid pattern with the volume are stored

lines with the 2D silhouette image. The problem is that one set of rays may indicate that a point is inside the volume, while another ray set says that point is outside the object. This is not a problem to identify, but removing such inconsistencies is non-trivial. When rendering a structure with errors, the marching cubes algorithm may get to a cube which intersects the surface and should be rendered but all the intersection points are not available. In this case, unavailable points must be interpolated from available data.

This method was used in combination with dynamic texturing for the system described in Section 6.1.2. Figure 3.5 shows an example of a model captured with this algorithm.

Image Based Methods

For real time applications, it is often the case that you would like to capture a model, and then display it from an arbitrary view-point immediately (possibly after transmission over a network). For these applications, as you can imagine, the 3D model can only be viewed from one view-point at a time. So every time the 3d model is computed, it is only used once. The idea with image-based methods is to generate particular view of the model, without the extra step of creating a 3D representation.

Matusik et al. [38] was the first to develop an image-based SFS. The method is exact to the resolution of output image. Each pixel is traced back along a ray, and if that ray intersects the visual hull then the pixel is rendered.

The HAVH (hardware accelerated visual hulls) algorithm developed by Li et al. [33] makes use of consumer graphics hardware to render arbitrary views of the polyhedral visual hull directly from the input silhouettes. Originally, the number of reference images was limited by the number of texture units available on the hardware, but with increasing programmability, this is no longer a constraint since many images could be placed in one texture, and unlimited texture accesses are

available in shader model 3.0.

From each silhouette image a 3D cone is generated as a polyhedron. Every polygon on every silhouette cone is then rendered. The trick is that all the silhouette images are applied to all the cones with projective texture mapping (using the projection matrix from their corresponding cameras). These textures have alpha=1 inside the object and alpha=0 outside, and are used as a mask that eliminates the portions of each cone that do not lie on the surface of the visual hull. All polygons are still rendered entirely, but only the correct parts of them actually generate pixels in the image. Some results using this algorithm are shown in Figure 3.6.

3.2.3 Results

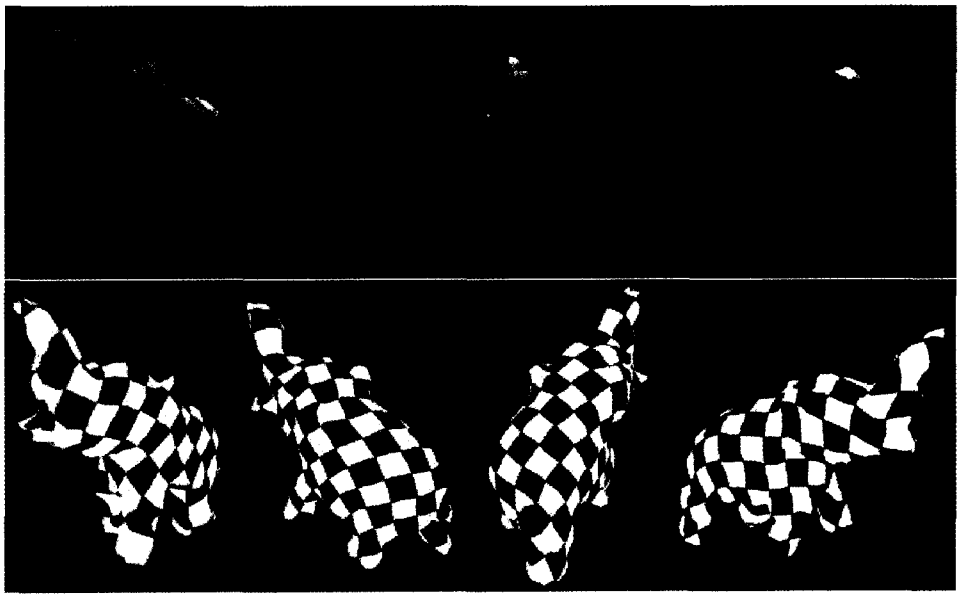

Figure 3.5: An elephant captured using SFS with a stationary camera and turntable. A few of the input images are shown on top, and the reconstructed elephant is shown on the bottom.

Figure 3.6: Results using the HAVH on algorithm on a hand with three cameras

Here we show a couple of examples of some objects captured with shape from silhouette. Fig-

ure 3.5 shows an elephant which was captured on a turntable. Several images were taken from a stationary camera as the object was rotated on a turntable. The geometry shown was captured using the marching intersections SFS algorithm.

In Figure 3.6 we see a visual hull reconstruction using far fewer images. Only three cameras are being used here. However, in this case, the visual hull is being constructed and rendered on the same machine in real-time using the HAVH algorithm.

3.3 Discussion and Comparison

Although both SFS and SFM have been used in this research, we have favored the silhouette-based approach. For a small object capture system, SFS is much more robust and simpler from a user's perspective. Minimal to no user input is required for a turntable system.

In our SFM system, the user must identify the points to be tracked, then carefully move the object or camera around on a tripod, then check the automatic triangulation and potentially change it. All that work and you can still only capture a small variation in viewing angle before some of the tracked points are lost. With the SFS system, the user simply identifies the background color, and the approximate location of the object in two images, and a full rotation around the object can be captured. In addition, SFS results in a much more accurate and detailed model.

The SFS system, however, can never be expected to capture a scene, like the interior of a room for example, which is possible with SFM. And SFS requires camera calibration in the form of a calibration pattern in each image. Each technique has its benefits, but for a small object capture system, with potentially unskilled users in mind, we have found SFS to be much more successful.

For the SFS intersection algorithm, we have used the marching intersections method in the system in Section 6.1.2. This algorithm was chosen for the following reasons: it is nearly as easy to implement and efficient as a voxel-based method; it uses less memory ($O(n^2)$ vs $O(n^3)$); resulting structure is more accurate due to the exact intersections.

Chapter 4

Dynamic Texture

4.1 Previous Work

Conventional graphics represents micro scale detail using parameterized lighting equations which attempt to simulate the surface's reflectance properties. The Phong lighting equation shown in Equation 2.4 is a classic example. More generally, the function being approximated by such models is called the bidirectional reflectance distribution function (BRDF). The BRDF gives the proportion of light transfered in the direction of the camera, given the direction from the light. For a real object, the reflectance properties often vary continuously over the surface. The bidirectional texture function (BTF) [11] is texture parameterized by viewing and lighting direction, which represents a spatially varying BRDF as well as parallax, self-occlusion, and self shadowing effects.

The BTF can be represented by just storing a large database of images. Our method, the *dynamic texture*, is a method for representing and rendering parameterized texture maps, such as the BTF, more efficiently [7]. Related work includes polynomial texture maps, which compress parameterized textures by storing only the coefficients of a biquadratic polynomial at each pixel in the texture [36]. Several other methods, including ours, use a linear basis to represent texture variation.

Freeman et al. showed that an image basis can be used to create the perceived effect of motion [19]. Jagersand showed how a basis can be used to represent motions as complicated as articulated agents (directly in the image plane with no geometry) [25, 26]. Certain types of animated textures such as water waves and fire were rendered using a basis and animated in eigenspace by Soatto et al. [50, 14]. These animated textures were also called dynamic textures, but not to be confused with our method by the same name.

The eigentexture method uses an image basis to represent lighting on a precise 3D model. Tensor-textures use a multilinear basis [52] to store the full BTF, with both view and lighting variation. Tensor-textures are interesting because they separate the lighting and view variation in a way that we can use a small number of basis elements for lighting variation, and a large number for view variation, resulting in better compression for the same quality. However, to capture a tensor texture, a full set of matching lighting conditions is required for every view. This is difficult to achieve with a real system, and so their implementation uses only computer generated inputs.

Our method, *dynamic textures*, can represent both view and lighting variation including lighting, self-shadowing, parallax, and self occlusion with a single linear basis. Where other methods often use planar material samples or acquire the basis in image space, we acquire our dynamic textures directly from real objects using approximate geometric models as shown in Sections 6.1.1 and 6.1.2 [6, 7]. In addition, our method, achieves additional compression by using the YUV color space as

described in Section 4.3. Other methods either ignore color, working in grayscale, or use the less optimal RGB color space.

4.2 Theory

A simple method for implementing any type of parameterized texture mapping is to store a large number of textures, by observations made with different known parameters, and when rendering just pick the observed texture with the nearest parameter. A slightly more intelligent method is to blend linearly between a few observed images with similar parameters (images observed from similar viewpoints in the case of view-dependent textures).

With dynamic texturing, instead of simply storing textures with various known parameters and interpolating them, we compute a linear basis from all observed textures, and blend this basis using interpolated coefficient vectors to generate new textures. Depending on the texture, the derived basis may have far fewer elements than there are observed textures, using much less memory, while still spanning the same texture variation.

To motivate the derivation, consider interpolated view dependent texturing as it is described in [12]. When rendering a new view with parameters identical to one of the original sample views, I_k, the texture derived from image I_k, $T_k = w(I_k)$, is used to texture the model, where w is a warp function defined by a 3D triangular mesh with texture coordinates.

At all other viewing positions, some linear blending of near views is used, with a vector of weights x based on their similarity to the current view. This can be expressed mathematically by a matrix multiplication, where the columns of T contain the sample views $T = [I_1, I_2, ..., I_m]$.

$$t = Tx \tag{4.1}$$

The major variability in T is due to geometric parallax error and illumination differences. Through an analytical derivation, a first order linear basis can be found to represent these types of variability [49, 21, 7]. This means that for large image sets, we can find a new basis B with far fewer columns than T, such that $T \approx \tilde{T} = BY$. Textures are then generated as $t = BYx$, and the number of basis images, and overall memory consumption is reduced.

While B could be be computed analytically, given exact geometric knowledge of the scene, camera and lighting, this is seldom feasible in image-based approaches. Instead we use the knowledge that there exists a subspace spanning T to obtain the best (in the least square sense) B through Principle components analysis. We calculate M as the eigenvectors of $T^T T$. A dimensionality reduction

is achieved by using only the first n eigenvectors $M_{1..n}$. Thus, our texture basis is $B = TM_{1..n}$ and our coefficients are $Y = M^T_{1..n}$.

To estimate the coefficients for intermediate poses, we interpolate between the coefficients of sampled poses. For efficient implementation, cubic interpolation is applied during preprocessing, and results are stored in a set of 2D look-up tables (one for each basis image) which map view direction to blending coefficient. Entries in the blending tables are then bilinearly interpolated during rendering.

The benefits of using the dynamic texture basis rather than standard view-dependent texturing, is that significantly less storage is required. The down-side is that every element of the texture basis will have an effect on the result at any view, instead of only the nearest few. This will require a little more work when rendering. However, in current graphics architectures, bandwidth and memory limitations are a greater problem than computation when it comes to image based rendering methods.

4.3 Color Space

The mathematics of this section allow for color dynamic textures by flattening entire images, including color dimensions, into single column vectors. However, we prefer to perform the process separately on color channels to improve compression by considering human perception. In general, we are more sensitive to high frequency intensity changes than we are to high frequency color changes, which is exploited by most image compression, video compression, and even analog television broadcast standards.

We use the YUV color space, where the Y channel is intensity, and U and V are color channels. By performing the PCA separately on the Y, U, and V channels, we are free to use both different resolutions and different numbers of basis images for each channel. We generally use many high resolution basis images for Y, while using few low resolution basis images for U and V. This results in both additional data compression, and improved rendering performance, with little or no difference in the resulting image quality.

4.4 Per-Pixel Rendering

When rendering a dynamic textured model with some parameters (viewing direction, light direction, etc.) we have some options as to how to set these parameters: we can choose to use the same parameter for a whole object; we can compute individual parameters for each polygon on the model; or, ideally, each pixel in the final rendered output will have it's parameter individually computed.

