REAL-TIME ACQUISITION AND RENDERING OF LARGE 3D MODELS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Szymon Marek Rusinkiewicz

August 2001

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Marc Levoy, Principal Advisor

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Patrick Hanrahan

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Leonidas Guibas

Approved for the University Committee on Graduate Studies:

Abstract

The digitization of the 3D shape of real objects is a rapidly expanding field, with applications in entertainment, design, and manufacturing. In order for 3D scanning to become more commonplace, methods are needed for quickly and robustly acquiring full geometric models of complex objects. This dissertation describes a scanning system that allows a user to rotate an object by hand and see a continuously-updated model as the object is scanned. This allows the user to see and fill holes in the model, and determine when the object has been completely covered.

The system consists of three components. First, it incorporates a new design for a structured-light range scanner capable of returning the shape of a moving object, as seen from one viewpoint, at a rate of 60 Hz. Next, the range images returned by this scanner are continuously aligned to each other using a variant of the Iterated Closest Points (ICP) algorithm. An analysis of the stages of the ICP pipeline suggests that a combination of variants, including projection-based matching and point-to-plane minimization, is suitable for real-time use. Finally, the aligned range data is merged into a discretized voxel grid, and point (splat) rendering is used to provide a real-time display of the partial 3D model.

Given the increasing capabilities of range scanning systems such as the above, traditional algorithms for display, simplification, and progressive transmission of meshes are too slow to be practical. QSplat is a multiresolution point rendering system capable of displaying scanned meshes containing hundreds of millions of range samples at interactive rates. It uses a single bounding sphere data structure for view frustum culling, backface culling, level-of-detail control, and splat rendering. The system may also be extended to progressive view-dependent transmission of large 3D models across a network of limited bandwidth.

"So long, and thanks for all the fish."

– Douglas Adams, 1952–2001

Acknowledgments

I would like to start by thanking my co-worker, collaborator, and co-conspirator Olaf Hall-Holt for his help, his ideas, and his friendship. The project that led to this dissertation was very much a joint effort, and working with Olaf on it was a great experience.

Over the past *n* years, I have had the privilege of being in probably the most exciting place for computer graphics. Stanford has assembled some really amazing people here, and I have had the opportunity to work and become friends with many of them. I'll start by singling out the DMich crowd, especially Lucas Pereira, James Davis, Sean Anderson, Dave Koller, Kari Pulli, Matt Ginzton, John Shade, and Daniel Wood. Take it from me, if you're ever stuck scanning rocks at 3 am, these are the people to call. Around the lab, I've interacted with my officemates Kekoa Proudfoot, Matt Pharr, and Pradeep Sen, as well as Cindy Chen, François Guimbretière, Steve Marschner, Tamara Munzner, Ravi Ramamoorthi, and many others. I thank them for making the Lab *the* spot for computer graphics, as well as a great place to do one's graduate studies.

I would like to thank my committee, consisting of Pat Hanrahan, Leo Guibas, Mark Horowitz, and Bernd Girod, for their encouragement and their enthusiasm about the project. I would also like to thank my advisor Marc Levoy for his support over the years, for the insightful conversations we've had on every imaginable topic, and for giving me the opportunity to work on so many interesting projects while at Stanford.

Finally, I thank my family for their continuous support.

Contents

Tables

Figures

"Reality is just a convenient measure of complexity."

– Alvy Ray Smith

Chapter 1

Introduction: Sampled Geometry in Computer Graphics

Research in the field of computer graphics has traditionally been focused on three goals: representing the shape of 3D objects (modeling), describing how those objects move (animation), and simulating light transport in a scene to produce photorealistic images (rendering). Producing convincing images and movies of of complex environments, however, requires considerable detail in the 3D models, skill in producing natural, nuanced motions, and experience in selecting appropriate models for light reflection and transport. Because the stages of this pipeline have traditionally required human input, large applications of computer graphics (e.g., computer-animated feature films) have been expensive and slow to produce.

The desire to reduce the dependence on human input in making realistic images of complex scenes, combined with the increasing capabilities and decreasing costs of computer-controlled sensors, has over the past ten years resulted in a broadening in the scope of computer graphics research. Since the images produced by computer graphics frequently depict actual objects (or at least things *based* on actual objects), interest has been growing in supplementing the traditional computer graphics pipeline with measurements of the real world. For example, the shape of objects may be acquired with 3D scanners, their movement may be measured with motion capture apparatus, and their reflectance can be obtained from images. This dissertation focuses on the first of these, namely measuring the 3D shape of real-world objects.

1.1 Applications of 3D Scanning

In addition to computer graphics, 3D scanners have been applied in a wide variety of fields:

- **Computer vision:** The goal of computer vision is usually considered to be object recognition and scene understanding. As part of this process, a 3D model of the scene is often constructed, either explicitly or implicitly.

- **Robotics:** Autonomous robot navigation requires some representation of the location of obstacles in the environment. Robot-mounted 3D scanners have been explored as a mechanism for building geometric models of unknown areas.

- **As-built models of buildings:** It is sometimes the case that blueprints corresponding to the actual state of a structure are not available. 3D scanning has been used to build models of buildings for structural analysis, walkthroughs, and restoration.

- **Medicine:** MRI and CT scanners are regularly used in medicine to visualize internal organs, perform diagnosis, and plan surgery. More conventional 3D scanners (i.e., surface-based, not volumetric) have been used to determine the position of a patient in the operating room, aligning previously-obtained volumetric 3D data with the patient's current position.

- **Art and art history:** Scanned models of sculptures make it easy to study the working techniques and design choices of artists. For art historians, such models provide lasting documentation of works of art, and allow planning and visualization of proposed restorations.

The last of these applications was demonstrated by the Digital Michelangelo Project [Levoy 00]. The purpose of the project was not only to push the state of the art in the design of range scanners and processing algorithms, but also to demonstrate the applicability of 3D scanning in the computational humanities. The project showed that current range scanning systems are capable of acquiring large, detailed models of complex objects. However, creating these models required a custom-designed scanner, a team of 30, months of scanning, and thousands of man-hours of postprocessing. Thus, it underscored the fact that present-day 3D scanners are typically slow, expensive, and difficult to use.

Scanning vs. Model Acquisition: Let us distinguish between two general ways in which 3D scanning may be used. First, the scanner could return depth information about a (possibly moving) object from a single point of view. In this case, the output of the scanner is a series of *range images*, which are similar to ordinary images in that they represent a single depth (as compared to a color) at each of a set of "pixels," i.e. along each of a set of rays in space. Such sequences of range images may be obtained for either rigid or nonrigid objects, and are useful for applications such as metrology (e.g., in manufacturing) or in cases where the data is actually a height field (e.g., terrain). In addition, sequences of range images may be used for image based modeling and rendering (IBMR) applications such as view interpolation or foreground/background segmentation. These range images, however, only represent the part of an object seen from a single point of view.

The other way of using 3D scanning is to build complete models of rigid objects. In this case, the object must be moved relative to the scanner (or the scanner moved relative to the object) in order to obtain and integrate views of the object from all sides. We shall refer to this use of scanning as *3D model acquisition*. Such a system must incorporate not only a 3D scanner, but also methods for aligning and merging together the multiple views. Most of the applications listed above fall into this category.

1.2 Statement of Problem

As stated above, present-day 3D scanners tend to be slow, expensive, and difficult to use, especially for model acquisition applications. Thus, although 3D scanning has been successfully applied in a variety of fields, one may imagine many more potential uses of 3D model acquisition that are currently not practical:

- Scanning of movie sets (for insertion of computer-generated special effects) has not been used widely, because current scanning systems are too slow.

- 3D models could be used for applications such as advertising or product catalogs on the World Wide Web, but current scanners are too difficult to use to make it practical to create such large numbers of models.

- One may imagine consumer applications of 3D scanners. For example, if it were easy to add 3D information to the photographs taken by digital cameras, an enhanced photo

editing package could automatically and robustly segment objects in the pictures, permitting per-object image processing or object removal. Current 3D scanners, however, are orders of magnitude too expensive for such mass-market applications.

This dissertation proposes a new design for a 3D model acquisition system that attempts to remedy some of the problems suggested above. In particular, the scanner is designed to be fast, inexpensive, and easy to use. We describe a complete pipeline that satisfies these criteria, and present results from a prototype implementation of this pipeline.

1.3 System Design

We have stated that our goals are to design a 3D model acquisition system that is fast, inexpensive, and easy to use. We hypothesize that one major reason that current systems do not meet these criteria is that their designs are not optimized for the entire model acquisition pipeline. Instead, they focus on optimizing just the scanner according to some criterion (speed, accuracy, or cost) and leave the rest of the pipeline to separate software packages. The result is that such systems require large amounts of human-assisted postprocessing at the alignment and merging stages, and provide poor feedback for planning scans. In contrast, we observe that if we examine our goals with the idea of optimizing for the *entire* pipeline, certain design choices are naturally suggested:

Speed: If one wishes to optimize an end-to-end scanning system for speed, one must obviously begin by acquiring range images as quickly as possible. This suggests range scanning technologies that obtain an entire range image at once, as opposed to just a single stripe or a single point.

A second aspect of scanning speed becomes relevant in the model acquisition application: it is necessary to move the scanner relative to the object to scan all parts of the surface. The question of where to move the scanner, that is finding the "next best view" given the data that has already been scanned, is the source of much of the inefficiency in the current methodology for model acquisition. Particularly in the later, "hole-filling" stages of scanning an object, it takes a long time for the user to perform a scan, evaluate whether it filled a hole, and plan the next scan, regardless of whether automatic [Maver 93] or manual next-best-view methods are used. Thus, one of the important factors in designing a model acquisition system is ensuring that rapid feedback about the efficacy of the current scan is available.

An example of the effectiveness of immediate feedback is the real-time display of the ModelMaker scanner by 3D Scanners Corp. Even though this scanner only obtains a single stripe of data at a time and is slightly cumbersome to use because it is mounted on a jointed arm, it was observed that it was useful for tasks such as filling small holes largely because it displayed the data being acquired in real time. For our system, of course, we would like our entire pipeline, including not only the scanning but also the alignment and display, to be fast enough to permit rapid planning of the next view.

Cost: In order to limit the cost of our system, we should use off-the-shelf components that either are currently inexpensive or that technological trends suggest will become inexpensive. As we will see later, our prototype system is based on the technology of projected structured-light triangulation. It requires only a consumer video camera and a portable video projector, both of which are rapidly declining in price. The software runs on a present-day PC, but could potentially take advantage of increasing CPU speeds over the next few years.

The other aspect of expense that we consider is motivated by the model acquisition application: some method is necessary for moving the scanner relative to the scene. As observed by [Davis 01], a major source of expense in many range scanner designs is the need for calibrated motion stages. Thus, our design should aim to not require any calibrated motion or tracking. As described later, our scanner has no requirements for calibrated motion, and in fact may be moved by hand (or, equivalently, the object may be moved).

Ease of Use: Two major obstacles to ease of use in current model acquisition systems are the difficulty in determining where to take scans and the need to perform manual initial alignment after uncalibrated motion. The first is the "next best view" problem mentioned above. Proposals have been made to try to solve the latter problem [Huber 01], but since these approaches typically involve an exhaustive search they are time consuming and may have robustness problems for certain objects.

As an alternative, we propose a system in which the entire pipeline of scanning, alignment, merging, and display runs in real time. This is a natural extension of trying to make feedback from the system as fast as possible (as proposed above), but actually making the pipeline real-time introduces a significant new paradigm for interaction with the system. In particular, if the user always sees the state of the model and the effectiveness of the current scans, there is a tight feedback loop wherein the user may continuously move the scanner or object to many orientations to fill holes.

The capability of always seeing the partially-completed model as it is being scanned appears to solve both of the usability problems mentioned above. Because the user sees where data has been acquired and where there are holes, it becomes easier to evaluate where to take scans. Because all alignment is done automatically in real time, there is no need for manual input. Finally, we observe that because the pipeline operates in real time, the distance between two consecutive frames is small; this leads to high robustness in alignment, compared to systems that must deal with scans that are far apart.

1.3.1 Technologies

Let us now examine some of the technologies and algorithms for 3D scanning, alignment, merging, and rendering that fit the above goals. Given our proposal to have the pipeline running interactively, we focus on those technologies that may be adapted for real-time use.

Rangefinding: Range scanning may be broadly divided into active and passive methods. Although real-time passive methods based on stereo [Faugeras 93a] or silhouettes [Matsumoto 97, Matusik 00] have been proposed, we will focus on active methods, since they typically perform well in the absence of scene texture, are computationally inexpensive and robust, and return accurate, densely-sampled range data.

Active range scanning methods may be based on time of flight [3DV Systems], depth from defocus [Nayar 96], photometric stereo [Rushmeier 97], or projected-light triangulation (see [Curless 97] for a more detailed taxonomy). Of these, the systems based on triangulation may be used with the largest range of scene sizes and have the lowest hardware costs, especially given the increasing capabilities and declining prices of computer-controlled projectors and video cameras. Triangulation-based scanners have been applied in industrial contexts, and some of them are capable of returning a range image in under a second (such as the scanning-stripe scanner by Minolta or the time-coded projected-light system by Steinbichler). Comparatively little research has been done, however, on applying triangulation methods to scenes containing moving or deforming objects. Previously-studied systems that allow object motion during scanning either use custom-designed hardware (e.g., the real-time scanning-stripe system of [Gruss 92]), make strong continuity assumptions about the scene, or are variants of passive vision methods that use a projected texture to aid in solving the correspondence problem.

For our model acquisition system, we introduce a new triangulation-based range scanner for moving scenes that returns dense range images in real time (60 Hz). The scanner uses a new

kind of illumination pattern, based on time-coding the boundaries between projected stripes. The boundaries are tracked from frame to frame, permitting the determination of depths even when there is relative motion between the scene and the scanner.

Alignment: Since the first stage of our pipeline only produces range images, it is necessary to obtain views of an object from multiple angles and to align these range images into a common coordinate system. In general, there are three classes of methods that have been considered for this application. The first involves known motion: the object and the scanner are moved relative to each other by a calibrated rotational or translational stage. As mentioned earlier, however, the key usability improvement in our design comes from lifting the restriction to calibrated motion and allowing the object and scanner to be moved freely with respect to each other.

A second way of obtaining the alignment between range images is to place a tracker on either the object or the scanner (whichever is moved relative to the other). Although we have chosen not to use this option because of accuracy and cost considerations, we believe that in many circumstances it would provide substantial benefits in the context of our proposed pipeline. As discussed in Section 7.3.3, a separate tracker would be especially useful for preventing the global drift that results from our use of scan-to-scan alignment.

The option we use for alignment in our pipeline is based on registering individual scans to each other based on the geometry in overlapping areas. Automatic alignment of 3D shapes has been studied extensively in the computational geometry and computer vision communities, mainly in the context of aligning two scans with completely unknown initial orientation. Methods such as identification and indexing of surface features [Faugeras 86, Stein 92], "spin-image" surface signatures [Johnson 97a], computing principal axes of scans [Dorai 97], and exhaustive search for corresponding points [Chen 99] have been proposed for either aligning two scans or identifying a scan given a database of examples.

In the context of 3D scanning, the Iterative Closest Points (ICP) algorithm [Chen 91, Besl 92] has become the most frequently used method because it converges to the correct scan-to-scan alignment with high accuracy. The traditional problem with ICP has been the fact that it requires a good initial guess of the alignment in order to reach the correct answer. In the context of our real-time range scanner, however, we will be aligning scans that start out very close to each other, so there is little danger of converging to an incorrect local minimum. Therefore, we have chosen to use a variant of ICP optimized for fast alignment of range images.

Merging and Rendering: Since the first stages of our pipeline, namely the rangefinding and alignment, produce data at such a high rate, it is necessary to perform some sort of merging or discarding of data in overlapping regions. This ensures that the number of primitives to be displayed does not grow linearly with time, and helps maintain an acceptable interactive frame rate for the display.

The most popular methods of merging individual range images into complete models are based on either stitching the aligned scans [Turk 94] or averaging implicit volumetric representations of the scans and extracting an isosurface [Curless 96]. As has been observed, the first of these methods is not topologically robust in the presence of noise and small surface features. The implicit volumetric methods, conversely, are robust and offer high quality, but prove too slow for real-time implementation.

The key observation behind our approach to merging range scans is that the real-time merging procedure only needs to be good enough to give the user an idea of how much of the surface has been scanned and whether any holes remain. Thus, our merging algorithm consists of simply quantizing each measured range sample to a 3D grid, and merging all the samples that fall within a single grid cell. An average normal is computed at each cell, and point rendering is used to draw the accumulated samples. A high-quality final reconstruction may be completed offline as a postprocess, since the user knows that the entire surface has been scanned.

The rendering is done using a method called "splatting," in which a screen-aligned splat (e.g. a circle or an alpha-blended Gaussian) is drawn for each point. These are scaled such that the splats for neighboring points overlap without leaving a gap. Thus, a complete rendering of the surface is produced directly from the scattered point data, without the need to triangulate the points or to reconstruct a consistent manifold surface.

1.3.2 Pipeline

Given the above design choices, our model acquisition pipeline is illustrated in Figure 1.1. A time-varying pattern consisting of black and white stripes is projected onto the scene using a DLP projector, and a standard NTSC video camera is used to capture video frames. The frames are processed to find and track the boundaries between the stripes, and a range image is obtained for each video frame. Successive range images are aligned to each other, integrated into a 3D grid, and rendered in real time.

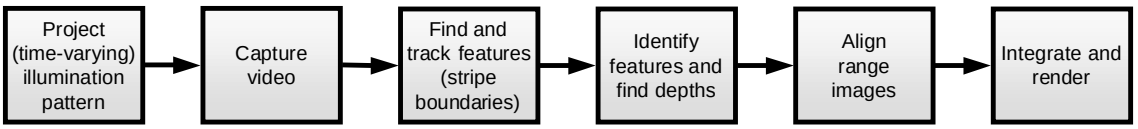


Figure 1.1: Real-time model acquisition pipeline.

1.4 Rendering of Large Models

As model acquisition systems such as the above become more powerful, it is becoming difficult to display the output of these systems at interactive rates. Traditional algorithms for display, simplification, and progressive transmission of meshes are impractical for data sets of this size. Moreover, many such techniques focus on optimizing the placement of individual edges and vertices, expending a relatively large amount of effort per vertex. Scanned data, however, has a large number of vertices and their locations are often imprecise due to noise. This suggests an alternative approach in which individual points are treated as relatively unimportant, and consequently less effort is spent per primitive. Recent research employing this paradigm includes the point-based display of our model acquisition system, the spline-fitting system by Krishnamurthy and Levoy [Krishnamurthy 96], the range image merging system by Curless and Levoy [Curless 96], and Yemez and Schmitt's rendering system based on octree particles [Yemez 99]. These algorithms do not treat range data as exact, and in fact do not preserve the 3D locations of any samples of the original mesh.

QSplat is a system for representing and progressively displaying large meshes that combines a multiresolution hierarchy based on bounding spheres with a rendering system based on points. A single data structure is used for view frustum culling, backface culling, level-of-detail selection, and rendering. The representation is compact and can be computed quickly, making it suitable for large data sets. The implementation launches quickly, maintains a user-settable interactive frame rate regardless of object complexity or camera position, yields reasonable image quality during motion, and refines progressively when idle to a high final image quality. The system may also be extended to progressive view-dependent transmission of large 3D models across a network of limited bandwidth. We have demonstrated the system on scanned models containing hundreds of millions of samples.

1.5 Outline of the Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 describes the design of a video-rate 3D scanner for moving scenes. Chapter 3 analyses the ICP algorithm, used for aligning 3D meshes, and describes a variant that can run in real time on present hardware. Chapter 4 describes the design of a system that combines the real-time 3D scanner, real-time alignment, and algorithms for merging and display of the partial model. Results are presented from a prototype of this model acquisition pipeline.

Chapter 5 describes QSplat, a system for real-time interactive display of large 3D models. It presents results from the models produced by the Digital Michelangelo Project, which contain 100 million to 1 billion range samples. Chapter 6 describes how QSplat models may be streamed across a network of limited bandwidth, and examines user interaction issues for this streaming system. Finally, Chapter 7 presents conclusions about both the model acquisition system and QSplat, and suggests ideas for future work.

"Begin to cast a beam on th' outward shape."

– John Milton

Chapter 2

Structured-Light Range Scanning for Moving Scenes

As mentioned in Chapter 1, our 3D model acquisition pipeline begins with a real-time (60 Hz) range scanner based on projected structured-light triangulation. In designing this scanner, we start by presenting a classification of structured-light scanning methods according to the reflectance, spatial, and temporal coherence assumptions they make. We show how previous approaches fit in this taxonomy, and present a new set of assumptions that leads to a system optimized for moving scenes. We show how to formulate the search for a suitable set of codes as a graph problem, and present one possible set of illumination patterns. Finally, we describe the boundary tracking and decoding algorithms used by our prototype implementation.[†]

2.1 Analysis of Continuity Assumptions

The assumptions made in designing a structured light system are often stated only implicitly, despite their importance. We characterize the space of structured light methods in terms of the underlying assumptions they make about the reflectance, space, and time coherence of the scene. We then argue that there exists a relatively unexplored, practical combination of assumptions that leads to a new class of structured light scanning methods.

[†]The author wishes to acknowledge that both the system described here and much of the text of this chapter were written together with Olaf Hall-Holt.

2.1.1 Background

Our range scanner is based on the principle of *projected-light triangulation* [Posdamer 82, Besl 88]. In its simplest variant, this consists of illuminating the object being scanned with a stripe of light, and observing it with a detector (typically a CCD camera) placed at a known angle with respect to the light source. The side-to-side wiggles of the observed stripe correspond to the shape of a contour on the object; mathematically, we may determine the 3D locations of points on this contour by computing the intersections between camera rays and the plane of the projected light stripe (see Figure 2.1).

In order to increase the amount of data acquired by a triangulation-based scanner at each point in time, we may simply project multiple stripes onto the object. If this is done, however, some method is necessary for determining which stripe is which in the camera image. A variety of methods have been proposed for finding this correspondence between camera pixels and locations in the projected pattern (without loss of generality, we will refer to these pattern locations as "projector pixels"). Each of these methods, however, makes certain assumptions about how the scene transforms the projected light into the camera image. In abstract terms, we may think of the scene as a function that transforms projector pixels into camera pixels, and we may consider the restrictions each scanning method places on this transfer function.

One class of scanners uses color coding to determine the correspondence between camera and projector pixels. A pattern consisting, for example, of colored dots [Davies 96] or a gray wedge [Carrihill 85] is projected onto the scene, then the color observed at each loca-

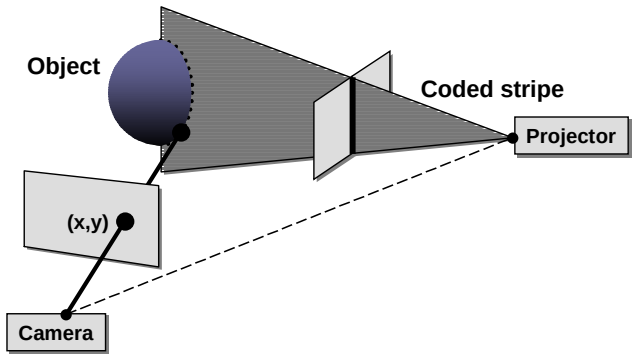


Figure 2.1: Layout of a single-camera, single-source triangulation system. The 3D positions of points on the object are determined from the intersection between the camera ray and the plane of light produced by the illumination source.

tion is used to determine the camera-projector correspondence. This type of method assumes that the scene does not modify the colors from the projector to the camera, thus restricting the allowable transfer functions to those that do not modify colors. Effectively, this makes a *reflectivity* assumption about the scene.

Another class of scanners encodes information into a pattern in some neighborhood of projector pixels. For example, Boyer and Kak describe a system in which location in the projector image is identified by a color coding of adjacent stripes [Boyer 87]. The correspondence between camera pixels and projector location can therefore be made only if the scene "transfer function" preserves spatial neighborhoods. We will call this a *spatial coherence* assumption. Note that for this system, violation of the coherence assumptions may cause small areas of incorrect depths around the discontinuities. Thus, the system assumes *local* scene coherence. In contrast, there exist systems for which the spatial coherence assumption must be valid *globally*. For example, Proesmans et al. describe a system in which a grid pattern is projected onto the scene, and the grid lines visible in the camera image are followed to determine correspondences between points in the camera and projector images [Proesmans 96]. In this case, a single visible surface is assumed, since the system relies on counting grid lines.

A third way of encoding information into the projector signal is to group pixels over time rather than space. For example, in the system of Sato and Inokuchi [Sato 87], the projector pixels are turned on-and-off over time, so that when a camera pixel records a particular on and off intensity pattern, the corresponding projector pixel can be identified. A popular choice for this on-off pattern is a set of Gray codes [Bitner 76], though other approaches using black-and-white [Gartner 96], gray-level [Horn 99], or color [Caspi 96] stripes, or swept laser stripes or dots [Rioux 94], have been examined. The common underlying assumption is *temporal coherence* of the scene, namely that neighborhoods of pixels can be identified over time. Note that all of the methods mentioned here make a *global* temporal coherence assumption (i.e., that the scene is static); we will discuss what it means to make a *local* temporal coherence assumption in the following section.

The design space for projected light illumination patterns can therefore be described in terms of the reflectance, spatial coherence, and temporal coherence assumptions they make. The strength of a spatial coherence assumption can be measured by the number of pixels involved: if the patterns to be identified in a given camera image require a minimum of n pixels, then the smallest identifiable features in the scene must occupy at least n camera pixels. The

reflectance assumption impacts the range of colors permitted in the scene, as well as the frequency of textures.

Our goal in designing an illumination pattern is to enable the scanning of moving scenes, with small feature sizes and largely unknown reflectance properties. Stated in terms of the above assumptions, this means that we would like to simultaneously minimize the spatial coherence, reflectance, and temporal coherence assumptions required by our system. This goal leads to the adoption of the following assumptions:

- Most of the time, two horizontally adjacent camera pixels will see the same surface.
- Most of the time, the reflectance of two horizontally adjacent pixels is similar.
- Most of the time, projected features that persist for n frames can be used in making correlations. In our implementation, we use $n = 4$.

The last of these assumptions has not to our knowledge been used in the same way before, so we will now look at the implications of motion for structured light systems, in particular explaining this last assumption more fully.

2.1.2 Space-time Coherence

In order to formalize the notion of temporal coherence, as well as to define a metric of the strength of such assumptions (as we did for spatial coherence), we consider the appearance of a scene illuminated by a sequence of patterns in (four-dimensional) space-time. We begin by examining the simplified case of two-dimensional range scanning (i.e., working in a plane), with time as a third dimension.

Let us consider a scene with objects moving in the plane. By stacking up a series of snapshots of the scene at different times, we see that the objects trace out volumes in three-dimensional space-time (Figure 2.2). A time-varying light pattern projected onto the objects can then be visualized as a two-dimensional light pattern projected onto these volumes, and a series of (one-dimensional) camera images of the illuminated objects may be thought of as a 2D picture of the volumes. Thus, there is a direct correspondence between moving-object, multiple-pattern range scanning in the plane and single-frame correspondence methods (sometimes called one-shot methods) in 3D.

Given this analogy, we may apply some concepts of 3D one-shot methods to the case of 2D moving-object range scanning. For example, 3D one-shot methods often use light pat-

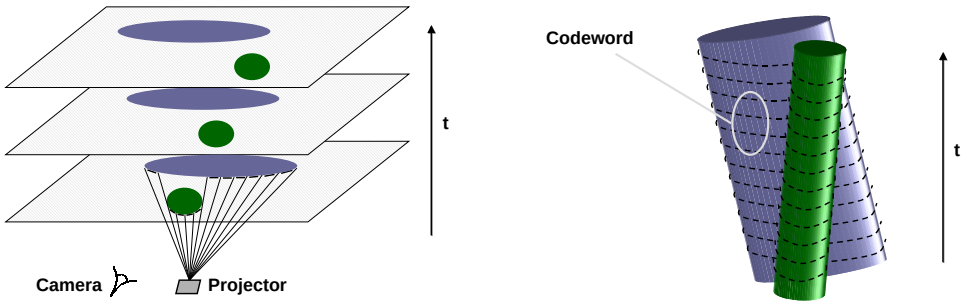


Figure 2.2: A 2D (planar) scanning application examined in 3D space-time. At left, we show a scene being scanned at three points in time. At right, we show how the trajectory of each object may be thought of as defining a solid in space-time. A sequence of 1D projected patterns becomes a 2D pattern projected onto the objects in space-time, and local neighborhoods (defining codewords) have both spatial and temporal extent. The relation between 3D scanning and 4D space-time is analogous.

terns in which two-dimensional neighborhoods specify codes. In the 2D moving-object case, such neighborhoods have both spatial and temporal extent, but the idea of communicating a code via these two-dimensional neighborhoods can still be used. Similarly, an analogy may be formed between following a continuous grid line (in 3D grid-based methods) and tracking a feature across several frames (in the 2D case). The same analogies may be made between moving objects in 3D and a single pattern in 4D.

Thus, we see that the notions of temporal and spatial coherence may be combined into a single concept of coherence in space-time. We may then classify particular spatio-temporal coherence assumptions according to the extent of space-time neighborhoods over which the moving scene is assumed to be continuous, and may therefore convey codes. Thus, our assumption of temporal coherence over 4 frames means that we are assuming neighborhoods of eight pixels to form correspondences in space-time: two in each frame. This implies the ability to *track* the movement of this pair of pixels from one frame to the next.

2.1.3 Motivation for Stripe Boundary Codes

The assumptions given above limit the set of scenes that our system will be able to scan. Making these assumptions explicit, in our case, also provides a set of tools for designing an illumination pattern. We proceed with a sort of worst-case analysis: if the scene behaves arbi-

trarily badly, except where constrained by the above assumptions, what approach to forming correspondences will still work?

- Since our spatial coherence assumption involves only horizontally adjacent pixels, pixels on different scan lines of the camera may not have anything in common, and must be treated independently. Due to the epipolar constraint, the independence of scan lines of the camera implies independence of rows of pixels on the projector, and therefore we can design the projector illumination pattern independently for different horizontal rows. In particular, we use the same illumination pattern for each row, which forms a stripe pattern.

- In order to allow the greatest possible variation in scene reflectances, our system uses only black and white stripes.

- Since at most two pixels of a given frame can be used together to infer correspondences, the only projected feature that we can reliably identify is the boundary between two stripes. Thus, we will use a *stripe boundary code* to convey information between projector and camera.

Focusing on stripe boundaries has several advantages. For example, if the stripes on either side of a stripe boundary can each be assigned n different codes over time, then roughly n^2 distinct stripe boundaries can be identified in a camera image, even if no scene feature is as wide as a stripe. Also, under appropriate assumptions on the smoothness of the scene geometry and texture, the stripe boundary may be located with sub-pixel accuracy, thus increasing the ultimate accuracy of the scanning system.

At this point the design of an illumination pattern has been considerably constrained. The next section describes a method for generating stripe boundary codes that can incorporate a variety of additional design criteria.

2.2 Designing a Stripe Boundary Code

Designing a stripe boundary code involves assigning a color (black or white) to each stripe at each point in time, such that each stripe boundary has a unique code (consisting of the black/white illumination history on both sides of the boundary) over the sequence of four

frames. To maximize spatial resolution, we would like the pattern to contain as many boundaries as possible. In order to generate such patterns, we form an analogy between stripe boundary codes and paths in a graph, and search for a maximal-length path that satisfies certain conditions.

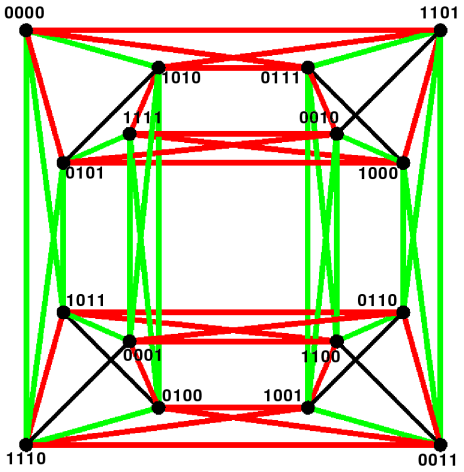
Consider a graph in which each node represents the black/white illumination of a single stripe over four frames, and all pairs of nodes are connected with edges. Each edge represents a (time-varying) stripe boundary, since it lies between two (time-varying) stripes. This graph has $2^4 = 16$ nodes and 120 (undirected) edges. A path through this graph that traverses each edge at most once in each direction corresponds to a stripe boundary code, since it specifies the assignment of colors to stripes over time, with the property that the code of each boundary is unique.

