UC San Diego

UC San Diego Electronic Theses and Dissertations

Title

Capturing geometry in real-time using a tracked Microsoft Kinect

Permalink

<https://escholarship.org/uc/item/1nc8q289>

Author

Tenedorio, Daniel

Publication Date

2011

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Capturing Geometry in Real-time using a Tracked Microsoft Kinect

A thesis submitted in partial satisfaction of the
requirements for the degree
Master of Science

in

Computer Science

by

Daniel Tenedorio

Committee in charge:

Jürgen Schulze, Chair
Henrik Wann Jensen
David Kriegman

2011

Copyright
Daniel Tenedorio, 2011
All rights reserved.

The thesis of Daniel Tenedorio is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2011

DEDICATION

I dedicate this thesis to my brother, Theodore Tenedorio. I hope that he finds his education to be as rewarding as mine has been.

TABLE OF CONTENTS

Signature Page	iii
Dedication	iv
Table of Contents	v
List of Figures	vii
Acknowledgements	viii
Vita and Publications	ix
Abstract of the Thesis	x
Chapter 1 Introduction	1
Chapter 2 Geometry Scanning Pipeline	4
2.1 Kinect Hardware	4
2.2 Real-Time Camera Pose Acquisition	6
2.2.1 ART Tracking System	6
2.2.2 ARToolkit	6
2.3 Transforming Points into World Space	8
Chapter 3 Redundant Data Elimination	10
3.1 Binned-Points Spatial Hashing	10
3.2 The Surface Invariant	12
3.3 High Depth-Gradient Filter	15
Chapter 4 Triangle Meshing	18
4.1 Asynchronous Processing	18
4.2 Ball-Pivoting	19
4.3 Marching Cubes	20
4.4 Texturing Algorithm	20
Chapter 5 User Interface and StarCAVE Display	24
5.1 User Interface	24
5.2 StarCAVE Display	26
Chapter 6 Results and Analysis	28
6.1 Real-Time Pose Acquisition	28
6.2 Redundant Data Elimination Algorithms	28
6.3 Triangle Meshing	33

Chapter 7	Conclusions	36
	7.1 Limitations	36
	7.2 Future Work	38
Bibliography		40

LIST OF FIGURES

Figure 1.1:	Stages of a scanning session with a stuffed bear	1
Figure 2.1:	Geometry scanning pipeline	5
Figure 2.2:	Labeled photograph of the tracked Kinect	7
Figure 2.3:	RGB, infrared, and depth frames from the Kinect	7
Figure 3.1:	Enforcing the surface invariant with intersection tests	13
Figure 3.2:	The AddPoint algorithm	14
Figure 3.3:	High depth-gradient filtering diagram	15
Figure 3.4:	The DepthGradFilter algorithm	16
Figure 3.5:	Kinect depth-map artifacts at close range	17
Figure 4.1:	Illustration of the texturing algorithm	21
Figure 4.2:	The TextureTriangle algorithm	23
Figure 5.1:	User interface of the scanning system	25
Figure 5.2:	UCSD’s StarCAVE: diagram and photograph	27
Figure 6.1:	Results: textured meshes generated using ARToolkit vs. tracker	29
Figure 6.2:	Results: textured meshes of stuffed bear	30
Figure 6.3:	Graphs: memory usage and point list length vs. bin size	31
Figure 6.4:	Graph: processing time per point vs. bin width	32
Figure 6.5:	Graph: points filtered out vs. max angle between camera direc- tion and surface normal vector	34
Figure 6.6:	Graph: texture generation time for one triangle vs. size of allotted texture square	35
Figure 7.1:	Texturing limitation for non-Lambertian surfaces	37

ACKNOWLEDGEMENTS

This research benefited from conversations with many intelligent people in the fields of computer graphics and computer vision at the University of California, San Diego. Professors Henrik Wann Jensen, David Kriegman, and Serge Belongie gave lots of high-level advice from their years of experience. Postdoctoral researcher Florian Schoff provided useful information about structure-from-motion algorithms for purposes of comparison with structured-light-based systems. Daniel Knoblauch helped install Kinect drivers and plug into the Kinect-hacking literature. Toshiya Hachisuka, Iman Sadeghi, and Oleg Bisker offered lots of performance optimization ideas over the course of several discussions at the UCSD Computer Graphics Lab. Calit2 staff researchers Andrew Prudhomme and Philip Weber helped calibrate tracking systems and provided feedback for debugging computer graphics code. Finally, I would like to thank Dr. Larry Smarr for his expert directorship of the California Institute for Telecommunications and Information Technology.

VITA

2006	Summer Internship – SKF Condition Monitoring
2007	Summer Internship – Concrete Games, a division of THQ
2008	Pacific Rim Undergraduate Research Experiences (PRIME) – Hsinchu, Taiwan
2008	B. S. in Computer Science, University of California, San Diego
2008-2009	Software Analyst I at Calit2 – Atlas In Silico project
2009	Summer Internship – Palm, Inc.
2009-2010	Graduate Teaching Assistant – University of California, San Diego
2010	Summer Internship – Google, Inc.
2011	M. S. in Computer Science, University of California, San Diego

PUBLICATIONS

West, Ruth, et. al, “Algorithmic Object as Natural Specimen: Meta Shape Grammar Objects from ATLAS in Silico”, *Leonardo Electronic Almanac*, Vol. 16 Issues 6-7, 2009.

Tenedorio, Daniel, et. al “Optimized Rendering for a Three-Dimensional Videoconferencing System”, *Proceedings for the 4th IEEE International Conference on e-Science*, 2008

ABSTRACT OF THE THESIS

Capturing Geometry in Real-time using a Tracked Microsoft Kinect

by

Daniel Tenedorio

Master of Science in Computer Science

University of California, San Diego, 2011

Jürgen Schulze, Chair

We propose a new geometry scanning system that permits the user to obtain detailed triangle models of real-world places with a tracked Microsoft Kinect device. The system generates a texture map for the triangle mesh using video frames from the Kinect's color camera and displays a continually-updated preview of the textured model in real-time, allowing the user to re-scan the scene from any direction to fill holes or increase the texture resolution. We also present filtering methods to maintain a high-quality model of reasonable size by removing overlapping or low-precision range scans. Our approach works well in the presence of degenerate geometry or when closing loops about the scanned subject, making it especially suited for scanning large areas. We demonstrate the ability of our system to acquire 3D models at human scale with a prototype implementation

in the StarCAVE, a holographic virtual-reality environment at the University of California, San Diego.