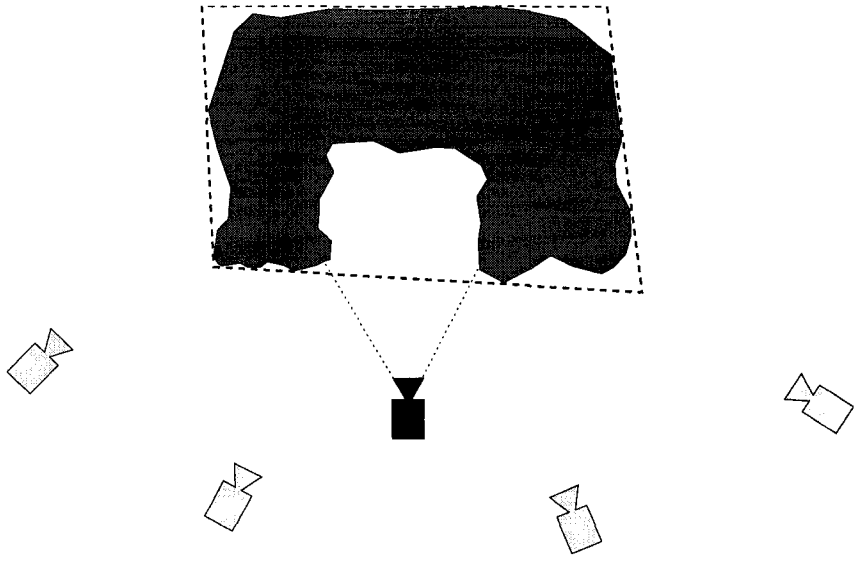

Figure 4.1: An example where per pixel dynamic texture rendering is necessary. The solid gray object is the true object. The dashed quadrilateral is the approximate geometry that the dynamic texture will be applied to. The light gray cameras indicate sampled viewing directions, the black camera is the new desired view. Note that none of the sample views can see the entire indentation while the new view can.

Consider view-dependent texturing of the object as depicted in Figure 4.4. None of the 4 camera views from which sample textures were taken can simultaneously see all sides of the indentation, yet the entire indentation is visible in the desired new view. Clearly using one viewing-direction parameter for the whole object, or for each individual polygon will not accurately render the new image. Per pixel accurate rendering is required. Although this scenario may seem convoluted, and appear to be solved by simply adding more cameras, note that if the cameras are distant relative to the size of indentations, which is generally the case, this situation will occur often.

To achieve per-pixel rendering, we first assume the sample images are taken from a distant camera (this assumption is roughly valid if we use a telephoto lens at capture time). Then, when rendering new views, at each pixel, we use the direction of the ray from the virtual camera center through the current pixel as the viewing direction parameter to the dynamic texture at that pixel. With no rendering speed constraints, this is simple, and our Maya plug-in, discussed in Section 4.6, easily accomplishes per pixel rendering. In order to apply this technique in real-time, we can use the algorithm shown in Section 4.5.3, which stores coefficient lookup tables in textures, and indexes them with the view direction at each pixel.

4.5 Hardware Accelerated Rendering

Rendering of dynamic textures consists of blending a potentially large basis of images. These operations are very well suited for implementation in graphics hardware. Through the rapid progression of graphics hardware during this research, several hardware implementations were developed.

4.5.1 Multi-pass Blending

The first and simplest method is to use frame-buffer blending. This method is supported by even very old graphics hardware, but requires the use of RGB color space, not taking advantage of the memory benefits of YUV as described in Section 4.3.

Image blending features are intended to enable rendering of transparent surfaces and other effects in standard model-based 3D graphics applications. However, most hardware is capable of performing more general blending, including the combination of scaled basis textures required for our rendering. However, the rendering hardware used is designed for textures containing positive values only, while the spatial basis B is a signed quantity. We rewrite this as a combination of two textures with only positive components:

$$I(t) = B^{+}\mathbf{y}(t) - B^{-}\mathbf{y}(t) + \bar{I}_q$$

Here B^{+} contains only the positive elements from B (and 0 in the place of negative elements) and B^{-} contains the absolute values of all negative elements from B. Then, before drawing each basis texture, the blending mode can be set to either scale by a coefficient and add to the frame buffer, or scale by a coefficient and subtract from the frame buffer (depending on the sign of the coefficient). A new view is rendered as in the following pseudo-code:

```
// draw the mean
BindTexture(Ī);
DrawGeometry();

// add basis textures
for(each i)
{
  SetBlendCoefficient(|y_i(t)|);
  BindTexture(B_i^+);
  if(y_i(t) > 0) SetBlendEquation(ADD);
  else SetBlendEquation(SUBTRACT);
  DrawGeometry();

  BindTexture(B_i^-);
  if(y_i(t) > 0) SetBlendEquation(SUBTRACT);
  else SetBlendEquation(ADD);
  DrawGeometry();
}
```

4.5.2 Multi-pass with Programmable Hardware

As the pixel pipeline began to become programmable, with *Shader Model 1.0* cards, we took advantage of several useful features: Textures could now be stored as signed 8 bit values, cutting memory consumption in half; multiple textures could be combined in a single pass, increasing performance, and reducing the chance of overflow; enough processing was available to convert between color spaces during rendering enabling us to take advantage of YUV color space as described in Section 4.3.

In this implementation, each RGBA texture image represents four basis textures from a single color channel (Y,U or V) scaled and biased to fit in the range (0,1). In each rendering pass, as many basis images as possible (four times the number of available texture units) are multiplied by their coefficients, the results are summed, and multiplied by the row of the color conversion matrix that applies to the current color channel. Between passes, we use OpenGL blending to add/subtract results with the frame buffer contents. Since signed frame buffer blending is still not supported, we are still required to render one pass for addition and one for subtraction for all passes except the first. Some simple effects, such as relighting, can easily be achieved in a single pass with just a handful of basis vectors.

Three implementations of this algorithm were written: one using the OpenGL shading language(GLSLang), one with ARB_fragment_program 1.0, and one with NVidia's register combiners. The fragment program and GLSlang shaders, which run on shader model 2.0 and 3.0 hardware respectively, also benefit from floating point computation within the shader, avoiding any overflow or underflow within each pass. The GLSLang shader is shown here:

```
uniform vec4 colormat;
uniform vec4 coeff[gl_MaxTextureUnits];
uniform sampler2D tex[gl_MaxTextureUnits];
varying vec2 texcoord;

void main (void)
{
  vec4 tval;
  float col=0.0;
  for(int i=0;i<gl_MaxTextureUnits;i++)
  {
    tval=2.0*(texture2D(tex[i],texcoord)-0.5);
    col+=dot(tval,coeff[i]);
  }
  gl_FragColor = col*colormat;
}
```

Pseudo-code for rendering with this shader is given here:

```
for(each color channel B,y)
{
  for(i = 1 to N/(4*MaxTextureUnits))
  {
    for(j = 1 to MaxTextureUnits)
    {
      SetShaderConstant(coeff[j], y_{4*i}(t), y_{4*i+1}(t), y_{4*i+2}(t), y_{4*i+3}(t));
      BindTexture(j, B_{4*i:4*i+3});
    }

    SetBlendEquation(ADD);
    DrawGeometry();

    for(j = 1 to MaxTextureUnits)
    {
      SetShaderConstant(coeff[j], -y_{4*i}(t), -y_{4*i+1}(t), -y_{4*i+2}(t), -y_{4*i+3}(t));
    }

    SetBlendEquation(SUBTRACT);
    DrawGeometry();
  }
}
```

4.5.3 Single Pass Rendering

Although the method in Section 4.5.2 is very efficient even on up-to-date hardware, we have implemented a third method which will blend even a large basis in a single pass. Although this method is slightly slower, it is cleaner, and easier to use since all blending is done in one place. Overflow and underflow are completely avoided since all blending computations take place in floating point. It also looks up coefficients per pixel, rather than having the application place them in constants, which could be used to vary the interpolation if parameters (viewing, lighting etc.) change per pixel.

Shader model 3.0 hardware has no limits on the number of texture accesses within a shader. However, there are limits to the number of textures that can be bound at once, and limits to the number of constants available. To implement blending within a single pass we have tiled basis textures and coefficient interpolation tables into six large textures: three for the Y,U and V bases, and three for the Y, U, and V interpolation tables. During rendering, an index into the coefficient interpolation

tables, the number of basis images, and the color matrix are provided by the application, and the geometry is simply rendered. The GLSlang shader is shown below.

```
uniform mat4 colormat;
uniform sampler2D B[3]
uniform sampler2D L[3]
uniform vec3 count;
varying vec2 texcoord;
varying vec2 lutcoord;

void main (void)
{
  vec2 tc,tc2;
  vec2 lc,lc2;
  int i,j;
  vec4 col={0,0,0,0};
  vec2 scale;
  vec4 lutvalue;
  vec4 basisvalue;


  for(int channel=0;channel<3;channel++)
  {
    scale.x=1.0;
    scale.y=1.0/count[j];

    tc=texcoord*scale;
    lc=lutcoord*scale;

    for(i=0;i<count[j];i++)
    {
      lutvalue=2*texture2D(L[channel],lc)-1;
      basisvalue=2*texture2D(B[channel,tc)-1;
      col[channel]+=dot(lutvalue,basisvalue);
      tc.y+=scale.y;
      lc.y+=scale.x;
    }
  }

  gl_FragColor = colormat*col;
  gl_FragColor.a=1;
}
```

4.6 Maya Plug-in

In addition to the various hardware accelerated implementations, a dynamic texture rendering plug-in was implemented in software for AliasWavefront's Maya modeling and rendering system. Using this plug-in, the user can add any dynamic textured objects to a conventional graphics scene, and even use the animation tools to make movies using dynamic textured objects combined with anything else Maya can render. Integrating our system into commercial rendering software makes our research much more accessible to the mainstream computer graphics community.

This rendering plug-in was used, together with a capture system based on shape from silhouette, for rendering view-dependent dynamic textured artifacts in a virtual heritage setting. A set of carvings depicting traditional Inuit seal hunting was captured using shape from silhouette with dynamic

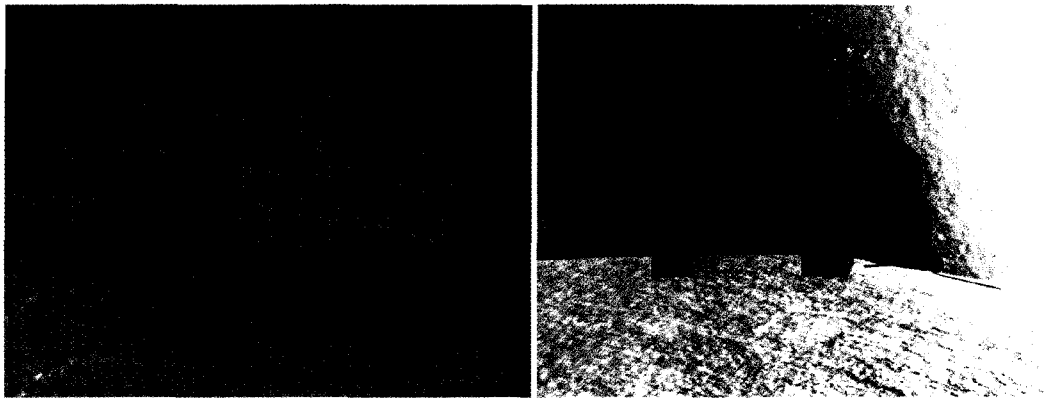

Figure 4.2: Two renderings from Maya that mix dynamic textured objects captured from images with conventional hand modeled objects.

textures. The resulting models were then combined with traditional geometric models in Maya and animated to tell a historically interesting story [17]. An interactive informational website, and a short movie were created using this technology. Figure 4.2 shows two renderings from this project showing some carvings captured using our system, and placed into a scene with conventionally modeled background objects.