We may place additional restrictions on the stripe boundary codes by modifying the graph and adding requirements on how it is traversed. First, we avoid any stripe boundaries that do not vary over time, since they will be indistinguishable (from the point of view of the camera) from texture boundaries. This restriction is manifested in the graph by deleting the edge between the two time-invariant color progressions, namely the edge between all black (0000) and all white (1111).

A second, more sweeping restriction arises as follows. In the complete graph, we allowed "boundaries" between any two stripes, including two stripes of the same color. Such a boundary cannot be seen in a camera image, so we call it a "ghost" boundary. In our case, if we eliminated all ghost boundaries, we would eliminate all but eight edges in the graph, which would lead to very short solution paths.

In order to allow some ghost boundaries, there must be a way to identify them implicitly in camera images. We enable such a determination with the help of the following restriction: we number the stripe boundaries according to their position along a projector row, from 1 to m, and restrict the locations of ghost boundaries to odd-numbered positions in frames 1 and 3 and even-numbered positions in frames 2 and 4. Thus, a given stripe boundary will be visible at least every other frame, and in any frame there will be at most one ghost between any two visible boundaries.

To embed this restriction in the graph, we color edges according to the ghost boundaries they contain: an edge is green (light gray) if there are no ghost boundaries in frames 1 and 3, and red (dark gray) if there are no ghost boundaries in frames 2 and 4. Edges that are both green and red are colored black, and edges that are neither red nor green are deleted from



Traversal

0000	0001	1011	1001	1100	1001	1001	0011
0101	0110	0110	1110	0001	1100	0100	0100
1011	1101	1100	0000	1010	0011	1111	1110
1100	0000	1001	1101	1111	1100	1010	0011
0111	1110	0110	0111	0101	0110	0100	1101
1010	1001	0001	0010	1010	0011	1011	1010
0000	0011	1011	1101	0101	1000	0001	0001
0111	0110	1110	0010	0000	0111	0100	1100
1000	1000	0001	1000	1010	1101	1010	0010
0101	1111	1110	1101	1101	1000	0111	0101
1111	0100	0100	0011	0110	0110	1100	1110
1000	1001	0001	1110	1011	1001	1011	1011
0010	0111	1111	0101	0100	0010	0101	0000
1111	0000	0010	0010	0011	0111	1000	

Figure 2.3: At left, a graph used to determine a set of stripe boundary codes. We look for a traversal of the 55 edges in this graph (with the understanding that each edge may be traversed once in each direction), with the added constraint that red (dark gray) and green (light gray) edges must alternate along the path. A black edge may be substituted for either red or green. As mentioned in the text, the edge between 0000 and 1111 is missing: we disallow this stripe boundary as a valid code, since it is too easy to confuse with static texture. At right, a maximal-length ($2 \cdot 55 = 110$ directed edges, 111 nodes) traversal.

the graph, since they will not be used in any path. This restriction reduces the number of (undirected) edges to 55, resulting in the graph of Figure 2.3. We then constrain our search to paths in which the colors of traversed links alternate between red and green (where black is understood to stand for either).

Since the problem as stated has many solutions, we may impose additional conditions:

- We would like the effect of errors (i.e., misidentifying a stripe boundary) to be as large as possible, so that outliers are easier to identify and filter away. (This is the opposite of the strategy adopted by [Jiang 94], in which the aim is to minimize the size of errors.)
- We look for codes in which the distribution of stripes of widths 1 and 2 (or, equivalently, the distribution of ghosts) is as uniform as possible within each frame.

A maximal-length code will traverse each of the 55 edges of the graph twice (once in each direction), so will have $2 \cdot 55 = 110$ stripe boundaries (and 111 stripes). Since we only need to find the code once, we may use a brute-force algorithm to search for paths in the graph. An example of such a code is shown in Figure 2.4.

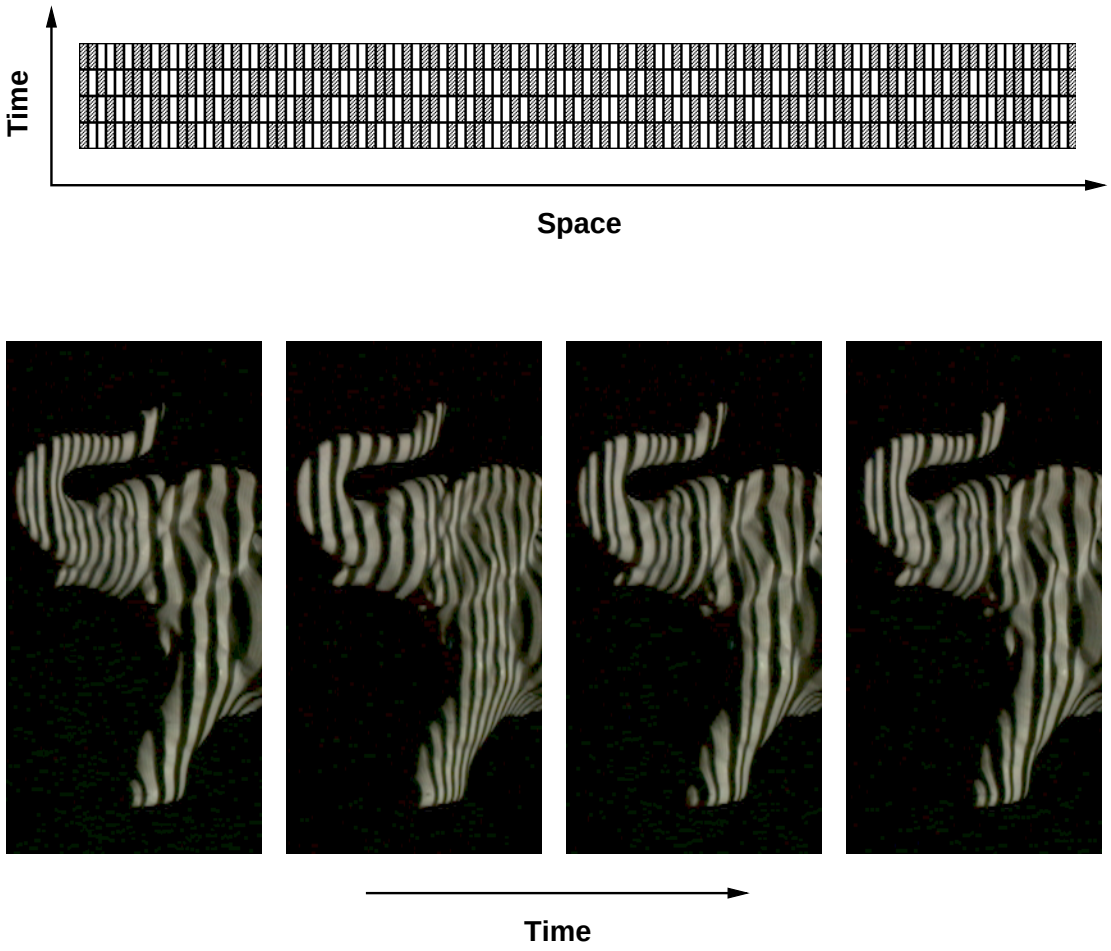


Figure 2.4: At top, a four-frame sequence of projected patterns satisfying the conditions in Section 2.2. Each pattern has 110 stripe boundaries (111 stripes), with the property that each stripe boundary has a unique code (consisting of the black/white illumination history on either side of the boundary over the sequence of four frames). At bottom, a sequence of video frames of an elephant figurine illuminated with a stripe boundary code.

2.3 Implementation

We have implemented a prototype range scanning system based on the stripe boundary codes described above – see Figure 2.5. In our system, a micromirror-based projector cycles through the illumination patterns at 60 Hz., and a standard NTSC video camera is used to capture images. The video is digitized and processed in real time, and the system generates a new range image every $^1/_{60}$ sec.

Our algorithm for depth extraction consists of segmenting each video field into illuminated and unilluminated regions (i.e., "black" and "white"), finding stripe boundaries, matching these boundaries to those on the previous field, and using information about the illumination history of each boundary to determine the plane in space to which it corresponds. Depth is then obtained via ray-plane triangulation.

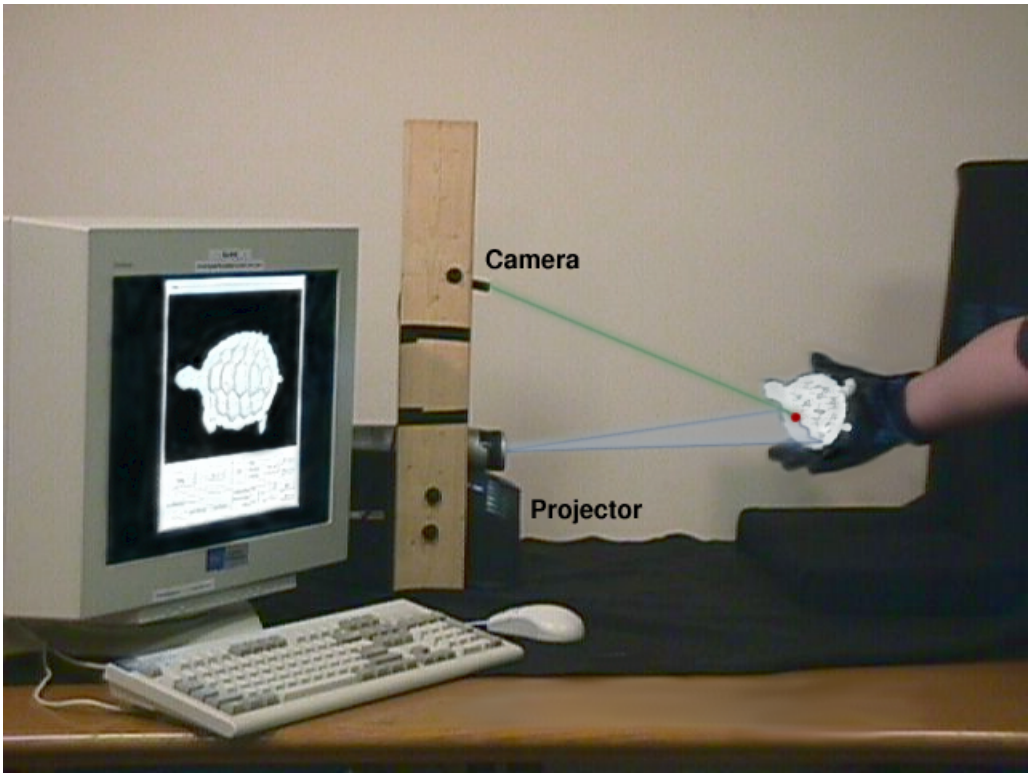


Figure 2.5: Photograph of our prototype system.

We now discuss the tradeoffs made in implementing each stage of this pipeline on current hardware, as well as possible extensions to these algorithms to make them more robust as hardware capabilities increase.

Pattern Projection and Video Capture: As mentioned above, our patterns are projected using a projector based on digital light processing (DLP) technology [Hornbeck 88]. These projectors have the advantage of being relatively inexpensive, and have very short transition times between patterns.

Because our projector and camera must be synchronized (so that we capture exactly one video frame for each projected frame), we have chosen to drive our projector with an S-video signal and to genlock our video camera to this signal. In addition, to prevent interpolation between projector pixels, we orient our pattern such that the stripes run along the scanlines of the projector. For this reason, we are currently limited to 240 projected stripes, as compared to the 1024 potentially available from our projector. Since we currently use a 4-frame sequence consisting of 111 stripes, this limitation is not significant. However, expanding to a larger number of stripes (to increase the working volume) would require driving the projector with a VGA or DVI signal, thus requiring a different method of projector-camera synchronization.

Since we use a standard video camera to capture frames, our captured video fields are interlaced. This results in a slight shift in the position of stripe boundaries from field to field. Since the effect is small, we currently do not correct for it in our processing pipeline, but because the effect of interlacing is completely known it would be possible to compensate for it. Note that any translation in the 3D model resulting from not considering the interlacing is corrected by our frame-to-frame alignment (as described in the next chapter).

Segmentation Algorithm: The problem of finding the stripes (and hence the stripe boundaries) in a captured video frame may be considered a special case of the general segmentation and edge detection problems. Both of these problems have been studied extensively in the computer vision community and many sophisticated algorithms are available [Faugeras 93b]. In our application, however, we need a method that is robust and runs in real time, while taking advantage of the known features of the projected illumination.

In particular, given the assumption of local reflectance coherence, we may assume that the highest-frequency variations in the captured frames are due to illumination, not texture. Moreover, we may assume that the projected stripes (and hence the edges we wish to find) are roughly perpendicular to the camera scanlines. Therefore, we process each scanline in-

dependently, looking for local maxima and minima along each row, and assume that these correspond to white and black projected stripes, respectively. Between each adjacent local maximum and minimum, we look for a pixel with intensity halfway between that of the minimum and maximum (optionally using subpixel interpolation), and use this as the location of a stripe boundary.

For scenes without high-frequency textures, we have found this method to be effective and robust, while still running in real time. In particular, we have found this algorithm less sensitive to variations in reflectivity and changes in ambient illumination than both threshold-based segmentation methods and derivative-based edge detectors.

Stripe Matching Algorithm: Since our approach relies on time-coding the boundaries between stripes, a critical part of our algorithm is matching the boundaries visible in each frame to those in previous frames. This is a nontrivial problem for two reasons. First, the boundaries move from frame to frame, potentially with large velocities. Second, the fact that our code contains "ghost" boundaries means that not all boundaries are visible in each frame.

It is the presence of ghosts (i.e., the inferred black-black and white-white stripe "boundaries") that distinguishes our stripe matching problem from the traditional feature tracking literature. To make the problem tractable, we must use the constraints presented in Section 2.2, namely that there may be at most one ghost between each pair of visible stripe boundaries, and that ghost must match to a visible stripe boundary in the previous and following frames. These conditions limit the possible matches and allow us to determine, in many cases, whether certain boundaries should match to other visible boundaries or to ghosts. Even these conditions, however, are not enough to disambiguate the situation shown in Figure 2.6. The two possibilities of having the center stripes match to each other and having them match to ghosts in the other frame are both allowed by the constraints mentioned above.

Although there is a large literature on tracking algorithms that could potentially be adapted to our application, including multiple-hypothesis methods [Reid 79] and methods that use velocities [Brown 97], most of these approaches are too slow for real-time implementation. Therefore, we currently implement only a simple matching algorithm that hypothesizes all possible locations of ghosts and matches each visible boundary to the closest stripe or hypothesized ghost in the previous frame. As discussed later, this places a constraint on the maximum allowable velocity of stripes, hence limiting the speed at which objects in the scene can move.

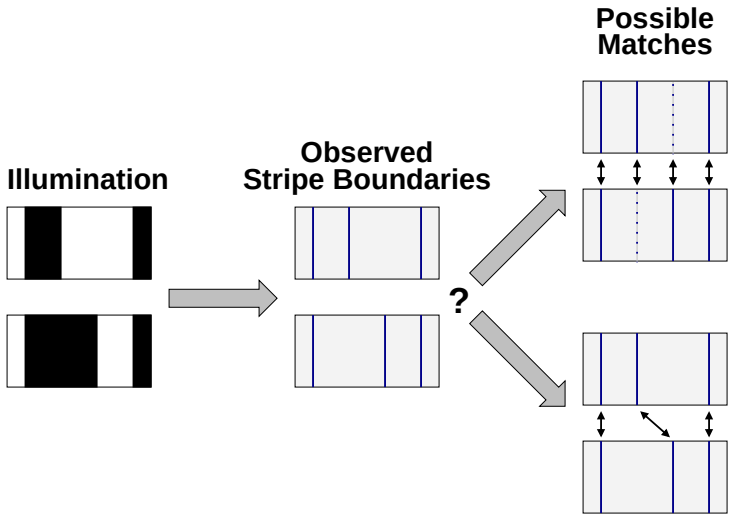


Figure 2.6: Matching stripe boundaries becomes difficult in the presence of "ghosts" (i.e., boundaries that could not be observed). Here, we show one possible ambiguity, namely whether the center stripe boundaries in the two frames should match to each other or should match to "ghosts" in the other frame.

We anticipate that future systems may incorporate better matching heuristics, permitting correct stripe matching in the presence of greater frame-to-frame motion.

Decoding Algorithm: Once we have matched the stripe boundaries in one frame to those in the previous frame, we propagate the illumination history (i.e., the color of the stripes on either side of the boundary over the past four frames) of the old boundaries to the new ones. If we have seen and successfully tracked this boundary for at least four frames, this history identifies it uniquely. Note that the boundary remains identified at *every* frame thereafter, since the four-frame illumination history contains all four patterns.

Triangulation: Given a stripe boundary identification, we determine the plane in space to which the boundary corresponds. We then find the intersection of that plane with the ray corresponding to the camera position at which the boundary was observed; this determines the 3D location of a point on the object being scanned. An important difference between our approach and traditional projected-stripe systems based on Gray codes is that this scheme only gives us depth values at stripe boundaries. These depths, however, are very accurate: we triangulate with an exact plane (the stripe boundary), rather than a wedge formed by two planes (the stripe itself). For smooth surfaces without high-frequency texture, we may perform

sub-pixel estimation of the location of the stripe boundaries to further reduce depth errors (see Section 4.3).

This triangulation process requires knowing the internal parameters of both the camera and projector, as well as their relative pose. In order to calibrate intrinsics, we currently use the method of [Heikkilä 97]. We follow this by moving a target to known 3D positions and optimizing to find the relative pose of the camera and projector. In the future, one could imagine an automatic calibration method that would permit a calibration target to be moved around (by hand), then simultaneously solve for the scanner calibration and the positions to which the target was moved (a similar approach was demonstrated by [Pollefeys 99]).

2.4 Results

We have tested our implementation on a scene consisting of an elephant figurine, approximately 10 cm in size, rotated by hand at approximately 1 cm/sec. Figure 2.7 shows the performance of the stripe finding and matching stages. We see that the system identifies most of the stripes four frames after they are introduced. In Figure 2.8, we compare our complete stripe boundary code system to a conventional Gray code-based system for this scene. Note that because the elephant was moving, the Gray code method gave erroneous results in regions in which stripes moved from frame to frame. In contrast, our system produced correct results in the moving-object case.

2.4.1 Limitations

Although we have demonstrated a system capable of real-time range scanning, our implementation has several limitations on its applicability:

Texture: As mentioned in Section 2.3, we currently assume that scene texture varies slowly in order to segment illuminated and unilluminated regions. If the scene does contain step-edges in texture, our segmentation algorithm may report false positives (as well as some false negatives). Even though we have designed our code such that static stripe boundaries correspond to illegal codewords, our system may obtain incorrect geometry in the presence of moving texture.

Structured-light methods for static scenes often compensate for texture by adding an additional all-white frame. This allows them to determine the reflectance seen at each pixel, and

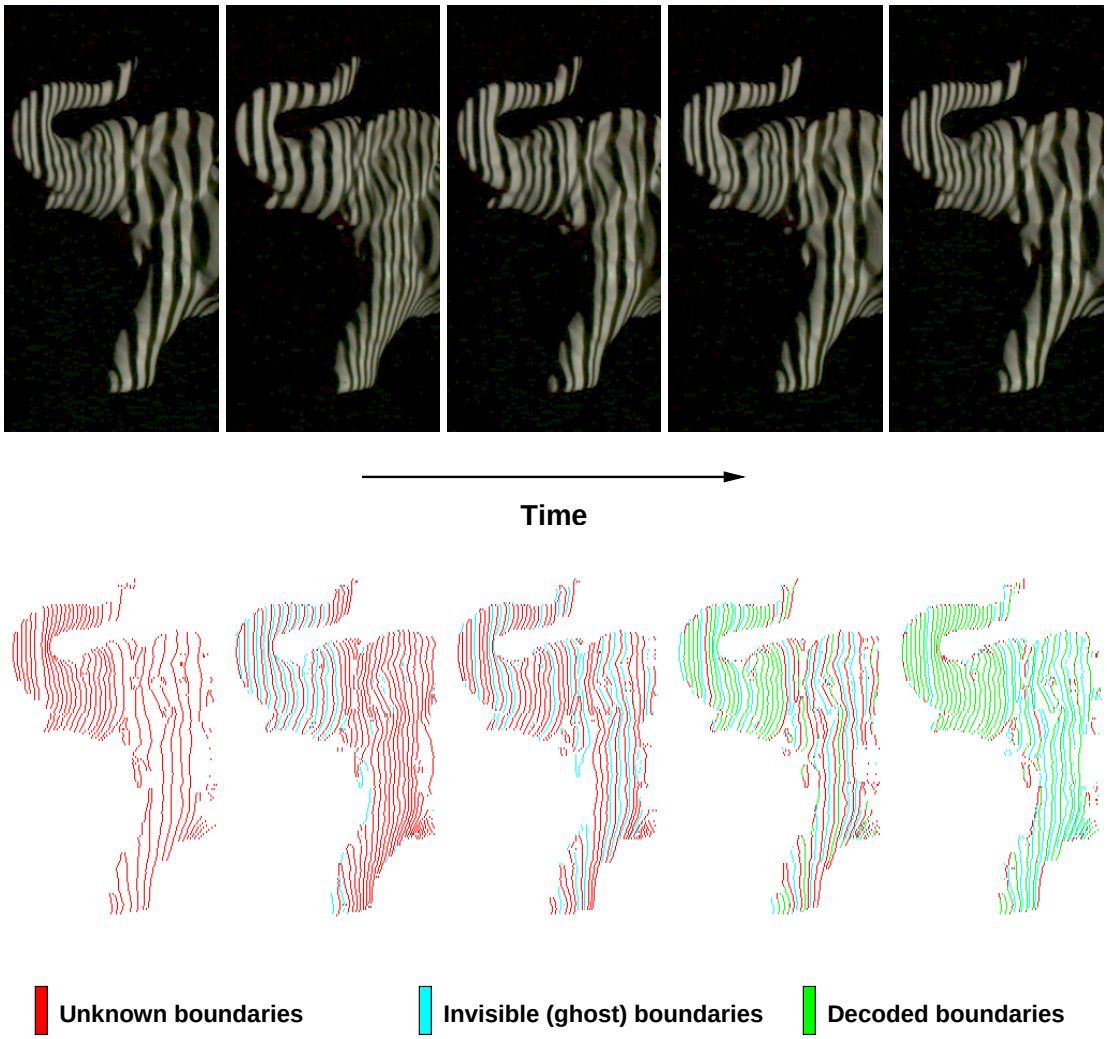


Figure 2.7: At top, video frames of an elephant figurine illuminated by the stripe code in Figure 2.4. At bottom, the recovered stripe boundaries. Stripe boundaries drawn in red indicate boundaries whose identity is not yet known, while those drawn in green have been successfully tracked for four frames and can be identified. Drawn in blue are the inferred positions of ghosts (i.e., "boundaries" between two stripes of the same color). These are not seen directly, but their presence can be inferred by assuming that stripes move with some maximum velocity.

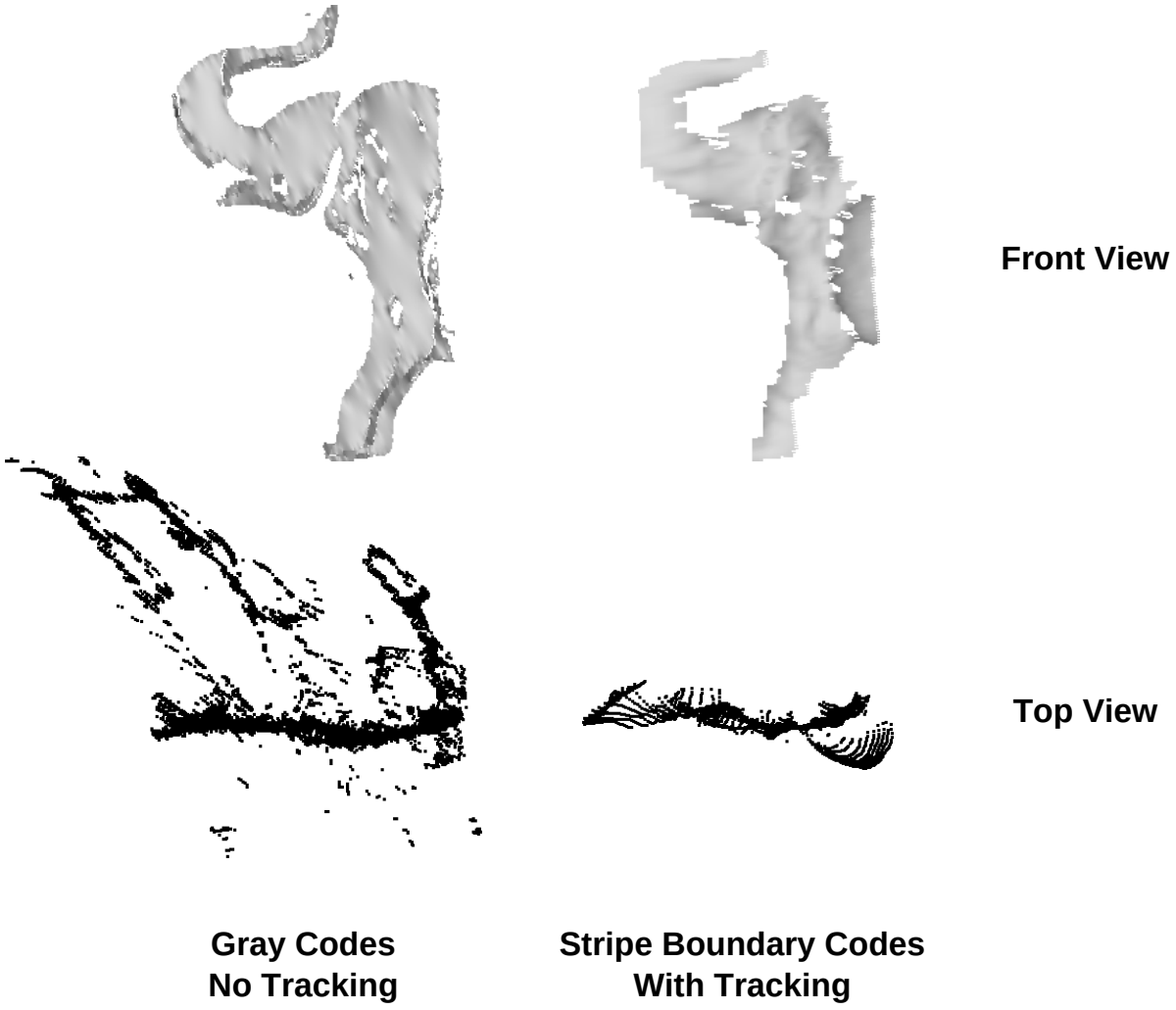


Figure 2.8: Comparison of range acquired using Gray codes and stripe boundary codes on a moving scene consisting of an elephant figurine moving over a black background (some raw video frames are shown in Figure 2.4). At left, a scene moving at approximately one pixel per frame, scanned using Gray codes. Note the presence of incorrect geometry, especially near discontinuities. At right, the same scene scanned using our codes. Note that in the top view we see multiple adjacent contours from different heights along the model. The contours lie close to each other, as we expect, and do not contain the outliers seen on the left.

use this reflectance estimate during segmentation of the remaining frames. Such an approach could also be implemented in the moving scene case, though it would require frame-to-frame tracking of the reflectance at each point. In addition, the presence of extra all-white frames would reduce the rate at which depths are returned. An alternative solution might be based on simultaneous acquisition at multiple wavelengths – this would not reduce the rate of capture.

Silhouettes: Another aspect of our current system that could be made more robust is the handling of silhouette edges and disocclusion of geometry from behind silhouettes. Currently, we must wait at least four frames before we have enough data to identify a new stripe, since the code is four frames long. Shortening this delay would be possible by attempting to identify a new stripe as soon as it appears by *looking ahead* three frames. Such a look-ahead scheme could also be used improve robustness, by providing additional hints of whether a given stripe was misidentified (i.e., it is likely that there was an error in tracking if the observed code at a stripe is not the same as the codes four frames ago and four frames into the future). Introducing such an algorithm based on look-ahead, however, would not only introduce additional latency into the system, which might be undesirable in certain applications, but also reduce the ability to identify short-lived features (specifically, those that appear then disappear again between four and eight frames later).

Object Motion: Because our prototype implementation was designed to run in real time on present hardware, we were limited to simple algorithms for stripe boundary matching and decoding. In practice, this requires objects in the scene to move relatively slowly (between one-fourth and one-half stripe-width per frame) in order for boundaries to be matched correctly. For our prototype, this corresponds to a constraint that objects move a maximum of approximately 10% of our working volume per second. We anticipate that future systems could incorporate more sophisticated tracking algorithms to allow for greater object speeds. In particular, we expect that algorithms such as Kalman filtering could be used to take advantage of the fact that both the scanner and the object have some mass, and therefore there will not be discontinuities in velocity when one or the other is moved. Thus, it should be possible to predict the positions of stripe boundaries from frame to frame, leading to more robust matching. Prediction is further discussed in Section 7.3.1.

Intrusiveness: For many potential applications of a range scanning system (in particular, any applications involving people), it is distracting to have visible flashing lights. Such applications

would require making the illumination patterns imperceptible, either by projecting them in the infrared or by using a time-multiplexed light cancellation technique [Raskar 98].

2.5 Summary

We have presented a new design for a structured-light range scanner capable of returning range images of moving objects at video rates. The system uses a time-coded projected pattern that assigns a unique illumination code to each pair of adjacent stripes. Features (stripe boundaries) in the projected illumination are tracked over time, permitting projector-camera correspondences to be found even if the scene and camera move relative to each other.

Although this range scanner may be used on any type of scene, including those containing nonrigid or deforming objects, this dissertation focuses on its application in the case of *rigid* objects. The next two chapters therefore explain how this scanner may be used as the first stage of a model acquisition pipeline that performs alignment, merging, and rendering of the range images produced by this scanner.

"I shall try to correct errors when shown to be errors, and I shall adopt new views so fast as they shall appear to be true views."

– Abraham Lincoln

Chapter 3

Fast Alignment of 3D Meshes

The previous chapter described the design of a range scanner capable of returning the 3D shape of a moving object as seen from a single viewpoint in real time. As mentioned in the introduction, however, the goal of our system is to produce complete models of rigid objects. Therefore, we must align the range images from different viewpoints that are produced as a rigid object is moved relative to the range scanner. The algorithm we use to do this is ICP (Iterative Closest Points). This algorithm is widely used for geometric alignment of three-dimensional models when an initial estimate of the relative pose is known. In the real-time range scanner application, the relative motion between two consecutive range images is small, so we may simply align each range image to the previous one.

This goal of this chapter is to explore the space of ICP design variants and search for one that is fast enough to use in a real-time application. Many variants of ICP have been proposed, affecting all phases of the algorithm from the selection and matching of points to the minimization strategy. We enumerate and classify many of these variants, and evaluate their effect on the speed with which the correct alignment is reached. We conclude by proposing a combination of ICP variants optimized for high speed. We demonstrate an implementation that is able to align two range images in a few tens of milliseconds, assuming a good initial guess. As described above, this is the algorithm that is used by the real-time model acquisition pipeline.

3.1 Taxonomy of ICP Variants

The ICP (originally Iterative Closest Point, though Iterative Corresponding Point is perhaps a better expansion for the abbreviation) algorithm has become the dominant method for aligning three-dimensional models based purely on the geometry, and sometimes color, of the meshes. The algorithm is widely used for registering the outputs of 3D scanners, which typically only scan an object from one direction at a time. ICP starts with two meshes and an initial guess for their relative rigid-body transform, and iteratively refines the transform by repeatedly generating pairs of corresponding points on the meshes and minimizing an error metric. Generating the initial alignment may be done by a variety of methods, such as tracking scanner position, identification and indexing of surface features [Faugeras 86, Stein 92], "spin-image" surface signatures [Johnson 97a, Huber 01], computing principal axes of scans [Dorai 97], exhaustive search for corresponding points [Chen 99], or user input. In this chapter, we assume that a rough initial alignment is always available. In addition, we focus only on aligning a single pair of meshes, and do not address the *global registration* problem [Bergevin 96, Stoddart 96, Pulli 97, Pulli 99].

Since the introduction of ICP by Chen and Medioni [Chen 91] and Besl and McKay [Besl 92], many variants have been introduced on the basic ICP concept. We may classify these variants as affecting one of six stages of the algorithm:

1. **Selection** of some set of points in one or both meshes.
2. **Matching** these points to samples in the other mesh.
3. **Weighting** the corresponding pairs appropriately.
4. **Rejecting** certain pairs based on looking at each pair individually or considering the entire set of pairs.
5. Assigning an **error metric** based on the point pairs.
6. **Minimizing** the error metric.