Chapter 1

Introduction

Figure 1.1: The leftmost image is a photograph of a stuffed bear sitting on a cardboard box. We use a tracked Kinect to scan the bear, producing a textured triangle mesh. The system displays an interactive preview of the model, as shown in the remaining images.

Measurement of the geometric structure of real-world objects is an active field of research with promising applications in virtual and augmented reality. In this paper, we discuss methods of using 3D scanning to construct high-resolution models with texture-mapped primitives for use in rendering.

Most 3D scanners record data from only one view. To build a complete model from several range scans, the user must move the scanner relative to the object and obtain enough views to cover all desired surfaces. A complete scanning system must decide where to record scans, how to align them with one another,

and how to reconstruct a surface by merging them together. Systems with human operators may also present real-time feedback about the model’s construction to help ensure that the user has scanned all required surfaces before leaving the scene.

Researchers have proposed different methods of implementing this pipeline. Computer vision scientists have developed techniques to infer structure from motion using multiple images of a non-moving scene from a single moving camera [16,22]. Although experts have successfully constructed accurate models using these systems, most of them share several common limitations. For example, establishing correspondence between scans with degenerate geometry can be difficult in the absence of identifiable visual features. Software implementations that attempt to do so usually execute offline due to their considerable computational expense, producing a single result and leaving the user no option to fill holes or otherwise improve the reconstruction, although recent work proposes the use of parallel hardware to solve the problem online [17].

On the other hand, range scanning systems based on projected structured-light patterns can be efficient and robust [10,11,14]. Such systems produce several range scans per second, usually in the form of depth maps. Many implementations use alignment algorithms based on ICP (iterative closest points) to produce a geometric alignment between range scans given some initial estimate of their relative pose [12,19,21,25]. Some robotics systems elect to merge the scans using calibrated motion of the scanner relative to the subject (or vice versa), an approach known as time-of-flight [23]. A third class of alignment methods involves placing markers on the subject and using their images to transform scans relative to each other [7].

We present a novel geometry scanning system that uses a tracked Microsoft Kinect device to construct detailed 3D triangle models of real-world objects. As the user moves the Kinect around the scene, the system provides an interactive visual preview of the model’s construction, allowing the user to fill holes in the geometry and re-scan areas where higher detail is required. The tracking system provides a real-time, highly accurate report of the Kinect’s position and orientation, allowing the system to combine the Kinect’s range scans into a global point cloud. We link nearby points together into a triangle model, copy parts of the image from the

Kinect’s color camera into a texture, and assign texture coordinates to the triangle vertices. The final result is a textured triangle mesh that we can simply rasterize from any camera angle to produce a realistic image, as illustrated in Fig. (1.1).

Our approach makes several novel contributions to the science of model acquisition. In Section 2, we perform a rigorous analysis of the Kinect’s capabilities in the context of a general 3D scanning system, controlling against noise from pose estimation by using a calibrated tracking system. We propose a software data structure that allows the system to scale to large scanning volumes in Section 3, and discuss how our application removes overlapping areas of range scans to prevent unbounded growth of the model size. We also describe a class of depth outliers that the Kinect produces at close range and introduce an efficient method to filter them out of the depth map. In Section 4, we demonstrate that it is possible to perform triangle meshing and texturing from the Kinect’s RGB camera in real-time. In Section 5, we reveal how having knowledge of the camera pose without necessarily adding new scans allows the operator to improve the model’s quality and hedge against the effects of miscalibration and noise. We quantify the performance of our system and discuss its limitations in section 6, and present ideas for future research in section 7.

Chapter 2

Geometry Scanning Pipeline

Our geometry scanning system uses a Microsoft Kinect for rangefinding and a tracking system to register the scans together relative to a global coordinate system. In Fig. (2.1), we list the steps our system performs to construct a triangle model from these data in real-time. In the rest of this section, we describe the hardware components in the Kinect and tracking system, and explain how we use the real-time camera pose to transform points into world-space.

2.1 Kinect Hardware

Our system uses data from each component in the Kinect (see Fig. (2.2)). The infrared projector illuminates the scene with a structured dot pattern, which the infrared camera on the right records to produce a depth map in VGA resolution with eleven bits of precision per sample. The depth reconstruction is robust in the absence of strong infrared interference from other sources like other Kinects or direct sunlight. The camera in the middle records visible-color images at 30 frames per second. It has a fixed focal-length lens with low radial lens distortion and automatic brightness adjustment. Fig. (2.3) shows an example of the RGB and infrared video frames, as well as the reconstructed depth map.

The device has an angular field of view of 57 degrees horizontally and 43 degrees vertically. Its sensing range is 0.7-6 meters from the subject, although we

Figure 2.1: Our system selects high-quality points from each range scan that are within a reasonable distance from the Kinect and are not too close to existing geometry in the model, then adds them to a spatial hashing data structure. A triangle meshing step links groups of these points into triangles, samples from recent RGB frames to populate a new part of the texture image, and assigns appropriate texture coordinates to each triangle vertex.

obtained the best quality results within a range of 1-2.5 meters. At a distance of two meters, 3D points corresponding to neighboring pixels in the depth map are about 3.3 mm apart from each other in the tangent direction.

A small baseline translation separates the RGB and IR cameras; therefore, a color sample at any pixel (x, y) does not necessarily correspond with the depth sample at that pixel. Our system uses Nicolas Burrus' Kinect calibration to associate depth samples with color values [3].