4.7 Experiments

Here, we describe a few experiments which test the dynamic texture independent from the other components of the system presented in this thesis.

4.7.1 Light Variation

Figure 4.3: A face lit using a dynamic texture with five basis images.

A set of images with different known lighting directions can be acquired using a hemispherical contraption with flashes mounted at various positions. By synchronized triggering of the flashes and a single camera, images with many different lighting conditions, and identical viewing conditions can be captured in a matter of seconds. Using a set of images captured with such a device, we have built a dynamic texture parametrized in lighting direction. Since light variation is easily represented

by a basis, the renderings shown in Figure 4.3 are generated using only 5 basis images, even though there were more than 60 sample images. The renderings were generated in real time very efficiently, requiring only two 3-component dot products and one addition per pixel (possible in a single pass even in most hardware with two texture units) The basis images and mean image are shown in Figure 4.4.

Figure 4.4: The dynamic texture basis used for lighting the face in Figure 4.3. The mean is shown in the upper left. Other elements scaled so that black is -1, white is +1 and gray is 0.

4.7.2 View Variation

The wreath shown in Figure 4.5 is nearly planar, but has small intricate details that could not possibly be captured by a simple planar texture map. We have computed a dynamic texture parameterized by viewing direction, and compare the results to simple texture mapping.

To build the dynamic texture, first, we warp the set of sample images to a set of square textures using the 4 corners of a rectangle surrounding the wreath which were marked with stickers. This warp is achieved accounting correctly for perspective distortion by using projective homographies. once we have all views in this common texture space, we perform the PCA on the textures to compute the dynamic texture basis.

Figure 4.5 shows renderings from three different views using both simple texture mapping, and view dependent dynamic texture mapping. The wireframe box is shown in the figure simply to illustrate the viewing direction, only a single quadrilateral is used in the actual rendering. It is clear, especially in the view on the far right, that the dynamic texture renderings are much more natural looking.

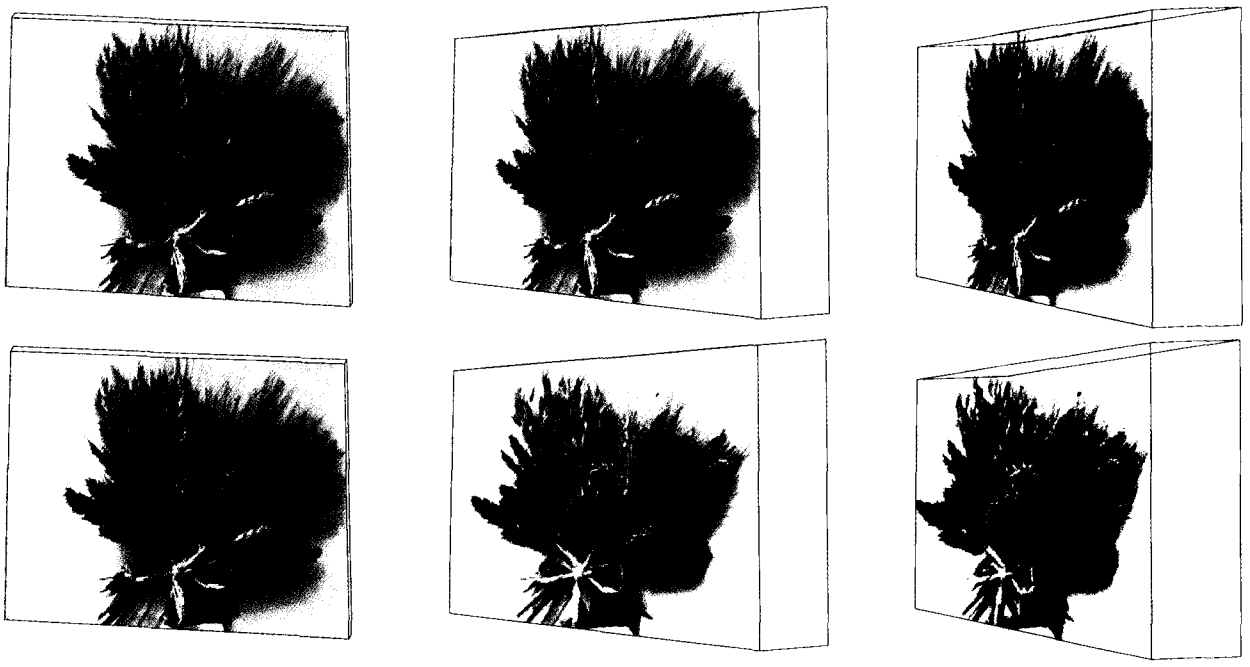

Figure 4.5: Texturing a rotating quadrilateral with a wreath. Top: by warping a flat texture image. Bottom: by modulating a dynamic texture basis which is then warped onto the same quad

We have found that although representing view dependency using basis modulation is possible, it becomes inefficient when the range of views is very large. In particular, this problem relates to how textures are warped from sample views into a common texture space. When a view is at a nearly grazing angle to a polygon, there will be very few samples for the texture on that polygon, and when it is warped into texture space, it will generate a very blurred sample. The result of this effect is that even if the object being captured is in fact geometrically planar, an image basis will be computed which incorporates the effect of blurring the texture at grazing angles. Although there will be no visual artifacts caused by this, when combined with texture variation caused by parallax, a larger basis is required than one would need for parallax alone. In addition, basis modulation can only account for a few pixels of parallax, but when viewed from a grazing angle, any out-of-plane variation in depth can be made to generate a very large translation of features in texture space. For this reason, we suggest that displacement mapping, which we will describe in Chapter 5, is much better suited for representing significant magnitude parallax caused by approximate geometry (or what we elsewhere refer to as meso scale structure). However, acquiring dynamic textures is very simple and convenient, which in some cases outweighs the inefficiency just mentioned, making their use for representing meso scale structures somewhat more attractive. This issue could be somewhat improved with some modifications to the acquisition strategy, which are discussed in Section 7.1.

Chapter 5

Displacement Mapping

Displacement mapping is a surface representation where objects are coarsely modeled with a conventional geometry, and fine details are represented with a displacement function giving a displacement to the true surface from each point on the reference surface along the surface normal [9]. Displacement mapping is used in our model to represent meso scale details.

Various more approximate methods are available for representing meso structure. With bump mapping, the surface normals are modified by a *bump map* affecting only the lighting of the surface [2]. Parallax mapping roughly approximates parallax by adjusting texture coordinates along the view direction [55]. Displacement mapping, in comparison, accurately renders parallax and self occlusion, as well as lighting when combined with a bump map or, in our system, a dynamic texture.

We have developed a novel algorithm for rendering displacement maps using modern graphics hardware, which will be presented in this chapter. Although we have designed this model in the interest of making it both easily rendered and captured from images, we have thus far focused on the rendering and not implemented or developed any algorithm which captures displacement maps from images given a macro scale geometric structure. See Section 7.1 for further discussion on this topic.

5.1 Previous Work

The concept of displacement mapping has existed for quite some time. Since Cook introduced the idea in 1984, [9] techniques for rendering them have been evolving continually. Traditionally, displacement maps have been rendered by uniformly subdividing each polygon into micro-polygons, and displacing the newly created vertexes using the displacement map [10]. In both ray-tracers and real-time systems, these high polygon counts lead to memory/bandwidth inefficiency, and high geometric transformation costs, which limit performance. Even today, offline renderers still use this method as it is easy to implement and speed is not necessarily an issue. Hardware has recently become capable of uniform subdivision displacement mapping by allowing vertex programs to sample from textures (an input to the vertex program from texture data in Figure 2.1). However, since it is possible for an object to extend from near the viewer to far from the viewer - a ground plane for example - uniform subdivision becomes either inefficient or inaccurate: either distant surfaces have many triangles smaller than a single pixel, or near surfaces have noticeably coarse resolution.

Adaptive geometric subdivision becomes complicated since higher resolution parts must connect to lower resolution parts without forming cracks in geometry. Other difficulties include noticeable popping when mesh resolution in an area changes as the camera moves. These problems have been well-researched in software systems [34, 15] but due to the complexity of the problem, adoption in

hardware has not been widespread.

Some image warping algorithms have been adapted from image-based rendering research for displacement mapping. Relief textures [44] is an implementation which is suitable for hardware. However, image warping only usually applies to images with depth, or displacement mapped planes. Relief textures have recently been generalized to cylinders [16], but generalization to arbitrary meshes would be difficult.

Both geometric-subdivision-based algorithms and image-based algorithms perform forward mappings: geometric methods create triangles and then project them onto the screen, and image-based methods directly warp images with depth onto the screen. For a fragment shader based technique, we need an inverse mapping, which determines which part of the geometry is visible at each fragment. Some approximations of this form exist, such as parallax mapping [55], but the only way to get geometrically accurate inverse displacement mapping is by ray-tracing.

Displacement maps have also been directly rendered in ray-tracing, using iterative root-finding methods [22] in software rendering. In older hardware, before fragment programs, slicing planes were used which approximate ray-tracing by sampling at discrete points a rendering pass for each sample [13]. Now, programmable fragment processors are beginning to be used for ray-tracing. View-Dependant displacement mapping precomputes all possible ray intersections, and looks them up at render time, using large amounts of texture memory [54]. Most like our approach is the sample-based ray-tracing of displacement maps in [23]. However, our method uses a different method for determining exact ray entry and exit points, takes more than twice as many samples per ray, and achieves higher frame rates.

Fragment-based solutions benefit from automatic level of detail (LOD): far or small parts of an object that appear small on the screen, contain less fragments, rendering more quickly, and large or near parts render more accurately. In comparison, achieving this type of LOD geometrically would require adaptive tessellation, which is complicated to implement and not available in most hardware. In addition, fragment based algorithms benefit from early Z rejection which completely avoids processing already occluded fragments.

5.2 Algorithm

We assume that displacements are between 0 and some maximum depth called the displacement scale s. Displacements are stored in a grayscale texture image with elements in $[0, 1]$, where the actual displacement is s times the value stored in the displacement image.

5.2.1 Planar Displacement

Here, we make some constraining assumptions which simplify the problem, and in Section 5.2.2 we will show how to transform the general case into this case. We assume that the reference surface is planar and rectangular. We also assume that its texture coordinates form a rectangle in texture space. This gives us a simple linear transformation between world space and texture space. It also allows us to simply reject intersections outside the texture space rectangle to get correct silhouettes.

From the reference plane, we generate a volume which we will call the displacement volume in object space V_o and the corresponding volume in texture space V_t. V_o is the volume created by sweeping the reference plane a distance of s along its normal. V_t is the 2D texture rectangle of the plane, swept along a 3rd axis from 0 to 1. The transformation matrix that transforms V_o to V_t will be called M_{ot}.

The displacement volume is rendered as six quadrilaterals in OpenGL. View rays are first transformed into object space, and then transformed by M_{ot} into texture space. The texture space ray origin and direction are interpolated and passed to a fragment program. At each fragment we sample the displacement map at N discrete points along the ray through the displacement volume. N is chosen based on the number of texture accesses available in particular hardware, and the complexity of the surface. This type of sampling is used since texture samples whose location depends on the results of previous samples (such as in standard iterative root-finding techniques) are expensive and limited in current graphics hardware. Sampled displacement values d_i are compared to the ray heights h_i at each point to determine which side of the surface the ray is on at each sample position.