In this chapter, we will look at variants in each of these six categories, and examine their effects on the performance of ICP. Although our main focus is on the speed of convergence, we also consider the accuracy of the final answer and the ability of ICP to reach the correct solution given "difficult" geometry. Our comparisons suggest a combination of ICP variants

that is able to align a pair of meshes in a few tens of milliseconds, significantly faster than most commonly-used ICP systems.

3.2 Previous Work

Our comparisons will consider variants from a number of different ICP implementations, summarized in Table 3.1. For completeness, the table also includes information about global registration algorithms, though we do not consider them here. The terms in this table will be defined later in this chapter. The papers are:

- [Chen 91] and [Besl 92], which first introduced the ICP algorithm. Besl and McKay used a point-to-point error metric, while Chen and Medioni used point-to-plane error. [Besl 92] also extrapolated transforms in order to accelerate convergence.
- [Turk 94] introduced a number of improvements on ICP, including rejection of edge points, repeated ICP on low-to-high resolutions of meshes, and weight proportional to the dot product between the surface normal and the vector from the surface point to the camera. In addition, the paper aligns individual scans to a single cylindrical scan of the object in order to prevent accumulation of scan-to-scan alignment errors.
- [Godin 94] introduced the ICCP (iterative closest *compatible* point) algorithm, which only matches samples if they are compatible by some metric. In addition, this implementation weights the point pairs by compatibility and the distance between the points.
- [Blais 95] performs projection to find corresponding points, instead of using closest points. This is one of the major variants incorporated in our high-speed ICP algorithm. Unlike our implementation, [Blais 95] combined this matching strategy with a search in the space of transforms based on simulated annealing.
- [Stoddart 96] introduced a gradient-descent-based global registration framework, in which the transforms of all scans are found simultaneously.
- [Masuda 96] introduced a robust ICP algorithm, which computes a number of ICP results using random subsets of points, then finds the one that minimizes the least-median-of-squares residual.
- [Bergevin 96] describes a global registration algorithm that repeatedly tries to minimize the ICP errors for all pairs of overlapping scans.

Table 3.1: Comparison of ICP variants.

Algorithm	Selection of Points	Matching Points	Weighting and Rejecting Pairs	Error Metric	Global Registration	Notes
[Chen 91]	Uniform subsampling, smooth regions	Normal shooting		Point-to-plane	Align new to all previous	Minimization equations can be approximated as linear if transform is close to correct
[Besl 92]	All	Closest point		Point-to-point		Accelerated by extrapolating transforms
[Turk 94]	Uniform subsampling	Closest point	Distance threshold; reject edge points; weight is normal dot camera vector	Point-to-point	Align all to cylindrical anchor scan	Perform ICP on low-to-high resolution versions of meshes
[Godin 94]	All in both meshes	Closest point with compatible color	Distance threshold; weighted by compatibility and distance	Point-to-point		
[Blais 95]	Uniform subsampling	Projection	Distance threshold	Point-to-point	Search for all transforms simultaneously	Search in transform space using simulated annealing
[Stoddart 96]	Assumed given	Assumed given		Point-to-point	Find all transforms simultaneously	Gradient descent
[Masuda 96]	Random sampling	Closest point accelerated with *k-d* tree	Distance threshold	Point-to-point	New to integration of all previous	Find transform that minimizes median of squared distances after several random subsamplings
[Bergevin 96]	Uniform subsampling, smooth regions	Normal shooting	Reject if the dot product of normals is negative	Point-to-plane	Iterated all-to-all ICP	
[Simon 96]	All	Closest point accelerated with *k-d* tree and point cache	Distance threshold	Point-to-point		Perturb starting positions to avoid local minima

Table 3.1 (cont.): Comparison of ICP variants.

Algorithm	Selection of Points	Matching Points	Weighting and Rejecting Pairs	Error Metric	Global Registration	Notes
[Dorai 97]	All	Normal shooting	Weighted based on effect of scanner noise on normal	Point-to-plane		
[Dorai 98]	All	Normal shooting accelerated by projection and search	Rejection based on pair-to-pair compatibility	Point-to-plane		
[Benjemaa 97]	All	Closest point accelerated using z-buffer search		Point-to-point	Iterated all-to-all ICP	
[Johnson 97b]	All	Closest point in shape+color space accelerated with k-d tree		Point-to-point, shape + color		
[Neugebauer 97]	Uniform subsampling	Projection	Reject points with distance greater than 3σ	Point-to-plane	Align all scans simultaneously	Terminate by statistically testing the hypothesis $\delta = 0$
[Weik 97]	Points with high intensity gradient	Projection followed by search for sample with similar image intensity and gradient	Reject projected points that are occluded in the source mesh	Point-to-point		
[Pulli 97]	All	Like [Weik 97], but project complete images and do image alignment		Point-to-point	Scan-to-scan ICP, then global optimization of transforms using pre-computed point pairs	
[Pulli 99], [Levoy 00]	Random sampling in both meshes	Closest point with compatible normals	Distance threshold; reject edge points; reject a percentage of pairs with largest distances	Point-to-plane	Like [Pulli 97], but process scans in order of how many others they overlap	

- [Simon 96] implemented a high-speed ICP algorithm (including accelerated closest-point computations and a closest-point cache), and analyzed the stability of ICP convergence.
- [Dorai 97] introduced a way of weighting the contribution of each point pair in ICP based on its expected uncertainty and its contribution to the ICP error.
- [Dorai 98] shows how to examine the set of corresponding point pairs to reject those that are mutually inconsistent, hence likely to be outliers.
- [Benjemaa 97] accelerates closest-point computation using a z-buffer-based search.
- [Johnson 97b] uses a closest-point metric that considers not only distance but also color. This permits ICP to be used on surfaces that are geometrically smooth but have some texture.
- [Neugebauer 97] presents an ICP algorithm very similar to the one we suggest for high-speed use, including projection-based matching and the point-to-plane error metric. The paper also examines the question of when ICP should be terminated, by statistically testing the hypothesis that the incremental transform is zero.
- [Weik 97] uses a closest-point search that considers texture intensity and gradient. [Pulli 97] is similar, but projects complete images to a common coordinate system, then performs image alignment.
- [Pulli 99] describes a practical combination of ICP variants for real-world use, including matching of points with compatible normals, percentage-based outlier rejection, and a point-to-plane error metric. In addition, it describes a global registration method based on simultaneously considering an arbitrary (overconstrained) set of ICP results among a set of scans. This is the algorithm used for the Digital Michelangelo Project [Levoy 00].

3.3 Comparison Methodology

Our goal is to compare the convergence characteristics of several ICP variants. In order to limit the scope of the problem, and avoid a combinatorial explosion in the number of possibilities, we adopt the methodology of choosing a *baseline* combination of variants, and examining performance as individual ICP stages are varied. The algorithm we will select as our baseline is

essentially that of [Pulli 99], incorporating the following features (these are described in detail in the following section):

- Random sampling of points on both meshes (new random points are selected at each iteration).
- Matching each selected point to the closest sample in the other mesh that has a normal within 45 degrees of the source normal.
- Uniform (constant) weighting of point pairs.
- Rejection of pairs containing edge vertices, as well as a percentage of pairs with the largest point-to-point distances.
- Point-to-plane error metric.
- The classic "select-match-minimize" iteration, rather than some other search for the alignment transform.

We pick this algorithm because it has received extensive use in a production environment [Levoy 00], and has been found to be robust for scanned data containing many kinds of surface features.

In addition, to ensure fair comparisons among variants, we make the following assumptions:

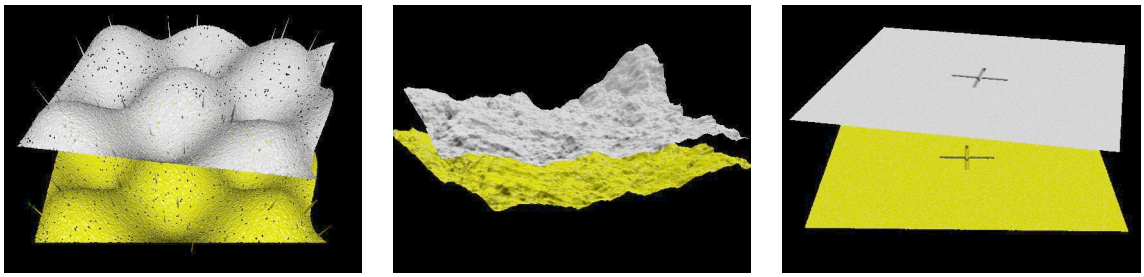
- The number of source points selected is always 2,000. Since the meshes we will consider have 100,000 samples, this corresponds to a sampling rate of 1% per mesh if source points are selected from both meshes, or 2% if points are selected from only one mesh.
- All meshes we use are simple perspective range images, as opposed to general irregular meshes, since this enables comparisons between "closest point" and "projected point" variants (see Section 3.4.2).
- Surface normals are computed simply based on the four nearest neighbors in the range grid.
- Only geometry is used for alignment, not color or intensity.

With the exception of the last one, we expect that changing any of these implementation choices would affect the quantitative, but not the qualitative, performance of our tests. Although we will not compare variants that use color or intensity, it is clearly advantageous to use such data when available, since it can provide necessary constraints in areas where there are few geometric features.

Although we will be focusing on separately varying each stage of the ICP pipeline, it is clear that the stages are not "orthogonal" to each other, in that changing one stage might change the relative performance of variants at another stage. Although it is not practical to explore all possible interactions of this sort, the following section notes a number of instances in which we have noticed this interdependence among stages.

3.3.1 Test Scenes

We use three synthetically-generated scenes to evaluate variants. The "wave" scene (Figure 3.1a) is an easy case for most ICP variants, since it contains relatively smooth coarse-scale geometry. The two meshes have independently-added Gaussian noise, outliers, and dropouts. The "fractal landscape" test scene (Figure 3.1b) has features at all levels of detail. The "incised plane" scene (Figure 3.1c) consists of two planes with Gaussian noise and grooves in the shape of an "X." This is a difficult scene for ICP, and most variants do not converge to the correct alignment, even given the small relative rotation in the illustrated starting position. Note that the three test scenes consist of low-frequency, all-frequency, and high-frequency features, respectively. Though these scenes certainly do not cover all possible classes of scanned objects, they are representative of surfaces encountered in many classes of scanning applications. For example, the Digital Michelangelo Project [Levoy 00] involved scanning surfaces containing



(a) Wave (b) Fractal landscape (c) Incised plane

Figure 3.1: Test scenes used throughout this chapter.

low-frequency features (e.g., smooth statues), fractal-like features (e.g., unfinished statues with visible chisel marks), and incisions (e.g., fragments of the Forma Urbis Romæ).

The motivation for using synthetic data for our comparisons is so that we know the correct transform exactly, and can evaluate the performance of ICP algorithms relative to this correct alignment. The metric we will use throughout this chapter is root-mean-square point-to-point distance for the *actual* corresponding points in the two meshes. Using such a "ground truth" error metric allows for more objective comparisons of the performance of ICP variants than using the error metrics computed by the algorithms themselves.

We only present the results of one run for each tested variant. Although a single run clearly can not be taken as representing the performance of an algorithm in all situations, we have tried to show typical results that capture the significant differences in performance on various kinds of scenes. Any cases in which the presented results are not typical are noted in the text.

All reported running times are for a C++ implementation running on a 550 MHz Pentium III Xeon processor.

3.4 Comparisons of ICP Variants

We now examine ICP variants for each of the stages listed in Section 1. For each stage, we compare the performance of the variants in the literature (as introduced in Section 3.2) on our test scenes.

3.4.1 Selection of Points

We begin by examining the effect of the selection of point pairs on the convergence of ICP. The following strategies have been proposed:

- Always using all available points [Besl 92].
- Uniform subsampling of the available points [Turk 94].
- Random sampling (with a different sample of points at each iteration) [Masuda 96].
- Selection of points with high intensity gradient, in variants that use per-sample color or intensity to aid in alignment [Weik 97].
- Each of the preceding schemes may select points on only one mesh, or select source points from both meshes [Godin 94].

In addition to these, we introduce a new sampling strategy: choosing points such that the distribution of normals among selected points is as large as possible. The motivation for this strategy is the observation that for certain kinds of scenes (such as our "incised plane" data set) small features of the model are vital to determining the correct alignment. A strategy such as random sampling will often select only a few samples in these features, which leads to an inability to determine certain components of the correct rigid-body transformation. Thus, one way to improve the chances that enough constraints are present to determine all the components of the transformation is to bucket the points according to the position of the normals in angular space, then sample as uniformly as possible across the buckets. Normal-space sampling is therefore a very simple example of using surface features for alignment; it has lower computational cost, but lower robustness, than traditional feature-based methods [Faugeras 86, Stein 92, Johnson 97a].

Let us compare the performance of uniform subsampling, random sampling, and normal-space sampling on the "wave" scene (Figure 3.2). As we can see, the convergence performance is similar. This indicates that for a scene with a good distribution of normals the exact sampling strategy is not critical. The results for the "incised plane" scene look different, however (Figure 3.3). Only the normal-space sampling is able to converge for this data set.

The reason is that samples not in the grooves are only helpful in determining three of the six components of the rigid-body transformation (one translation and two rotations). The other three components (two translations and one rotation, within the plane) are determined entirely by samples within the incisions. The random and uniform sampling strategies only place a few samples in the grooves, especially at low sample rates (Figure 3.4a). This, together with the fact that noise and distortion on the rest of the plane overwhelms the effect of those pairs that *are* sampled from the grooves, accounts for the inability of uniform and random sampling to converge to the correct alignment. Conversely, normal-space sampling selects a larger number of samples in the grooves (Figure 3.4b).

Sampling Direction: We now look at the relative advantages of choosing source points from both meshes, versus choosing points from only one mesh. For the "wave" test scene and the baseline algorithm, the difference is minimal (Figure 3.5). However, this is partly due to the fact that we used the closest compatible point matching algorithm (see Section 3.4.2), which is symmetric with respect to the two meshes. If we use a more "asymmetric" matching algorithm, such as projection or normal shooting (see Section 3.4.2), we see that sampling from both

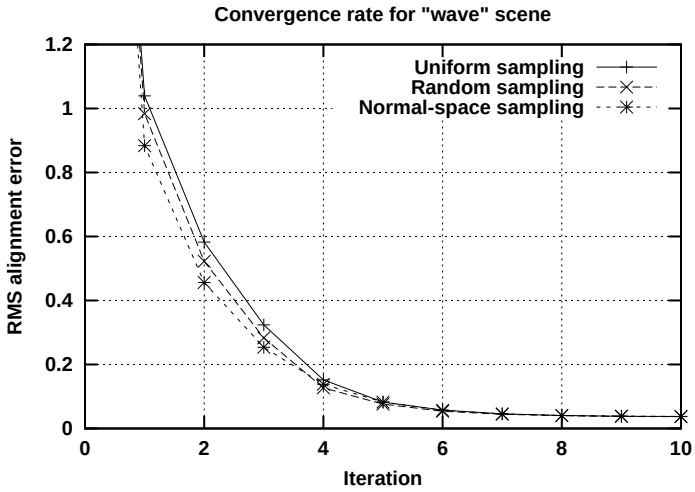


Figure 3.2: Comparison of convergence rates for uniform, random, and normal-space sampling for the "wave" meshes.

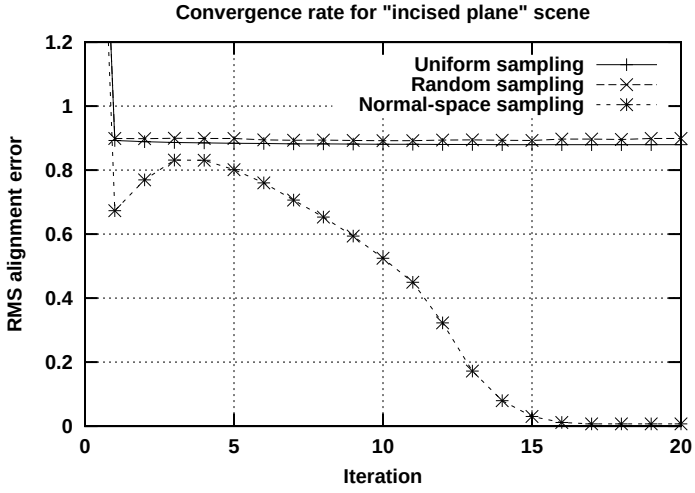


Figure 3.3: Comparison of convergence rates for uniform, random, and normal-space sampling for the "incised plane" meshes. Note that, on the lower curve, the ground truth error increases briefly in the early iterations. This illustrates the difference between the ground truth error and the algorithm's estimate of its own error.

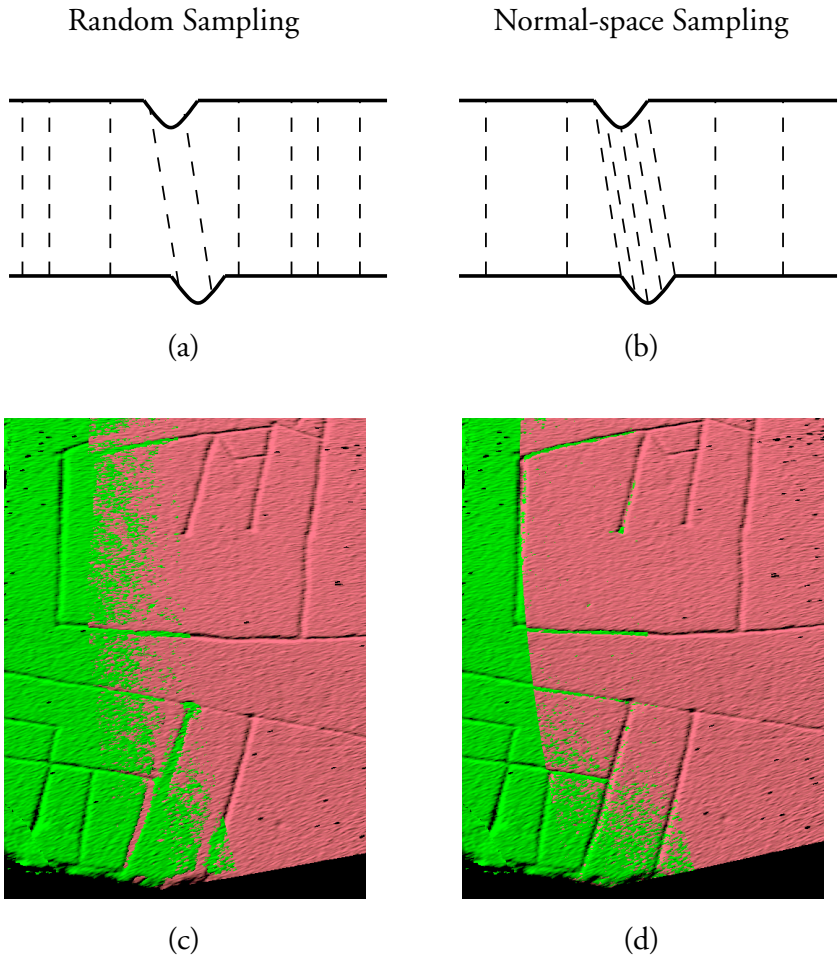


Figure 3.4: Corresponding point pairs selected by the (a) "random sampling" and (b) "normal-space sampling" strategies for an incised mesh. Using random sampling, the sparse features may be overwhelmed by presence of noise or distortion, causing the ICP algorithm to not converge to a correct alignment. Here, two range images are shown in different colors; note the misalignment of the incisions (c). The normal-space sampling strategy ensures that enough samples are placed in the feature to bring the surfaces into near-perfect alignment, despite the presence of very pronounced distortion (d). "Closest compatible point" matching (see Section 3.4.2) was used for this example. The meshes in (c) and (d) are scans of fragment 165d of the Forma Urbis Romæ.

meshes appears to give slightly better results (Figure 3.6), especially during the early stages of the iteration when the two meshes are still far apart. In addition, we expect that sampling from both meshes would also improve results when the overlap of the meshes is small, or when the meshes contain many holes.

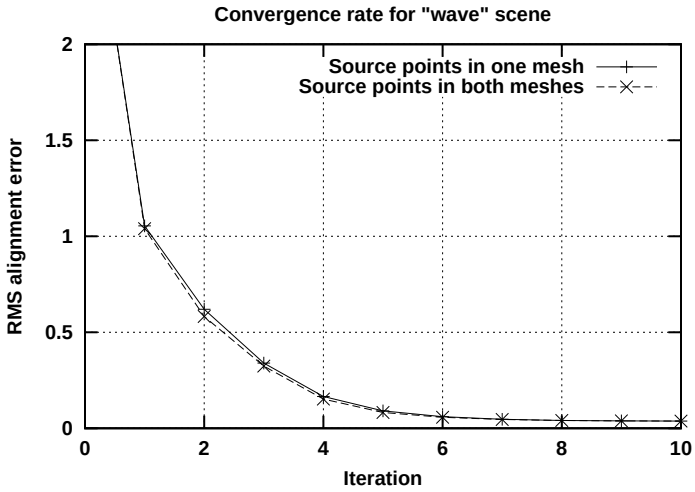


Figure 3.5: Comparison of convergence rates for single-source-mesh and both-source-mesh sampling strategies for the "wave" meshes.

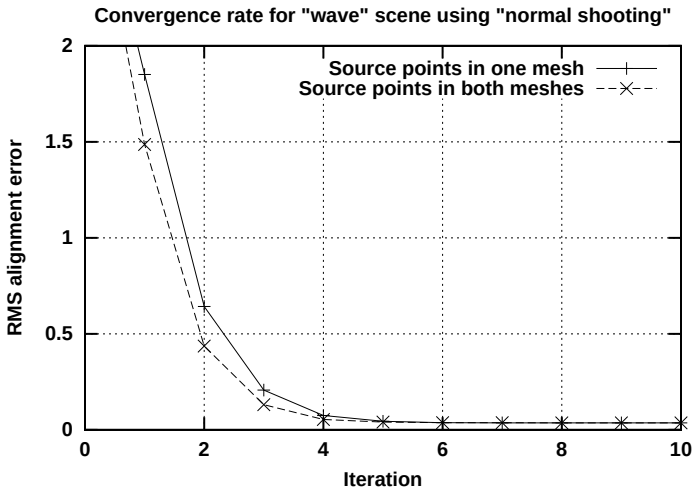


Figure 3.6: Comparison of convergence rates for single-source-mesh and both-source-mesh sampling strategies for the "wave" meshes, using normal shooting as the matching algorithm.

3.4.2 Matching Points

The next stage of ICP that we will examine is correspondence finding. Algorithms have been proposed that, for each sample point selected:

- Find the closest point in the other mesh [Besl 92]. This computation may be accelerated using a *k-d* tree and/or closest-point caching [Simon 96].
- Find the intersection of the ray originating at the source point in the direction of the source point's normal with the destination surface [Chen 91]. We will refer to this as "normal shooting."
- Project the source point onto the destination mesh, from the point of view of the destination mesh's range camera [Blais 95, Neugebauer 97]. This has also been called "reverse calibration."
- Project the source point onto the destination mesh, then perform a search in the destination range image. The search might use a metric based on point-to-point distance [Benjemaa 97], point-to-ray distance [Dorai 98], or compatibility of intensity [Weik 97] or color [Pulli 97].
- Any of the above methods, restricted to only matching points compatible with the source point according to a given metric. Compatibility metrics based on color [Godin 94] and angle between normals [Pulli 99] have been explored.

Since we are not analyzing variants that use color, the particular variants we will compare are: closest point, closest compatible point (normals within 45 degrees), normal shooting, normal shooting to a compatible point (normals within 45 degrees), projection, and projection followed by search. The first four of these algorithms are accelerated using a *k-d* tree. For the last algorithm, the search is actually implemented as a steepest-descent neighbor-to-neighbor walk in the destination mesh that attempts to find the closest point. We chose this variation because it works nearly as well as projection followed by exhaustive search in some window, but has lower running time.

We first look at performance for the "fractal" scene (Figure 3.7). For this scene, normal shooting appears to produce the best results, followed by the projection algorithms. The closest-point algorithms, in contrast, perform relatively poorly. We hypothesize that the reason

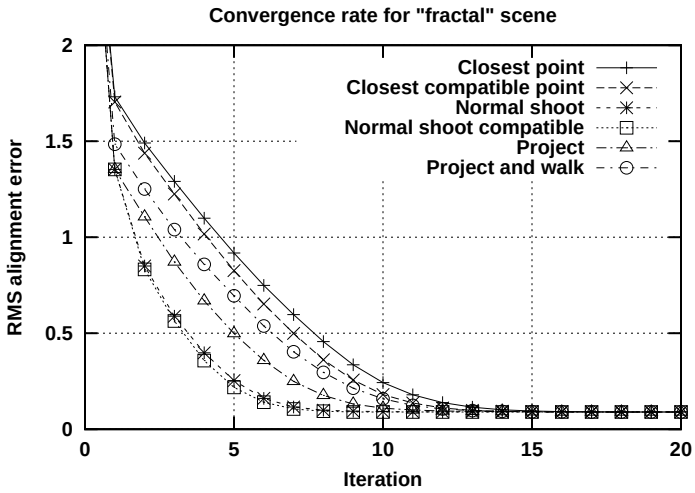


Figure 3.7: Comparison of convergence rates for the "fractal" meshes, for a variety of matching algorithms.

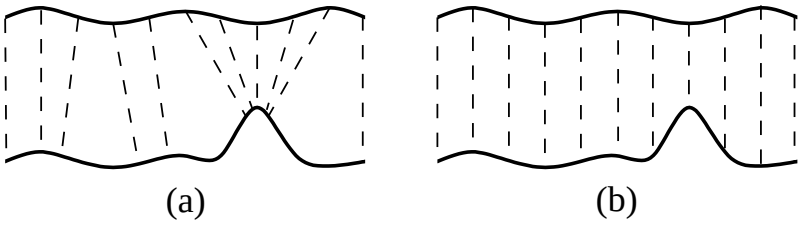


Figure 3.8: (a) In the presence of noise and outliers, the closest-point matching algorithm potentially generates large numbers of incorrect pairings when the meshes are still relatively far from each other, slowing the rate of convergence. (b) The "projection" matching strategy is less sensitive to the presence of noise.

for this is that the closest-point algorithms are more sensitive to noise and tend to generate larger numbers of incorrect pairings than the other algorithms (Figure 3.8).

The situation in the "incised plane" scene, however, is different (Figure 3.9). Here, the closest-point algorithms were the only ones that converged to the correct solution. Thus, we conclude that although the closest-point algorithms might not have the fastest convergence rate for "easy" scenes, they are the most robust for "difficult" geometry.

Though so far we have been looking at error as a function of the number of iterations, it is also instructive to look at error as a function of running time. Because the matching stage of ICP is usually the one that takes the longest, applications that require ICP to run quickly (and that do not need to deal with the geometrically "difficult" cases) must choose the matching algorithm with the fastest performance. Let us therefore compare error as a function of time for these algorithms for the "fractal" scene (Figure 3.10). We see that although the projection algorithm does not offer the best convergence per iteration, each iteration is faster than an iteration of closest point finding or normal shooting because it is performed in constant time, rather than involving a closest-point search (which, even when accelerated by a *k-d* tree, takes $O(\log n)$ time). As a result, the projection-based algorithm has a significantly faster rate of convergence vs. time. Note that this graph does not include the time to compute the *k-d* trees used by all but the projection algorithms. Including the precomputation time (approximately 0.64 seconds for these meshes) would produce even more favorable results for the projection algorithm.

3.4.3 Weighting of Pairs

We now examine the effect of assigning different weights to the corresponding point pairs found by the previous two steps. We consider four different algorithms for assigning these weights:

- Constant weight
- Assigning lower weights to pairs with greater point-to-point distances. This is similar in intent to dropping pairs with point-to-point distance greater than a threshold (see Section 3.4.4), but avoids the discontinuity of the latter approach. Following [Godin 94], we use

$$Weight = 1 - \frac{Dist(p_1, p_2)}{Dist_{max}}$$

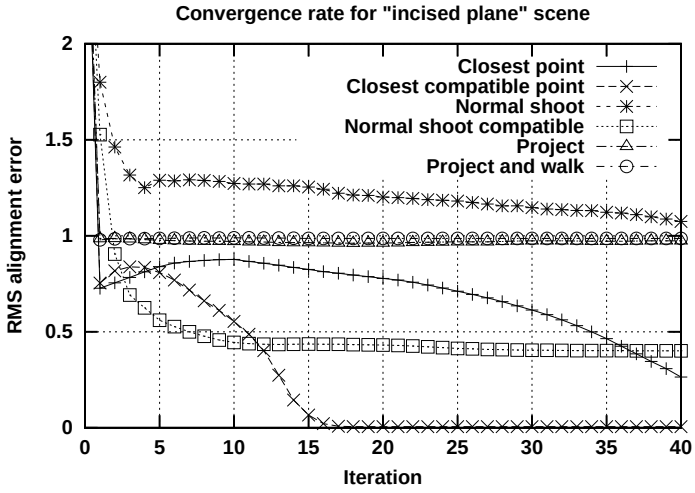


Figure 3.9: Comparison of convergence rates for the "incised plane" meshes, for a variety of matching algorithms. Normal-space-directed sampling was used for these measurements.

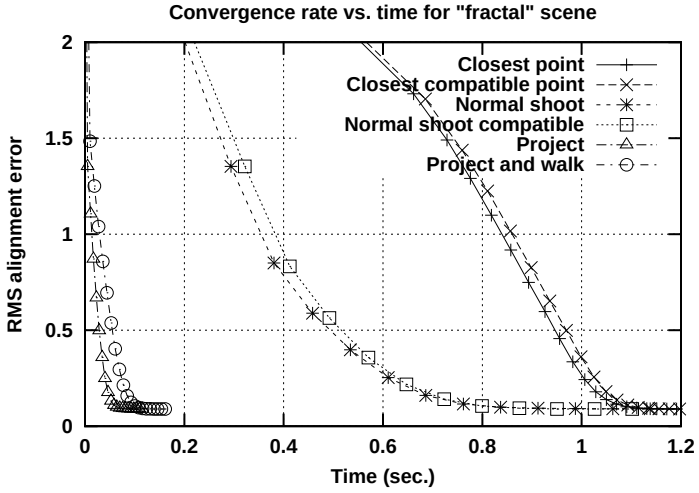


Figure 3.10: Comparison of convergence rate vs. time for the "fractal" meshes, for a variety of matching algorithms. (cf. Figure 3.7) Note that these times do not include precomputation (in particular, computing the *k-d* trees used by the first four algorithms takes 0.64 seconds).

- Weighting based on compatibility of normals:

$$Weight = n_1 \cdot n_2$$

  Weighting on compatibility of colors has also been used [Godin 94], though we do not consider it here.

- Weighting based on the expected effect of scanner noise on the uncertainty in the error metric. For the point-to-plane error metric (see Section 3.4.5), this depends on both uncertainty in the position of range points and uncertainty in surface normals. As shown in the Appendix, the result for a typical laser range scanner is that the uncertainty is lower, hence higher weight should be assigned, for surfaces tilted away from the range camera.

We first look at a version of the "wave" scene (Figure 3.11). Extra noise has been added in order to amplify the differences among the variants. We see that even with the addition of extra noise, all of the weighting strategies have similar performance, with the "uncertainty" and "compatibility of normals" options having marginally better performance than the others. For the "incised plane" scene (Figure 3.12), the results are similar, though there is a larger difference in performance. However, we must be cautious when interpreting this result, since the uncertainty-based weighting assigns higher weights to points on the model that have normals pointing away from the range scanner. For this scene, therefore, the uncertainty weighting assigns higher weight to points within the incisions, which improves the convergence rate. We conclude that, in general, the effect of weighting on convergence rate will be small and highly data-dependent, and that the choice of a weighting function should be based on other factors, such as the accuracy of the final result.

3.4.4 Rejecting Pairs

Closely related to assigning weights to corresponding pairs is rejecting certain pairs entirely. The purpose of this is usually to eliminate outliers, which may have a large effect when performing least-squares minimization. The following rejection strategies have been proposed:

- Rejection of corresponding points more than a given (user-specified) distance apart.

- Rejection of the worst n% of pairs based on some metric, usually point-to-point distance. As suggested by [Pulli 99], we reject 10% of pairs.

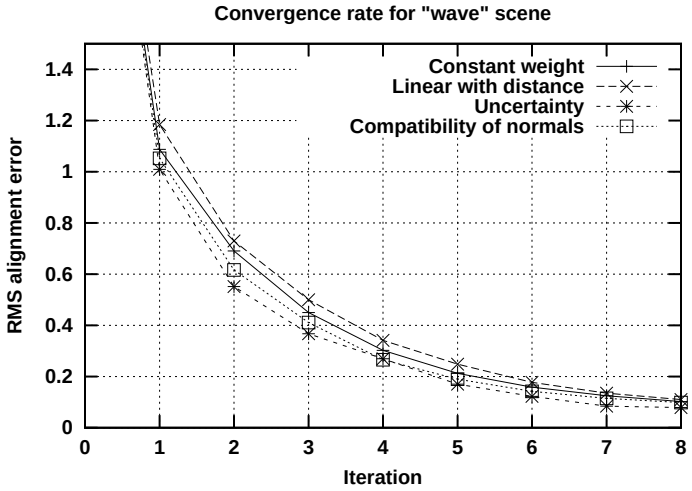


Figure 3.11: Comparison of convergence rates for the "wave" meshes, for several choices of weighting functions. In order to increase the differences among the variants we have doubled the amount of noise and outliers in the mesh.