The Kinect sends the video data to the host computer over a USB 2.0 cable. Each RGB pixel uses three bytes and each depth sample uses two, so a pair of frames uses 1.46 MB at 640×480 resolution. The device sends a total of 43.94 MB/sec through the cable when running at 30 Hz, a considerable throughput given the 60 MB/sec maximum provided by the USB 2.0 interface.

2.2 Real-Time Camera Pose Acquisition

2.2.1 ART Tracking System

We obtain an accurate real-time representation of the camera pose by attaching a tracker to the top of the Kinect and recording its position and orientation using a wireless, optical tracking system by AR Tracking [1]. The system uses four ceiling-mounted infrared cameras to report the precise 6-DOF pose of the tracking target with sub-millimeter accuracy at 60 Hz. Interestingly, we have noticed that the Kinect and tracking system do not interfere with each other, but we have not investigated why this occurs.

2.2.2 ARToolkit

We have investigated an alternate means of obtaining the camera pose using ARToolkit, an augmented reality library that computes the relative pose between printed markers by searching for their images within video frames of several consumer cameras [15]. ARToolkit's list of supported cameras does not include the Kinect, but we were able to modify an existing configuration for a VGA webcam

Figure 2.2: The projector on the left casts a structured pattern of infrared dots onto the scanned object. The infrared camera on the right creates a map of depth values using the dots' projected locations. The middle RGB camera records standard video frames. The tracking sensor obtains the Kinect's real-time camera pose.

Figure 2.3: The color and infrared cameras capture VGA image frames of the scene, which the hardware uses to construct a depth map. Black pixels in the depth map show where the Kinect cannot calculate depth due to glossy surfaces that reflect the infrared light in unpredictable ways.

by replacing the intrinsic camera parameters with those of our Kinect. We can now perform marker detection using the Kinect’s RGB camera.

ARToolkit’s most common role is obtaining the marker’s pose relative to the camera for purposes of rendering 3D objects near the markers virtual location. We invert that matrix to obtain the camera’s pose relative to a single marker that we place somewhere in the scene. Without loss of generality, we use the inverted matrix to align all range scans relative to the marker, setting the origin at its center and coordinate axes parallel with its edges. We can align any range scan where the associated RGB frame contains at least one full marker image. This method does not accumulate alignment error, as we compute the camera pose for each frame independently, without using any previous range scans or model data.

2.3 Transforming Points into World Space

We perform a series of steps to generate 3D model data from the raw color and depth samples. We begin by creating a 3D point with coordinates $[x \ y \ z]^T$ from each sample at (column x , row y , depth value z). We then place the point at its perspectively-correct position relative to the Kinect using free code from the libfreenect library [5]. To transform the point into world space, we note that the 3D point $X_0 = [X \ Y \ Z \ 1]^T$ projects to point $p' = [x' \ y' \ 1]^T$ as follows:

$$\lambda x' = \Pi X_0 = K \Pi_0 g X_0$$

We wish to solve for the 3D point in the scene X_0 given the camera pose g , ideal projection matrix Π_0 , intrinsic camera parameters K , projected 2D point x' and associated depth λ . Simplifying the equation, we have:

$$\begin{aligned} \Pi^{-1}(\lambda x') &= \Pi^{-1} \Pi X_0 \\ \Pi^{-1}(\lambda x') &= X_0 \end{aligned}$$

The Kinect’s depth map already provides $(\lambda x')$, but we still need $\Pi^{-1} = (K \Pi_0 g)^{-1}$. Since we have calibrated our Kinect, we simply assume $K = I$, yielding

$\Pi_0 g = [R \ T]$ where R is the camera's rotation and T is its translation relative to the tracking coordinate system. The tracker gives us these data, so we have everything we need to solve for the original 3D point:

$$X_0 = [R \ T]^{-1} (\lambda x')$$

Chapter 3

Redundant Data Elimination

Real-time scanning systems accumulate data rapidly as they align range scans together. Without some way to merge or discard redundant 3D data, it becomes impossible to render the points at interactive framerates or perform real-time triangle meshing. Some scanning systems address this issue by collapsing points in cells of a high-resolution voxel grid [20], but we find that this approach does not scale well to large scanning volumes due to the voxel grid’s high memory requirements. These systems also sometimes fail to merge consecutive range scans in the presence of misalignment larger than the width of one voxel, creating multiple layers of occupied voxels. This effect causes the systems to render many points in the vicinity of the true surface and impairs the user’s ability to determine which parts of the model are missing. We address these issues with the following redundant point-elimination algorithms and data structures.

3.1 Binned-Points Spatial Hashing

Our goal is to store points in a data structure that supports efficient insert and lookup operations. While several such structures exist, we require the ability to process almost one hundred million points per second, either discarding each point or adding it to the model. We have decided to use a custom data structure that divides the scannable area into cubes called bins, where each bin contains a list of points. The bins are equal-sized and axis-aligned. One may equivalently

consider the data structure as a spatial hash table mapping 3D positions to lists of points, where indexing involves simply dividing each point’s offset from the edge of the world by the size of one bin.

Before adding a new point to the data structure, we also add a copy of it to several monotonically-growing dynamic arrays. These arrays store the points’ 3D positions, colors, and texture coordinates, allowing the system to use OpenGL’s `glDrawArrays` function to render the points quickly. Active bins can then store lists of indices into these arrays, where each index refers to all of the arrays at the same time. This data structure is simple to implement and has low memory requirements because of the compact index lists and relatively large bin size. We use it to efficiently perform approximate nearest-neighbor queries by determining which bin contains the query point, then returning the list of points within that bin. When new points lie near bin edges, the system adds them to the bins on either side of the edge to ensure that queries work correctly in these areas.

Some other applications use k-d trees for 3D nearest-neighbor queries [8,24]. Balanced trees generally support $O(\log N)$ insert and lookup costs. In practice, we find that they quickly become unbalanced after many insert operations, leading to unpredictable behavior. Periodic rebalancing operations can address this issue at the cost of some jerkiness in the processing frame rate. In addition, many k-d tree implementations store the actual 3D point positions in the data structure rather than indexing into arrays like our data structure does, resulting in a loss of rendering flexibility.