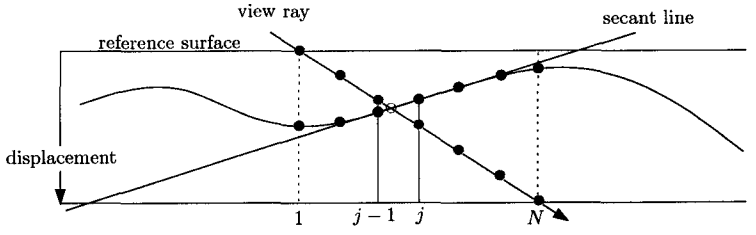

Figure 5.1: Algorithm: d_i are shown in blue, h_i in green, and the intersection point in red

We take the interval between the sample nearest to the viewer, j, where $h_j > d_j$ and its neighboring sample, $j - 1$, to contain the intersection of the ray with the surface (see Figure 5.1). The surface is then approximated by the secant line from d_j to d_{j-1}. Finally, we calculate texture coordinates, which are used to index texture and normal maps, as the intersection of this secant line with the view ray.

In comparison, using the position of sample j to index texture and normal maps texture coordi-

nates, rather than performing the final step, will generate results equivalent to volumetric slicing, as in [13], but with a single rendering pass.

5.2.2 Arbitrary Meshes

In the planar case, in order to render silhouettes accurately, we assume that any intersections outside of the plane's rectangle in texture space are not on the plane, and don't render them. This assumption, along with the assumption of planarity, must be removed in order to support arbitrary meshes.

There are two problems when generalizing this algorithm. First, exact entry and exit points are required for rays intersecting V_o (the volume generated by displacing a single triangle by the maximum displacement). Second, finding a linear transformation that maps rays in V_o into the corresponding volume in texture space: V_t.

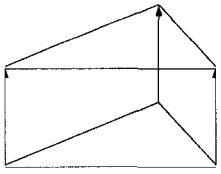

Figure 5.2: A displacement volume in texture space (triangular prism)

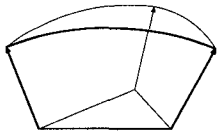

Figure 5.3: A displacement volume in object space assuming spherically interpolated normals.

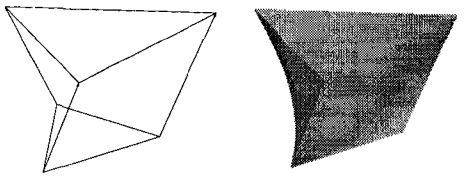

Figure 5.4: A displacement volume in object space assuming linearly interpolated normals, shown shaded on the right to emphasize that the volume is curved.

But what do the volumes V_o and V_t look like? V_t is a triangular prism defined by the texture coordinates of the current triangle, with a depth of 1 as shown in Figure 5.2. V_o is more complicated. If we were to assume spherical interpolation of normals - as displacement mapping is usually implemented in software renderers with geometric subdivision - we would get the volume shown in Figure 5.3. It would be very difficult to compute entry/exit points into such a volume, and there is clearly no linear mapping from this curved volume to a prism. For simplicity, we must assume linear interpolation of normals, which generates the volume shown in Figure 5.2.2. Although two of the boundaries are now planes, the other three boundaries to this volume are curved bilinear surfaces. Intersecting rays with those surfaces would still be too complicated, and there is still no linear transform between this volume and a prism. Now we choose to approximate V_o by subdividing it into volumes with planar boundaries, and for which there exists a linear transformation into texture space. We subdivide V_t into three tetrahedrons, and we approximate V_o by three corresponding tetrahedrons. In a connected mesh, we must assure that neighboring displacement volumes are split into tetrahedrons along the same edges. This can be done simply by reordering all triangle vertexes in order of their index into the vertex array as shown in Hirche et al. [23].

Where our method differs substantially from the tetrahedral renderer in [23] is in how we calculate entry and exit points through the tetrahedrons. Now that each tetrahedron has a linear mapping to texture space, we compute entry and exit points of each ray by simple line/plane intersections. We simplify this further by performing the intersections in axis aligned tetrahedron space where the four planes are $x = 0$, $y = 0$, $z = 0$, and $x + y + z = 1$.

Detailed Algorithm Overview

Initialization:

- generate three tetrahedrons for each triangle
- build transformation matrices from object space to tetrahedron space
- build transformation matrices from tetrahedron space to texture space

Rendering:

- render all four triangles of each tetrahedron passing vertex position and both matrices to the vertex program.
- The vertex program computes the ray entry point and direction in tetrahedral space and texture space, these are interpolated linearly over each triangle and passed to the fragment program.
- The fragment program intersects the ray, in tetrahedral space, with the the planes of the tetrahedron. The nearest positive intersection parameter is chosen (ignoring the intersection with the face being rendered). This parameter is then used directly on the ray in texture space.

  Sample points for the ray-tracing algorithm are computed as:

  $p_1 = r * q * \frac{1}{N+1}$

  $p_{i+1} = p_i + r_1$

  Here N is the number of samples, p_1 is the ray entry point in texture space, r is the ray direction in texture space, and q is the exit point intersection parameter. Using these sample points, the rest of the ray-tracing algorithm proceeds as shown in Section 5.2.1

5.3 Displacement Mapping for LOD

Since the performance and accuracy of this algorithm depends directly on the the number of fragments rendered, level of detail (LOD) is automatic: parts of the object appearing larger on the screen are rendered in more detail. This makes this method useful for LOD of any object, not just ones which would be naturally displacement mapped (such as brick walls). This can be achieved by precomputing a displacement map and bump map given a low-polygon and a high-polygon version of the same object.

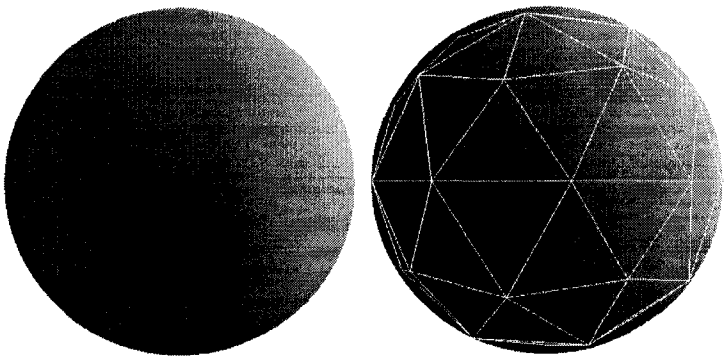

Figure 5.5: A smooth sphere rendered as a low detail sphere plus a displacement map.

Most current game engines already use similar techniques, computing bump-maps to make a low-polygon model look almost as good as the high-polygon version. However, a common complaint is that the silhouette of the object clearly shows the low resolution of the geometry when bump mapping is used. Using true displacement mapping gives much higher quality including correct silhouettes and parallax effects. Some example renderings of our algorithm being used for LOD are shown in Figures 5.5 and 5.9.

5.4 Improving performance

5.4.1 Early Z Rejection

The performance of our technique is bound by fragment processing. Many fragments may be occluded, and do affect the final image, but are processed anyway depending on the order they are rendered. Modern GPUs can discard fragments before running the fragment program if the contents of the depth buffer indicate that they are already occluded. This means that it is in our best interest to render scenes from front to back in scenes with high fragment program costs. Sorting polygons, or just tetrahedrons, can save us from this overdraw problem, but introduces extra CPU and bandwidth load. For opaque objects, another solution is to simply render the reference object - just the

original low-resolution triangles, not the extruded volumes - into the depth buffer before rendering the displaced volumes with the expensive ray-tracing shader. This achieves a similar speed improvement without the CPU and bandwidth hit. Note that early z rejection optimization requires that the shader does not compute the displaced depth value; this works fine unless it is necessary for the intersections of objects to be rendered accurately.

5.4.2 Tight Fitting Displacement Volumes

In our formulation thus far, each displacement volume has been extruded by the same distance along the object's normals. Depending on the displacement map and the tessellated resolution of the mesh, all displaced volumes are not likely to contain actual displacements over the full range $(0, 1)$. We can optimize displaced volumes by checking the minimum and maximum displacements for each triangle given a certain displacement map. Fitting the displacement volume to actual minimum and maximum displacements will improve both performance and accuracy. Ray-tracing through tighter bounding volumes means more accurate ray tracing results, since the same number of samples are taken over a smaller distance. In addition, smaller volumes means that fewer fragments are processed, particularly at grazing angles.

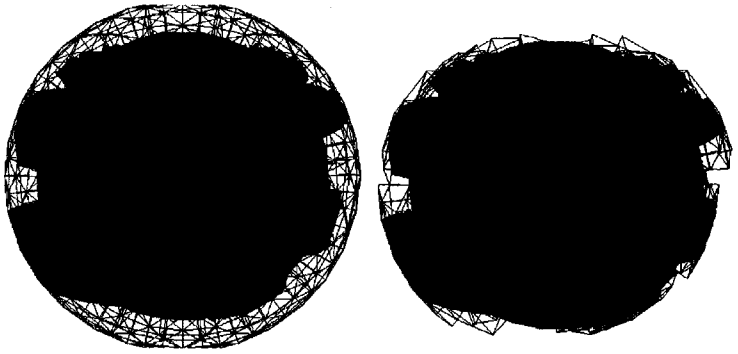

Figure 5.6: A displacement mapped object with full size displacement volumes on the left, tight fitting volumes on the right.

There are cases where the accuracy and performance improvements are noticeable. Notice the horizontal line in near the center of the sphere in Figure 5.6. It is not straight on the left due to sampling aliasing but in the optimized case it is rendered correctly. LOD displacement maps, as in Section 5.3, commonly contain a large range of displacements over the whole object, but not necessarily all in a single triangle. Therefore, LOD displacement maps often benefit from performance increases with tighter displacement volumes. For the model in Figure 5.6 we have measured frame rate increases approximately 65% when using tight fitting displacement volumes over full size ones.

5.4.3 Variable Sampling Rate

With shader model 3.0 hardware, looping support and conditionals provide us with the possibility of taking a variable number of samples depending on the ray. Rays that are nearly normal to the surface are vertical in texture space and require few samples. Rays that are nearly parallel to the surface polygon are horizontal in texture space and require more samples. Ideally, one sample would be taken for each pixel in the displacement map that the ray crosses (when projected on the the texture); this would eliminate all aliasing. This could be achieved by separating sample points by a uniform distance in the 2D texture plane (1 pixel size), rather than taking a uniform number of samples. This would reduce aliasing at grazing angles, where more samples are needed, and improve efficiency in direct views. The number of samples taken per fragment would be spatially coherent, which is required for current hardware to actually achieve a speed improvement since a group of fragments renders only as fast as the slowest fragment. We have not yet implemented this method, but it seems that it should improve performance and/or quality.

5.4.4 Silhouette Displacement Mapping

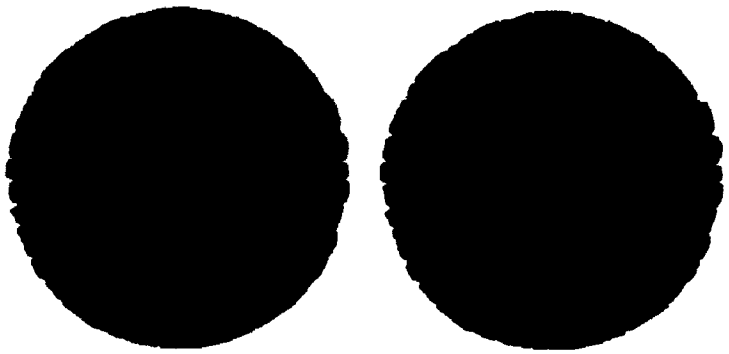

Figure 5.7: True displacement mapping is shown on the left, on the right, only the silhouette regions are displacement mapped, the center is simply bump-mapped.

We have tested a more drastic optimization, that we call silhouette displacement mapping, which applies full displacement mapping only to polygons near the silhouette of the object. Other polygons receive only bump mapping. With geometric-subdivision-based methods, this would be difficult, requiring adaptive tessellation to line up the detailed part with the coarse part. With our method, we simply reduce the displacement scale s away from the silhouette. Where s is zero, we render only bump-mapped triangles. Parts with s non-zero are near the silhouette and render with full displacement mapping.