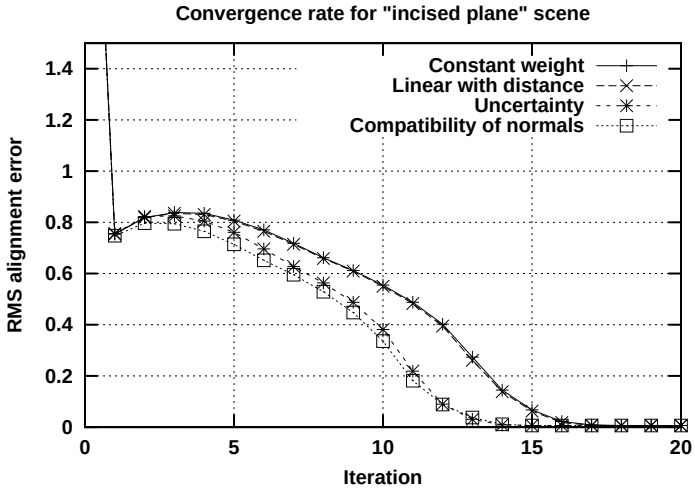


Figure 3.12: Comparison of convergence rates for the "incised plane" meshes, for several choices of weighting functions. Normal-space-directed sampling was used for these measurements.

- Rejection of pairs whose point-to-point distance is larger than some multiple of the standard deviation of distances. Following [Masuda 96], we reject pairs with distances more than 2.5 times the standard deviation.

- Rejection of pairs that are not consistent with neighboring pairs, assuming surfaces move rigidly [Dorai 98]. This scheme classifies two correspondences (p_1, q_1) and (p_2, q_2) as inconsistent iff

$$|Dist(p_1, p_2) - Dist(q_1, q_2)|$$

  is greater than some threshold. Following [Dorai 98], we use

$$0.1 \cdot \max(Dist(p_1, p_2), Dist(q_1, q_2))$$

  as the threshold. The algorithm then rejects those correspondences that are inconsistent with most others. Note that the algorithm as originally presented has running time $O(n^2)$ at each iteration of ICP. In order to reduce this, we have chosen to only compare each correspondence to 10 others, and reject it if it is incompatible with more than 5.

- Rejection of pairs containing points on mesh boundaries [Turk 94].

The latter strategy, of excluding pairs that include points on mesh boundaries, is especially useful for avoiding erroneous pairings (that cause a systematic bias in the estimated transform) in cases when the overlap between scans is not complete (Figure 3.13). Since its cost is usually low and in most applications its use has few drawbacks, we always recommend using this strategy, and in fact we use it in all the comparisons in this chapter.

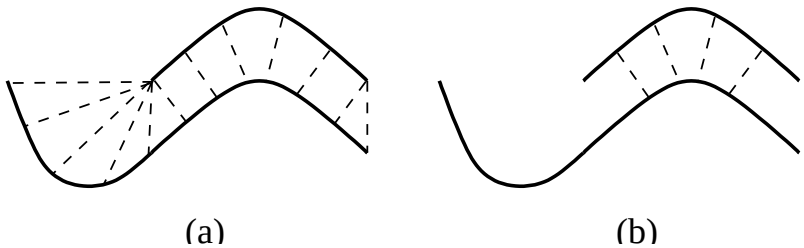


Figure 3.13: Effect of disallowing edge matches. (a) When two meshes to be aligned do not overlap completely (as is the case for most real-world data), allowing correspondences involving points on mesh boundaries can introduce a systematic bias into the alignment. In this case, the points on the left half of the lower mesh all match to the edge of the upper mesh, thus pulling the upper mesh to the left of its correct position. (b) Disallowing edge pairs eliminates these incorrect correspondences.

Figure 3.14 compares the performance of no rejection, worst-10% rejection, pair compatibility rejection, and 2.5 σ rejection on the "wave" scene with extra noise and outliers. We see that rejection of outliers does not help with convergence. In fact, the algorithm that rejected pairs most aggressively (worst-10% rejection) tended to converge more slowly when the meshes were relatively far from aligned. Thus, we conclude that outlier rejection, though it may have effects on the accuracy and stability with which the correct alignment is determined, in general does not improve the speed of convergence.

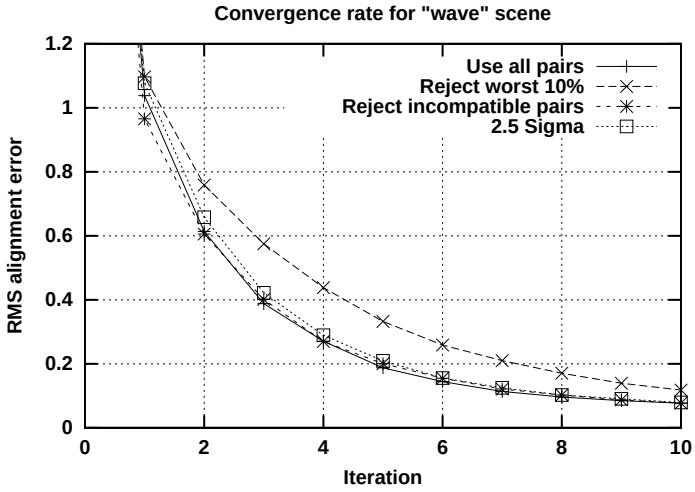


Figure 3.14: Comparison of convergence rates for the "wave" meshes, for several pair rejection strategies. As in Figure 3.11, we have added extra noise and outliers to increase the differences among the variants.

3.4.5 Error Metric and Minimization

The final pieces of the ICP algorithm that we will look at are the error metric and the algorithm for minimizing the error metric. The following error metrics have been used:

- Sum of squared distances between corresponding points. For an error metric of this form, there exist closed-form solutions for determining the rigid-body transformation that minimizes the error. Solution methods based on singular value decomposition [Arun 87], quaternions [Horn 87], orthonormal matrices [Horn 88], and dual quaternions [Walker 91] have been proposed; Eggert et al. have evaluated the numerical accuracy and stability of each of these [Eggert 97], concluding that the differences among them are small.

- The above "point-to-point" metric, taking into account both the distance between points and the difference in colors [Johnson 97b].

- Sum of squared distances from each source point to the plane containing the destination point and oriented perpendicular to the destination normal [Chen 91]. In this "point-to-plane" case, no closed-form solutions are available. The least-squares equations may be solved using a generic nonlinear method (e.g. Levenberg-Marquardt), or by simply linearizing the problem (i.e., assuming incremental rotations are small, so $\sin\theta \sim \theta$ and $\cos\theta \sim 1$).

There are several ways to formulate the search for the alignment:

- Repeatedly generating a set of corresponding points using the current transformation, and finding a new transformation that minimizes the error metric [Chen 91].

- The above iterative minimization, combined with extrapolation in transform space to accelerate convergence [Besl 92].

- Performing the iterative minimization starting with several perturbations in the initial conditions, then selecting the best result [Simon 96]. This avoids spurious local minima in the error function, especially when the point-to-point error metric is used.

- Performing the iterative minimization using various randomly-selected subsets of points, then selecting the optimal result from among those trials using a robust (least median of squares) metric [Masuda 96].

- Stochastic search for the best transform, using simulated annealing [Blais 95].

Since our focus is on convergence speed, and since the latter three approaches tend to be slow, our comparisons will focus on the first two approaches described above (i.e., the "classic" ICP iteration, with or without extrapolation). The extrapolation algorithm we use is based on the one described by Besl and McKay [Besl 92], with two minor changes to improve effectiveness and reduce overshoot:

- When quadratic extrapolation is attempted and the parabola opens downwards, we use the largest x intercept instead of the extremum of the parabola.

- We multiply the amount of extrapolation by a dampening factor, arbitrarily set to $^1/_2$ in our implementation. We have found that although this occasionally reduces the benefit of extrapolation, it also increases stability and eliminates many problems with overshoot.

On the "fractal" scene, we see that the point-to-plane error metric performs significantly better than the point-to-point metric, even with the addition of extrapolation (Figure 3.15). For the "incised plane" scene, the difference is even more dramatic (Figure 3.16). Here, the point-to-point algorithms are not able to reach the correct solution, since using the point-to-point error metric does not allow the planes to "slide over" each other as easily.

3.5 High-Speed Variants

We may now construct a high-speed ICP algorithm by combining some of the variants discussed above. Like Blais and Levine, we propose using a projection-based algorithm to generate point correspondences. Like Neugebauer, we combine this matching algorithm with a point-to-plane error metric and the standard "select-match-minimize" ICP iteration. The other stages of the ICP process appear to have little effect on convergence rate, so we choose the simplest ones, namely random sampling, constant weighting, and a distance threshold for rejecting pairs. Also, because of the potential for overshoot, we avoid extrapolation of transforms.

All of the performance measurements presented so far have been made using a generic ICP implementation that includes all of the variants described in this chapter. It is, however, possible to make an optimized implementation of the recommended high-speed algorithm, incorporating only the features of the particular variants used. When this algorithm is applied to the "fractal" testcase of Figure 3.10, it reaches the correct alignment in approximately 30 milliseconds. This is considerably faster than our baseline algorithm (based on [Pulli 99]), which takes over one second to align the same scene. It is also faster than previous systems that used the constant-time projection strategy for generating correspondences; these used computationally expensive simulated annealing [Blais 95] or Levenberg-Marquardt [Neugebauer 97] algorithms, and were not able to take advantage of the speed of projection-based matching. Figure 3.17 shows an example of the algorithm on real-world data: two scanned meshes of an elephant figurine were aligned in approximately 30 ms.

This dissertation is not the first to propose a high-speed ICP algorithm suitable for real-time use. David Simon, in his Ph. D. dissertation [Simon 96], demonstrated a system capable of aligning meshes in 100-300 ms. for 256 point pairs (one-eighth of the number of pairs

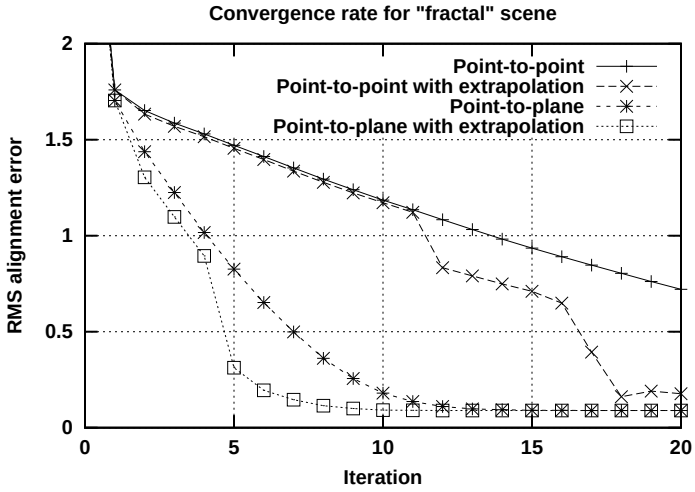


Figure 3.15: Comparison of convergence rates for the "fractal" meshes, for different error metrics and extrapolation strategies.

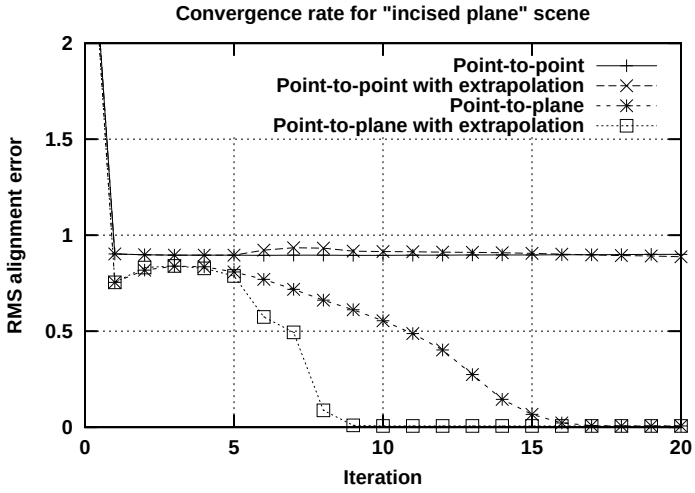


Figure 3.16: Comparison of convergence rates for the "incised plane" meshes, for different error metrics and extrapolation strategies. Normal-space-directed sampling was used for these measurements.

Figure 3.17: High-speed ICP algorithm applied to scanned data. Two scans of an elephant figurine from a prototype video-rate structured-light range scanner were aligned by the optimized high-speed algorithm in 30 ms. Note the interpenetration of scans, suggesting that a good alignment has been reached.

considered throughout this chapter). His system used closest-point matching and a point-to-point error metric, and obtained much of its speed from a closest-point cache that reduced the number of necessary *k-d* tree lookups. As we have seen, however, the point-to-point error metric has substantially slower convergence than the point-to-plane metric we use. As a result, our system appears to converge almost an order of magnitude faster, even allowing for increase in processor speeds. In addition, our system does not require preprocessing to generate a *k-d* tree.

3.6 Dual Perspective Range Images

The fast ICP algorithm described above requires range samples that are organized in a regular grid in order to find matching points via projection. The simplest way of creating such a system would be to store a range image in which depth samples are indexed by row and column in the camera image. Given the range scanner described in Chapter 2, however, we obtain range data that is irregularly spaced along each camera row. Moreover, the x location of each sample is obtained with sub-pixel precision. Thus, a resampling step would be necessary in order to generate a range image indexed by camera x and y.

As an alternative to this resampling step, we make the observation that each range sample is uniquely indexed by the camera row and projector column. Although this no longer corresponds to a simple perspective range image, it is nevertheless possible to compute the pro-

jection of any 3D point onto this range image in constant time. Thus, having a range image indexed in this way avoids a resampling while still maintaining the property of fast projection. Figure 3.18 illustrates various kinds of range images – our scanner corresponds to case (d). To perform the projection-based matching, we would project samples from another range image (not shown) back along the blue lines in Figure 3.18d.

Note that historically other kinds of range scanners have also generated "range images" that did not correspond to simple perspective projection. For example, translating laser-stripe scanners produce "half-perspective" range images (Figure 3.18c).

3.7 Summary

This chapter has presented a classification and comparison of several variants on the ICP algorithm. A comparison of the running times of several matching and minimization strategies suggests that combining a constant-time variant for finding point pairs (projection-based matching) with point-to-plane minimization results in a high-speed ICP algorithm capable of aligning two meshes in a few tens of milliseconds, without requiring expensive preprocessing.

This high-speed ICP variant is the one we incorporate into our real-time model acquisition pipeline. As described in the following chapter, we align successive range images to each other, permitting a complete model of an object to be obtained without the need for calibrated motion. After alignment, the samples of these range images are merged and displayed, letting a user see a model of the object as it is being built up.

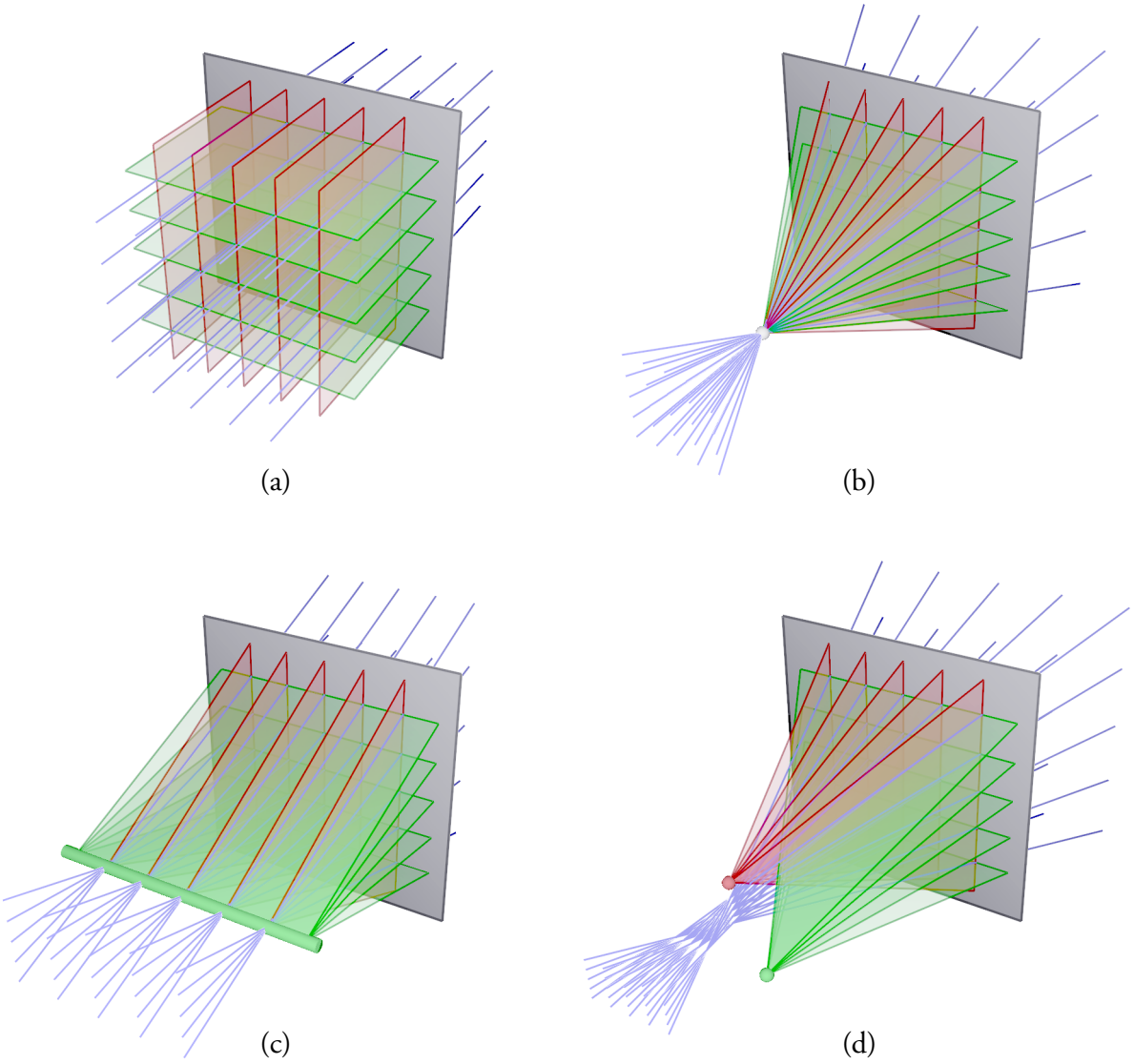


Figure 3.18: Geometry of range images. (a) In an orthographic range image, a depth is stored along each of a 2D family of parallel rays (blue). These rays are defined by the intersections of two families of horizontal (green) and vertical (red) planes. (b) In a perspective range image, range samples are defined along rays that are the intersections between two families of planes passing through a single point of projection. (c) Many translating laser scanners produce a range image in which depth is stored along rays defined by a family of horizontal planes passing through a line, and a family of parallel vertical planes. (d) Our structured-light range scanner produces a range image in which depth is stored at the intersections of two families of planes passing through different points.

"... as large as life, and twice as natural."

– Lewis Carroll (Charles Lutwidge Dodgson)

Chapter 4

Real-time Model Acquisition: Merging, Rendering, and System Integration

We now describe the design of a complete 3D model acquisition system that produces aligned range images in real time and a complete model of a rigid object in a few minutes. Recall our real-time model acquisition pipeline, reproduced in Figure 4.1. The first four boxes of this pipeline were the subject of Chapter 2; the result is a series of range images of a moving object, generated at 60 Hz. The next stage of the pipeline involves aligning the range images to each other, using the ICP algorithm of Chapter 3. In the simplest case, this just consists of aligning each range image to the previous one, though we later describe how to refine this algorithm to accommodate cases in which ICP fails. These aligned scans form the input to the final stages of the pipeline, namely the algorithms for real-time merging and display.

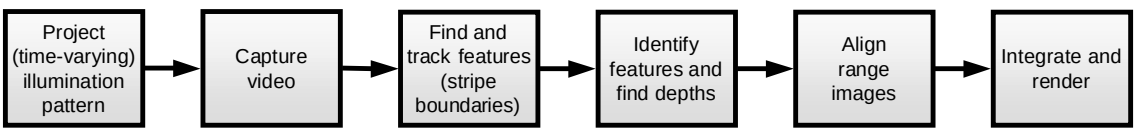


Figure 4.1: Real-time model acquisition pipeline.

In this chapter, we begin by describing the algorithm for combining range data in real time, which uses a 3D grid data structure. The data is then displayed using *point rendering*, a method that draws samples directly without reconstructing a triangulated manifold surface. These algorithms do not result in the highest possible quality, but they are efficient enough on current hardware to display the partially-complete model as it is being scanned and are good enough to let the user see and fill holes. In Section 4.2 we describe how a user might interact with this model acquisition system, focusing on situations in which the automatic alignment fails. Finally, in Section 4.3 we present quantitative measurements and statistics about the system.

4.1 Merging and Rendering

The goal of a live preview of the scanned model is to give the user feedback about which areas of the model have been scanned, and whether any holes are left. Since the data is accumulating so rapidly, it is necessary to perform some sort of merging or discarding of redundant, overlapping 3D data in order to restrict the number of primitives that must be rendered, thus maintaining an acceptable interactive frame rate.

4.1.1 Previous Work

A variety of algorithms have been proposed for merging (aligned) range scans into a single model (see [Curless 97] for an in-depth survey). In general, these may be classified as either operating purely on point clouds, or using the adjacency information in each constituent range image. In the former category are algorithms such as alpha-shapes [Edelsbrunner 92], the implicit reconstruction algorithm of [Hoppe 92], crusts [Amenta 98], and ball-pivoting [Bernardini 99]. These algorithms are general and conceptually simple, but often do not produce good results in the presence of misalignment and noise. Using the connectivity of each range scan, as in the Zipper [Turk 94] and VRIP [Curless 96] algorithms, is usually faster and produces better results in the presence of noise.

All of the algorithms mentioned above have the significant drawback of being unsuitable for real-time use. Though they can produce high-quality results, running times are typically on the order of several seconds or minutes for the sizes of scans produced by our prototype range scanner (on the order of several thousand samples per range image – see Table 4.1). Thus, we make the decision to sacrifice quality in the reconstruction in order to perform merging

at interactive rates; we may still use one of the above algorithms to perform a high-quality reconstruction as a post-process.

4.1.2 Merging Algorithm

The algorithm we implement for merging data combines some features of the above two categories (i.e., point cloud- and range image-based). We use adjacency information to reconstruct a range image at each frame, and use this triangulated mesh to compute per-vertex normals. We then discard the connectivity information, and perform merging based only on the points themselves. This consists of quantizing all points to a 3D grid, and combining all points that map to the same location in this grid (the idea of collapsing all points in a voxel grid cell is similar to the one used in the mesh simplification method of Rossignac and Borrel [Rossignac 93]). A running-average normal is maintained at each grid cell, and point rendering is used to display the combined grid data structure (see Section 5.1 for more background on point rendering). The rendering is done using a method called "splatting," in which a screen-aligned splat (e.g. a circle or an alpha-blended Gaussian) is drawn for each point. These are scaled such that the splats for neighboring points overlap without leaving a gap. When the points are regularly arranged on a grid, such as in this application, their spacing is known *a priori* and it is simple to select a splat size for a given viewpoint that guarantees that the splats for adjacent samples overlap without leaving holes.

Although this merging algorithm runs quickly, it does not produce the highest-quality results. If scans are misaligned by more than the grid spacing, samples from those scans will obviously not be merged, creating two or more "layers" of occupied voxels. After many range images have been incorporated, the result is a region of occupied voxels in the vicinity of the true surface. Since these voxels are rendered with Z-buffering, the visible voxels will be those that are outermost. Thus, the surface will appear thickened and possibly noisy. This still results in acceptable quality, however, since shading provides a much greater visual cue than shape [Levoy 88], and the voxels are rendered using the correct normals. Moreover, as stated earlier, the quality of the real-time reconstruction only needs to be good enough to guide the user in positioning the object and determining the presence of holes in the model; a high-quality reconstruction is performed offline.

Outlier Elimination: As mentioned above, moderate misalignment and noise does not degrade the appearance of the merged grid unacceptably. Large outliers, however, *are* visible, and

it is necessary to eliminate them before merging. Fortunately, since we acquire data at such a high rate, we can be aggressive in eliminating outliers; any correct data that is mistakenly discarded is highly likely to be acquired again, because of the feedback given to the user.

Our outlier rejection algorithm operates after a range image has been triangulated. We eliminate all long and thin triangles, as well as triangles that are backfacing with respect to the camera or projector. If this elimination results in points that are not part of any triangle, those points are eliminated as outliers.

Depending on the parameter in the above algorithm (that is, the "skinniest" permitted triangles), this method may eliminate a certain amount of data that was seen at a large tilt with respect to the camera. This is not necessarily a drawback, since that data is likely to be of lower quality. Discarding this data encourages the user to turn those regions of the object towards the range scanner (so that higher-quality data may be obtained). For creases in some objects, however, it may be impossible to orient the object such that a given portion of the surface is both visible from and normal to the camera. Therefore, the outlier elimination should not be set to be *too* aggressive. In practice, we currently discard triangles in which the smallest angle is less than 10 degrees. This could be increased even further by adopting the strategy of not permanently discarding points deemed to be outliers, but retaining them for use (with low confidence) during the offline merging.

Grid Size: In order to perform this voxel-based merging and rendering, it is necessary to choose the grid size. Smaller grid cells result in greater detail, at the expense of greater memory usage, lower frame rates, and greater sensitivity to noise. Larger grids give higher frame rates, but the individual splats become more visible. In practice, we use a grid size on the order of the spacing of samples given by the range scanner; higher grid resolution results in substantially lower frame rates without corresponding increases in perceived quality. For our prototype, we usually use a grid size of $^1/_2$ mm., which is roughly equal to the range sample spacing near the front of our working volume (see the discussion in Section 4.3).

4.1.3 Results

Some sample frames produced by the grid-based merging and point rendering implementation are shown in Figure 4.2. Once all the data has been acquired, the grid data structure used for interactive rendering may be discarded, and a high-quality model is produced by the following (offline) pipeline:

- Starting from scan positions and transforms computed by the real-time algorithm, ICPs are performed on consecutive scans, as well as additional pairs of overlapping scans. The algorithm used at this stage is not the high-speed ICP variant but the algorithm of of [Pulli 99], which was used as our high-quality "baseline" algorithm in Chapter 3.
- A globally-optimal alignment is computed by simultaneously considering the results of all the scan-to-scan alignments and performing a global relaxation [Pulli 99].
- The scans are triangulated, and merged using the VRIP algorithm [Curless 96].
- A final, merged triangular mesh is extracted using marching cubes [Cline 88].

Additional processing (such as decimation, smoothing, or hole-filling) may now be performed on the model, or the model may be converted to QSplat format for rendering (as described in the following chapter).

4.2 Interaction with the System

So far, we have suggested that the alignment stage of the pipeline always involves performing an ICP between a range image and the previous one. Under ideal conditions, this usually works, but in practice an ICP will occasionally fail. This might happen if the object is moved too fast, and will certainly happen if the object is moved out of the field of view of the scanner. Once such a situation has been corrected (i.e., the object slows down to a reasonable velocity and/or comes back within the field of view), it is necessary to regain the alignment of the new scans to the previously-acquired model.

A reasonable approach to restarting after a failed alignment might be to treat it as a general problem of aligning two 3D models given an unknown initial pose. As mentioned in the previous chapter, there exist algorithms to solve this problem, but they are typically slow and not robust. Instead, we may take advantage of the human operator's strengths in performing pattern matching by simply displaying one or more range images, asking the user to position the object so it roughly lines up with one of them, and attempting to perform ICP to those range images until one of the ICPs succeeds.

If we adopt this strategy, we must choose which range images to present to the user. We have observed that just using the last range image before the ICP failure is not a good choice. If the object is moving out of the field of view, the last few range images see smaller and smaller

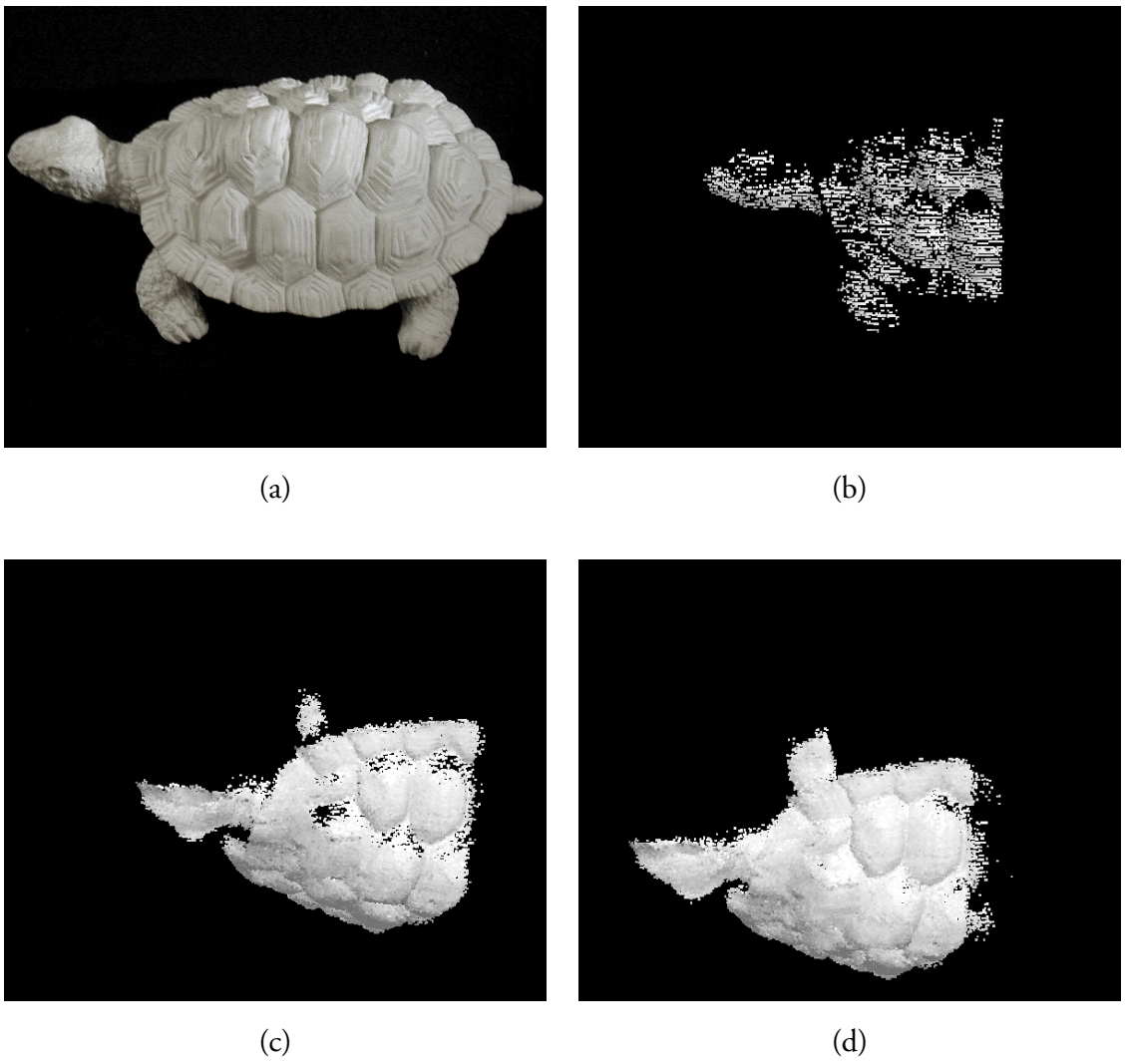
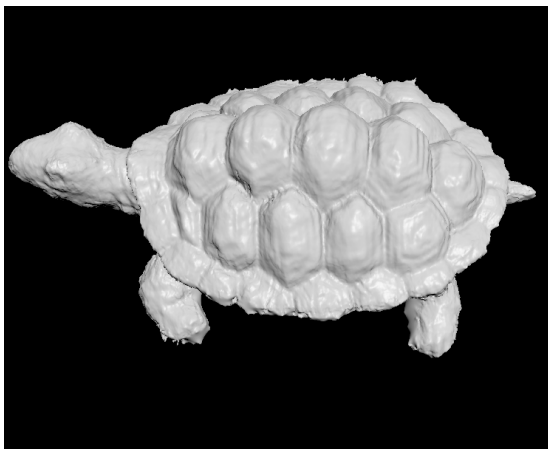
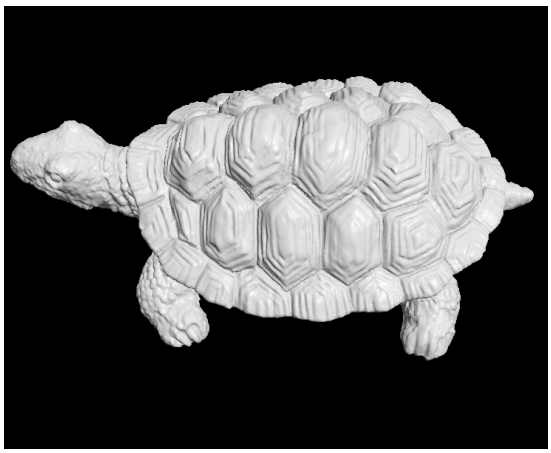


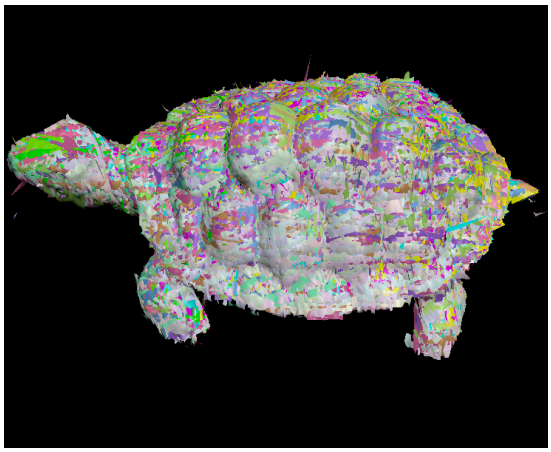
Figure 4.2: Results of point-based merging and rendering. (a) Photograph of a turtle figurine, about 18 cm. long. (b) Shortly after the start of scanning, data has been accumulated relatively sparsely, and individual point primitives are visible. (c) After a few seconds of scanning, the front part of the turtle has been covered relatively well. However, the user sees some remaining holes, and is able to rotate the object (d) to fill those holes.