If each bin contains a maximum of x points, then inserting a point uses $O(\log x)$ time due to periodic list resizing, looking up all points within one bin executes in $O(1)$ time, and deleting any point uses $O(1)$ time with lazy deletion. If the size of one bin is $b \text{ m}^3$ and the points stored are no closer together than $r \text{ m}$, then it is likely that $x \leq (\frac{b}{r})^3$ if we assume that each bin contains a 3D volume of points. We examine an interesting way to enforce this assumption in the next subsection.

3.2 The Surface Invariant

With an ideal scanning system, one could eliminate redundant data by simply performing a hash-table lookup at each new point’s location to determine whether to add it to the model. In practice, we find that different scans of the same real-world location sometimes fail to merge together, resulting in an accumulation of several points in a neighborhood about the true surface. The extra data causes noisy renders and reduces rendering framerates, in addition to other problems. Considerably decreasing the hash table resolution fixes the problem, but eliminates the ability to scan detailed surfaces.

To construct a better data-reduction algorithm, we note that each depth map stores one value per pixel. Because any camera at position c can only see the surface of an object at that point, we can assert the following invariant: for new point p with depth value λ , all space in the direction $(p - c)$ with depth $< \lambda$ is empty. We enforce this invariant by testing whether any part of a short line segment through p in direction $(p - c)$ intersects a small sphere centered about each other point in the model (see Fig. (3.1)). To reduce the search space, we first perform a lookup into our spatial hash table, allowing us to only consider the list of points within the same bin as the new point. The algorithm produces a surface of spheres about scanned points through which no new line segment can penetrate. We state it more formally in Fig. (3.2).

The algorithm quickly stops adding new points in the vicinity of the surface. Using a line segment that is longer than the sphere radius produces a high-resolution surface that is robust to scanning noise and miscalibration. We note that it is possible to further reduce the search space by excluding new points that are close to old ones in a rasterization of the model from the perspective of the Kinect. We did not implement this idea as our data structure performed well enough for our purposes, but it would be interesting to investigate in future work.

Figure 3.1: For each new 3D point, we perform an intersection test between spheres about nearby points and a short line segment in the view direction. When the test fails, we merge the new point into the old one, preventing a buildup of multiple layers due to noise or miscalibration. A surface of spheres quickly emerges; thereafter, the system refuses to add new points in its vicinity.

Algorithm AddPoint(p, c)

Parameters: point p , camera position c

Returns: true if point p is OK to add to model, false if not

Constants: distances dz, r

```

b = get_bin_from_point(p)
L = points_in_bin(b)
x = dz * (p - c)
y = line segment from (p - x) to (p + x)

for each point q in L, from most to least recent:
    sphere s = (center q, radius r)
    if intersect(y, s) == true:
        return false

return true

```

Figure 3.2: The AddPoint algorithm determines whether to reject new point p by searching for nearby points in the model. We actually test against points in bins next to p if p is close to a bin edge, but the algorithm omits this description for readability. The intersect function operates quickly since it is only responsible for determining whether an intersection exists, not its exact location. The algorithm's runtime increases and memory usage decreases with $\|L\|$.

Figure 3.3: The system performs filtering by setting a maximum allowed ratio of change in depth (dz) to change in tangent direction (dx or dy). Surfaces that are perpendicular to the Kinect have a relatively low ratio, whereas more parallel surfaces have a higher ratio.

3.3 High Depth-Gradient Filter

We find that our system produces the most accurate model when the scanner is perpendicular to the scanned surface. Many shiny surfaces reflect a variety of colors depending on the viewing angle, and we find that restricting the relative camera pose results in more uniform recorded colors for these surfaces. Perpendicular orientation also minimizes noise from the Kinect’s limited depth sample resolution.

We implement this restriction by filtering the depth map using an efficient image processing technique to produce a Boolean mask. We skip over depth samples where the corresponding mask values are equal to true, allowing us to forgo the expense of sending them through the rest of the processing pipeline. The algorithm sets mask values to false for depth samples with high changes in depth relative to their neighbors, similar to how 2D edge-detection filters find areas with high gradients in pixel intensity.

Fig. (3.3) illustrates how the algorithm detects surfaces of high depth-gradient, and Fig. (3.4) lists its steps in greater detail. Its runtime is linear on the number of pixels in the depth map. Mask calculation is completely parallelizable, permitting future use of a GPU to perform the filtering.

The filtering provides the additional benefit of removing depth samples that are complete outliers. The Kinect’s depth map generally lacks such outliers when the distance between object and camera is large enough. When brought too close to the surface, the device produces artifacts like those shown in Fig. (3.5).

Algorithm DepthGradFilter(m, k)

Parameters: depth map m , matrix k projecting depth map samples to
3D points relative to Kinect

Returns: Boolean mask f indicating whether each sample should
be removed

Constants: threshold Tau , $0 < \text{Tau} < 1$

```
f = init_boolean_mask(480, 640, false)
m = gaussian_blur(m)
for j = 0, 1, ..., 638:
    for i = 0, 1, ..., 478:
        p1 = k * (j, i, m[i][j])^T
        p2 = k * (j, i + 1, m[i + 1][j])^T
        p3 = k * (j + 1, i, m[i][j + 1])^T
        dzx = |(p2.z - p1.z) / (p2.x - p1.x)|
        dzy = |(p3.z - p1.z) / (p3.y - p1.y)|
        if dzx > Tau or dzy > Tau:
            f[i][j] = true

return f
```

Figure 3.4: The DepthGradFilter produces a Boolean mask to filter out points with high depth gradient. Note that the loops are completely parallelizable. We already use the tracking system to compute the matrix k , so that comes at no additional cost. The `init_boolean_mask` function creates a VGA-resolution Boolean mask, initialized to all false values. The `gaussian_blur` function approximates a standard Gaussian filter to within about 3 percent by performing three passes of a box blur with a kernel size of eight pixels.