5.5 Results

Our algorithm has been implemented using OpenGL with ARB vertex program and fragment program extensions. The algorithm will run on a wide variety of hardware (anything supporting ARB_vertex_program), but has been tested primarily on Radeon 9700/9800 graphics cards. Performance results given here were measured on a Radeon 9700/Athlon 2500+ system in a full 640x480 window.

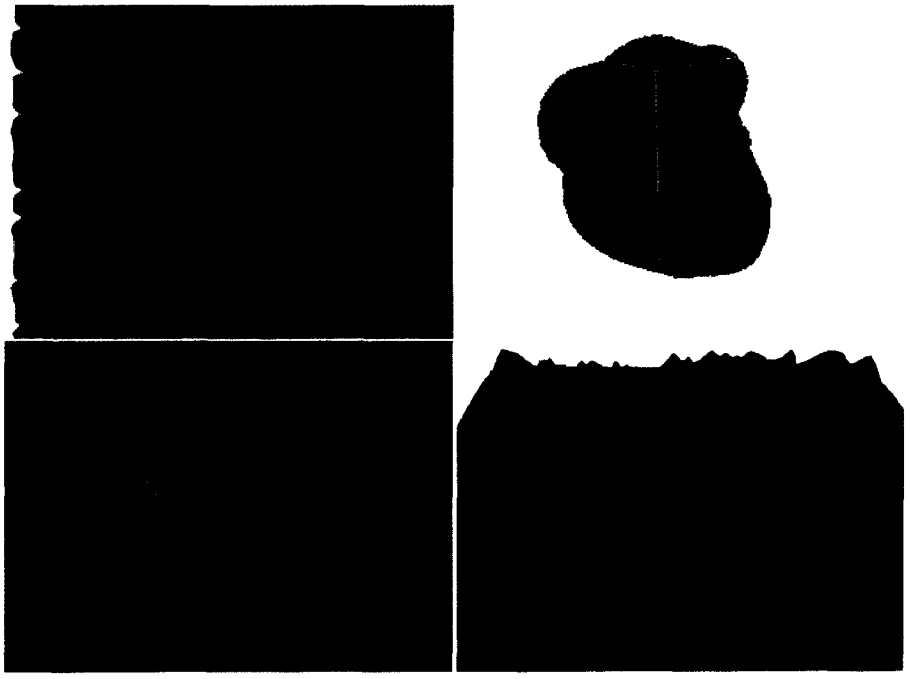

Figure 5.8: Results of the planar displacement mapping algorithm: upper right object is made from six displacement mapped planes rendered in four passes, others are rendered in a single pass with a single plane.

For the planar algorithm we have been able to get 15 samples per ray before running out of resources (we are limited by ALU instructions) using ARB_fragment_program. To achieve higher sampling rates, multiple rendering passes are performed. The single pass planar algorithm gets up to 150 fps in with every pixel in a 640x480 window being processed. Results are shown in Figure 5.8.

Figure 5.10 shows the results of first (on top) rendering a rock displacement map directly on an approximate sphere, and second (on the bottom) blending the LOD displacement map used for Figure 5.5 with the rock displacement map to create a much smoother result.

In the generalized algorithm we were able to take 11 samples per ray, with the added overhead of computing ray entry and exit points. Not that this is significantly more than the 4 samples that a

previous algorithm took [23]. Frame rates vary from model to model with the generalized method, since there can be a lot of overdraw. The face model shown in Figure 5.9 renders at 45 fps with. the spheres in Figures 5.10 and 5.5 render at minimum 40 fps. Early Z rejection gives the following improvement: with no sorting we get 25 fps for the sphere in Figure 5.10; pre-drawing to the depth buffer increases the frame rate to 36; sorting tetrahedrons increases the frame rate to 40.

The algorithm proposed here can render displacement mapped planes and arbitrary objects using fragment processing hardware. The method we have described is more efficient than previous methods. It is useful for both planes and arbitrary objects, and has also been shown to work well for level of detail rendering.

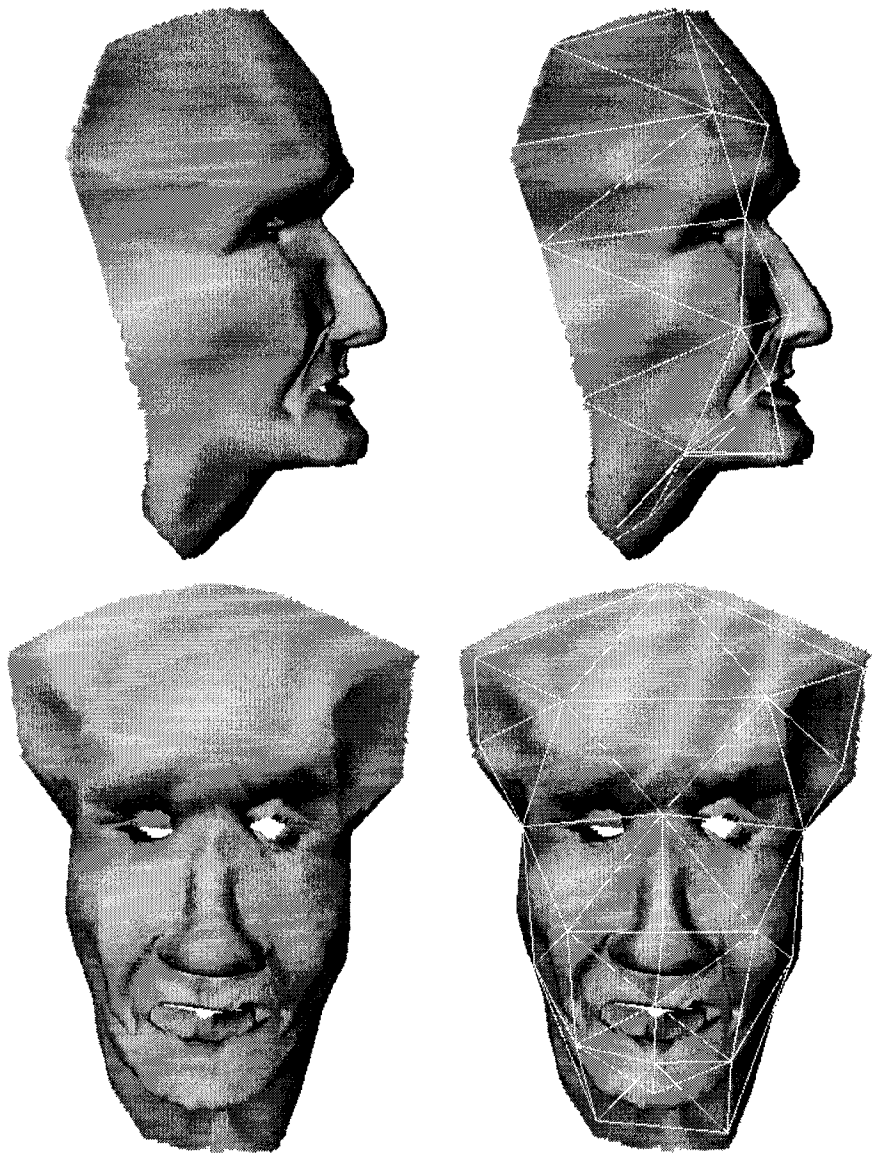

Figure 5.9: Level of detail rendering with displacement mapping. Two views of an object rendered with our algorithm. The overlay on the right shows the coarse geometric resolution of the base mesh.

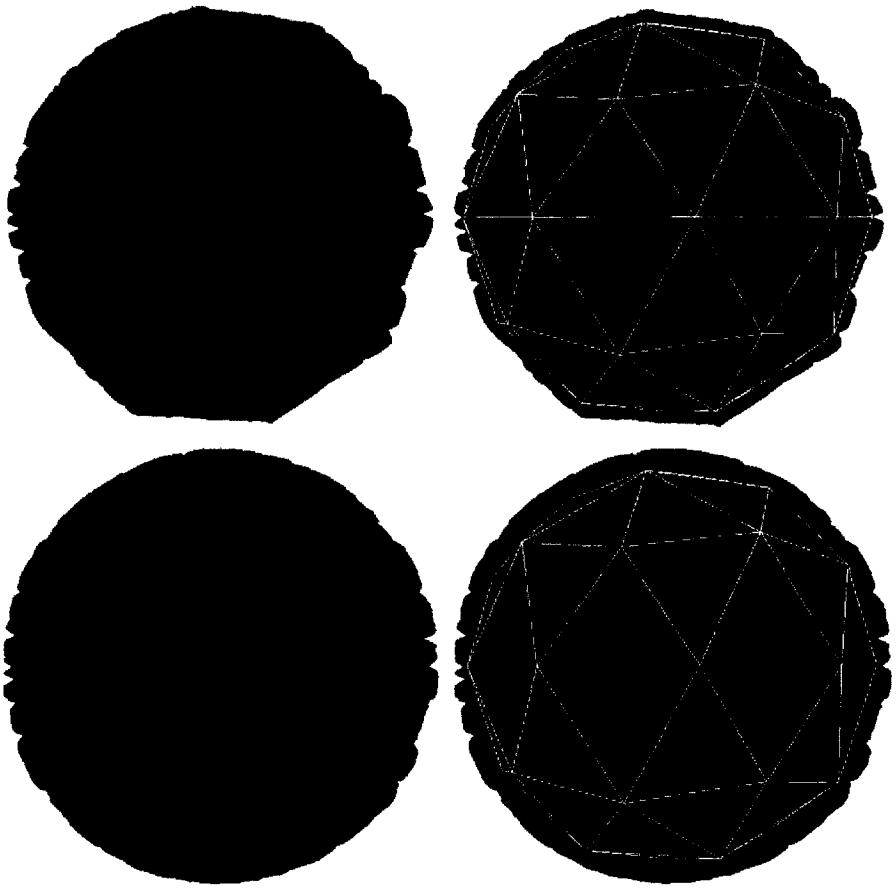

Figure 5.10: A coarse sphere displaced by a rock displacement map on the top, and combined LOD displacement map and rock displacement map on the bottom. Note how the bottom sphere is smoothly curved while the top one has a more polygonal shape.

Chapter 6

Systems and Experiments

We have developed two systems for acquiring and rendering different types of hierarchical graphics models. The first uses triangle meshes for the macro scale, and represents both the meso and micro scale with a dynamic texture. The second system also represents the macro scale using a triangular mesh. However, in our second system the meso scale is represented separately with a displacement map, and only the micro scale is modeled using a dynamic texture.

6.1 Dynamic Textured Geometry

We have performed many experiments using dynamic textures in combination with rough polygonal meshes. Considering that the geometric information acquired by the methods described in Chapter 3 are often inaccurate and sparse, dynamic textures parameterized in view direction can be used to account for small scale geometric inaccuracy (meso scale). In addition, the dynamic texture will reproduce complex surface reflectance (micro scale). Unfortunately, in this case, the meso and micro information are linked, and both only parameterized in view direction, meaning the light cannot be made to move around independently of the view (but can potentially move with respect to the object when the view changes if that is how the object was captured). However, a more elaborate setup with a rotating camera, and an array of lights could be used to overcome this problem in the future.

Objects modeled in this way are rendered in graphics hardware by modulating the dynamic texture basis, as described in detail in Chapter 4. We have implemented two types of geometry acquisition in our experiments, structure from motion, and shape from silhouette. The separate acquisition systems and results with the two methods will be detailed in the following subsections.

6.1.1 Structure from Motion

SFM acquires geometry based on corresponding feature positions over many views. In order to get the feature correspondences, we use tracking. Textured regions are tracked using the XVision2 tracking libraries which provide very useful and efficient SSD trackers [30]. These trackers work best on textured regions that are locally unique. We have features manually selected by the user, since trackable regions are often sparse and we prefer to place them in strategic places for modeling purposes, such as the apex of a roof for example.