(e)



(f)



(g)

	Our scanner (e)	**Cyberware** (f)
Range images	1,830	22
… used in VRIP	183	22
Avg. samples / scan	3,140	75,380
Sample spacing (x)	0.75 mm.	0.25 mm.
Sample spacing (y)	0.5 mm.	0.33 mm.
Alignment	automatic	manual
Scanning time	4 min.	30 min.

(h)

Figure 4.2 (cont.): (e) After the remainder of the turtle has been scanned, an offline global registration is run on all of the original range images, and the scans are merged at 0.25 mm. resolution. Every tenth range image is used in this reconstruction. (f) For comparison, a model created with a Cyberware Model 15 scanner. The sample-to-sample spacing for this scanner is aproximately half that in our prototype, resulting in a sharper model. (g) The raw range images used for the reconstruction in (e). Each range image is shown in a different color. (h) Statistics about the merged models in (e) and (f).

pieces of the object. A user is likely to have difficulty in aligning the object to one of these. More seriously, having smaller and smaller amounts of range data decreases the constraints on ICP, thus increasing the possibility of obtaining an incorrect alignment.

Anchor Scans: In order to make it possible to display good range images for restarting ICP, we maintain some number of *anchor scans* to which we attempt to align. We would like these to have the following characteristics:

- The anchor scans should be large enough to be recognizable to the user and for ICP to operate reliably.
- We would like the anchors to have relatively low overlap among themselves, so that they cover as much of the object as possible.

In order to maintain the above properties, given anchors $A_1 \ldots A_n$ and a new range image R the alignment and recovery algorithms are as follows:

0. If R is the first scan, we set $A_1 \leftarrow R$, and proceed to the next scan. Otherwise, we copy the transform produced by the last successful ICP to the current scan. (Note that this currently does not perform any prediction – see the discussion in Section 7.3.1.)
1. If R is empty or has fewer than *align_min* points we do not attempt alignment, since such an alignment would be more likely to be incorrect. We indicate to the user that the object is out of the working volume.
2. Otherwise, we attempt to align R to A_1.
   - If the initial ICP fails, we repeat the ICP a number of times, using more iterations, larger thresholds, and several perturbations to the starting position of R.
   - If all of the above attempts fail, we attempt to ICP R to each of $A_2 \ldots A_n$. If any of these succeeds, we move that anchor to the front of the list (so it becomes A_1).
   - If ICP to all of the anchors has failed, we draw R and $A_1 \ldots A_n$ (in different colors), and ask the user to move the object to line up R with one of the anchors.
3. Once an ICP has succeeded, we evaluate whether R should become a new anchor. This is done if:
   - R is large (more than *anchor_min* points)

- The overlap between R and the anchor for which ICP succeeded is less than *overlap_max*.

If both of these conditions hold, R becomes A_1, $A_1 \ldots A_{n-1}$ become $A_2 \ldots A_n$, and the old A_n is deleted.

There are four constants that affect the operation of the above algorithm, and an optimal choice depends on the characteristics of the particular scanner. For our prototype, we have made the following choices:

- *align_min*, the minimum size of a range image for which we attempt alignment, must be large enough for ICP to converge reliably. In our implementation, we set this to 500 points.

- *anchor_min*, the minimum size of an anchor scan, should be large enough to contain human-recognizable features. We typically set this to 1000 points.

- *overlap_max* is the overlap threshold below which we declare the current scan to be a new anchor. We would like this to be low, so that different anchors cover as much of the surface of the object as possible. We have found that an overlap of $^1/_2$ works well in practice.

- n, the number of anchors we maintain, affects the quality of the user's interaction with the model acquisition system. Keeping more anchors gives the user greater flexibility in restarting the alignment, but also slows down the system (though only in the failed-ICP case), since ICP must be attempted to each of the anchors. In our current system we typically use $n = 5$ or 10.

The above rules for deciding when a scan becomes an anchor were chosen so that anchors tend to be large and reasonably spaced out. This makes it easy for the user to restart the "normal" ICP after a failed alignment. In addition, it has the benefit of preventing the accumulation of alignment errors when the object is not moving. This is because, when the object is not moving, the overlap between each new scan and the anchor will be large, so the anchor will not change. Thus, all of the scans will be aligned to the same anchor. If instead we performed simple scan-to-scan alignment, the small errors introduced by each ICP would have the potential of accumulating, leading to more global drift in the alignment.

Camera Positioning: During the normal course of scanning, we position the virtual camera in the same location relative to the model as the actual position of the real camera relative to the object. This provides a natural user interface in which the user may look at different parts of the model simply by moving the object. In some cases, however, it may be more convenient for the user to examine the model in detail without moving the object. For this case, we implement a virtual trackball that lets the user directly fly around the model.

Another camera-placement possibility might be to position the virtual camera at the user's current head position. This could be obtained from a separate camera pointed at the user, an active tracker worn on the head, or simply hardcoded given the layout of the scanner and the user's probable location. A further possible refinement would be a partially-transparent head-mounted display with the model overlaid on the actual object. We have not experimented with any of these options.

4.3 System Summary – Measurements and Statistics

We now describe some measurements and statistics related to various aspects of our current implementation. These represent only one point in the design space for such a scanner, and we anticipate that future systems might further explore the design possibilities for different working volume, resolution, and accuracy.

Hardware: The system uses a Compaq MP1800 DLP projector, with a maximum resolution of 1024x768. Because of the need to synchronize it with the video camera, we currently send an S-Video signal to the projector, limiting us to a resolution of 640x240 interlaced. The camera we use is a Sony DXC-LS1 NTSC camera, with a 1/500 sec. shutter speed. The video is digitized by a Pinnacle Studio DC10+ capture card, yielding interlaced 640x240 video fields at 60 Hz.

Layout: The layout of the system determines its working volume and resolution. For the scans presented here, we have positioned the camera and projector 20 cm. apart, with a triangulation angle of 21 degrees. This configuration produces a working volume approximately 10 cm. across, but, as illustrated in Figure 4.3, the working volume has a long "tail" (with progressively worse sample spacing, focus, and accuracy with increasing distance from the camera). Near the front of the working volume, samples are spaced roughly every 0.5 mm. in Y (parallel to the stripe direction) and every 0.75 mm. in X (perpendicular to the stripes).

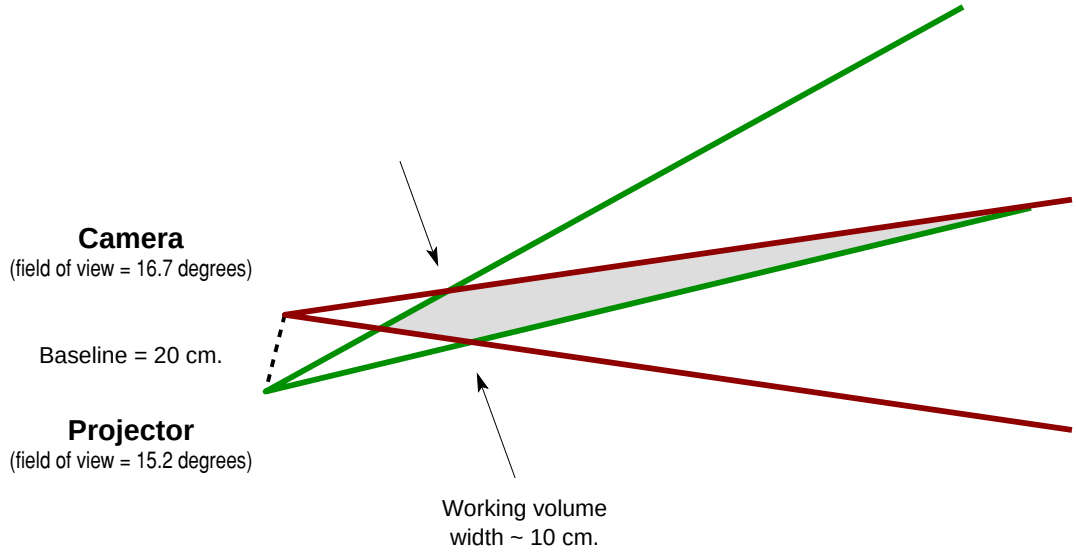


Figure 4.3: Camera-projector layout in our prototype. Note that the working volume has a long "tail," in which sample spacing, focus, and accuracy get progressively worse with increasing distance from the camera. Therefore, it is desirable to perform most scanning towards the front of the working volume. In order to enforce this, we frequently place a non-reflective backdrop to cut off the tail of the working volume.

Accuracy and Precision: There are several ways in which the accuracy of our range scanner may be characterized. We may look at the spacing between samples on the surface, the noise in the location of each sample, the presence of outliers, and the distortion in each range image due to miscalibration. All of these depend on the physical arrangement of camera and projector, and so must be compared to the size of the working volume. The figures below all apply to the front 10 cm. section of working volume discussed above.

The sample spacing in our prototype, as we have stated, is 0.5-0.75 mm. The noise in each of these samples is primarily due to the error in locating a stripe boundary, which may be due to noise in the camera and digitizer or due to object texture. For surfaces without high-frequency texture, we may find the locations of the stripe boundaries with subpixel precision, and we estimate the noise in each sample to be under 0.1 mm. This is due almost entirely to noise in the camera and capture card: using a higher-quality camera and digitizer (Toshiba IK-TU40A 3-CCD camera and DPS-465 digitizer) we obtain lower per-sample noise – under 0.03 mm. Figure 4.4 illustrates the noise using this high-quality camera, both with and without subpixel estimation.

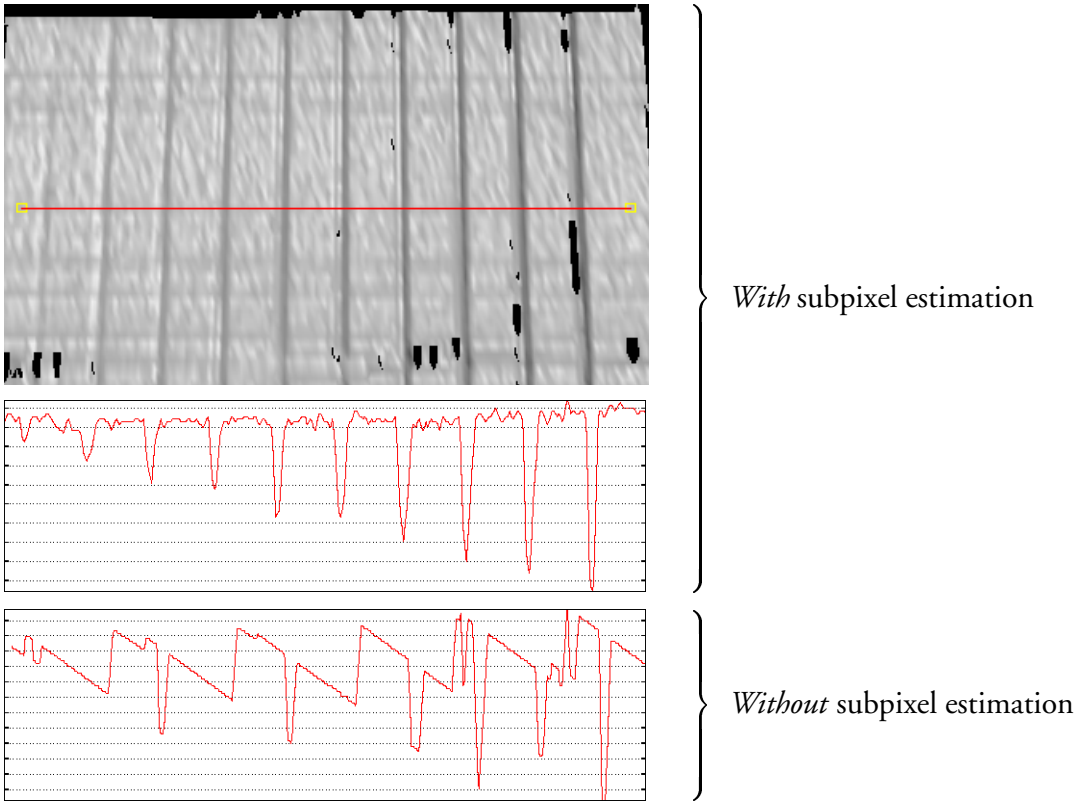


Figure 4.4: Comparison of decoding accuracy with and without sub-pixel estimation of stripe boundaries. At top, a rendering of a scanned target (approximately 150x50 samples) with grooves ranging from 0.1 mm to 1.0 mm in depth. At center, the depth profile of a slice through the target, if sub-pixel estimation of stripe boundaries *is* used. At bottom, the corresponding depth profile when sub-pixel estimation *is not* used. A sawtooth pattern results from the quantization of the locations of stripe boundaries to the nearest pixel in the camera image. The grooves in the target are 1 cm. apart, and the vertical tick marks in the graphs are 0.1 mm. apart.

Ultimately, using a high-quality camera and digitizer, the the limit on the minimum achievable local noise depends on two factors. The first is the focus of the camera and projector. Using a smaller aperture (especially on the projector – most commercially-available models use large apertures) would permit better localization of stripe boundaries. The second major limit on the accuracy of this system is scene texture. When the reflectance of the surface varies rapidly (on the order of the camera pixel spacing), we are not able to perform accurate subpixel estimation, and so the per-sample error is substantially larger, on the order of 0.2 mm.

Outliers: As discussed earlier, our range scanner occasionally generates outlier points, and it is necessary to eliminate them before merging. We have tuned our outlier detector to be fairly aggressive at discarding suspicious data, and as a result we discard practically all outliers while still retaining the majority of good data. By manually evaluating a large number of range images, we estimate the rate of undetected outliers to be under 0.01%, which corresponds to roughly one undetected outlier point per range image. It would be possible to be even more aggressive at discarding these outliers, at the expense of rejecting more valid data.

Calibration and Warping: The final characteristic of accuracy is the distortion in our scanner due to miscalibration. We have adopted a calibration procedure in which known 3D points in the scene are measured using a Faro arm touch probe, and their (u, v) camera locations as well as projector p coordinate are found. The optimal set of intrinsic and extrinsic calibration parameters are found by minimizing the error in all the $(u, v, p) \rightarrow (x, y, z)$ mappings simultaneously.

Although we may estimate the error in the calibration directly from the convergence of the minimization algorithm, a more meaningful estimate arises from considering the maximal misalignment between range images of the same object taken at a variety of different positions and orientations. For the turtle data set of Figure 4.2, we observe a misalignment of approximately 0.5 mm. (after high-quality ICP and global registration), leading us to conclude that the distortion is of this order of magnitude.

CPU Usage: Our prototype uses a dual-CPU system, with Intel Pentium III Xeon processors running at 1 GHz. One CPU is used for the first few stages of the range scanning pipeline, namely grabbing video frames, finding stripe boundaries, matching the boundaries across time, and identifying the boundaries from the accumulated illumination history. The second CPU performs triangulation to find 3D points, aligns the scans using the fast ICP algorithm, in-

tegrates range images into the 3D grid, and renders the updated grid. The first piece of this pipeline operates at full speed (60 Hz.), while the second operates slower, approximately 10 Hz.

The reason for choosing this unequal division of stages among CPUs is to have the matching stage not drop frames; this permits the highest-possible speeds for object motion. It is not as critical for the rest of the pipeline to run at the full 60 Hz. camera rate, since it only results in a lower frame rate for the display.

Statistics about our implementation are summarized in Table 4.1.

Table 4.1: Statistics about our implementation.

Hardware

Projector	Compaq MP1800
Camera	Sony DXC-LS1
CPU	Dual Pentium III Xeon, 1 GHz.

Layout

Baseline	20 cm.
Triangulation angle	21 degrees
Working volume	10 cm. wide; long "tail" (see Figure 4.3)

Acquired data

Camera field size	640 x 240
Projected stripe boundaries	110
Samples per range image	26,400 (maximum); 5,000 (typical)
Acquisition rate	60 Hz

Accuracy and precision

Sample spacing (near front of working volume)	0.75 mm. (x); 0.5 mm. (y)
Per-sample noise	< 0.1 mm. (typical); 0.2 mm. (if highly textured)
Distortion across working volume	0.5 mm. (estimated)

Typical CPU usage

CPU #1: Finding boundaries	11 ms.
Matching boundaries	4 ms.
Identifying boundaries	2 ms.
CPU #2: Determining positions and normals	18 ms.
Aligning scans	15 ms.
Merging into grid	10 ms.
Rendering	Depends on amount of data (typically 15–50 ms.)

"Large elements in order brought,
And tracts of calm from tempest made"

– Alfred, Lord Tennyson

Chapter 5

QSplat: Rendering of Large Models

The Digital Michelangelo Project [Levoy 00] has demonstrated high-quality range scanning of large objects, producing detailed models ranging from 100 million to 1 billion range samples. In the previous three chapters, we have argued that model acquisition technologies are evolving towards making the creation of large 3D meshes faster, easier, and less expensive. Using the output of these systems, however, remains a challenge: traditional algorithms for display, simplification, and progressive transmission of meshes are impractical for data sets of this size.

In this chapter we describe QSplat, a system for representing and progressively displaying large meshes that combines a multiresolution hierarchy based on bounding spheres with a rendering system based on points. A single data structure is used for view frustum culling, backface culling, level-of-detail selection, and rendering. The representation is compact and can be computed quickly, making it suitable for large data sets. The implementation, originally written for use in a large-scale 3D digitization project, launches quickly, maintains a user-settable interactive frame rate regardless of object complexity or camera position, yields reasonable image quality during motion, and refines progressively when idle to a high final image quality. We have demonstrated the system on scanned models containing hundreds of millions of samples.

After examining previous work on point rendering and level-of-detail control, we present the QSplat data structure and the preprocessing and run-time rendering algorithms. We discuss some of the tradeoffs and design decisions involved in making it practical for large meshes,

including quantization, file layout, and splat shape. Finally, we describe the rendering performance of the system, and discuss its preprocessing costs. A network-streaming version of QSplat is presented in the following chapter.

5.1 Previous Work

Previous approaches for representing and displaying large models can be grouped into point rendering, visibility culling, level-of-detail control, and geometric compression.

Point Rendering: Computer graphics systems traditionally have used triangles as rendering primitives. In an attempt to decrease the setup and rasterization costs of triangles for scenes containing a large amount of geometry, a number of simpler primitives have been proposed. The use of points as a display primitive for continuous surfaces was introduced by Levoy and Whitted [Levoy 85], and more recently has been revisited by Grossman and Dally [Grossman 98]. Point rendering has been incorporated into commercial products – the Animatek Caviar system, for example, uses point rendering for animated characters in video games [Animatek]. Particles have also been used in more specialized contexts, such as rendering fire, smoke, and trees [Csuri 79, Reeves 83, Max 95].

A concept related to point rendering is *splatting* in volume rendering [Westover 89]. For large volumes, it is natural to use a hierarchical data structure to achieve compression of regions of empty space, and Laur and Hanrahan have investigated hierarchical splatting for volumes represented using octrees [Laur 91]. Although splatting is best suited to the case in which the projected voxel size is on the order of the pixel size, other regimes have also been examined. The *dividing cubes* algorithm proposed by Cline et al. is intended for use when voxels are larger than pixels [Cline 88]. For voxels smaller than pixels, Swan et al. have proposed algorithms for producing correctly antialiased results [Swan 97].

Visibility Culling: Frustum and backface culling algorithms, such as those used by QSplat, have appeared in a large number of computer graphics systems. Hierarchical frustum culling based on data structures such as octrees has been a standard feature of most systems for rendering large scenes [Samet 90]. Backface culling of primitives is commonly implemented in hardware, and Kumar and Manocha have presented an algorithm for hierarchical backface culling based on cones of normals [Kumar 96].

Another class of visibility culling algorithms includes methods for occlusion culling. Greene et al. describe a general algorithm to discard primitives that are blocked by closer geometry using a hierarchical Z-buffer [Greene 93]. Other, more specialized occlusion algorithms can also be used if the scene is highly structured. Systems for architectural flythroughs, for example, often use the notion of *cells* and *portals* to cull away entire rooms that are not visible [Teller 91]. QSplat currently does not perform any sort of occlusion culling – it would provide minimal benefit for viewing the scanned models we are considering. Occlusion culling would, however, be a useful addition for scenes of greater depth complexity.

Level of Detail Control: Rendering a large data set at low magnification will often cause primitives to be smaller than output device pixels. In order to minimize rendering time in these cases, it is desirable to switch to a lower-resolution data set with primitives that more closely match the output display resolution. Among LOD algorithms, one may differentiate those that store entire objects at discrete levels of detail from methods that perform finer-grained LOD control. The algorithms in the latter class can control the number of primitives continuously, minimizing "popping" artifacts, and often vary the level of detail throughout the scene to compensate for the varying magnification of perspective projection.

Multiresolution analysis represents an object as a "base mesh," with a series of corrections stored as wavelet coefficients [Eck 95]. Certain et al. have implemented a real-time viewer based on multiresolution meshes that can select an arbitrary number of wavelet coefficients to be used, and so draw a mesh with any desired number of polygons [Certain 96]. Their viewer also includes features such as progressive transmission and separate sets of wavelet coefficients for geometry and color.

Progressive meshes store a base mesh together with a series of vertex split operations that are used generate the higher-resolution versions [Hoppe 96]. Progressive meshes have been incorporated into a real-time viewer that performs view-dependent refinement for real-time flythroughs of scenes of several million polygons [Hoppe 97, Hoppe 98]. The viewer can not only select an arbitrary number of polygons to draw, but also refine different parts of an object to different resolutions. Other recent systems that allow level of detail to vary throughout the scene include the *ROAM* terrain rendering system [Duchaineau 97], and *LDI trees* [Chang 99]. The implementation of LOD control in QSplat has the same goal as these systems, permitting the level of detail to vary smoothly throughout a scene according to projected screen size.

Geometric Compression: The goal of geometric compression is to reduce the storage and memory requirements of large meshes, as well as their transmission costs. Deering has presented a system for compression of mesh connectivity, vertex locations, colors, and normals, which was later implemented in hardware [Deering 95]. More recent research, such as the *Topological Surgery* scheme by Taubin and Rossignac, has focused on reducing the cost of representing mesh connectivity and improving the compression of vertex positions [Taubin 98]. Pajarola and Rossignac have applied compression to progressive meshes, yielding a scheme that combines level-of-detail control and progressive refinement with a compact representation [Pajarola 99]. Their algorithm, however, has higher preprocessing and decoding costs than QSplat.

5.2 QSplat Data Structure and Algorithms

QSplat uses a hierarchy of bounding spheres [Rubin 80, Arvo 89] for visibility culling, level-of-detail control, and rendering. Each node of the tree contains the sphere center and radius, a normal, the width of a normal cone [Shirman 93], and optionally a color. One could generate such a bounding sphere hierarchy from polygons, voxels, or point clouds, though for our application we only needed an algorithm for generating the hierarchy from triangular meshes. The hierarchy is constructed as a preprocess, and is written to disk.

5.2.1 Rendering Algorithm

Once the hierarchy has been constructed, the following algorithm is used for display:

```
TraverseHierarchy(node)
{
   if (node not visible)
      skip this branch of the tree
   else if (node is a leaf node)
      draw a splat
   else if (benefit of recursing further is too low)
      draw a splat
   else
      for each child in children(node)
         TraverseHierarchy(child)
}
```

We now examine several stages of this basic algorithm in detail.

Visibility Culling: As we recurse the bounding sphere hierarchy, we cull nodes that are not visible. Frustum culling is performed by testing each sphere against the planes of the view frustum. If the sphere lies outside, it and its subtree are discarded and not processed further. If the sphere lies entirely inside the frustum, this fact is noted and no further frustum culling is attempted on the children of the node.

We also perform backface culling during rendering, using the normal and cone of normals stored at each node. If the cone faces entirely away from the viewer, the node and its subtree are discarded. We also detect the case of a cone pointing entirely towards the viewer, and mark its children as not candidates for backface culling.

Determining When to Recurse: The heuristic used by QSplat to decide how far to recurse is based on projected size on the screen. That is, a node is subdivided if the area of the sphere, projected onto the viewing plane, exceeds a threshold. The cutoff is adjusted from frame to frame to maintain a user-selected frame rate. We currently use a simple feedback scheme that adjusts the threshold area by the ratio of actual to desired rendering time on the previous frame. Funkhouser and Séquin have demonstrated a predictive algorithm for LOD control that results in smaller frame-to-frame variation of rendering times [Funkhouser 93]; however, we have not implemented this. We also have not incorporated any algorithm for smooth transitions as sections of the model change from one level of detail to another, such as the geomorphs in Hoppe's progressive mesh system [Hoppe 98]. Given the modest changes in appearance as we refine and the quick changes in viewpoint typical in our application, we have not found the absence of smoothing visually significant; other applications, however, might benefit from smoother transitions.

Although screen-space area is the most popular metric for LOD control, other heuristics have been proposed for determining how far to recurse [Duchaineau 97, Hoppe 97]. Within the framework of our system, one could incorporate rules for recursing further around silhouette edges (using per-node normals), in areas of high curvature (using normal cone widths), or in the central "foveal" region of the screen (which uses only projected position).

The above implementation of frame rate control is used during interactive manipulation of the model. Once the user stops moving the mouse, we redraw the scene with successively smaller thresholds until a size of one pixel is reached. Figure 5.1 shows a sample scene rendered by QSplat at several levels of refinement.

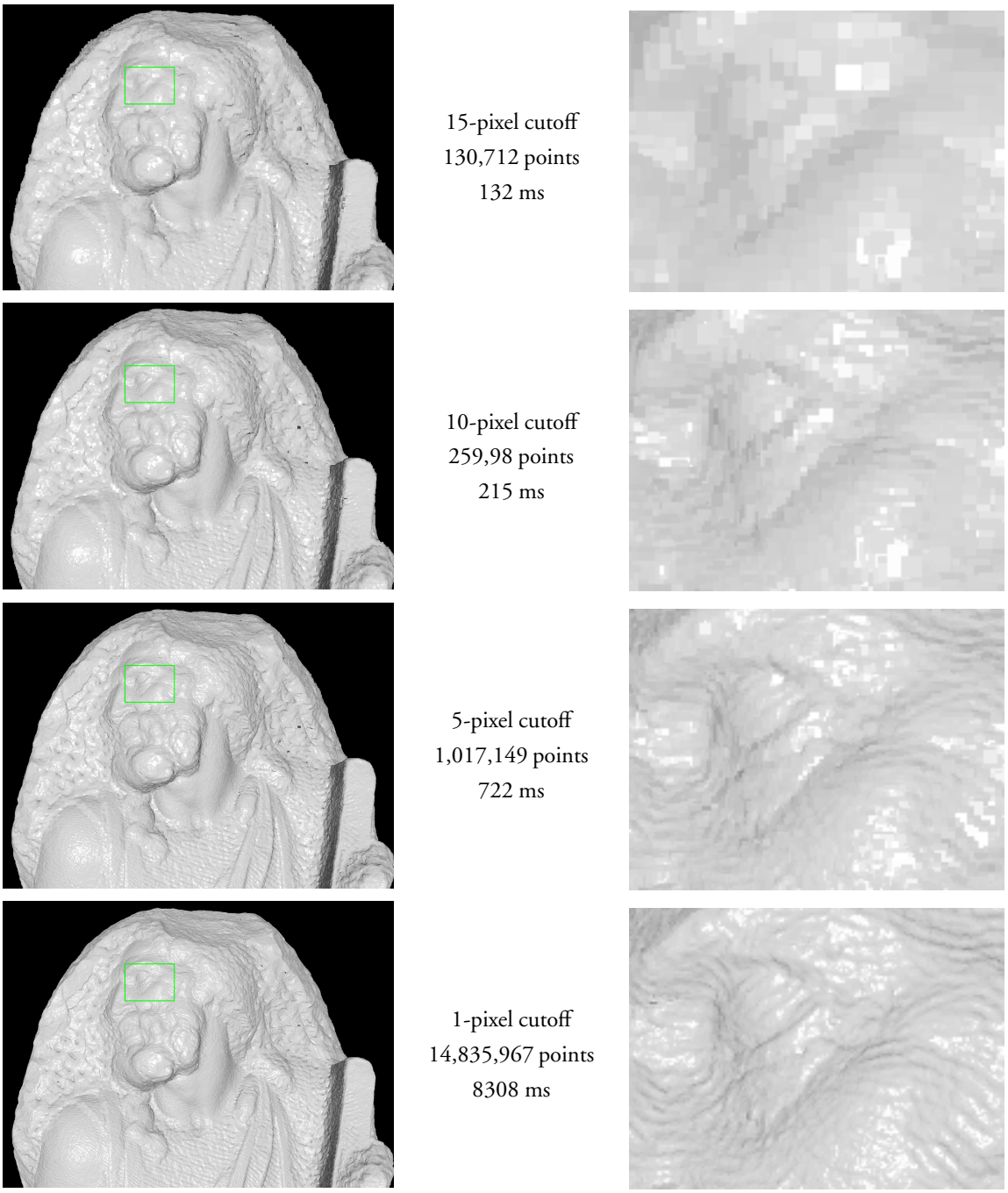


Figure 5.1: A model of Michelangelo's statue of St. Matthew rendered by QSplat at several levels of refinement. Rendering was done on an SGI Onyx2 with InfiniteReality graphics, at a screen resolution of 1280x1024. The model was generated from a mesh with 127 million samples, representing a statue 2.7 meters tall at 0.25 mm resolution. The images at right are closeups of the outlined areas at left.

Drawing Splats: Once we have either reached a leaf node or decided to stop recursing, we draw a splat representing the current sphere [Westover 89]. The size of the splat is based on the projected diameter of the current sphere, and its color is obtained from a lighting calculation based on the current per-sphere normal and color. Splats are drawn with Z-buffering enabled to resolve occlusion. We discuss the shape of each splat in Section 5.3.3.

5.2.2 Preprocessing Algorithm

Our preprocessing algorithm begins with a triangular mesh representing the model to be encoded. Although one could build up a QSplat hierarchy directly from a point cloud, starting with a mesh makes it easy to compute the normals at each node. If we did not have a mesh, we would have to compute normals by fitting a plane to the vertices in a small neighborhood around each point. Beginning with a mesh also makes it possible to assign sphere sizes to the input vertices (which become the leaf nodes in our bounding sphere hierarchy) such that no holes are left during rendering. In order to guarantee this, the sizes must be chosen such that if two vertices are connected by an edge of the original mesh, the spheres placed at those vertices are large enough to touch. Our current algorithm makes the size of the sphere at a vertex equal to the maximum size of the bounding spheres of all triangles that touch that vertex. This is a conservative method – it may result in spheres that are too large, but is guaranteed not to leave any holes.