Figure 3.5: The left image is a visualization of the depth map from a Kinect held at proper range, using example code from libfreenect [5]. The right image uses the same software to illustrate how the depth map becomes corrupted when the scanned object is too close. Our high depth-gradient filter successfully removes these artifacts before processing the depth map each frame.

We note that Rusinkiewicz, Hall-Holt, and Levoy have proposed an alternative offline outlier elimination method based on removal of skinny triangles [20]. However, our system filters the depth map before undergoing the expense of triangle meshing. This advantage provides enough of a performance boost that we can execute our filtering in real-time.

Chapter 4

Triangle Meshing

Rendering the point model provides enough of a preview for most users to determine which areas require further scanning. However, most 3D applications use triangle mesh models, leaving the user responsible for performing offline meshing using a software package like Meshlab [2]. Creating small triangles from all of the points generates a model that is visually accurate but has too many triangles to efficiently render. It is possible to join triangles in flatter areas without significant loss in geometric accuracy, but visual quality suffers in the absence of a texture map to apply to the triangles. We discuss an algorithm to produce such a texture map using color images saved from the Kinect’s RGB camera. We then apply the texture to a triangle mesh that we compute from the point data in real-time. The textured mesh provides better feedback to the user than that obtained from directly rendering the point model.

4.1 Asynchronous Processing

While we receive a depth map from the Kinect each frame, our accumulating point model is not in the form of a heightmap. Therefore, we cannot trivially construct a triangle mesh from the points by connecting each point to two of its immediate neighbors and must instead rely on a generalized meshing algorithm to construct an incremental mesh as we gather new points. Since meshing a large point cloud is a computationally expensive operation, our system leaves unchanged

parts of the model alone and re-meshes areas with new points. Our spatial hashing data structure conveniently divides the scanning volume into such areas, populating a modified-bin list that we can use to determine which areas to process each frame.

We process a bin by iterating through each of the points in its list, then writing to some output memory storing the meshed triangles associated with that bin. Triangle mesh processing runs asynchronously, ensuring a high framerate and allowing the system to use multiple processing cores for increased performance. The edges of some triangles can cross bin edges, leading to a small amount of overlap in these areas. With texturing enabled, this effect becomes invisible. The system removes the bin from the modified-bin list after meshing all points inside it.

4.2 Ball-Pivoting

Bernardini, et al. proposed the original “Ball-Pivoting Algorithm for Surface Reconstruction” [9]. We found this approach attractive for our system because of its simplicity and ability to enforce a maximum distance apart to connect points together. The algorithm repeatedly selects a point at random and builds a seed triangle from nearby points within the maximum radius. It then pivots a ball around each triangle edge, forming another triangle about each un-meshed point that the ball intersects during its revolution. The algorithm produces the final triangle mesh by repeating these steps until no free points remain.

We have integrated the ball-pivoting implementation from the VCG library, the same library that powers Meshlab [2], using a ball-pivoting radius equal to three times the sphere radius described in the above `AddPoint` algorithm. This serial CPU implementation performs acceptably well in our system because we only process small groups of new points each frame. A nice feature of this approach is that it only uses existing points as triangle vertices, allowing us to look up the RGB frames associated with the points to accelerate the texture mapping process later.

4.3 Marching Cubes

As an alternative triangle meshing approach, we integrated nVidia’s CUDA implementation of the Marching Cubes algorithm [4, 18]. The algorithm iterates through a voxel grid, using the values to produce combinations of triangles from a precomputed array of 256 possible configurations. We initialize an $8 \times 8 \times 8$ voxel grid in each active bin and set the appropriate voxel after adding each point to the bin. To aid the voxel-assignment process, we calculate normal vectors for each point by fitting planes to its neighbors using the Mobile Robot Programming Toolkit’s RANSAC plane-fitting routines [6]. The algorithm guarantees that generated triangles have similar size and lie no further than one voxel’s distance from existing points. We find that the ball-pivoting algorithm produces a more accurate result, while the marching-cubes approach executes faster.

4.4 Texturing Algorithm

After generating triangles about scanned points, we create a texture for each triangle using image frames from the Kinect’s RGB camera. Our goal is that with reasonably correct triangle geometry, a render of the textured mesh looks similar to a photo of the same scene taken from the same pose.

The goal is easy to achieve in the case of a single image: we simply use the image as the texture, project the three vertices of each triangle back towards the image plane and use the intersection points as the triangle’s texture coordinates. There is no need to copy image data from place to place, and it is possible to inexpensively rasterize the vertices’ position with respect to the image plane on the GPU.

On the other hand, the problem becomes more interesting with our general-purpose scanner: each point in the model may originate from a different image frame. To account for this possibility, we store several recent images from the RGB camera to check against when computing the texture map, as well as the camera location and world-space position of the centermost point in the frame. We then define each image plane using the latter point and the vector from the

Figure 4.1: The system sets color values for a square-shaped portion of the texture image for each new triangle. It begins by calculating barycentric coordinates (U, V, W) for each texture pixel with respect to three of the four of the squares corners (a, b, c, d). It then calculates an interpolated position x' inside the triangle with corners (p_1, p_2, p_3). For each saved RGB frame, if a line segment from x' to the saved Kinect camera position intersects the frame within its four corners (c_1, c_2, c_3, c_4), the system assigns the corresponding color value to the current destination texture pixel.

former point to the latter. We form a bounding rectangle for each image along with its pose by projecting the four corner-most points to this image plane.

For each triangle in the scanned model, we copy color values from the saved RGB planes to pixels in a fixed-size triangle of a large texture image, packing two triangles per axis-aligned square. We store the saved RGB planes in a queue data structure, pushing them onto the back and deleting them from the back once the queue reaches a maximum size. The diagram In Fig. (4.1) illustrates the process of sampling from the RGB planes to fill the pixels in the texture.