After the user identifies several features in the camera view of the target object. Then, from the live video stream, we track the locations of these features in real-time. We record the video stream, along with the feature positions while the user rotates the object through various viewing directions. We make the simplifying and restrictive assumption that all tracked points are visible in all video frames.

Using the tracked feature positions from many different views, we can compute a rough 3D structure using methods described in Section 3.1.

Early experiments, such as the flower in Figure 6.1, use separate dynamic textures for each quadrilateral. Each one is warped from each frame of the video sequence, to a square textures.

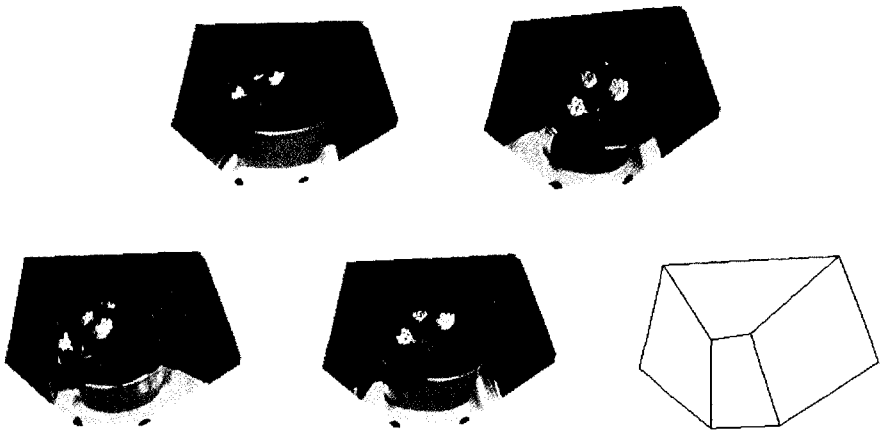

Figure 6.1: A flower rendered with a very simple geometry of four quadrilaterals (shown on the bottom right), each dynamic textured with respect to viewing direction.

In more recent experiments, such as the house shown in Figure 6.3, memory usage was significantly reduced. First, we use only a single dynamic texture. The mapping from image space to texture space is defined by assigning each vertex a texture coordinate, which we generate automatically as the vertex's average camera space projection. A set of triangles is generated for the model using Delaunay tessellation of the texture coordinates, and potentially edited manually with our GUI shown in Figure 6.2. Each view is warped to texture space by affine warping of the pixels in each triangle. Additionally, the dynamic texture is computed in YUV color space as described in Section 4.3, further reducing memory requirements by allowing reduced number and resolution of U and V basis elements.

Figure 6.2: The system built for viewing and editing tracking data.

In our system, the camera or object could be oriented by hand. However, the best results were obtained using a tripod and stick to orbit the camera around the object. This setup achieves uniform sampling (if the tripod is moved smoothly), and also keeps the object centered in the camera view, which is essential for the structure from motion algorithm. After tracking feature points, the system allows the user to interact with the data in order to potentially identify and remove or correct inaccurately tracked features. Figure 6.2 shows a screenshot of the system's user interface.

As can be seen, in extreme cases, such as the flower in Figure 6.1, a complex and intricately detailed object can be represented with very few polygons at the expense of a large texture basis. The polygonal mesh for this flower contains only four quadrilaterals. However, this puts a heavy weight on the dynamic texture to account for very large geometric inaccuracies. It was necessary, in this case, to use 100 basis images for rendering, and blurriness artifacts were still present.

The house shown in Figure 6.3 has a more reasonable geometric structure, but still requires a fairly large basis (50 to 100 elements). In addition, our constrained capture setup which requires the visibility of all feature points in all video frames, results in fairly small possible variation in viewing angle for most objects. The next subsection describes a system which acquires more accurate geometry and allows larger variation in viewing direction.

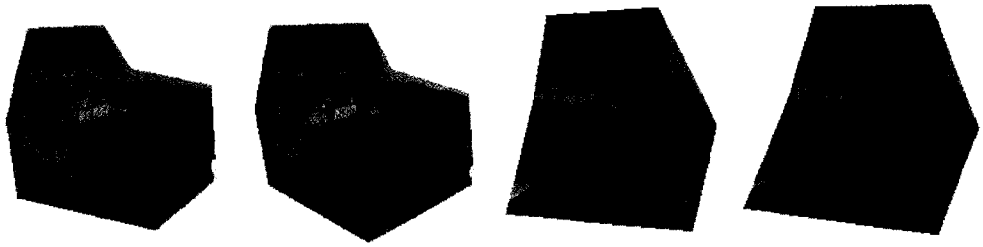

Figure 6.3: 4 new views of a house rendered with dynamic textures (parameterized in viewing direction)

6.1.2 Shape from Silhouette

To capture geometry using shape from silhouette, we first place the object on an automatic turntable. Using a stationary camera, we take a set of images from views on a ring around the object. We may move the camera to get a second or more rings of views from different heights. A calibration pattern on the turntable is used determine the camera's position and orientation with respect to the object. A colored piece of paper is mounted behind the object for use when detecting the silhouette.

We have implemented the method described in 3.2.1 to segment the object from the background, giving us a silhouette image. Using the segmented image, our implementation of the marching intersections SFS algorithm described in Section 3.2.2 was implemented to compute the visual hull.

As in the SFM case, we apply a view-dependent dynamic texture to the resulting geometric model to account for its inaccuracy. However, applying the dynamic texture is slightly more difficult since we have a much larger range of views, and a closed mesh.

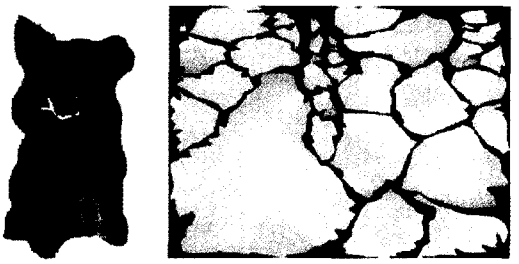

Figure 6.4: Texture atlas: on the left is a rendering visualizing the different charts on the object; on the right is the texture atlas.

Proper behaviour of our system depends on computing a one-to-one mapping from each point on the object's surface into a 2D texture image. We represent this mapping by a 2D texture coordinate stored along with each vertex, and linear interpolation within each triangle. Computing such a set of texture coordinates for a general mesh is a very difficult problem. This problem was solved easily in the case of SFM in Section 6.1.1 where observed views were restricted to a small range. However, in the case of SFS, we have an arbitrary polyhedral mesh, which requires a much more general method for automatically generating texture coordinates.

The method from Levy et al. [32] was implemented, and works sufficiently well. This method breaks a polygonal mesh into *charts* using a region growing algorithm. The algorithm segments the mesh into the desired number of charts preferring to split along regions of high curvature. Each chart is then flattened into 2D using conformal mapping. Conformal mapping preserves angles so that the error between the three angles of each triangle in texture space and the same triangle in 3D is minimized (in the least squares sense). We have also implemented a version which uses multi-dimensional scaling (MDS) as described in [60]. MDS works similarly, minimizing the error in distance along the surface of the 3D mesh to distance in 2D texture space; results are comparable.

All of the charts are then packed together into a *texture atlas*. Packing is performed using a greedy algorithm, ordering the charts from largest to smallest. Figure 6.4 shows an example where the texture atlas is shown next to the object with charts colored uniquely.

After texture coordinates are generated, warping the example views into texture space requires an extra trick. Some parts of the model are occluded from some views, so we cannot simply warp triangles from the camera views into texture space. We overcome this using a depth buffer. For each view, we generate a depth image where each pixel represents the depth from the camera to the visual

Figure 6.5: Two views of a pig rendered with view dependent dynamic textures.

Figure 6.6: A scene with several dynamic textured objects (10 dynamic textures in total)

hull. We then warp this depth image into texture space. Now, when we warp the actual image pixels into texture space, we interpolate the actual depth of each vertex relative to the camera across each triangle. So, at each pixel in the texture, we know the depth of that point on the geometry, and we know (from the depth texture) the depth that the pixel we are potentially placing there has. If they are not the same (to some tolerance) that element of the texture is occluded. After marking occluded areas in the texture for each view, we fill those parts in with the mean value as computed over all views where that part was visible. In this way, we can use a single dynamic texture even though parts of it are not visible in every view.

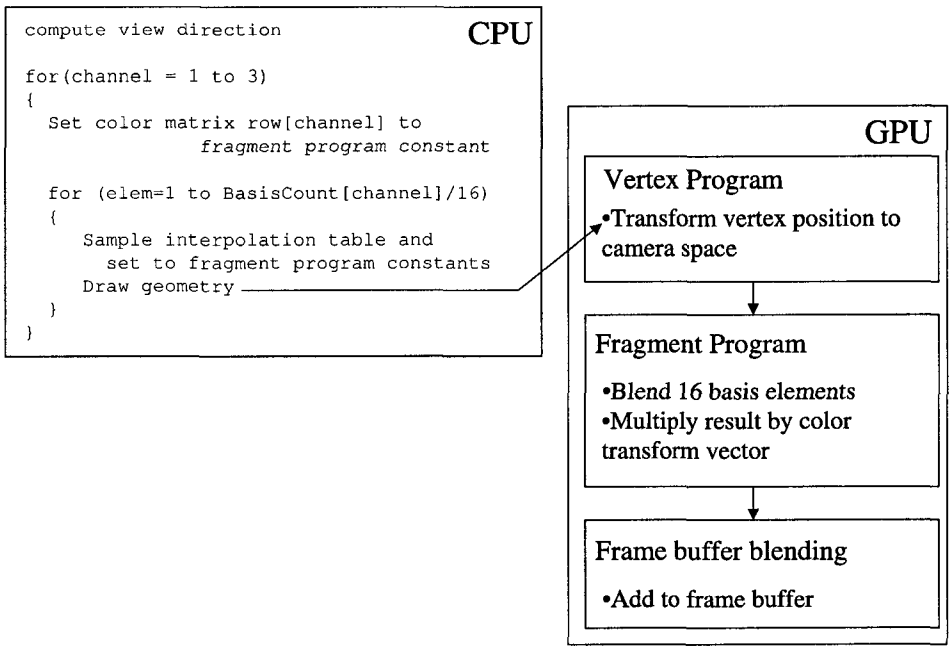

Figure 6.7: Algorithm overview for rendering geometry with dynamic texture

After this warping step, the computation and rendering of the dynamic texture take place in the same way as described in Chapter 4. An overview of how the rendering algorithm is executed on the CPU and GPU is given in Figure 6.7. An example object captured with this system is shown in Figure 6.5 and a scene filled with several objects is shown in Figure 6.6. On a 3.2 GHz CPU with NVidia GeForceFX 6800GT graphics accelerator, the pig model renders at 115 fps, and the scene with several dynamic textured models (a total of 10 dynamic textures) renders at 35 fps.

6.2 Geometry, Displacement Mapping, and Dynamic Texture

This thesis is focused on the rendering aspects of graphics models, and since an implementation of this type of algorithm was not readily available, we have used other methods in order to test the integrated rendering system. Ideally, we would like to use SFS or SFM, in combination with an algorithm which computes displacement maps from images such as the method recently proposed by Vogiatzis et al [53].

The geometry and texture have been captured in independent steps. First, very detailed geometry was captured using a laser scanner. Using 3D modeling software, we performed post-processing on

the detailed geometry to compute a low-polygon model and corresponding displacement map.

Then, the dynamic texture part of the model was acquired separately and aligned manually with the geometric model. To acquire lighting variation in a dynamic texture, we mounted the object and camera rigidly on opposite ends of a rod. Next to the object we mount a matte sphere, and the rod is attached to a rotating tripod as shown in Figure 6.8.