Once we have assigned leaf sphere sizes, we use the following algorithm to build up the rest of the tree:

```
BuildTree(vertices[begin..end])
{
   if (begin == end)
      return Sphere(vertices[begin])
   else
      midpoint = PartitionAlongLongestAxis(vertices[begin..end])
      leftsubtree = BuildTree(vertices[begin..midpoint])
      rightsubtree = BuildTree(vertices[midpoint+1..end])
      return BoundingSphere(leftsubtree, rightsubtree)
}
```

The algorithm builds up the tree by splitting the set of vertices along the longest axis of its bounding box, recursively computing the two subtrees, and finding the bounding sphere of the

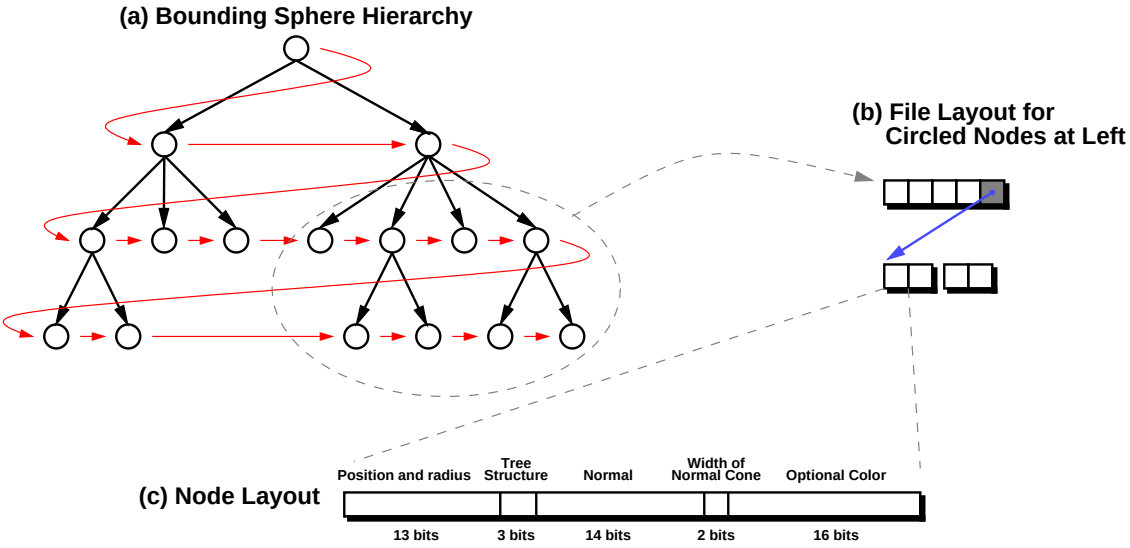


Figure 5.2: QSplat file and node layout. (a) The tree is stored in breadth-first order (i.e., the order given by the red arrows). (b) The link from parent to child nodes is established by a single pointer from a group of parents to the first child. The pointer is not present if all of the "parent" siblings are leaf nodes. All pointers are 32 bits. (c) A single quantized node occupies 48 bits (32 without color).

two children spheres. As the tree is built up, per-vertex properties (such as normal and color) at interior nodes are set to the average of these properties in the subtrees. When the recursion reaches a single vertex, we simply create a sphere whose center is the position of the vertex. Because the total size of a tree depends on the branching factor at each node, we combine nodes in the tree to increase the average branching factor to approximately 4. This reduces the number of interior nodes, thereby reducing the storage requirements for the tree. The final step of preprocessing is quantizing all of the properties at each node, as described in Section 5.3.1.

5.3 Design Decisions and Tradeoffs

Let us now consider some of the decisions made in the implementation of QSplat that make it suitable for our application of visualizing large scanned data sets. We describe how tradeoffs in quantization, file layout, splat shape, and the choice of splatting were affected by our goals of fast rendering and compact representation.

5.3.1 Node Layout and Quantization

The layout of each node in the bounding sphere hierarchy is shown in Figure 5.2c. A node contains the location and size of a sphere relative to its parent, a normal, the width of a cone of normals, an optional color, and a few bits used in representing the structure of the tree. We discuss the structure of the tree and the layout of nodes within the file in Section 5.3.2.

Position and radius: The position and radius of each sphere is encoded relative to its parent in the bounding sphere hierarchy. In order to save space, these quantities are quantized to 13 values. That is, the radius of a sphere can range from $^1/_{13}$ to $^{13}/_{13}$ of the radius of its parent, and the offset of the center of a sphere relative to the center of its parent (in each of X, Y, and Z) is some multiple of $^1/_{13}$ of the diameter of the parent sphere. The quantization proceeds top-down, so the position and size of a child sphere is encoded relative to the quantized position of its parent; thus, quantization error does not propagate down the mesh. In order to guarantee that the quantization process does not introduce any holes, the quantized radius is always rounded up to the nearest representable value that ensures that the quantized sphere completely encloses the true sphere.

Note that not all of the 13^4 possible combinations of (x, y, z) center offset and radius ratio are valid, since many result in child spheres that are not enclosed by their parents. In fact, only 7621 of the possible combinations are valid, which means that we can encode the quantized position and radius using only 13 bits (using a lookup table). For a parent sphere of radius 1, this encoding scheme gives a mean quantization error of 0.04 in the x, y, and z components of a child sphere, and a mean error of 0.15 in the child sphere's radius. The error in the radius is larger than the error in position because the radius is first increased by the quantization error in the position of the sphere (to ensure that the quantized sphere encloses the true sphere), and is then always rounded up to the next representable value. We could obtain lower quantization error in the radius by not insisting that the quantized sphere completely enclose the original. Doing so, however, would introduce the possibility that spheres that should touch no longer do so after the quantization. This could produce holes in our renderings.

The idea of representing geometric quantities such as sphere positions by encoding them incrementally, thereby essentially spreading out the bits of the quantities among the levels in the hierarchy, represents a departure from traditional approaches to mesh compression, which rely on encoding the differences between vertex positions along some path along the edges of the mesh [Taubin 98]. This "hierarchical delta coding" is, in fact, closer to the wavelet repre-

sentation of geometry used in the multiresolution analysis of Eck et al. [Eck 95]. Our space requirement of 13 bits per node appears competitive with state-of-the-art geometric compression methods, which average 9-15 bits per vertex depending on initial quantization of vertex positions. This is not an entirely valid comparison, however, since traditional geometric compression methods also represent mesh connectivity (which we discard), and since our 13 bits per node also includes sphere radius.

The position and radius of each node are decoded on-the-fly during rendering. Because of this, our data structure is not only compact on disk, but also requires less memory during rendering than methods that must decompress their data before rendering.

Normals: The normal at each node is stored quantized to 14 bits. The representable normals correspond to points on a 52×52 grid on each of the 6 faces of a cube, warped to sample normal space more uniformly. A lookup table is used during rendering do decode the representable normals. In practice the use of only $52 \cdot 52 \cdot 6 = 16224$ different normals (leading to a mean quantization error of approximately 0.01 radian) produces no visible artifacts in the diffuse shading component, but some banding artifacts are visible around specular highlights in broad areas of low curvature. It would be possible to eliminate these artifacts, as well as achieve better compression, by moving to an incremental encoding of each normal relative to the normal of the parent sphere. This would, however, increase the computational complexity of the inner loop of the algorithm, resulting in a time-space tradeoff. Unlike the range of node positions, the space of normals is bounded, so a fixed quantization table suffices for encoding the normals of arbitrary scenes. Therefore, at this time we have chosen to use a fixed quantization for the normals, which requires only a single table lookup at run time. As processor speed increases, we anticipate that the incremental quantization scheme will become more attractive.

Colors: Colors are currently stored quantized 5-6-5 to 16 bits. As in the case of normals, an incremental encoding of colors would save space but be more expensive at run time.

Normal cones: After some experimentation, we have decided to quantize the width of the cone of normals at each node to just 2 bits. The four representable values correspond to cones whose half-angles have sines of $1/16$, $4/16$, $9/16$, and $16/16$. On typical data sets, backface culling with these quantized normal cones discards over 90 percent of nodes that would be culled using exact normal cone widths. Note that we are always conservative in representing normal cone widths, so we never discard geometry that should be displayed. As with normals and

colors, the normal cone widths could be represented relative to the widths at the parent nodes, but this would slow down rendering.

5.3.2 File Layout and Pointers

The nodes of the bounding sphere hierarchy are laid out (both in memory and on disk) in breadth-first order. A primary consequence of this is that the first part of the file contains the entire mesh at low resolution. Thus, we only need to read in the first part of a file in order to visualize the model at low resolution; we see greater detail as more of the file is read in from disk. We currently use OS-provided memory mapping as the basis for working-set management, so high-resolution data for a given section of the model is read in from disk when the user looks at it. This progressive loading is important for usability with large models, for which the time to load the entire data set from disk may be several minutes. Because data is loaded as it is needed, rendering performance will be lower the first time the user zooms in on some area of the model – due to our feedback-based approach to frame rate control, there is a glitch in the frame rate. Subsequent frames that touch the same area of the model, however, are rendered at full speed. Speculative prefetching has been explored as a method for reducing this performance variation [Funkhouser 92, Funkhouser 96, Aliaga 99], but we currently do not implement this.

Several pointerless schemes have been proposed for tree encoding, including linear octrees and methods based on complete trees [Samet 90]. These data structures, however, are inappropriate for our application. Linear octrees and related ideas require the entire tree to be traversed to recover its structure, which is impractical in our system. Data structures based on complete trees can be used for partial traversals, but because the algorithm we use to generate our trees is based on axis-aligned bisections, we can not guarantee that the resulting trees will be complete and balanced. Furthermore, modifying the preprocessing algorithm to generate complete trees would not be desirable, since putting an equal number of vertices in each subtree can potentially put the splitting planes significantly off-center. Given the amount of quantization we perform on child sphere centers, this could lead to significant inaccuracies in the compressed tree.

Although we can not use pointerless encodings for our trees, we should at least attempt to minimize the number of pointers required. Given that we store the tree in breadth-first order, it is sufficient to have one pointer for each group of siblings in the tree (i.e. children of a single parent sphere), that points to the children of these nodes. Furthermore, that pointer is not

necessary if none of these spheres have children (i.e. they are all leaf nodes). Using this scheme, approximately 8 to 10 percent of the total storage cost is devoted to pointers, which we judged to be sufficiently small that we did not pursue more complicated schemes for reducing pointer costs further. In order to be able to traverse the tree, we store at each node two bits encoding the number of children of the node (0, 2, 3, or 4 children – nodes with a single child are not permitted), and one bit indicating whether all children of this node are leaf nodes.

The total storage requirements for a tree may now be computed as the number of nodes in the tree multiplied by the cost per node, plus the overhead due to pointers. For a tree with average branching factor 3.5, the total number of nodes will be 1.4 times the number of leaf nodes, making the net storage requirements for the entire tree approximately 9 bytes times the number of leaf nodes, or 6 bytes if colors are not stored.

5.3.3 Splat Shape

The choice of kernel used to represent a rendered point sample can have a significant effect on the quality of the final image. The simplest, fastest option is a non-antialiased OpenGL point, which is rendered as a square. A second choice is an opaque circle, which may be rendered as a group of small triangles or, less expensively in most OpenGL implementations, as a single texture-mapped polygon. Another possibility is a fuzzy spot, with an alpha that falls off radially with a Gaussian or some approximation. The particular approximation we use is a spline in opacity that falls to $^1/_2$ at the nominal radius of the splat. These last two options will be slower to draw, since they require sending more data to the graphics pipeline. In addition, drawing a Gaussian splat requires special care regarding the order in which the splats are drawn, because of the interaction between blending and Z-buffering. Levoy and Whitted discuss this problem in the context of a software-only renderer [Levoy 85]; they propose an approach based on buckets to ensure that both occlusion and blending happen correctly. In OpenGL we can use multipass rendering to implement an approximation to the correct behavior. For the first pass, depth is offset away from the viewer by some amount z_0, and we render only into the depth buffer. For the second pass we turn off depth offset and render additively into the color buffer, with depth comparison but not depth update enabled. This has the effect of blending together all splats within a depth range z_0 of the surface, while maintaining correct occlusion. Figure 5.3 compares these three choices of splat kernel. Because per-splat drawing time on current hardware is different for each kernel, we present comparisons at both constant splat size and constant running time.

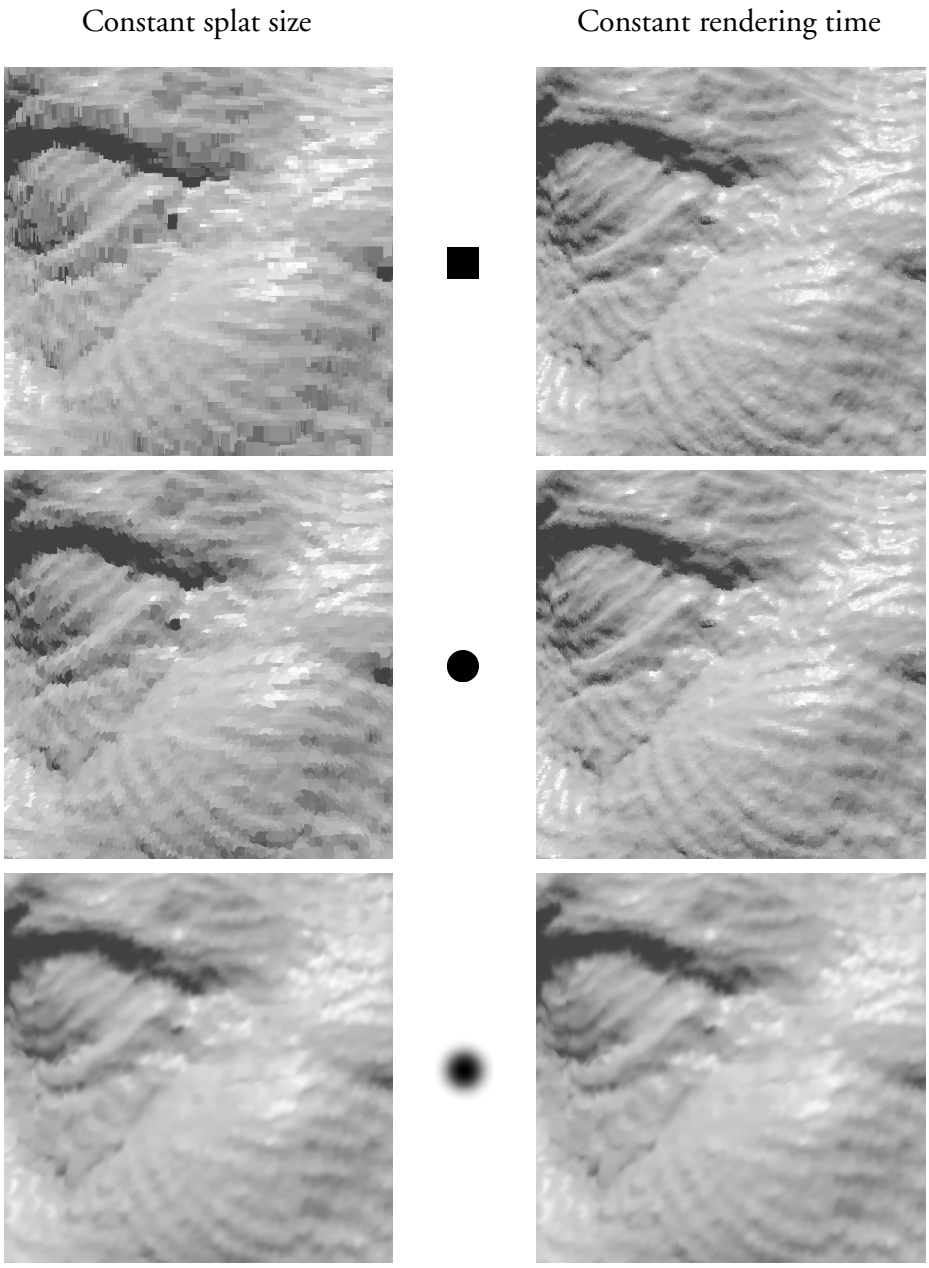


Figure 5.3: Choices for splat shape. We show a scene rendered using squares, circles, and Gaussians as splat kernels. At left, each image uses the same recursion threshold of 20 pixels. Relative to squares, circles take roughly twice as long to render, and Gaussians take approximately four times as long. The Gaussians, however, exhibit significantly less aliasing. At right, the splat size threshold for each image is adjusted to produce the same rendering time in each case. According to this criterion, the square kernels appear to offer the highest quality.

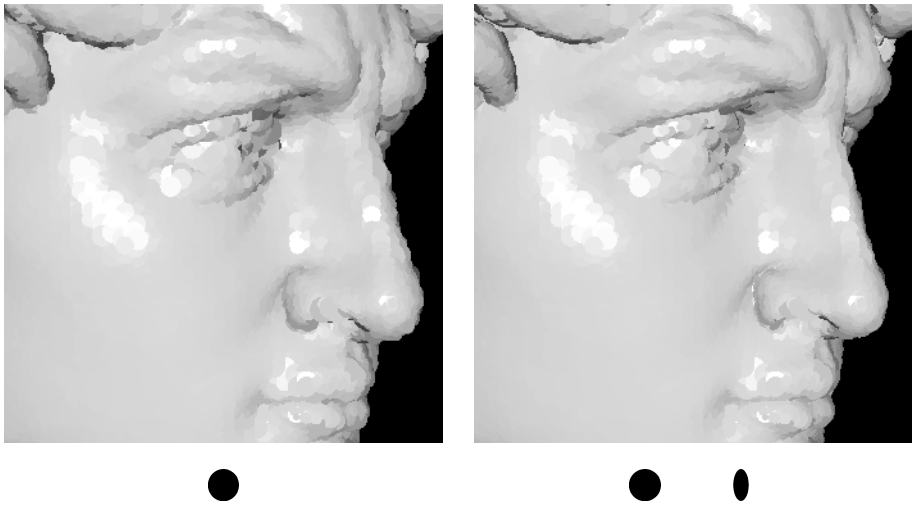


Figure 5.4: Circular vs. elliptical splats. In the left image, all splats are circular with diameter 20 pixels. In the right image, we draw elliptical splats rotated and foreshortened depending on per-node normals. This reduces thickening and noise around silhouette edges. Recursion depth has deliberately been limited to make the splats large enough to see in this visualization.

Another option we have in choosing splat shape is the choice of whether the splats are always round (or square in the case of OpenGL points) or elliptical. In the latter case, the normal at each node is used to determine the eccentricity and orientation of the ellipse. When the normals point towards the viewer, the splats will be circular. Otherwise, the minor axis of each ellipse will point along the projection of the normal onto the viewing plane, and the ratio of minor to major axes will equal $\hat{\mathbf{n}} \cdot \hat{\mathbf{v}}$, where $\mathbf{n}$ is the normal of the splat and $\mathbf{v}$ is a vector pointing towards the viewer. This improves the quality of silhouette edges compared to circular splats, reducing noise and thickening. We compare the use of circular and elliptical splats in Figure 5.4.

Because we construct our bounding sphere hierarchy such that spheres placed along a continuous surface will never leave holes, we can guarantee that the square and circular kernels will always result in hole-free reconstructions. Our approximation to a Gaussian kernel is also guaranteed to produce full opacity in areas that started out as continuous surfaces, because of the same property. Since the alpha of each Gaussian splat falls to $1/2$ at the nominal radius, the sum of the alpha channels of two adjacent splats is guaranteed to be at least 1.

When we move to elliptical kernels, we can no longer guarantee hole-free reconstructions because normals need not be continuous along the surface. In practice, we do occasionally see holes when using elliptical kernels, especially around silhouette edges. We have found that restricting the maximum foreshortening of ellipses (e.g. clamping the maximum ratio of major to minor axis to 10) fills in practically all of these holes.

5.3.4 Consequences of a Point-Based System

The fact that QSplat uses points as its rendering primitives makes it most suitable for certain kinds of scenes. In particular, point rendering systems are most effective for objects with uniformly-sized geometric detail, and in applications where it is not necessary to look at the model at significantly higher resolution than the spacing between samples. If the model has large, flat or subtly curved surfaces, polygonal models can be more compact and faster to draw. Similarly, if it is necessary to zoom in such that the spacing of samples is large compared to pixel size, polygons offer higher visual quality, especially near sharp edges and corners. Figure 5.5 shows a comparison between point- and polygon-based renderings.

QSplat was developed with the intent of visualizing scanned models that contained significant amounts of fine detail at scales near the scanning resolution. We used the Volumetric Range Image Processing (VRIP) system [Curless 96] to merge raw scans into our final models, and the marching cubes algorithm [Cline 88] to extract a polygonal mesh. Since the latter produces samples with a uniform spacing, point rendering was well-suited for our application domain. For scenes with large, smooth regions, we expect that QSplat would be less effective relative to polygon-based systems. The visual quality of the resulting models would still be good, however, if the large polygons were diced, as in the REYES architecture [Cook 87]. For applications containing both high-frequency detail and large flat regions, hybrid point/polygon schemes might be appropriate.

5.4 Performance

As described in Section 5.3, the goal of interactivity dictated many design decisions for our system. In addition to these, we have optimized our implementation in several ways in order to increase the size of the models we can visualize.

(a)
Points

(b)
Polygons – same number of primitives as (a)
Same rendering time as (a)

(c)
Polygons – same number of vertices as (a)
Twice the rendering time of (a)

Figure 5.5: Comparison of renderings using point and polygon primitives.

5.4.1 Rendering Performance

The majority of rendering time in our system is spent in an inner loop that traverses the hierarchy, computes the position and radius of each node, performs visibility culling, and decides whether to draw a point or recurse further. This inner loop was tuned to eliminate expensive operations, especially at lower levels of the tree. For example, we do not perform an exact perspective divide at the low levels of the tree, switching to an approximation when the screen-space size of a node reaches a few pixels. As a result, on average our algorithm can render between 1.5 and 2.5 million points per second on an SGI Onyx2 once data has been read in from disk. The exact rate varies depending on caching effects (for example, we observe a speedup when the working set fits in L2 cache) and how much data is culled at which levels in the tree.

Our display rate may be compared to the 480 thousand polygons per second (on identical hardware) reported by Hoppe for his implementation of progressive meshes [Hoppe 98] or the 180 thousand polygons per second for the ROAM system [Duchaineau 97]. For our application, we typically use frame rates of 5-10 Hz, meaning that we draw 200 to 300 thousand points per frame during interactive rendering. Note that unlike the above two systems, QSplat makes no explicit use of frame-to-frame coherence, such as cached lists of primitives likely to be visible. QSplat's rendering performance is summarized in Table 5.1.

The simplicity of our algorithm makes it well suited for implementation on low-end machines. As an extreme example, we have implemented QSplat on a laptop computer with no 3D graphics hardware (366 MHz Intel Pentium II processor, 128 MB memory). Because rendering is performed in software, the system is fill limited. For a typical window size of 500x500 and frame rate of 5 Hz, the implementation can traverse 250 to 400 thousand points per second, has a 40 million pixel per second fill rate, and typically draws 50 to 70 thousand splats per frame. At this resolution the implementation is still comfortably usable. Although most present desktop systems do have 3D graphics hardware, the same is not true for portable and handheld systems, and in applications such as digital television set-top boxes. We believe that QSplat might be well-suited for such environments.

5.4.2 Preprocessing Performance

Although preprocessing time is not as important as rendering time, it is still significant for practical visualization of very large meshes. Hoppe reports 10 hours as the preprocessing time

David's head, 1mm

David, 2mm

St. Matthew, 0.25mm

Typical performance	Interactive	Static	Interactive	Static	Interactive	Static
Traverse tree	22 ms	448 ms	30 ms	392 ms	27 ms	951 ms
Compute position and size	19 ms	126 ms	30 ms	307 ms	31 ms	879 ms
Frustum culling	1 ms	4 ms	1 ms	3 ms	1 ms	3 ms
Backface culling	1 ms	22 ms	2 ms	25 ms	1 ms	35 ms
Draw splats	77 ms	364 ms	46 ms	324 ms	50 ms	1281 ms
Total rendering time	**120 ms**	**838 ms**	**109 ms**	**1051 ms**	**110 ms**	**3149 ms**
Points rendered	**125,183**	**931,093**	**267,542**	**2,026,496**	**263,915**	**8,110,665**

Preprocessing statistics			
Input points (= leaf nodes)	2,000,651	4,251,890	127,072,827
Interior nodes	974,114	2,068,752	50,285,122
Bytes per node	6	4	4
Space taken by pointers	1.3 MB	2.7 MB	84 MB
Total file size	**18 MB**	**27 MB**	**761 MB**
Preprocessing time	**0.7 min**	**1.4 min**	**59 min**

Table 5.1: Typical QSplat rendering and preprocessing statistics for three models. The columns marked "interactive" indicate typical performance when the user is manipulating the model. The columns labeled "static" are typical of performance when the user has stopped moving the mouse and the scene has refined to its highest-quality version. Variation of up to 30% has been observed in these timings, depending on details such as cache performance. All times were measured on an SGI Onyx2 with InfiniteReality graphics; rendering was done at 1280x1024 resolution.

for a progressive mesh of 200 thousand vertices [Hoppe 97]. Luebke and Erikson report 121 seconds as the preprocessing time for 281 thousand vertices for their implementation of hierarchical dynamic simplification [Luebke 97]. In contrast, our preprocessing time for 200 thousand vertices is under 5 seconds (on the same hardware). Table 5.1 presents some statistics about the preprocessing time and space requirements of the models used in this chapter's figures.

Another class of algorithms with which we can compare our preprocessing time is algorithms for mesh simplification and decimation. Although these algorithms have different goals than QSplat, they are also commonly used for generating multiresolution representations or simplifying meshes for display. Lindstrom and Turk have published a comparison of several recent mesh simplification methods [Lindstrom 98]. They report times of between 30 seconds and 45 minutes for simplification of a bunny mesh with 35000 vertices. One method that paper did not consider was the voxel-based simplification of Rossignac and Borrel [Rossignac 93], which takes under one second on identical hardware to that used by Lindstrom and Turk. Our preprocessing time for this mesh is 0.6 seconds. Thus, our algorithm is significantly faster than most of the contemporary mesh decimation algorithms, and competitive with Rossignac and Borrel's method.

5.5 Summary

The QSplat system has shown that a single, simple bounding sphere data structure can be used for representing and rendering large models at interactive rates. QSplat takes advantage of the simplicity resulting from not maintaining mesh connectivity in order to perform fast display with view-dependent level-of-detail, to achieve compression ratios close to those of current geometric compression techniques, and to require preprocessing times comparable to the fastest presently-available mesh decimators.

"... steal such gentle shape"

– Shakespeare, *Richard III, Act II, Scene II*

Chapter 6

Streaming QSplat

Although the QSplat system, presented in the previous chapter, makes it possible to interactively navigate large models at interactive rates, it assumes that the entire model is present on a local disk. Recent growth in the speeds of network links and consumer uses of the World Wide Web, however, have brought increased interest in streaming transmission of three-dimensional data sets. In this chapter, we show how QSplat may be extended to progressive download of models over a network of limited bandwidth, thus enabling practical visualization of 3D data sets containing hundreds of millions of samples.

We demonstrate how to incorporate view-dependent progressive transmission into QSplat, by having the client request visible portions of the model in order from coarse to fine resolution. In addition, we investigate interaction techniques for improving the effectiveness of streaming data visualization. In particular, we explore color-coding streamed data by resolution, examine the order in which data should be transmitted in order to minimize visual distraction, and propose tools for giving the user fine control over download order.

6.1 Introduction

In the past, interactive 3D content has not had a large presence on the World Wide Web. Despite the availability of standards such as VRML, 3D models have been constrained to specialized niches because of long download times and poor interactive performance. Today,

however, the availability of low-cost, high-performance graphics cards and the introduction of high-speed residential Internet connectivity are making it practical to include 3D models as important components of web sites.

Given the presently available network bandwidths, however, it would not be feasible to use large 3D models if those models had to be downloaded entirely before they could be viewed. As we have seen, however, the size of currently attainable models is increasing rapidly because of the availability of devices and algorithms for scanning large objects at high resolution. For these large meshes, the only practical way of allowing remote download and visualization is to stream the data as it is needed, and to permit the viewer to look at and interact with partially-downloaded models.

In this chapter we introduce a streaming version of QSplat, allowing large models to be progressively streamed across a network of limited bandwidth. The system retains the advantages of QSplat, such as low preprocessing costs and high rendering performance, but adds view-dependent network streaming of geometry. The extension to streaming is based on the fact that we can terminate the recursion of our data structure at any time during rendering if we find that portions of the hierarchy are not yet present on the client; a low-resolution model is rendered, and the missing nodes are requested from the server. Thus, portions of the model are downloaded as the user looks at them.

Though progressive transmission of 3D data has been explored before, most of the effort has focused on either high-speed streaming from a local disk or low-speed streaming across a slow connection. In the former case, the available bandwidth is often adequate to mask the presence of streaming, and research has concentrated on techniques such as prefetching that attempt to hide the fact that data is being read progressively at all. With low-speed links, attention has mostly focused on achieving good approximations to the final model while transmitting as little data as possible, regardless of the required CPU time.

In contrast to the high- and low-bandwidth extremes, comparatively little effort has been devoted to the user interaction issues that become relevant at intermediate speeds (e.g. a few hundred kbps, which is becoming an increasingly common rate for residential Internet connectivity). These speeds are high enough that it is often not worthwhile to implement expensive compression and optimization techniques, but are sufficiently low that there is little hope of concealing the presence of streaming for large models. Thus, we accept that the streaming process will be visible to the user, and focus on designing a user interface that lets the user

know how much data is present, minimizes the visual distraction due to streaming, and gives the user fine control over the streaming process.

We first examine some previous systems that have been used for 3D streaming and large data visualization. Next, we describe the extensions to QSplat that must be made to support 3D streaming. Finally, in Section 6.4 we discuss interaction issues that influence the design of the streaming QSplat user interface, focusing on how to aid the user in interpreting the data and understanding and controlling the streaming process.

6.2 Previous Work on 3D Streaming

Several schemes have been proposed for transmitting 3D data across a network. The simplest ones rely on transmitting a full polygonal model (either directly [VRML 97] or in a compressed format [Taubin 98]), and therefore require the entire model to be transmitted before the user can look at it.

More sophisticated systems transmit low-resolution data first, so the user can begin to interact with the model, then progressively stream higher-resolution data, time permitting [Guéziec 99]. The progressive mesh framework [Hoppe 96] represents a mesh as a simple "base mesh" plus a series of refinements to the mesh based on a vertex split primitive. Progressive meshes, therefore, are well-suited to streaming [Prince 00], especially with the addition of compression [Pajarola 00].

Corrections to a base mesh may also be encoded using wavelets, as was first proposed in multiresolution analysis [Eck 95]. The model may then be transmitted by sending the base mesh and streaming the wavelet coefficients in order of magnitude [Khodakovsky 00]. One advantage of this approach, explored by Certain et al., is that color and geometry wavelets may be streamed independently [Certain 96].

Commercial systems incorporating some of these algorithms are beginning to appear. MetaStream's MTS products, for example, represent geometry as a base mesh together with a series of vertex split operations [Abadjev 99], similar in spirit to progressive meshes. Other products are available that stream polygonal models (e.g. [RealityWave]) or voxelized volumetric data (e.g. [Octree]).

Many systems for architectural walkthrough and terrain flythrough are designed to work with scenes larger than the available memory [Funkhouser 92, Funkhouser 96, Aliaga 99]. In order to achieve high-quality renderings, they explicitly manage the way data is transferred

between memory and disk. These systems typically employ the notion of a potentially-visible set (PVS) of data, comprising both currently-visible data and data that may come into view in the near future, given some assumptions about where the user is likely to move and look next. These systems then perform prefetching to ensure that off-screen data is loaded into memory before the user looks at it. Network streaming of potentially-visible sets for such applications has been explored by Cohen-Or and Zadicario [Cohen-Or 98].