As listed in Fig. (4.2), the algorithm tests all saved RGB frames for a color

value rather than just those frames corresponding to the triangle vertices. Its runtime is $O(q^2F)$, where q is the size of output square in the texture, in pixels, and F is the maximum queue size of saved RGB frames. In practice, we find that the algorithm samples from the most recent frame almost every time, reducing the runtime to $O(q^2)$ in these cases. We used $q = 16$ and $F = 50$, with a maximum texture size of 8192^2 pixels. For simplicity, we have only described sampling from the first intersected RGB frame, although our code can sample from several frames and interpolate the sample colors to average out the effects of noise and hide the projected edges of the rectangular RGB frames.

Algorithm TextureTriangle(v_1, v_2, v_3, tx, ty)

```

for y = 0, 1, ..., q - 1:
    for x = 0, 1, ..., q - 1:
        (U, V, W) = get_barycentric_coords_2d(
            (x, y), (1, 0), (q - 1, 0), (q - 1, q - 2))
        if U < 0 or V < 0 or W < 0:
            continue
        point t = (U * v1) + (V * v2) + (W * v3)

        for each saved RGB plane r = (camera_pos, center_point,
            corners, image), from most to least recent:
            plane p = (r.center_point, r.center_point - r.camera_pos)
            point q = calc_intersection(p, r.camera_pos - t)
            (x, y, z) = get_barycentric_coords_3d(
                q, r.corners[0], r.corners[1], r.corners[2])
            if x >= 0 and y >= 0 and x < 1 and y < 1:
                set_texture_pixel(tx + x, ty + y,
                    r.image[x * 640, y * 480])
            break

```

Figure 4.2: The TextureTriangle algorithm populates the square of pixels in the output texture image from (tx, ty) to $(tx + q - 1, ty + q - 1)$. The `get_barycentric_coords` functions takes four locations and return the barycentric coordinates of the first location with respect to the triangle formed by other three. The `calc_intersection` function calculates the point of intersection between a plane and a line segment. The `set_texture_pixel` function copies the given color value to a location in the output texture. After the algorithm finishes, the triangles texture coordinates, in pixels, are (tx, ty) .

Chapter 5

User Interface and StarCAVE Display

5.1 User Interface

We propose the use of a tracking system for alignment, unlike other approaches like ICP that use range scan data for this purpose [12, 19, 21, 25]. We find that in addition to producing more robust pose estimation, the system is capable of calculating the scanner’s pose without using any of the camera data at all, provided that it stays within the tracking volume. This freedom allows the system developer to implement several interesting user-interface features. Fig. (5.1) shows a splat rendering of a finished scan with several of these features listed in a menu.

For example, we have added a scanning toggle: by pressing a button on a separate controller, the user can turn the scanner off, retaining the ability to visualize its real-time pose without adding new data. Another button turns the scanner back on again, ready to add data from a different location. The user can exploit this feature by turning the scanner off before moving it past troublesome surfaces like mirrors that would otherwise contribute incorrect data to the model.

Pressing a different button on the same controller triggers an undo feature, removing all points scanned within the past several frames from the model. Implementing the undo feature simply requires periodically copying the point lists

Figure 5.1: This screenshot illustrates how the tracker permits development of useful user-interface features. The cone visualizes the Kinect’s pose relative to the scanned model at all times. The color of the cone indicates the scanning mode: red for scanning on, purple for scanning off, yellow for cleanup mode. The menu allows the user to display triangles or normal vectors, change the splat size, and edit the maximum scanning distance to add new points.

and spatial hashing data structure to memory or disk, then restoring them upon a call to the undo function. The feature makes particular sense when using the Kinect as a range scanner because the device produces undesirable data in a variety of common circumstances, including when the scanned object is too close, when the surface is reflective, and in the presence of infrared interference. In cases of garbage data, the user can simply undo the last few frames, turn the scanner off, and re-scan the desired surface from elsewhere.

After discovering inaccurate data that is more than a few seconds old, the user may wish to erase and rescan the affected area without discarding subsequent frames. We have implemented a cleanup scanning mode that deletes any previously-scanned points that fall within a cone oriented in the scanner’s view direction, allowing the user to re-scan the area afterwards for a higher-quality result. The user can fine-tune the cleanup, targeting specific areas using a visualization of the cone relative to the point model. This feature is especially useful when scanning people, as the user can erase and re-scan the subject’s face as desired.

5.2 StarCAVE Display

To showcase our scanning system and user-interface features, we have installed the system in the StarCAVE virtual-reality environment at the University of California, San Diego [13]. The StarCAVE comprises a fully horizontally enclosed space using fifteen rear-projected wall screens, stacked three high, with the bottom and top trapezoidal screens tilted inward by fifteen degrees to increase immersion. The viewer wears polarized glasses to view a separate image in each eye. A head computer synchronizes several video servers, one for each wall screen. Each video server runs a copy of the same program, rendering the same scene from many angles, as shown in Fig. (5.2) A head-mounted tracker allows the system to adjust the positions of rendered objects based on the viewer’s location. Our system exploits the StarCAVE’s holographic effect by displaying the real-time model in the precise virtual position of the scanned subject.

We attach the Kinect to the head computer and extract the RGB and depth frames at about 20 Hz. The head computer compresses the RGB frame using JPEG compression code and compresses the depth frame using a custom implementation of lossless Huffman coding. It then computes a header with the 6-DOF Kinect pose and sends it with the compressed frames to the video servers using TCP network sockets. The video servers receive the data and decompress the frames, then simultaneously execute the algorithms described above to generate a textured triangle mesh. Each video server also runs a copy of the rendering software to asynchronously display the current version of the triangle mesh from the unique perspective of its associated screen at 60 Hz.

(a) StarCAVE CAD Model

(b) Microscopic Geometry Viewer

Figure 5.2: The figure on top shows the layout of the screens and supporting brackets in the StarCAVE. In the below image, an example of a StarCAVE application allows multiple viewers to examine the structure of small organisms on the cellular level.

Chapter 6

Results and Analysis

We show some examples of scanned models in this section, with photographs of the scanned objects for comparison. Each model’s silhouette reveals the coarseness of the triangle mesh, but the texture image gives it a realistic appearance elsewhere. In the following subsections, we compare the performance and accuracy of our system’s processing components with other established methods.