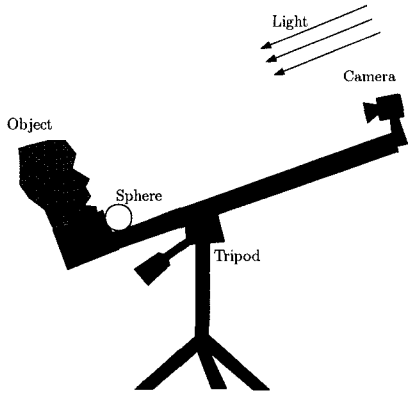

Figure 6.8: The apparatus used for acquiring light variation

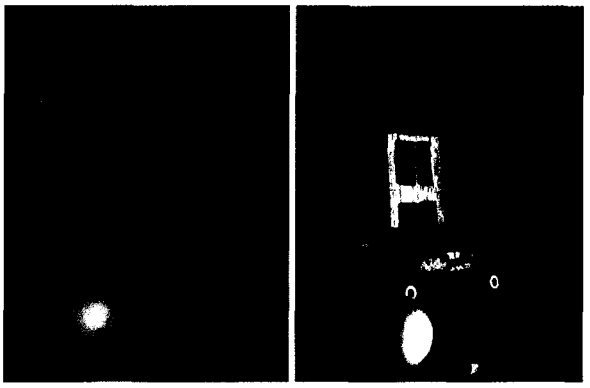

Figure 6.9: Example images taken as shown in the adjacent figure

Placing the apparatus in natural, directional light (from the sun), we rotated the tripod manually in a grid pattern, while capturing many images. Some example images are shown in Figure 6.9. After sampling the images, the user identifies the sphere in a single image.

The light direction is calculated in each image by solving a simple set of linear equations. We can compute the normal of each pixel in the sphere, and then by using the standard diffuse lighting equation, we relate the normal and pixel color to the light vector by the standard diffuse lighting equation $c = n \bullet l + a$, where c is the pixel brightness, n is the pixel normal, l is the light direction, and a is the ambient light. With many pixels, a system of equations is built and solved for l and a. Using the light direction as a parameter, we compute a dynamic texture as described in Chapter 2.

A ceramic mask and a model house made from natural wood, bark, etc. were captured using this method and the resulting renderings are shown in Figures 6.10 and 6.13, respectively. The algorithm used for rendering, and how it is mapped to the CPU and GPU is shown in Figure 6.11. The examples render at approximately 100 fps in a 640 by 480 window using NVidia GeForce 6800GT graphics hardware and a 3.2GHz CPU.

Representing lighting variation requires very few basis elements. In both the house and face examples, only eight grayscale basis images were used, and no basis images were used for color information. The first four basis images are shown in Figure 6.12. The hardware rendering imple-

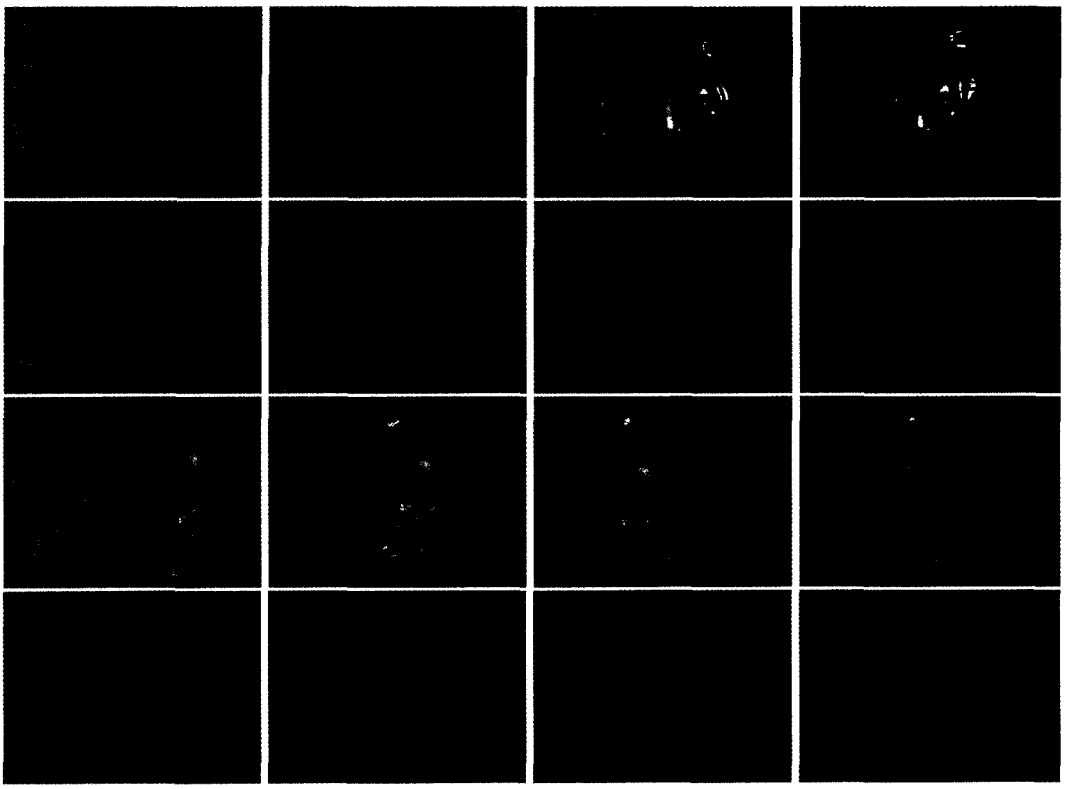

Figure 6.10: Four views each with four different lightings of a displacement-mapped and dynamic-textured Korean face mask from the Shilla period.

mentation for only 8 basis images requires only three texture accesses, two dot products and two additions, which allowed it to easily be inserted into the displacement mapping fragment program in place of the standard lighting computation. This allows the composite rendering to take place in a single pass, and achieve performance approximately equal to the displacement mapping code with standard bump-mapped lighting.

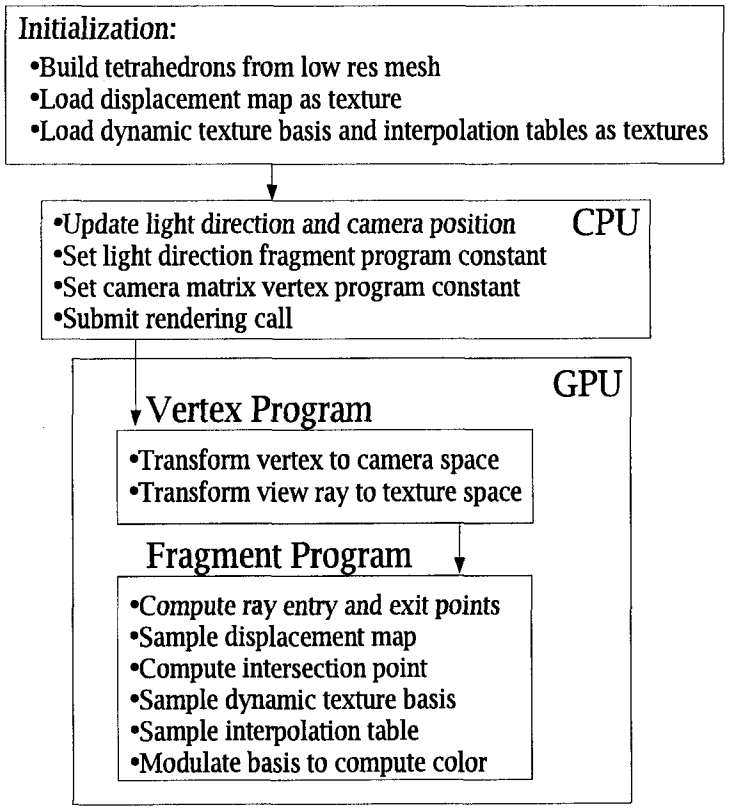

Figure 6.11: Algorithm overview for rendering geometry, displacement map, and dynamic texture.

Figure 6.12: First 4 basis images (of 8) used for lighting the face artifact. Intensities have been remapped so that black is -1.0, gray is 0.0 and white is 1.0.

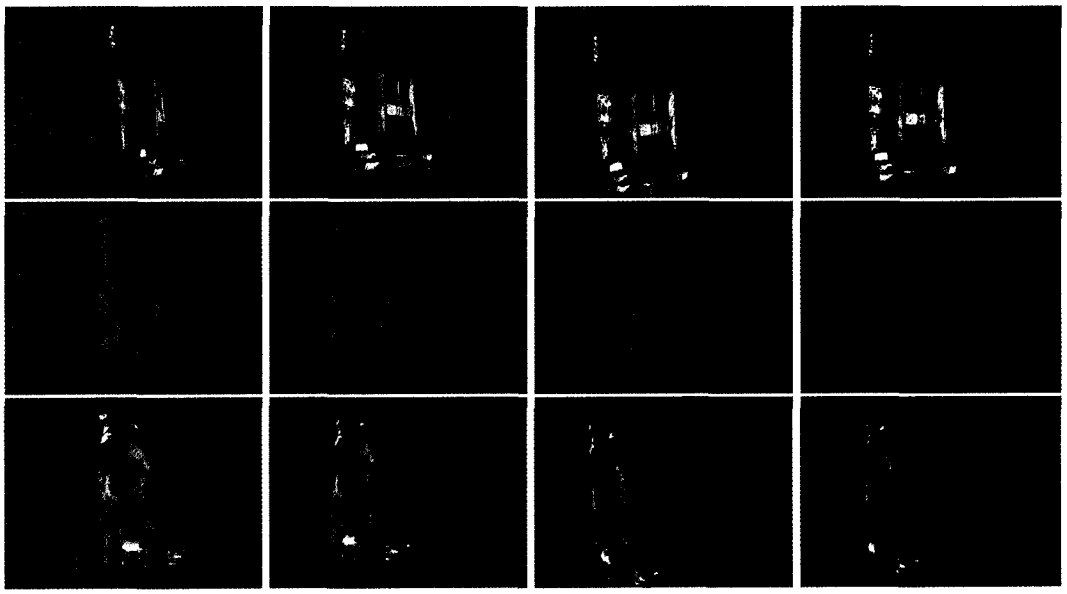

Figure 6.13: Four views each with three different lightings of a displacement-mapped and dynamic-textured model house.

Chapter 7

Discussion

7.1 Future Work

7.1.1 Geometric Capture from Images

The system shown in Section 6.1.2, which captures objects using shape from silhouette and view-dependent dynamic textures, is based completely on 2D images, and is therefore quite easy to use. However, the system used for acquiring geometry, displacement maps, and dynamic texture in Section 6.2, additionally makes use of 3D input for acquiring displacement map data. Future work would be to compute displacement maps from images given an approximate geometry. Some work has been done on this problem by making use of the parallax between multiple views [53]. In addition, such a method should be incorporated into a system easy enough to be used by people not trained in computer graphics, like the system in Section 6.1.2.

7.1.2 Displacement Mapping

The displacement mapping algorithm shown in Chapter 5 renders in a pixel shader on graphics hardware. Recent advances in hardware allow texture accesses during vertex processing, as well as hardware-accelerated geometric tessellation. Since both vertex and pixel based methods for displacement mapping in hardware are quite new, the performance of these methods should be compared, to determine which is actually most efficient using available graphics cards.

7.1.3 Dynamic Texture

In this thesis, we have shown examples which use dynamic textures for representing lighting variation, and others which use dynamic textures for representing view-dependency. However, we have yet to use dynamic textures for both effects simultaneously. There are two reasons for this. First, it is very time consuming to simultaneously capture many lighting directions and viewing directions, since it would mean sampling a 4-dimensional function. Second, interpolation of coefficients in two dimensions is simple, but becomes complicated when the number of dimensions increases. *Tensor Textures* have shown how different dimensions of variation can be separated using tensor mathematics [52]. However, this requires that for each viewing direction there is a set of images for each lighting condition, rather than a random collection of images where each has a view and lighting direction. This makes capturing such textures very difficult, which explains why the examples in their papers are made from computer renderings, and not real scenes. Allowing tensor texture like separation, with more flexible inputs would be a very useful future result.