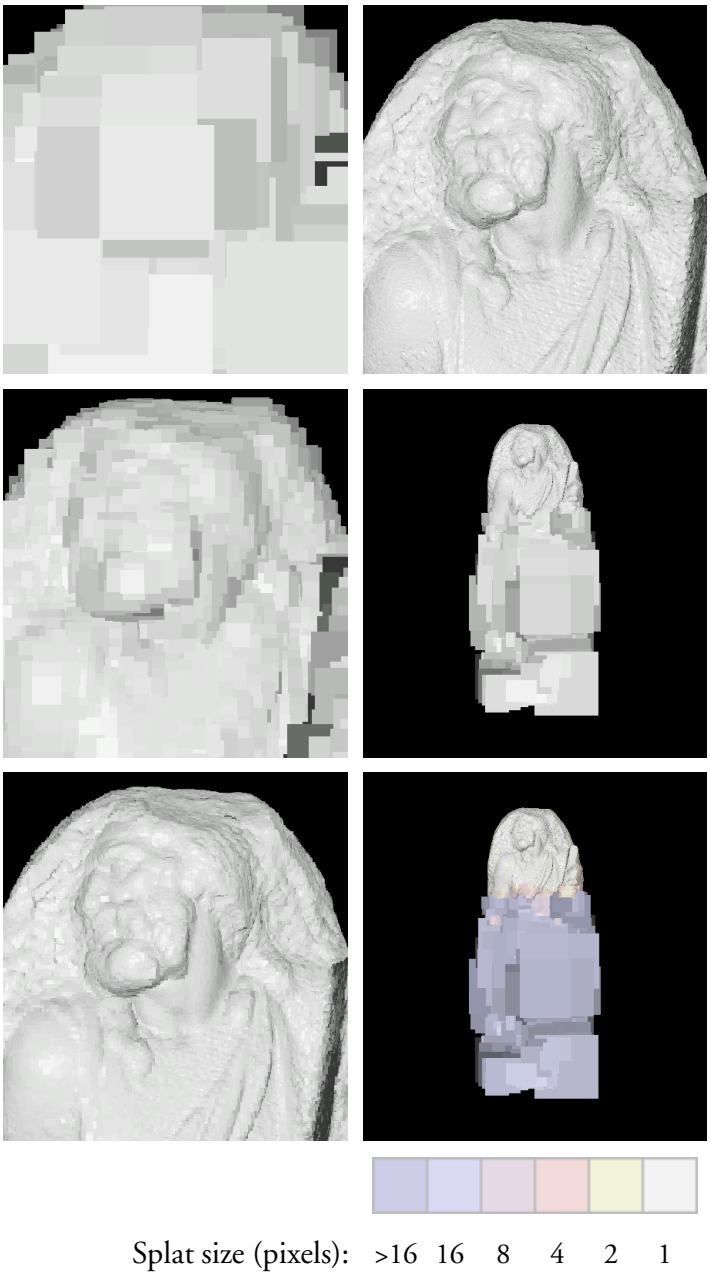
Compared with most of the above systems (both research and commercial), our streaming QSplat implementation has higher rendering performance (both because it uses simpler rendering primitives and because it does not require CPU time to be devoted to decompression), requires less preprocessing time, and uses a standard HTTP server rather than a custom streaming server. As we discuss in Section 6.3.4, this makes QSplat well-suited for streaming large models across networks of moderate bandwidths. Our system, however, is not as bandwidth-efficient as some systems that incorporate more sophisticated geometric compression. In addition, since it uses splats as the rendering primitive, it will have lower visual quality than polygon-based systems for certain kinds of scenes.

6.3 Adding Network Streaming to QSplat

The key to network streaming of QSplat models is the observation that during rendering we can terminate recursive decent of our hierarchical representation at any time. In place of missing geometry, QSplat displays a splat corresponding to the parent node in the hierarchy. In the system of Chapter 5, recursion is terminated under two conditions: if the children of a given node are smaller than a threshold, or if we reach a leaf node. To accommodate streaming, we need to add one additional condition: we stop recursion if the children of a given node have not yet been transmitted from the server to the client. Thus, with low run-time cost we transparently accommodate the presence or absence on the client of various portions of the hierarchy, including the possibility of having different resolutions of data present throughout the model. Note that we still perform the usual feedback-driven frame rate control when the user is dragging the mouse, so although the frame rate may be higher than the user setting if not enough data is present, it will not drop lower than requested. Figure 6.1 illustrates the results of streaming transmission.

To allow for network streaming, therefore, we need three components:

- A bitmask indicating which regions of the model are present on the client.



a) Appearance of the model immediately after the start of streaming.

b) 1 second after (a).

c) 10 seconds after (a).

d) 60 seconds after (a).

e) Appearance of the model after zooming out from (d). High-resolution data has been streamed only in the region on which we were zoomed in.

f) Color-coded visualization of the relative resolution of data present in (e).

Figure 6.1: View-dependent streaming data transmission of a 130 million sample model over a network limited to 384 kbps.

- A prioritized request queue containing a set of regions of the model that the client would like to receive, given the current camera position.
- A separate thread on the client that makes requests to a server, listens for responses, and updates the tree data structure and availability mask as data is received.

Given these, the rendering algorithm is as follows:

```
TraverseHierarchy(node)
{
   if (node not visible)
      skip this branch of the tree
   else if (node is a leaf node)
      draw a splat
   else if (benefit of recursing further is too low)
      draw a splat
   else if (any child is not present)
      draw a splat
      RequestQueue.insert(children(node), priority)
   else
      for each child in children(node)
         TraverseHierarchy(child)
}

DrawFrame(model)
{
   RequestQueue.clear
   TraverseHierarchy(model.root)
   if (not RequestQueue.empty)
      n ← (estimated net bandwidth) / (frame rate)
      for i ← 1 .. n
         SendRequest(RequestQueue.top)
         RequestQueue.pop()
}
```

6.3.1 Availability Mask

In order to perform rendering correctly, we must have a data structure that maintains information about which portions of the model have been received from the server. For maximum

flexibility, we would like to have the mask as fine-grained as possible – ideally, we would store availability information at the granularity of a single node of the tree – so that we can download precisely the areas of the model in which we are interested. For efficiency of download and to minimize the memory spent on the mask, however, we must use a larger granularity. In our system, we represent availability at the granularity of fixed-size (typically 1 kilobyte) blocks. In addition, to simplify the bookkeeping, we increase the storage per chunk to two bits, so that we can represent four states for each block: not present, desired (i.e., present in the request queue), requested from the server, and present.

6.3.2 Request Queue

As we traverse the hierarchy during rendering and encounter chunks that are not present on the client, we push requests for these blocks onto a priority queue (implemented as a max-heap). As we will see in Section 6.4.2, the priority for a node is determined from the projected screen size and position of that node's parent (which triggered the request for the node). The priority of a chunk is the highest priority of all nodes within that chunk.

The request queue is cleared before every rendered frame. This ensures that:

- The request queue never gets too large, since its size will be proportional to the number of rendered nodes, rather than the total number of nodes in the model.
- A chunk that moves out of the field of view will be dropped from the request queue, preventing the system from wasting time on downloading sections of the model that are no longer relevant to the user's viewpoint.

If the request queue is ever empty after rendering a frame, meaning that all currently-visible data was already present on the client, we first download data in the vicinity of the viewpoint, then revert to downloading any remaining parts of the model in order from the root of the tree to the leaves.

6.3.3 Network Communication

The streaming QSplat client uses a separate thread to make requests from the server and listen for responses. The number of requests to make per frame is based on an estimate of the network bandwidth, so that there are never too many outstanding requests. The data requested from the server consists of ranges of the original file; thus, the server need not have any special

knowledge of the QSplat file format. In our implementation of the streaming QSplat client we have chosen to use the HTTP/1.1 protocol (including the byte-range and persistent connection features [Fielding 97]) to issue requests, so we may stream models from any standard web server (e.g. Apache); a separate streaming server is not required.

6.3.4 Discussion

Let us now examine some of the advantages and disadvantages involved in using QSplat, as compared to traditional polygonal representations, as the basis of a network streaming system.

Suitability of QSplat for Network Streaming: Streaming QSplat retains most of the advantages and disadvantages of QSplat in its suitability for representing various classes of geometric models. In particular, streaming QSplat will work best for large, dense models containing relatively regular, uniformly-spaced data points (e.g. as produced by VRIP [Curless 96] and marching cubes [Cline 88]) and high geometric detail at fine scales. In contrast, a QSplat representation of a model with large flat regions, subtle curves, or sharp corners will not look as good as a polygonal or spline model of equal size. Moreover, a low-resolution version of any model, when rendered with splats, will contain visible artifacts of the splat shape.

A second property of QSplat that becomes useful for streaming is the fact that parts of a model may be transmitted in any order, subject only to the constraint that parent nodes must be transmitted before children. This makes it easy to incorporate various strategies for choosing the order in which parts of the model are transmitted. By contrast, some geometric compression techniques require that the model be transmitted in a particular order, since they represent vertex positions and connectivity by encoding deltas along a particular path through the vertices of the model.

Compressed Data Size: One difference between QSplat and most other geometric compression techniques is that QSplat uses the same data representation on disk and in memory, thus not requiring extra time or space for decompression. In designing streaming QSplat, we have chosen to use this same data representation for network transmission as well. By eliminating the need to encode and decode a compressed format, we simplify the requirements for the network server, and we minimize run-time overhead in the client when using moderate- or high-speed links.

The tradeoff is that QSplat may not be as bandwidth-efficient as algorithms that incorporate more sophisticated geometric compression. As an example, QSplat requires per-vertex normals to be stored and transmitted explicitly. Although QSplat's representation of normals is

reasonably efficient (14 bits per node), normals could instead be computed by the client from transmitted polygon geometry, thereby saving network bandwidth. (QSplat could not use this approach, since it does not use polygons.)

6.4 Interaction Techniques for 3D Streaming

As mentioned earlier, we have chosen to focus on the user interaction techniques that become relevant to streaming at moderate network bandwidths (e.g. a few hundred kbps), rather than on the low- or high-bandwidth extremes. Our motivation for this is the observation that, after remaining static for many years, typical network speeds appear to be rising, especially in residential settings. These speeds, however, are still not sufficiently high that streaming becomes invisible. Therefore, since the user will be able to observe the streaming, we explore color coding to communicate the relative resolution of data present at various points. In addition, we investigate several options for the order in which to stream data, including a user-controlled "magnifying glass" tool that directly controls download order. Finally, we examine the role of prefetching at these speeds.

6.4.1 Color-Coding by Resolution

In a view-dependent streaming system such as ours, the model may, because of previous camera movements, have different sections available at different resolutions. Similar situations arise in other multiresolution rendering systems, such as the hierarchical splatting of Laur and Hanrahan [Laur 91]. When looking at such models, it is possible to mistake low-resolution splats for plausible object geometry. Thus, users need visual cues that allow them to distinguish a transition between areas of different resolutions from an actual feature of the object. To accomplish this, streaming QSplat provides an optional user-selected color coding of the downloaded data, so that areas of different resolutions appear in different colors. This provides visual feedback for the user about the resolution at which various areas of the model are being rendered, and which areas are still being downloaded. The color coding is used in Figures 6.1, 6.2, and 6.3.

6.4.2 Streaming Order

When the camera is positioned to look at some portion of the model that has not been seen before, we must choose the order in which to stream the nodes within the view frustum. This

reduces to defining a priority function for a given node, since the position of nodes within the request queue determines the order in which they will be downloaded. There are several possibilities for this ordering:

1. We may base the priority function on the level of a node within the bounding sphere hierarchy. This will have the effect of downloading (a portion of) the tree in order from root to leaves, such that all nodes at any given level of the tree will be downloaded before we start on the next level within the tree. This has the benefit of being simple to compute, but has the drawback that it may assign the same weight to differently-sized pieces of the model. As a result, nodes downloaded at the same time may have different sizes.

2. We may prioritize nodes by size (i.e. sphere radius) in object space. This is also simple to compute, but has a drawback similar to option 1 because it may assign the same weight to equally-sized pieces of the model regardless of their distance to the viewer. Thus, far-away nodes occupying a relatively small area on the screen may be assigned the same priority as close-by nodes that appear larger on the screen.

3. To remedy the above problem, we may assign priorities based on a node's projected screen size. With this priority function, nodes that appear the same size for a given camera position will be downloaded at roughly the same time. This exposes a second problem, however: the (x, y) screen location at which data is being streamed will be constantly varying in a seemingly-random fashion. This proves to be somewhat distracting for the user, since data appears to be changing at unpredictable locations on the screen.

4. A potential fix for the above problem is to stream based on the screen-space y coordinate. This refines the model in a single pass from the top of the screen to the bottom, which appears more ordered and thus less objectionable for the user. This approach, however, has the drawback that the single pass over the screen is slow, since the user must wait for full-resolution data to be downloaded at each y location.

5. The advantages of approaches 3 and 4 can be combined by prioritizing the nodes such that we perform a number of top-to-bottom sweeps over the data. Each of these passes has its own screen-space cutoff for node size, and we download only the nodes larger

than this cutoff. A similar strategy has been used for progressive download of images, e.g. progressive GIFs. A priority function that implements this behavior is

$$Priority(n) = \lceil \log_k Splatsize(n.radius) \rceil \cdot 1000 + Project(n.center).y$$

This bases the priority on a (logarithmically) quantized version of the node's screen size, with a secondary ordering based on the screen y coordinate. The base of the logarithm, k, determines how much data is downloaded per pass. We have experimentally determined that using $k = \sqrt{2}$ produces acceptable results, roughly doubling the number of downloaded nodes on each pass.

We have chosen to use the algorithm described in option 5 in our implementation. The effect of using this priority function is demonstrated in Figure 6.2. We believe that it offers a good compromise of downloading the most relevant data as soon as possible while minimizing visual distraction.

Figure 6.2: Streaming within a frame is performed in a series of top-to-bottom sweeps that each download all nodes larger than a certain screen-space tolerance. Here, we show the appearance of the model at three points during refinement. Because the refinement order is based on screen-space size, the splats present within a frame tend to be close to each other in size (as long as the viewpoint is not changed).

6.4.3 Magnifying Glass

For certain model inspection tasks it is desirable to have finer-grained control over download order than the above algorithm provides. For example, in a large, complex model there may be a feature of interest that a user wishes to examine at the highest possible level of detail. Given only the above algorithm, the only way to accomplish this quickly (i.e., without waiting for the entire screen to be refined to the desired resolution) would be to zoom in on the given feature. Sometimes, however, it is desirable to see the feature of interest in the context of the surrounding geometry, for which lower resolutions are often sufficient. Under such circumstances, we can introduce tools that allow the user to boost the priority of certain points on the model or regions of the screen. As an example, we have implemented a "magnifying glass" tool that temporarily increases the priority of a region of the screen (the magnifying glass metaphor in user interfaces has been explored before, e.g. in the work on "Magic Lenses" by Bier et al. [Bier 93]). The magnifying glass may be dragged around to permit the user to focus on any locations on the screen. The effect is illustrated in Figure 6.3. Note that color coding is especially useful in this case to illustrate what sections of the model are present at what resolution.

Figure 6.3: A "magnifying glass" tool is used to provide fine control over download order. As illustrated by the color coding, higher-resolution data has been streamed in the area of the face.

6.4.4 Prefetching

Architectural walkthrough and terrain rendering systems often use prefetching to improve the quality of renderings and to avoid latencies in the availability of high-resolution data when the user moves to new parts of the model. We have implemented a prefetching algorithm for streaming QSplat that places nodes slightly outside the view frustum onto the request queue with a low priority. After some experimentation, however, we have found that using prefetching does not improve the quality of interaction with QSplat to the extent it does with architectural walkthrough and terrain rendering systems. The chief causes of this are:

- In contrast with walkthrough systems, QSplat is best suited to visualizing objects, not environments. Because of this, and because of the trackball interface used by QSplat (as compared to a "flythrough" interface), the camera movements during interaction with QSplat tend to be less predictable than in walkthrough systems. This results in larger potentially-visible sets, so resources devoted to prefetching are spread out over a larger area of the model.

- Systems in which prefetch is most effective stream data from disk, which can be done at a sufficiently high rate that they successfully create the illusion that high-resolution data is always available. In contrast, we assume a network link with significantly lower bandwidth. Coupled with the fact that QSplat draws more primitives per frame than most comparable polygon-based systems, we can not hope to maintain the illusion that highest-resolution data is always available.

- It is difficult to determine an acceptable value for the relative priority to be assigned to on-screen and off-screen data. If not enough weight is given to on-screen data, the refinement rate of the visible portion of the model slows down to an undesirable degree. If the off-screen data is not weighted enough, there is little visible difference compared to not performing any prefetching. This is because the off-screen data is downloaded at a slow rate compared to the speed at which it will be downloaded as soon as it comes into view.

Because of the above factors, it is difficult to find circumstances under which it is clearly useful to perform prefetching in QSplat. In fact, after some experimentation we have decided to abandon prefetching entirely, and only fetch off-screen data once the entire viewport is fully

refined (i.e., to a node size of one pixel), at which point the system is idle and might as well spend its time prefetching.

6.5 Summary

Our investigation of streaming QSplat has shown that, with a few additions, the QSplat data structure and rendering algorithm introduced in the previous chapter may be adapted for view-dependent network streaming. Because no connectivity information need be represented or transmitted, the system is flexible enough to permit experiments with various aspects of user interaction during streaming 3D data transmission. We have shown that large, complex models may be streamed and interactively displayed over network links of moderate bandwidth.

"Begin at the beginning, and go on till you come to the end: then stop."

– Lewis Carroll (Charles Lutwidge Dodgson)

Chapter 7

Conclusions and Future Work

This dissertation has described the design of a new real-time 3D model acquisition system and demonstrated results from a prototype implementation of the scanning, alignment, merging, and rendering pipeline. In contrast with previous systems, our design permits the user to rotate an object (by hand), and see a continuously-updated model as an object is scanned, thus providing instant feedback about the presence of holes and the amount of surface that has been covered. The system uses off-the-shelf components and runs on today's CPUs, thus achieving our goals of making 3D model acquisition easier, faster, and less expensive.

Because this dissertation has focused on the implementation of a complete pipeline, we have only explored one point in the design space of such systems. Thus, we anticipate future work both in exploring the stages of our real-time pipeline and in examining the various ways in which the system as a whole may be used.

7.1 Structured-Light Scanner and Coding

We have presented an analysis of structured-light scanning in terms of reflectance, spatial, and temporal coherence assumptions. Based on this examination, we have derived a particular new set of illumination codes optimized for moving scenes. More generally, however, the notion of conveying unique codes through spatio-temporal neighborhoods could be used to design other schemes that make lesser or greater continuity assumptions. One obvious extension might

be to use the "Y" dimension through which we currently convey no information (since our projected patterns use vertical stripes). By having the codes not be simple stripe patterns but instead vary in the vertical direction, we could potentially code a greater number of distinct codewords using a smaller number of frames. A further extension might involve designing codes for which the number of frames necessary to identify a feature depends on the degree of local spatial continuity. That is, these codes would have the property that in smooth regions it would take only a few frames to identify codewords, while in discontinuous regions it would take more frames to disambiguate them. In certain cases, this property might be achieved by projecting completely random time-varying patterns.

Another direction in which the design of projected patterns could be extended involves adaptive coding. Given the ease of controlling DLP projectors, it should be possible to dynamically adapt the projected pattern to the object being scanned. This might be done:

- to compensate for reflectance, e.g. by projecting brighter stripes in dark regions to obtain a good signal while projecting darker stripes in light regions to avoid saturating the camera;
- to obtain denser data in certain regions of interest;
- to permit projected stripes to have approximately equal widths from the point of view of the camera, despite the presence of foreshortening; or
- to obtain higher-quality data in cases when the projector is of higher resolution than the camera. This would involve performing small shifts of the projected pattern from frame to frame, while still keeping the stripe width in each frame large enough to be distinguishable in the camera images. Since currently-available projectors usually have higher resolutions than available cameras, we expect this scheme to be beneficial in the foreseeable future.

7.2 ICP

We have classified and compared several ICP variants, focusing on the effect each has on convergence speed. We have introduced a new sampling method that helps convergence for scenes with small, sparse features. Finally, we have presented an optimized ICP algorithm that uses a constant-time variant for finding point pairs, resulting in a method that takes only a few tens of milliseconds to align two meshes.

Because the present comparisons have focused largely on the speed of convergence, we anticipate future surveys that focus on the stability and robustness of ICP variants. In addition, a better analysis of the effects of various kinds of noise and distortion would yield further insights into the best alignment algorithms for real-world scanned data. Algorithms that switch between variants, depending on the local error landscape and the probable presence of local minima, might also provide increased robustness.

The normal-space sampling algorithm we have proposed is just one example of adding some amount of selectivity to ICP (i.e., not treating all points equally, but differentiating between them in some way). One could imagine other distinguishing criteria at the point-selection stage (based on, e.g., curvature), as well as the matching, weighting, and rejection stages. In this sense, there exists a relatively unexplored space of algorithms between the two extremes of ICP and feature matching. In many cases in which "plain" ICP fails, these algorithms might provide faster convergence and greater acceptable initial misregistration.

7.3 Model Acquisition System

As mentioned in Section 2.4.1, there are many areas in which the performance of our range scanner could be improved in order to improve the quality of the data returned and the ease of a user's interaction with the scanner. One general area of improvement involves gathering more data, by using multiple cameras or projectors, higher-resolution cameras or projectors, or high-speed cameras and projectors. Depending on the hardware added, this might result in faster allowable object motion, higher resolution, higher-quality data, or better coverage of the object. We have explored one simple multi-projector arrangement – see Figure 7.1.

A second general way of improving interaction with the model acquisition system involves using algorithms that are known today, but are too processor-intensive to be practical. Specifically, because of the CPU limitations of present-day systems, the quality of the merging and rendering is not as high as that achievable by offline systems. The real-time rendering in the current prototype is therefore only suitable as a preview to allow the user to find holes and evaluate coverage. As CPU speeds increase, we anticipate that it will become practical to incorporate higher-quality rendering using partially-transparent or fuzzy (Gaussian) splats and to perform better merging based on algorithms such as VRIP [Curless 96]. One difference between our current grid-based merging algorithm and VRIP is that the former allows for a higher grid resolution than the spacing between samples in each range image. Thus, the resolu-

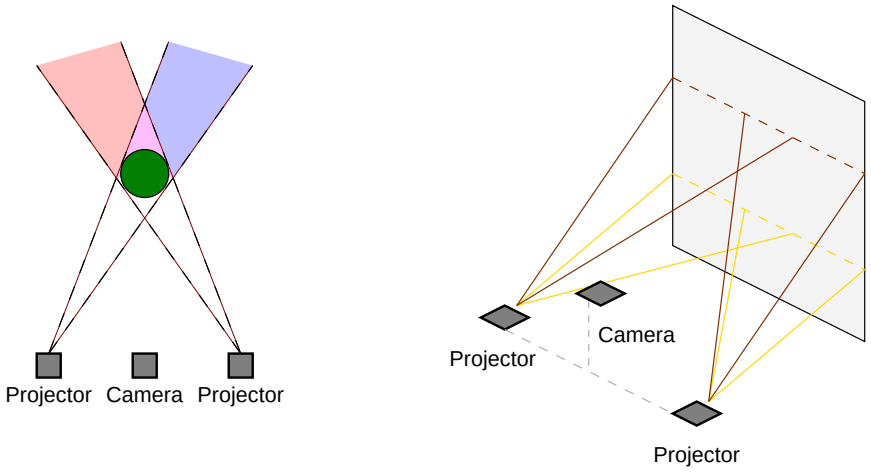


Figure 7.1: Simple multi-projector configuration. Both projectors display the same codes, and they are arranged such that the coded stripes correspond to the same planes in space. As a result, only the central region in purple is in shadow, instead of the regions marked in red and blue.

tion of the merged model is potentially higher than that of the constituent scans. Incorporating the capability for such "super-resolution" into a VRIP-like algorithm is an interesting area for future investigation.

The third general category of improvements to our model acquisition pipeline involves introducing new algorithms. The following sections explore several such possibilities.

7.3.1 Prediction

In order to increase the allowable speed of object motion without using high-speed cameras, our scanner could take advantage of the fact that the motion of the scanner and scene is likely to have a certain amount of continuity. Therefore, it should be possible to predict the effects of constant-velocity motion, and use these predictions at several stages in our pipeline. This would, to a large extent, change the constraints on the maximum speed of motion to instead be constraints on acceleration.

There are three stages in our pipeline at which prediction could be used. First, simple 2-D prediction could be applied to the velocities of individual stripe boundaries in the camera image. Second, the translational and rotational velocity of the whole object could be predicted based on ICP results, leading to better guesses for the starting position for the next ICP. Finally, these two approaches could be combined: the system could extrapolate the rigid-body motion

resulting from ICP, simulate the projector by finding the locations of the projected stripes on the partially-constructed model, and then project these points back into the predicted location of the camera. This would produce predictions for stripe boundary locations in the next camera frame that took advantage of the already-known shape of the object.

Although the first two of these ways of using prediction are likely to be easy to add and produce good results, the third possibility mentioned above involves actually using the partial model. Thus, it is likely to be sensitive to outliers, and would need special ways of handling the as-yet unscanned regions of the object. However, it suggests the more general idea of introducing more feedback between the stages of the pipeline. This will become more practical with increasing CPU speeds, and has the potential of resulting in greater robustness and accuracy. One might imagine a general statistical error estimation framework for the entire pipeline, in which all the data (stripe boundary locations, transforms, and the model itself) were described as probability distributions, and the maximum likelihoods of each could be computed by simultaneously considering all the available data.

7.3.2 Calibration

In describing our scanning system, we have not devoted much attention to the calibration of the intrinsics and extrinsics of the camera and projector. However, there is no reason why our goals of "faster, cheaper, easier to use" should not also be applied to calibration. Therefore, let us look at a few variants on calibration procedures, and how those could be adapted to give good accuracy while being easy to use.

The most direct form of calibrating a scanner is to use a target with known 3D point features, measure the camera (u, v) locations and projector p coordinates corresponding to those features, then solve for the intrinsics and extrinsics of both camera and projector (thus determining the optimal $(u, v, p) \rightarrow (x, y, z)$ mapping). This is the method we have adopted, with 3D point positions found using a Faro arm (a jointed-arm touch probe), and using a linear distortion model. More elaborate distortion models are possible, and Valkenburg and McIvor have explored using them for both the camera and projector, obtaining increases in global accuracy [Valkenburg 98].

In order to improve on the user-friendliness of this method, one possible first step is to not require measurement of absolute 3D point positions. Instead, the user could move move the calibration target freely to several positions, and the calibration algorithm could solve for both the locations of the target and the system parameters using a bundle adjustment method

[Gühring 01]. A second improvement in robustness and usability comes from separating the calibration of intrinsics and extrinsics. This method is used to good effect in the system of [Raskar 99], in which, once the camera is calibrated, the calibration of the projector is performed by placing an arbitrary target in the field of view and projecting known patterns onto it.

Perhaps the ultimate way of simplifying calibration is not requiring it at all. This is the approach taken by self-calibration systems, such as the one described by [Jokinen 99]. An arbitrary object is scanned from many orientations, the multiple scans are aligned, and the calibration of the scanner is iteratively improved by adjusting parameters so as to minimize this misregistration error. Using this method, Jokinen reports high calibration accuracy, but the method depends on having an object with sufficient complexity as well as a good initial guess for the calibration parameters.

By combining several of the above ideas, it should be possible to make the process of recalibrating the scanner fast and user-friendly. As discussed below, one possible application of our system involves digitizing large interior spaces such as rooms or corridors. For such an application, it might be desirable to have the capability of easily changing the working volume by adjusting the baseline or the angle between the camera and projector. By pre-calibrating camera intrinsics, obtaining an initial estimate of extrinsics from a target moved by hand, and performing self-calibration from acquired 3D data, calibrating for such changes to the system would be easy and efficient. The process could be simplified even further by reducing the degrees of freedom of the system, e.g. with a rig that only permitted constrained motion of the projector relative to the camera.

7.3.3 Alignment Drift and Global Registration

Even with good calibration of the range scanner, there will be some amount of scan-to-scan alignment drift due to only aligning each range image to one other. Despite our use of *anchor scans* to attempt to minimize this effect, we still observe accumulated misalignment on the order of several millimeters after scanning completely around an object. By computing additional scan-to-scan ICPs and running a global relaxation algorithm [Pulli 99] during scanning, it should be possible to reduce or eliminate this accumulation of alignment errors during scanning. This, however, is likely to introduce considerable pauses into the scanning, since it will require regenerating the grid data structure from the original scans after a global registration. Thus, a decision would have to be made about whether to attempt to hide this process (by

performing it in the background during scanning), or expose it in the user interface (by only performing global registration when the user explicitly asks for it).

In some applications, the scanning will involve an area large enough that it may be difficult to determine which of the anchor scans should be aligned to each other when attempting global registration. For example, using this pipeline in the context of scanning a building might involve moving the scanner through a cycle of corridors; after the scanner has been moved back to the starting position, enough alignment error might have accumulated that finding candidate anchor scans to attempt to align would be challenging. In this case, our scanning system might be integrated with a separate tracker. Since this tracker would be relied upon for global, not local, accuracy, technologies such as magnetic trackers and GPS might be usable for this purpose. An alternate possibility, especially for the highly-structured application of scanning building interiors, would be to impose *a priori* external constraints (e.g., that walls are perpendicular to each other and to floors) and try to maintain those constraints by performing segmentation and feature extraction on the returned range data.

7.3.4 Applications in Various Contexts

Because our system uses off-the-shelf components and is computationally inexpensive, it permits a variety of potential applications in such fields as tele-immersion or robot guidance. In addition, since moving the scanner is equivalent to moving the scene, making the scanner portable would permit real-time digitization of buildings, rooms, or movie sets. The following are a few possible contexts in which this system could be applied:

- **Small working volume and high accuracy:** This configuration might be useful for applications such as industrial inspection or metrology. Another potential application is medicine, in which such a system might be used to build 3D models of a patient that could be aligned with pre-acquired 3D diagnostic datasets. All of these applications are likely to require a more detailed analysis of the sources of noise in the scanner, better control over error sources such as focus and mechanical drift, and more robust handling of object texture.

- **Large working volume, cart- or shoulder-mounted:** The most obvious application for such a system is digitization of movie sets or building interiors. The issues to be addressed for such a system include the aperture and shutter settings of both the camera and projector (these affect depth of field and sensitivity to ambient illumination), and

the physical layout of the scanner (i.e., baseline and triangulation angle). Another design choice for this application involves the long "tail" of the working volume, as illustrated in Figure 4.3. Depending on the camera and projector field of view and the triangulation angle, it is possible for this tail to extend to infinity (such as in the common passive stereo setup of two parallel cameras). In many cases, having such a tail is undesirable, since the accuracy of data acquired in this region is low. For scanning a room or corridor, however, having even low-resolution data of distant parts of the scene provides valuable constraints on ICP, and helps prevent the accumulation of alignment error. Using such data, however, would require incorporating distance-dependent uncertainty estimates into both the alignment and merging (VRIP) algorithms.

- **Room- or stage-sized (fixed) scanner, moving people:** This configuration (involving just the range scanner, not the full model acquisition pipeline) might provide a markerless motion capture system for human movement, with the capability of capturing not only joint rotations but also surface deformations. In order to be applied to people, such a system would have to use multiple cameras and projectors (to observe the subjects from all angles at once) operating at high frame rates. In addition, the intrusiveness of the blinking illumination pattern would have to be eliminated. As mentioned in Section 2.4.1, this could be done by working in the infrared, or by using time-multiplexed light cancellation [Raskar 98].

A final direction for future research involves solving the *model acquisition* problem for nonrigid objects. Although the first stage of our current pipeline (the 3D scanner) can handle deforming objects, the alignment and merging stages would require considerable changes. There has been recent work on tracking non-rigid objects in the computer vision community [Costeira 98, Bregler 00], though much of it assumes either that an initial model is available or that the deformation is heavily constrained. Nevertheless, this work indicates that model acquisition of deforming objects may be tractable, especially if many features are present in the geometry or texture.

7.4 QSplat

The QSplat system has demonstrated real-time progressive rendering of large scanned models. QSplat's architecture matches the rendering speed of state-of-the-art progressive display algo-

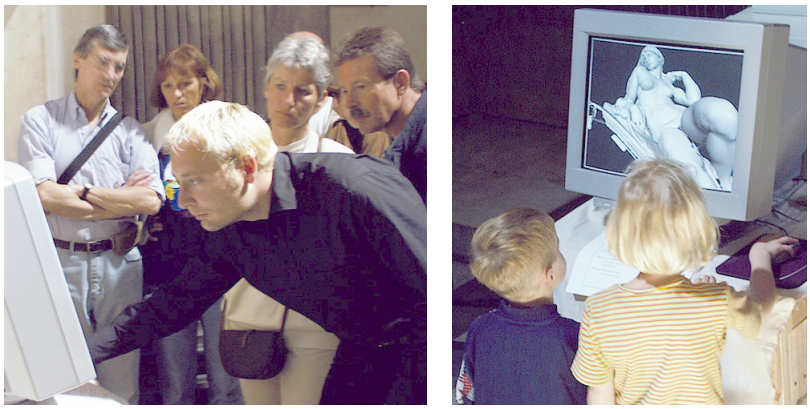


Figure 7.2: Tourists in the Medici Chapel using QSplat to fly around our 3D model of Michelangelo's statue of Dawn. We simplified the interface to only allow rotating, translating, and relighting the model. Nevertheless, some tourists managed to get the viewer into various confusing states, typically by zooming in too far. This underscores the need for a simple, robust, and constrained user interface. We found that most tourists appreciated having a computer model of the statue at which they were looking; having the capability to see the statue from other views, and to change its virtual lighting, made looking at the statue a more active, hands-on experience.

rithms, has preprocessing times comparable to the fastest presently-available mesh decimators, and achieves compression ratios close to those of current geometric compression techniques.