6.1 Real-Time Pose Acquisition

After analyzing the real-time Kinect pose calculated from ARToolkit versus the one produced by the tracking system, we discover that the average reported camera position differs by an average of 11 mm and the average camera orientation differs by an average of 13 degrees. We compare textured models generated from a series of frames scanned using both approaches in Fig. (6.1). Note that the camera must be able to see at least one marker in each image for ARToolkit to work. In Fig. (6.2), we juxtapose a photograph of our stuffed bear with some closeup renderings of generated triangle models.

6.2 Redundant Data Elimination Algorithms

We used a world space of $\pm 10 \text{ m}^3$ with a bin-size of 12 cm^3 for our spatial-hashing data structure. Each empty bin uses two bytes, so the initial data structure

(a) Photograph

(b) ARToolkit

(c) Tracking System

Figure 6.1: We constructed a scene using several stuffed animals and a board with printed ARToolkit markers. We scanned the scene with the Kinect and aligned the scans using two different methods: ARToolkit and the tracking system. Comparing the textured meshes, we discover that the infrared tracking system produces a better estimate of the camera pose over time.

Figure 6.2: The bottom image is a photograph of the head of a stuffed bear. The top two images are renderings of textured triangle meshes obtained using our scanning system. In the top image, we observe an almost photorealistic appearance to the triangle mesh, except for the silhouette of the model and a few triangles on the top of the bear’s head. In the middle image, borders between triangles are more visible and there are some artifacts from the bear’s fur reflecting different brightnesses at different times.

Figure 6.3: The graph on the left shows how memory usage is inversely proportional to the bin size. In the other graph, we see that larger bins generally have longer lists of points in each bin.

over a world size of $\pm w$ m³ with bin size of b m³ uses $2 \left(\frac{w}{b}\right)^3$ bytes of RAM, as shown in Fig. (6.3). The same figure reveals how, given a fixed minimum point spacing s , each bin through which a planar surface passes contains $O(s^2)$ points. The average processing time spent per bin is linear in the number of points it contains, generally rising exponentially with the bin width.

For purposes of comparison with previous work [19], we implemented a feature wherein we initialize a small voxel grid within each active bin. This feature uses a reasonable amount of memory because it does not waste space on empty bins. We implement the same intersection test of a line segment parallel to the view direction against a neighborhood about each existing point by performing lookups into the voxel grid in regular increments across the segment, from one edge of the bin to the other. We set the voxel size to be equal to the minimum point spacing c to maintain the same point model resolution as before. In Fig. (6.4), we discover that the average processing time per bin no longer rises with the bin width, as expected.

The high depth-gradient filter possibly removes points in each range scan, depending on the maximum-allowed angle between the Kinect view direction and surface normal vector, as shown in Fig. (6.5). We found that a mimaximum angle

Figure 6.4: As the bins get larger, the average number of points within each bin’s list becomes longer. Since we have to examine them all before adding a new point to any bin, the processing time generally rises with the bin size. However, if we initialize a small voxel grid within each active bin, the average processing time per point remains relatively constant at 2 – 3 μs . The latter approach does not consume unreasonably large amounts of RAM because it does not waste space allocating voxels for empty bins.

of about 52 degrees works best for our application. On average, the system uses 18.658 msec per frame to compute the filter mask.

6.3 Triangle Meshing

The serial Ball-Pivoting algorithm generates an average of 32,573 triangles per second when calculating high-quality normal vectors for the triangles using a CPU-based RANSAC plane-fitting approach [6] and 51,073 triangles per second without them. The GPU-accelerated Marching Cubes algorithm requires the high-quality normals and generates an average of 51,289 triangles per second. With the Ball-Pivoting algorithm, the density of triangles per unit volume depends on the minimum point spacing s , which we use for the ball-pivoting radius. In contrast, the triangle density of the marching cubes approach depends only on the voxel grid resolution.

Separately meshing the points in each bin creates some redundant triangles across bin edges. Applying the texture image to the triangle mesh makes these extra triangles invisible to the viewer. We can eliminate them completely by remeshing the entire point model at the end of the scanning process.

Our TextureTriangle algorithm uses an average of 0.58873 μsec to populate each pixel in the destination texture image. If each triangle uses n^2 texture pixels, the average processing time for each triangle is $0.58873 n^2 \mu\text{sec}$, as shown in Fig. (6.6). The choice of which saved RGB frame to sample from when processing each triangle affects both the algorithm's runtime and the quality of the resulting image. Sampling from the most recent RGB frames first allows all triangles in the vicinity to use the same RGB frame, reducing the effect of lighting issues in the presence of non-diffuse surfaces.

Figure 6.5: This graph shows how many points our high depth-gradient filter removes as a function of the maximum angle permitted between the camera direction and surface normal vector, over a reasonable range of maximum angles. Adjusting this parameter allows the operator to tune how “picky” the system is for adding new points into the model. Note that all values of the maximum angle shown here filter out the aforementioned depth outliers.

Figure 6.6: The time the system takes to generate the texture image for each triangle is proportional to the number of pixels in each triangle. All triangles use square-shaped slices of the texture image; the processing time rises quadratically as the operator increases the length of these squares.

Chapter 7

Conclusions

This paper has described a new 3D geometry scanning system designed to create high-quality models from multiple views of large scenes. The system uses a tracked Microsoft Kinect device to sidestep many of the problems associated with ICP-based scanning systems, including alignment in the presence of degenerate geometry. The system uses the Kinect’s infrared sensors to operate in a variety of environments without intrusively projecting visible light into the scene. Our redundant data elimination algorithms robustly generate a single surface of points using only a single core of a modern CPU, allowing the system to mesh the points into triangles and stream the 3D data to a rendering cluster in real-time. Our texturing system uses the Kinect’s RGB camera to produce a single texture image for the entire triangle mesh, producing a model that is easy to render using simple rasterization and no advanced shaders.