A problem which was pointed out in Section 4.7 is that when acquiring a view-dependent dynamic texture, we may warp some grazing angle views containing very few useful pixels into texture

space, and then end up unnecessarily simulating the resulting blurred image during rendering. The problem is that the current method solves for a basis that minimizes the error between recomputed texture images (after being projected onto the reduced rank basis) and the original texture images for any view. The key to solving this problem is to realize that we do not care about errors between texture images. The important error is the difference between an original camera view, and the recomputed texture for that view, mapped onto the object and projected onto that camera. A mathematical reformulation which minimizes this error would achieve much better results with fewer basis vectors.

7.2 Conclusions

We have presented a new graphics model based on a hierarchy of scales of detail. The model was designed to balance ease of acquisition from images with efficient rendering. The three levels of detail - macro, meso, and micro - are represented using triangle meshes, displacement mapping, and dynamic textures, respectively. We have shown how this model can be rendered in real time using current hardware graphics accelerators. Also, we have presented systems for capturing this type of graphics model.

The low resolution macro scale was captured using image-based modeling algorithms. Two methods, shape from silhouette and structure from motion, were implemented and described in detail. Shape-from-silhouette proved more useful for capturing the full range of views of a small object, while structure from motion is more useful for large scenes, where silhouette information is not available.

For the rendering of the meso scale, a novel displacement mapping algorithm was developed, presented and tested. The method uses modern graphics hardware to ray trace displacement-mapped surfaces in real-time.

For the micro scale, we described dynamic textures, which can represent a texture which varies with respect to light or view direction. Several efficient dynamic texture renderers were designed and implemented on various levels of consumer graphics hardware. Dynamic textures parameterized by viewing direction were used to represent meso scale structure in some cases. Micro scale structure was represented by dynamic textures parameterized in light direction.

An easy to use system was built for capturing models with shape from silhouette and view dependent dynamic textures. These models are very easy to capture, but have only two scales, geometry and dynamic texture, and cannot be rendered with varying lighting conditions. A plug-in was developed for AliasWavefront's Maya renderer which can render captured objects of this type

in scenes which can also contain traditional graphics models.

A second system was built for making the full 3-tiered models. This system uses a laser scanner to acquire the geometric structure, from which a low resolution triangle mesh and corresponding displacement map are extracted. The lighting variation was captured separately using cameras. The final results can be seen in Figures 6.10 and 6.13.

These methods were proven useful in several experiments where real objects were captured and rendered. We demonstrated successful rendering of otherwise difficult cases such as flowers and other natural materials.

Bibliography

[1] Tomas Akenine-Mller and Eric Haines. *Real-time Rendering, 2nd Edition*. A.K. Peters Ltd., 1999.

[2] James F. Blinn. Simulation of wrinkled surfaces. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 286–292, New York, NY, USA, 1978. ACM Press.

[3] J. E. Bresenham. Algorithm for computer control of a digital plotter. *Seminal graphics: poineering efforts that shaped the field*, pages 1–6, 1998.

[4] Darius Burschka, Zachary Dodds, Dana Cobzas, Gregory D. Hager, Martin Jgersand, and Keith Yerex. Tutorial on recent methods for image-based modeling and rendering. In *IEEE Virtual Reality*, 2003.

[5] Heung-Yeung Shum Shing-Chow Chan and Sing Bing Kang. *Image Based Rendering (Monographs In Computer Science)*. Springer, 2005.

[6] Dana Cobzas and Martin Jgersand. Tracking and re-rendering using dynamic textures on geometric structure from motion. In *European Conference on Computer Vision*, 2002.

[7] Dana Cobzas, Keith Yerex, and Martin Jagersand. Dynamic textures for image-based rendering of fine-scale 3d structure and animation of non-rigid motion. In *Proceedings of Eurographics*, 2002.

[8] Dana Cobzas, Keith Yerex, and Martin Jagersand. Editing real world scenes: Augmented reality with image-based rendering. In *IEEE Virtual Reality*, 2003.

[9] Robert L. Cook. Shade trees. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 223–231. ACM Press, 1984.

[10] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The reyes image rendering architecture. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 95–102. ACM Press, 1987.

[11] Kristin J. Dana, Bram van Ginneken, Shree K. Nayar, and Jan J. Koenderink. Reflectance and texture of real-world surfaces. *ACM Transactions on Graphics*, 18(1):1–34, 1999.

[12] Paul E. Debevec, Camillo J. Taylor, and Jitendra Malik. Modeling and rendering architecture from photographs: A hybrid geometry- and image-based approach. *Computer Graphics*, 30(Annual Conference Series):11–20, 1996.

[13] S. Deitrich. Elevation maps. Technical report, NVIDIA Corporation, 2000.

[14] G. Doretto and S. Soatto. Editable dynamic textures, 2002.

[15] Mark A. Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. ROAMing terrain: real-time optimally adapting meshes. In *IEEE Visualization*, pages 81–88, 1997.

[16] Mohamed A. ElHelw and Guang-Zhong Yang. Cylindrical relief texture mapping. *Journal of WSCG*, 11, feb 2003.

[17] Cleo Espiritu. Hunter on ice: a look through the inuit seal hunt, 2003. http://www.cs.ualberta.ca/ cleo/sealhunt.

[18] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer graphics: principles and practice (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[19] W Freeman, E Adelson, and D Heeger. Motion without movement. *Computer Graphics*, 25(4):27.

[20] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. *Computer Graphics*, 30(Annual Conference Series):43–54, 1996.

[21] Gregory D. Hager and Peter N. Belhumeur. Efficient region tracking with parametric models of geometry and illumination. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(10):1025–1039, 1998.

[22] Wolfgang Heidrich and Hans-Peter Seidel. Ray-tracing procedural displacement shaders. In *Graphics Interface*, pages 8–16, 1998.

[23] Johannes Hirche, Alexander Ehlert, Stefan Guthe, and Michael Doggett. Hardware accelerated per-pixel displacement mapping. In *GI '04: Proceedings of the 2004 conference on Graphics interface*, pages 153–158. Canadian Human-Computer Communications Society, 2004.

[24] Peter Sturm II and Bill Triggs. A factorization based algorithm for multi-image projective structure and motion. In *ECCV (2)*, pages 709–720, 1996.

[25] Martin Jägersand. Image based view synthesis of an articulated agent. 1997.

[26] Martin Jägersand. Image based animation from learned visual-motor models. In *IEEE Computer Animation*, 2000.

[27] Martin Jagersand, Dana Cobzas, and Keith Yerex. Modulating view-dependent textures. In *Eurographics Short Presentations*, 2004.

[28] Steven L. Kent. *The making of Doom III : the official guide.* McGraw-Hill, 2003.

[29] Kiriakos N. Kutulakos and Steven M. Seitz. A theory of shape by space carving. Technical Report TR692, 1998.

[30] S. Lang. The xvision 2 software system. Master's thesis, The Johns Hopkins University, 2001.

[31] Marc Levoy and Pat Hanrahan. Light field rendering. *Computer Graphics*, 30(Annual Conference Series):31–42, 1996.

[32] Bruno Levy, Sylvain Petitjean, Nicolas Ray, and Jerome Maillot. Least squares conformal maps for automatic texture atlas generation.

[33] Ming Li, Marcus Magnor, and Hans-Peter Seidel. Hardware-accelerated visual hull reconstruction and rendering. In *Proceedings of Graphics Interface 2003*, Halifax, Canada, 2003.

[34] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory A. Turner. Real-time, continuous level of detail rendering of height fields. In *SIGGRAPH '96*, pages 109–118, August 1996.

[35] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169, New York, NY, USA, 1987. ACM Press.

[36] T Malzbender, D Gelb, and H Wolters. Polynomial texture maps. In *SIGGRAPH*, 2001.

[37] W. Martin and J.K. Aggarwal. Volumetric descriptions of objects from multiple views. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5(2):150–158, 1983.

[38] Wojciech Matusik. Image-based visual hulls. Master's thesis, Massachusetts Institute of Technology, 2001.

[39] Wojciech Matusik, Chris Buehler, and Leonard McMillan. Polyhedral visual hulls for Real-Time rendering. In *12th Eurographics Workshop on Rendering*, pages 115–126, 2001.

[40] Nelson L. Max. Horizon mapping: shadows for bump-mapped surfaces. 4(2), March.

[41] Leonard McMillan and Gary Bishop. Plenoptic modeling: An image-based rendering system. *Computer Graphics*, 29(Annual Conference Series):39–46, 1995.

[42] M.Tarini, M.Callieri, C. Montani, and C. Rocchini. Marching intersections: an efficient approach to shape from silhouette. In *Proceedings of VMV 2002*, 2002.

[43] SK Nayar, Y Nakagawa, and TK Yoshida-Machi. Shape from focus: An effective approach for rough surfaces. pages 824–831, 1994.

[44] Manuel Oliveira and Gary Bishop. Relief textures. Technical Report TR99-015, March 1999.

[45] M. Pollefeys and L. Van Gool. A stratified approach to metric self-calibration, 1997.

[46] C. Rocchini, P. Cignoni, F. Ganovelli, C. Montani, P. Pingi, and R. Scopigno. Marching intersections: an efficient resampling algorithm for surface management. In *Proceedings of SMI 2001*, pages 296–305, Genova, Italy, 2001.

[47] D. Scharstein and R. Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision*, 47(1/2/3):7–42, 2002.

[48] Christophe Schlick. A survey of shading and reflectance models. *Computer Graphics Forum*, 13(2):121–131, 1994.

[49] Amnon Shashua. *Geometry and Photometry in 3D Visual Recognition*. PhD thesis, MIT, 1993.

[50] Ying Nian Wu Stefano Soatto, Gianfranco Doretto. Dynamic Textures. pages 439–446, July 2001.

[51] Carlo Tomasi and Takeo Kanade. Shape and Motion from Image Streams: A Factorization Method Part 2. Detection and Trcking of Point Features. Technical Report CMU-CS-91-132, April 1991.

[52] M. Alex O. Vasilescu and Demetri Terzopoulos. Tensortextures: multilinear image-based rendering. *ACM Trans. Graph.*, 23(3):336–342, 2004.

[53] George Vogiatzis, Philip Torr, S.M. Seitz, and Roberto Cipolla. Reconstructing relief surfaces. In *Proceedings of British Machine Vision Conference (BMVC'04)*, pages 117–126, 2004.

[54] Lifeng Wang, Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, and Heung-Yeung Shum. View-dependent displacement mapping. *ACM Trans. Graph.*, 22(3):334–339, 2003.

[55] Terry Welsh. Parallax mapping with offset limiting: A per-pixel approximation of uneven surfaces. Technical report, Infiscape Corporation, 2004.

[56] Keith Yerex, Neil Birkbeck, and Martin Jagersand. A quick and automatic image based modeling and rendering system. In *Eurographics Short Presentations*, 2004.

[57] Keith Yerex, Dana Cobzas, and Martin Jagersand. Predictive display models for telemanipulation from uncalibrated camera-capture of scene geometry and appearance. In *IEEE International Conference on Robotics and Automation*, 2003.

[58] Keith Yerex and Martin Jagersand. Displacement mapping with ray-casting in hardware. In *SIGGRAPH sketches*, 2004.

[59] Ruo Zhang, Ping-Sing Tsai, James Edwin Cryer, and Mubarak Shah. Shape from shading: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21(8):690–706, 1999.

[60] G. Zigelman, R. Kimmel, and N. Kiryati. Texture mapping using surface flattening via multidimensional scaling. *IEEE Transactions on Visualization and Computer Graphics*, 08(2):198.