Because the QSplat viewer is lightweight and can be implemented on low-cost hardware, we believe it has the potential for permitting 3D rendering in applications where it was previously impractical, for example built-in kiosks in museums. On May 6, 1999, we set up QSplat on a computer in the Medici Chapel in Florence, displaying our partially-completed computer model of the statue of Dawn, and we let the tourists play (see Figure 7.2).

Several previously-introduced techniques could be incorporated into the present QSplat framework to make it more time and space efficient:

- Huffman coding [Huffman 52] or another lossless compression scheme could be used to make the current representation more compact. This would be useful for offline storage or transmission across low-bandwidth communications links, but would require the model to be decompressed before rendering.

- For cases when rendering speed is more important than compact representation, the algorithm could be sped up by eliminating the compression and incremental encoding

of sphere positions and sizes (as described in Section 5.3.1), and simply storing these quantities as floats. In addition, *normal masks* and *visibility masks*, such as those used by Grossman and Dally, could speed up rendering if there is a significant amount of large-scale occlusion [Zhang 97, Grossman 98]. A further gain in speed could be achieved by parallelizing the rendering algorithm, distributing portions of the tree to different processors. We can already parallelize our preprocessing algorithm by breaking up the mesh into tiles, though we have reported single-processor results in this paper.

- Further analysis is necessary to understand the temporal coherence and caching behavior of QSplat. A large amount of systems research has been done on frame rate control and working set management techniques in terrain rendering and architectural walkthrough systems [Funkhouser 96], and those algorithms would improve the smoothness of user interaction with QSplat.

The following are potential areas of future research for combining the QSplat approach with different kinds of algorithms within computer graphics:

- The bounding sphere hierarchy used by QSplat is well-suited as an acceleration data structure for ray tracing. Potentially, this could be used for high-quality renderings with advanced rendering effects of models stored in the QSplat format (a proof of concept is illustrated in Figure 7.3). In addition, the availability of pre-filtered geometry and normals could help in generating correctly antialiased renderings of detailed geometry.

- Instancing would be easy to incorporate into our tree-based data structure and rendering algorithm, greatly reducing the memory requirements for many classes of procedurally-defined scenes. This could be thought of as a new form of view-dependent sprite, permitting efficient inclusion of geometry at multiple locations within a scene.

- Items other than normals and colors could be stored at each node. Transparency (alpha), BRDFs, and BTDFs would be obvious candidates that would increase the visual complexity representable by QSplat, giving it capabilities similar to those of modern volumetric renderers [Kajiya 89]. More complicated objects such as light fields, view-dependent textures, spatially-varying BRDFs, and layered depth images could potentially also be stored at each node, creating hybrids of point rendering systems and contemporary image-based renderers.

Figure 7.3: A ray-traced rendering (with ray-traced shadows) of Michelangelo's St. Matthew. The model was stored in QSplat format, and this rendering was generated by intersecting rays with the bounding sphere hierarchy.

7.5 Streaming QSplat

We have demonstrated a system for view-dependent network streaming and interactive display of large, complex 3D models. The implementation works with a standard web server, incurs low run-time overhead on the client, and takes advantage of the low preprocessing costs, compact storage, and real-time rendering capabilities of QSplat.

As mentioned earlier, the per-node storage requirements of QSplat are higher than those achievable by some other geometric compression algorithms, largely because QSplat must store per-vertex normals. Although it would not be practical to eliminate QSplat's per-vertex normals completely, their storage cost could be considerably reduced in cases in which low per-primitive cost is critical (e.g. low-speed modem links). By combining incremental encoding of normals (i.e., encoding the normal of each node as a displacement relative to the normal of its parent node) with an entropy coding technique (e.g. Huffman coding [Huffman 52]), we could reduce the storage requirements for a normal from the present 14 bits to perhaps 3-5 bits per node. In addition, using Huffman coding for vertex position, sphere radius, and color could further reduce the per-node storage requirements of QSplat, to be competitive with state-of-the-art polygonal compression techniques. Adding this extra compression, however, would require devoting CPU time to decompressing the network-streamed data before it could be rendered, thus decreasing rendering performance (especially on a single-CPU machine) and increasing the latency with which newly-downloaded blocks could be used in rendering.

A second improvement would be to eliminate the need for temporary storage on the client. Because the present implementation is based closely on QSplat, the client requires a local temporary file equal in size to the size of the model. This file is memory mapped, and blocks are written to the file as they are received. For widest applicability, such as a web browser plugin, the client machine should not be required to have this much free disk space (which for a model of hundreds of millions of samples may approach a gigabyte). The temporary file could be eliminated by adding an additional level of indirection to the mapping from the logical position of a section of a model to physical location in memory. This extra pointer would also permit sections of the model to be discarded in an LRU fashion, to limit total memory usage. For certain systems, the virtual memory implementation can provide the same capabilities.

Appendix: Scanner Noise and Weighting

During the later stages of ICP, the goal shifts from reducing the error quickly to finding the "correct" transformation as accurately as possible. In order to determine an accurate alignment, it is necessary to take into account the uncertainty in the contribution of each point pair to the error metric. If the weights on point pairs are assigned inversely proportional to the uncertainties, minimizing the weighted error metric will find the transformation that uses the data optimally.

We derive an expression for the uncertainty in point-to-plane distance (see Section 3.4.5) for the simplified case of a translating laser-plane triangulation scanner. To further simplify the problem, we only consider a single planar surface (Figure A.1a). The result derived here is used as the "uncertainty" weighting method in Section 3.4.3.

We begin by considering the width of the laser stripe on the surface of the object. This width varies as

$$W_{surf} = W_0 \sec\theta \tag{A.1}$$

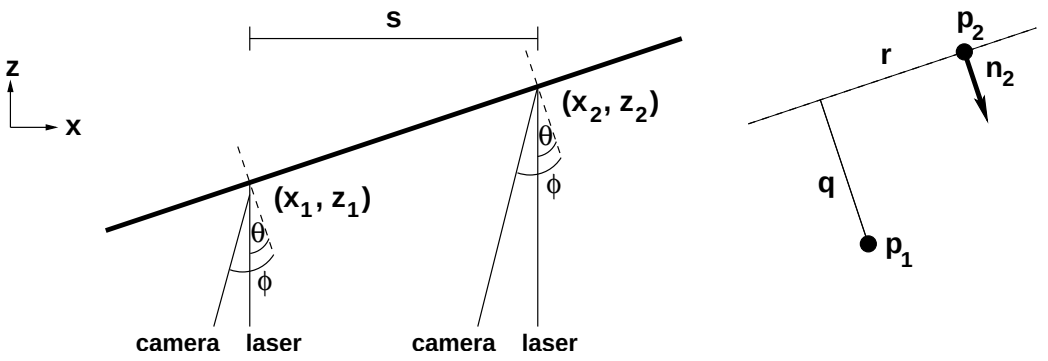
for some W_0. The width as seen by the camera is then

$$W_{cam} = W_{surf} \cos\phi \tag{A.2}$$

We now look at the x and z components of the uncertainty in the position of a point on the surface. We assume, as does [Turk 94], that the laser beam has a Gaussian profile, and that the z component of uncertainty is proportional to the uncertainty in finding the peak of the stripe in the camera image; thus, uncertainty in z is proportional to the width of the stripe as seen by the camera. The x component of the uncertainty is a function of scanner calibration, hence is a constant. Thus,

$$\Delta z = a \sec\theta \cos\phi \tag{A.3}$$

$$\Delta x = b \tag{A.4}$$



(a) (b)

Figure A.1: (a) Scanner configuration assumed for error analysis. We assume a laser-stripe triangulation scanner with a single camera. The scanner translates a distance s per frame, in a direction perpendicular to the laser plane. The angle between the surface normal and the laser is θ, and the angle between the camera and surface is ϕ. (b) The distance from p_1 to the plane containing p_2 and perpendicular to n_2 is denoted by q.

for some constants a and b.

As observed by [Dorai 97], in analyzing scanner errors we must consider not only the uncertainties in position, but also the uncertainty in computing surface normals:

$$\tan\theta = \frac{z_2 - z_1}{x_2 - x_1} \tag{A.5}$$

Differentiating,

$$(\sec^2\theta)\Delta\theta = \frac{\Delta(z_2 - z_1)}{x_2 - x_1} + \frac{z_2 - z_1}{x_2 - x_1}\frac{\Delta(x_2 - x_1)}{x_2 - x_1} \tag{A.6}$$

$$\Delta\theta = \left(\frac{a\sec\theta\cos\phi}{s} + \frac{b\tan\theta}{s}\right)\cos^2\theta \tag{A.7}$$

$$= \frac{a}{s}\cos\theta\cos\phi + \frac{b}{s}\cos\theta\sin\theta \tag{A.8}$$

Thus, we see that the uncertainty in surface normals is actually highest when the surface faces the camera and lowest when it is oblique to the camera.

We may now consider the uncertainty in the point-to-plane error (see Figure A.1b):

$$\Delta q = r\Delta\theta + \cos\theta(\Delta z_1 + \Delta z_2) + \sin\theta(\Delta x_1 + \Delta x_2) \tag{A.9}$$

$$= \frac{r}{s}\cos\theta(a\cos\phi + b\sin\theta) + 2a\cos\phi + 2b\sin\theta \tag{A.10}$$

This expression is a function of r, which is the point-to-point distance along the normal plane. When the two scans are close together, we expect r to be on the order of $s \cdot \sec\theta$, where s is the spacing in x of range samples. Substituting, we obtain

$$\Delta q = 3a\cos\phi + 3b\sin\theta \tag{A.11}$$

For most range scanners, the uncertainty along the line of sight (which is proportional to the constant a) will dominate the uncertainty in scanner position (given by b). In this case, the error in point-to-plane distance is just proportional to $\cos\phi$. In summary, the optimal weighting of point pairs for the point-to-plane algorithm is proportional to the secant of ϕ, the angle between the surface normal and the line of sight to the camera.

References

[**3DV Systems**] 3DV Systems, Inc. "ZCam," Web page: http://www.3dvsystems.com/

[**Abadjev 99**] Abadjev, V., del Rosario, M., Lebedev, A., Migdal, A., and Paskhaver, V. "MetaStream," *Proc. VRML*, 1999.

[**Aliaga 99**] Aliaga, D., Cohen, J., Wilson, A., Baker, E., Zhang, H., Erikson, C., Hoff, K., Hudson, T., Stuerzlinger, W., Bastos, R., Whitton, M., Brooks, F., and Manocha, D. "MMR: An Interactive Massive Model Rendering System Using Geometric and Image-Based Acceleration," *Proc. Symposium on Interactive 3D Graphics*, 1999.

[**Amenta 98**] Amenta, N., Bern, M., and Kamvysselis, M. "A New Voronoi-Based Surface Reconstruction Algorithm," *Proc. SIGGRAPH*, 1998.

[**Animatek**] AnimaTek International, Inc. "Caviar Technology," Web page: http://www.animatek.com/products_caviar.htm

[**Arun 87**] Arun, K., Huang, T., and Blostein, S. "Least-Squares Fitting of Two 3-D Point Sets," *Trans. PAMI*, Vol. 9, No. 5, 1987.

[**Arvo 89**] Arvo, J. and Kirk, D. "A Survey of Ray Tracing Acceleration Techniques," *An Introduction to Ray Tracing*, Glassner, A. S. ed., Academic Press, 1989.

[**Benjemaa 97**] Benjemaa, R. and Schmitt, F. "Fast Global Registration of 3D Sampled Surfaces Using a Multi-Z-Buffer Technique," *Proc. 3DIM*, 1997.

[**Bergevin 96**] Bergevin, R., Soucy, M., Gagnon, H., and Laurendeau, D. "Towards a General Multi-View Registration Technique," *Trans. PAMI*, Vol. 18, No. 5, 1996.

[**Bernardini 99**] Bernardini, F., Mittleman, J., Rushmeier, H., Silva, C., and Taubin, G. "The Ball-Pivoting Algorithm for Surface Reconstruction," *IEEE Trans. on Vis. and Comp. Graph.*, Vol. 5, No. 4, 1999.

[**Besl 88**] Besl, P. "Active Optical Range Imaging Sensors," *Machine Vision and Applications*, Vol. 1, 1988.

[**Besl 92**] Besl, P. and McKay, N. "A Method for Registration of 3-D Shapes," *Trans. PAMI*, Vol. 14, No. 2, Feb. 1992.

[**Bier 93**] Bier, E., Stone, M., Pier, K., Buxton, W., and DeRose, T. "Toolglass and Magic Lenses: The See-Through Interface," *Proc. SIGGRAPH*, 1993.

[**Bitner 76**] Bitner, J. R., Erlich, G, and Reingold, E. M. "Efficient Generation of the Binary Reflected Gray Code and its Applications," *CACM*, Vol. 19, No. 9, 1976.

[**Blais 95**] Blais, G. and Levine, M. "Registering Multiview Range Data to Create 3D Computer Objects," *Trans. PAMI*, Vol. 17, No. 8, 1995.

[**Boyer 87**] Boyer, K. L. and Kak, A. C. "Color-Encoded Structured Light for Rapid Active Ranging," *Trans. PAMI*, Vol. 9, No. 1, 1987.

[**Bregler 00**] Bregler, C., Hertzmann, A., and Biermann, H. "Recovering Non-Rigid 3D Shape from Image Streams," *Proc. CVPR*, 2000.

[**Brown 97**] Brown, R. G. and Hwang, P. Y. C. *Introduction to Random Signals and Applied Kalman Filtering*, 3 ed., John Wiley & Sons, 1997.

[**Carrihill 85**] Carrihill, B. and Hummel, R. "Experiments with the Intensity Ratio Depth Sensor," *Computer Vision, Graphics, and Image Processing*, Vol. 32, 1985.

[**Caspi 96**] Caspi, D. and Kiryari, N. "Range Imaging with Adaptive Color Structured Light," *Trans. PAMI*, Vol. 20, No. 5, 1996.

[**Certain 96**] Certain, A., Popović, J, DeRose, T., Duchamp, T., Salesin, D., and Stuetzle, W. "Interactive Multiresolution Surface Viewing," *Proc. SIGGRAPH*, 1996.

[**Chang 99**] Chang, C., Bishop, G., and Lastra, A. "LDI Tree: A Hierarchical Representation for Image-Based Rendering," *Proc. SIGGRAPH*, 1999.

[**Chen 91**] Chen, Y. and Medioni, G. "Object Modeling by Registration of Multiple Range Images," *Proc. IEEE Conf. on Robotics and Automation*, 1991.

[**Chen 99**] Chen, C., Hung, Y., and Cheng, J. "RANSAC-Based DARCES: A New Approach to Fast Automatic Registration of Partially Overlapping Range Images," *Trans. PAMI*, Vol. 21, No. 11, 1999.

[**Cline 88**] Cline, H. E., Lorensen, W. E., Ludke, S., Crawford, C. R., and Teeter, B. C. "Two Algorithms for the Three-Dimensional Reconstruction of Tomograms," *Medical Physics*, Vol. 15, No. 3, 1988.

[**Cohen-Or 98**] Cohen-Or, D. and Zadicario, E. "Visibility Streaming for Network-based Walkthroughs," *Proc. Graphics Interface*, 1998.

[**Cook 87**] Cook, R., Carpenter, L., and Catmull, E. "The Reyes Image Rendering Architecture," *Proc. SIGGRAPH*, 1987.

[**Costeira 98**] Costeira, J. and Kanade, T. "A Multi-Body Factorization Method for Motion Analysis," *IJCV*, Vol. 29, No. 3, 1998.

[**Csuri 79**] Csuri, C., Hackathorn, R., Parent, R., Carlson, W., and Howard, M. "Towards an Interactive High Visual Complexity Animation System," *Proc. SIGGRAPH*, 1979.

[**Curless 96**] Curless, B. and Levoy, M. "A Volumetric Method for Building Complex Models from Range Images," *Proc. SIGGRAPH*, 1996.

[**Curless 97**] Curless, B. "New Methods for Surface Reconstruction from Range Images," Ph. D. Dissertation, Stanford University, 1997.

[**Davies 96**] Davies, C. and Nixon, M. "Sensing Surface Discontinuities via Coloured Spots," *Proc. IWISP*, 1996.

[**Davis 01**] Davis, J. and Chen, X. "A Laser Range Scanner Designed for Minimum Calibration Complexity," *Proc. 3DIM*, 2001.

[**Deering 95**] Deering, M. "Geometry Compression," *Proc. SIGGRAPH*, 1995.

[**Dorai 97**] Dorai, C., Weng, J., and Jain, A. "Optimal Registration of Object Views Using Range Data," *Trans. PAMI*, Vol. 19, No. 10, 1997.

[**Dorai 98**] Dorai, C., Weng, J., and Jain, A. "Registration and Integration of Multiple Object Views for 3D Model Constrution," *Trans. PAMI*, Vol. 20, No. 1, 1998.

[**Duchaineau 97**] Duchaineau, M., Wolinsky, M., Sigeti, D,. Miller, M., Aldrich, C., and Mineev-Weinstein, M. "ROAMing Terrain: Real-time Optimally Adapting Meshes," *Proc. Visualization*, 1997.

[**Eck 95**] Eck, M., DeRose, T., Duchamp, T., Hoppe, H., Lounsbery, M., and Stuetzle, W. "Multiresolution Analysis of Arbitrary Meshes," *Proc. SIGGRAPH*, 1995.

[**Edelsbrunner 92**] Edelsbrunner, H. and Mücke, E. P. "Three-Dimensional Alpha Shapes," *Proc. Volume Visualization Workshop*, University of North Carolina at Chapel Hill, 1992.

[**Eggert 97**] Eggert, D. W., Lorusso, A., and Fisher, R. B. "Estimating 3-D Rigid Body Transformations: A Comparison of Four Major Algorithms," *MVA*, Vol. 9, No. 5/6, 1997.

[**Faugeras 86**] Faugeras, O. and Hebert, M. "The Representation, Recognition, and Locating of 3-D Objects," *Int. J. Robotic Res.*, Vol. 5, No. 3, 1986.

[**Faugeras 93a**] Faugeras, O., Hotz, B., Mathieu, H., Viéville, T., Zhang, Z., Fua, P., Théron, E., Moll, L., Berry, G., Vuillemin, P., and Proy, C. "Real Time Correlation-Based Stereo: Algorithm, Implementations and Applications," *Technical Report RR-2013*, INRIA, 1983.

[**Faugeras 93b**] Faugeras, O. *Three-Dimensional Computer Vision: A Geometric Viewpoint*, MIT Press, 1993.

[**Fielding 97**] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Berners-Lee, T. "Hypertext Transfer Protocol – HTTP/1.1," *RFC 2068*, UC Irvine, DEC, MIT/LCS, 1997.

[**Funkhouser 92**] Funkhouser, T., Séquin, C., and Teller, S. "Management of Large Amounts of Data in Interactive Building Walkthroughs," *Proc. Symposium on Interactive 3D Graphics*, 1992.

[**Funkhouser 93**] Funkhouser, T. and Séquin, C. "Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments," *Proc. SIGGRAPH*, 1993.

[**Funkhouser 96**] Funkhouser, T. "Database Management for Interactive Display of Large Architectural Models," *Graphics Interface*, 1996.

[**Gartner 96**] Gartner, H., Lehle, P., and Tiziani, H. "New, Highly Efficient, Binary Codes for Structured Light Methods," *SPIE*, Vol. 2599, 1996.

[**Godin 94**] Godin, G., Rioux, M., and Baribeau, R. "Three-dimensional Registration Using Range and Intensity Information," *Proc. SPIE: Videometrics III*, Vol. 2350, 1994.

[**Greene 93**] Greene, N., Kass, M., and Miller, G. "Hierarchical Z-buffer Visibility," *Proc. SIGGRAPH*, 1993.

[**Grossman 98**] Grossman, J. and Dally, W. "Point Sample Rendering," *Proc. Eurographics Rendering Workshop*, 1998.

[**Gruss 92**] Gruss, A., Tada, S., and Kanade, T. "A VLSI Smart Sensor for Fast Range Imaging," *Proc. IEEE Int. Conf. on Intelligent Robots and Systems*, 1992.

[**Guéziec 99**] Guéziec, A., Taubin, G., Horn, B., and Lazarus, F. "A Framework for Streaming Geometry in VRML," *IEEE Computer Graphics & Applications*, Vol. 19, No. 2, 1999.

[**Gühring 01**] Gühring, J. "Reliable 3D Surface Acquisition, Registration and Validation Using Statistical Error Models," *Proc. 3DIM*, 2001.

[**Heikkilä 97**] Heikkilä, J. and Silven, O. "A Four-Step Camera Calibration Procedure with Implicit Image Correction," *Proc. CVPR*, 1997.

[**Hoppe 92**] Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., and Stuetzle, W. "Surface Reconstruction from Unorganized Points," *Proc. SIGGRAPH*, 1992.

[**Hoppe 96**] Hoppe, H. "Progressive Meshes," *Proc. SIGGRAPH*, 1996.

[**Hoppe 97**] Hoppe, H. "View-Dependent Refinement of Progressive Meshes," *Proc. SIGGRAPH*, 1997.

[**Hoppe 98**] Hoppe, H. "Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering," *Proc. Visualization*, 1998.

[**Horn 87**] Horn, B. "Closed-Form Solution of Absolute Orientation Using Unit Quaternions," *JOSA A*, Vol. 4, No. 4, 1987.

[**Horn 88**] Horn, B., Hilden, H., and Negahdaripour, S. "Closed-Form Solution of Absolute Orientation Using Orthonormal Matrices," *JOSA A*, Vol. 5, No. 7, 1988.

[**Horn 99**] Horn, E. and Hiryati, N. "Toward Optimal Structured Light Patterns," *Image and Vision Computing*, Vol. 17, 1999.

[**Hornbeck 88**] Hornbeck, L. J. and Nelson. W. E. "Bistable Deformable Mirror Device," *OSA Technical Digest*, Vol. 8, *Spatial Light Modulators and Applications*, 1988.

[**Huber 01**] Huber, D. "Automatic 3D Modeling Using Range Images Obtained from Unknown Viewpoints," *Proc. 3DIM*, 2001.

[**Huffman 52**] Huffman, D. "A Method for the Construction of Minimum Redundancy Codes," *Proc. IRE*, Vol. 40, No. 9, 1952.

[**Jiang 94**] Jiang, X. and Bunke, H. "Range Data Acquisition by Coded Structured Light: Error Characteristic of Binary and Gray Projection Code," *SPIE*, Vol. 2252, 1994.

[**Johnson 97a**] Johnson, A. and Hebert, M. "Surface Registration by Matching Oriented Points," *Proc. 3DIM*, 1997.

[**Johnson 97b**] Johnson, A. and Kang, S. "Registration and Integration of Textured 3-D Data," *Proc. 3DIM*, 1997.

[**Jokinen 99**] Jokinen, O. "Self-Calibration of a Light Striping System by Matching Multiple 3-D Profile Maps," *Proc. 3DIM*, 1999.

[**Kajiya 89**] Kajiya, J. and Kay, T. "Rendering Fur with Three Dimensional Textures," *Proc. SIGGRAPH*, 1989.

[**Khodakovsky 00**] Khodakovsky, A., Schröder, P., and Sweldens, W. "Progressive Geometry Compression," *Proc. SIGGRAPH*, 2000.

[**Krishnamurthy 96**] Krishnamurthy, V. and Levoy, M. "Fitting Smooth Surfaces to Dense Polygon Meshes," *Proc. SIGGRAPH*, 1986.

[**Kumar 96**] Kumar, S., Manocha, D., Garrett, W., and Lin, M. "Hierarchical Back-Face Computation," *Proc. Eurographics Rendering Workshop*, 1996.

[**Laur 91**] Laur, D. and Hanrahan, P. "Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering," *Proc. SIGGRAPH*, 1991.

[**Levoy 85**] Levoy, M. and Whitted, T. "The Use of Points as a Display Primitive," *Technical Report TR 85-022*, University of North Carolina at Chapel Hill, 1985.

[**Levoy 88**] Levoy, M., "Display of Surfaces from Volume Data," *IEEE Computer Graphics & Applications*, Vol. 8, No. 3, 1988.

[**Levoy 00**] Levoy, M., Pulli, K., Curless, B., Rusinkiewicz, S., Koller, D., Pereira, L., Ginzton, M., Anderson, S., Davis, J., Ginsberg, J., Shade, J., and Fulk, D. "The Digital Michelangelo Project: 3D Scanning of Large Statues," *Proc. SIGGRAPH*, 2000.

[**Lindstrom 98**] Lindstrom, P. and Turk, G. "Fast and Memory Efficient Polygonal Simplification," *Proc. Visualization*, 1998.

[**Luebke 97**] Luebke, D., and Erikson, C. "View-Dependent Simplification of Arbitrary Polygonal Environments," *Proc. SIGGRAPH*, 1997.

[**Masuda 96**] Masuda, T., Sakaue, K., and Yokoya, N. "Registration and Integration of Multiple Range Images for 3-D Model Construction," *Proc. CVPR*, 1996.

[**Matsumoto 97**] Matsumoto, Y., Terasaki, H., Sugimoto, K., and Arakawa, T. "A Portable Three-Dimensional Digitizer," *Proc. 3DIM*, 1997.

[**Matusik 00**] Matusik, W., Buehler, C., Raskar, R., Gortler, S., and McMillan, L. "Image-Based Visual Hulls," *Proc. SIGGRAPH*, 2000.

[**Maver 93**] Maver, J. and Bajcsy, R. "Occlusions as a Guide for Planning the Next View," *Trans. PAMI*, Vol. 15, No. 5, 1993.

[**Max 95**] Max, N. and Ohsaki, K. "Rendering Trees from Precomputed Z-buffer Views," *Proc. Eurographics Rendering Workshop*, 1995.

[**Nayar 96**] Nayar, S. K., Watanabe, M., and Noguchi, M. "Real-Time Focus Range Sensor," *Trans. PAMI*, Vol. 18, No. 12, 1996.

[**Neugebauer 97**] Neugebauer, P. "Geometrical Cloning of 3D Objects via Simultaneous Registration of Multiple Range Images," *Proc. SMA*, 1997.

[**Octree**] Octree Corporation, Inc. "Octree Graphics," Web page: http://www.octree.com/graphics.shtml

[**Pajarola 00**] Pajarola, R. and Rossignac, J. "Compressed Progressive Meshes," *IEEE Transactions on Visualization and Computer Graphics*, Vol. 6, No. 1, 2000.

[**Pajarola 99**] Pajarola, R. and Rossignac, J. "Compressed Progressive Meshes," *Technical Report GIT-GVU-99-05*, Georgia Institute of Technology, 1999.

[**Pollefeys 99**] Pollefeys, M. "Self-Calibration and Metric 3D Reconstruction from Uncalibrated Image Sequences," Ph. D. Dissertation, Katholieke Universiteit Leuven, 1999.

[**Posdamer 82**] Posdamer, J. L. and Altschuler, M. D. "Surface Measurement by Space-encoded Projected Beam Systems," *Computer Graphics and Image Processing*, Vol. 18, 1982.

[**Prince 00**] Prince, C. *Progressive Meshes for Large Models of Arbitrary Topology*, M. S. Dissertation, University of Washington, 2000.

[**Proesmans 96**] Proesmans, M., Van Gool, L., and Oosterlinck, A. "One-Shot Active 3D Shape Acquisition," *Proc. ICPR*, 1996.

[**Pulli 97**] Pulli, K. *Surface Reconstruction and Display from Range and Color Data*, Ph. D. Dissertation, University of Washington, 1997.

[**Pulli 99**] Pulli, K. "Multiview Registration for Large Data Sets," *Proc. 3DIM*, 1999.

[**Raskar 98**] Raskar, R., Welch, G., Cutts, M., Lake, A., Stesin, L., and Fuchs, H. "The Office of the Future: A Unified Approach to Image-Based Modeling and Spatially Immersive Displays," *Proc. SIGGRAPH*, 1998.

[**Raskar 99**] Raskar, R., Brown, M., Yang, R., Chen, W., Welch, G., Towles, H., Seales, B., and Fuchs, H. "Multi-Projector Displays Using Camera-Based Registration," *Proc. Visualization*, 1999.

[**RealityWave**] RealityWave, Inc. "VizStream Technology," Web page: http://www.realitywave.com/technology.asp

[**Reeves 83**] Reeves, W. "Particle Systems – A Technique for Modeling a Class of Fuzzy Objects," *Proc. SIGGRAPH*, 1983.

[**Reid 79**] Reid, D. "An Algorithm for Tracking Multiple Targets," *Trans. Auto. Control*, Vol. 24, No. 6, 1979.

[**Rioux 94**] Rioux, M. "Digital 3-D Imaging: Theory and Applications," *SPIE*, Vol. 2350, 1994.

[**Rossignac 93**] Rossignac, J. and Borrel, P. "Multi-Resolution 3D Approximations for Rendering Complex Scenes," *Geometric Modeling in Computer Graphics*, 1993.

[**Rubin 80**] Rubin, S. M. and Whitted, T. "A 3-Dimensional Representation for Fast Rendering of Complex Scenes," *Proc. SIGGRAPH*, 1980.

[**Rushmeier 97**] Rushmeier, H., Taubin, G., and Guéziec, A. "Applying Shape from Lighting Variation to Bump Map Capture," *Proc. Eurographics Rendering Workshop*, 1997.

[**Samet 90**] Samet, H. *Applications of Spatial Data Structures*, Addison-Wesley, 1990.

[**Sato 87**] Sato, K. and Inokuchi, S. "Range-Imaging System Utilizing Nematic Liquid Crystal Mask," *Proc. ICCV*, 1987.

[**Shirman 93**] Shirman, L. and Abi-Ezzi, S. "The Cone of Normals Technique for Fast Processing of Curved Patches," *Proc. Eurographics*, 1993.

[**Simon 96**] *Fast and Accurate Shape-Based Registration*, Ph. D. Dissertation, Carnegie Mellon University, 1996.

[**Stein 92**] Stein, F. and Medioni, G. "Structural Indexing: Efficient 3-D Object Recognition," *Trans. PAMI*, Vol. 14, No. 2, 1992.

[**Stoddart 96**] Stoddart, A. and Hilton, A. "Registration of Multiple Point Sets," *Proc. CVPR*, 1996.

[**Swan 97**] Swan, J., Mueller, K., Möller, T., Shareef, N., Crawfis, R., and Yagel, R. "An Anti-Aliasing Technique for Splatting," *Proc. Visualization*, 1997.

[**Taubin 98**] Taubin, G. and Rossignac, J. "Geometric Compression Through Topological Surgery," *ACM Trans. on Graphics*, Vol. 17, No. 2, 1998.

[**Teller 91**] Teller, S. and Séquin, C. "Visibility Preprocessing for Interactive Walkthroughs," *Proc. SIGGRAPH*, 1991.

[**Turk 94**] Turk, G. and Levoy, M. "Zippered Polygon Meshes from Range Images," *Proc. SIGGRAPH*, 1994.

[**Valkenburg 98**] Valkenburg, R. J. and McIvor, A. M. "Accurate 3D Measurement Using a Structured Light System," *Image and Vision Computing*, Vol. 16, No. 2, 1998.

[**VRML 97**] *Virtual Reality Modeling Language*, ISO/IEC Standard 14772-1:1997.

[**Walker 91**] Walker, M., Shao, L., and Volz, R. "Estimating 3-D Location Parameters Using Dual Number Quaternions," *CVGIP: Image Understanding*, Vol. 54, No. 3, 1991.

[**Weik 97**] Weik, S. "Registration of 3-D Partial Surface Models Using Luminance and Depth Information," *Proc. 3DIM*, 1997.

[**Westover 89**] Westover, L. "Interactive Volume Rendering," *Proc. Volume Visualization Workshop*, University of North Carolina at Chapel Hill, 1989.

[**Yemez 99**] Yemez, Y. and Schmitt, F. "Progressive Multilevel Meshes from Octree Particles," *Proc. 3D Digital Imaging and Modeling*, 1999.

[**Zhang 97**] Zhang, H. and Hoff, K. "Fast Backface Culling Using Normal Masks," *Proc. Symposium on Interactive 3D Graphics*, 1997.