7.1 Limitations

Our system operates under the assumption that the scene remains static while the Kinect moves around to capture data from different poses. If a scanned object moves during the process, the scanner combines range scans from both old and new positions, producing an incorrect result. Our system’s real-time feedback mechanism helps the user capture range scans from enough angles to fully construct the model in one session, but the scene must nevertheless remain static for several

Figure 7.1: The texturing algorithm uses colors from the Kinect’s RGB camera to generate a texture for the triangle mesh. Scanning objects that reflect different colors at different angles can create visible brightness discontinuities between triangles, as shown in this render of a scanned carpet floor.

seconds. One way to overcome this limitation would be to use multiple inward-facing calibrated cameras instead of a single moving camera with real-time pose estimation.

An additional problem arises when scanning shiny surfaces where the reflected brightness depends on the viewing angle. This property causes color discontinuities in the reconstructed model which become especially pronounced when assigning textures to the triangle mesh, as shown in Figure (7.1). Furthermore, the Kinect’s RGB camera has an automatic brightness-correction feature that can cause the colors to change over time even when scanning diffuse surfaces. Explicit color normalization in the scanning system software would mitigate these effects and is probably worth considering in any future related work involving a moving camera.

Using the Kinect as a structured-light scanner provides welcome freedom from the intrusion of projecting visible light into the scene. Unfortunately, it is still subject to infrared interference from direct sunlight. Interestingly, our tracking system also uses infrared light, but we find that neither the Kinect nor the tracking cameras interfere with each other. We do not notice an interference problem from

other common sources of infrared light like remote control devices.

The texture generation algorithm selects recently-acquired RGB frames to sample color data from. The most intuitive idea is to use the RGB frames corresponding to the vertices of the current triangle, but this approach causes an unsightly brightness contrast between neighboring triangles meshed one or two frames apart from each other. We find that sampling from the most recently acquired RGB frames first can reduce the impact of this issue by forcing all triangles in a local region to use the same image. Asynchronous texture assignment generally lags five to ten frames behind the rest of the processing, giving the point scanning pass enough time to completely populate a surface before meshing and texturing it. However, problems arise if the user moves the Kinect quickly during those frames, allowing previously occluded surfaces to become visible. Doing so causes triangles to sample from incorrect image frames, although moving the scanner at a slow pace helps reduce the chance of these artifacts. Our choice of sampling from the most recent RGB frame first is a compromise that generates the best possible result given the design of our system.

7.2 Future Work

There are many possibilities for future improvement with this type of scanning system. Our redundant data elimination and triangle meshing algorithms use a single CPU core to create triangles about five millimeters wide in real-time. One could use a GPU to perform the same checks along the view direction for previous geometry near new points by rasterizing all existing primitives into a depth buffer. Comparing the depth buffer to the Kinect's depth map, one could quickly discard new samples with similar depths as the rasterized depth map. After using the tracker to project the accepted depth samples into 3D world-space points, the system could create a pair of triangles connecting each new point to its neighbors from the current frame. An asynchronous step could combine neighboring coplanar triangles and use the vertex colors to generate a texture for them in a ground-up fashion. This approach would increase the triangle mesh resolution, especially at

high-gradient areas. It would also allow the system to use each RGB frame’s colors immediately, preventing the Kinect’s brightness correction from causing a high brightness contrast between neighboring triangles.

An interesting computer vision-based idea would be to rely completely on ARToolkit to provide the camera pose. The user could scatter markers about the scene, then ensure that at least one marker is always visible in each frame. Setting the world-space origin using the first marker, the system could operate until two markers appear within an image frame and then store the world-space pose of the second marker relative to the first. Such a system would not rely on any extrinsic calibration at all, as it would discover the marker positions anew each session. Other proposed scanning systems use computer vision-based feature detection to establish correspondences between images, but these systems quickly accumulate error as the user moves the scanner away from the origin. On the other hand, ARToolkit’s markers would provide guaranteed high-quality features in the scene, and their known size would allow the system to store scanned data in real-world metric units.

The real-time range scanning system acquires more than 40 MB/sec of data. Instead of immediately adding each new depth map to the scene, future work could implement error correction from data in consecutive frames to obtain averaged point colors. This could address the problem of brightness discontinuities in surfaces with varying brightness by averaging colors over multiple scans. Performing such a temporal merging of the point colors using a Gaussian filter would be relatively inexpensive, but obtain a visible increase in color quality for scans of highly specular surfaces.

Bibliography

- [1] AR Tracking GmbH.
<http://ar-tracking.eu>.
- [2] MeshLab.
<http://meshlab.sourceforge.net>.
- [3] Nicolas Burrus' Kinect Calibration.
<http://nicolas.burrus.name/index.php/Research/KinectCalibration>.
- [4] NVIDIA's CUDA Implementation of Marching Cubes.
http://developer.download.nvidia.com/compute/cuda/1.1/Website/Graphics_Interop.html.
- [5] OpenKinect.
<http://openkinect.org>.
- [6] The Mobile Robot Programming Toolkit - Randomized Sample Consensus C++ Examples.
http://www.mrpt.org/RANSAC_C++_examples.
- [7] Brett Allen and Brian Curless. The space of human body shapes : reconstruction and parameterization from range scans. *Engineering*, 2003.
- [8] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.
- [9] F. Bernardini, J. Mittleman, H. Rushmeier, C. Silva, and G. Taubin. The ball-pivoting algorithm for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):349–359, 1999.
- [10] James R. Bitner, Gideon Ehrlich, and Edward M. Reingold. Efficient generation of the binary reflected gray code and its applications. *Communications of the ACM*, 19(9):517–521, September 1976.
- [11] D. Caspi, N. Kiryati, and J. Shamir. Range imaging with adaptive color structured light. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(5):470–480, May 1998.

- [12] D. Chetverikov, D. Svirko, D. Stepanov, and P. Krsek. The Trimmed Iterative Closest Point algorithm. *Object recognition supported by user interaction for service robots*, pages 545–548, 2002.
- [13] T Defanti, G Dawe, D Sandin, J Schulze, P Otto, J Girado, F Kuester, L Smarr, and R Rao. The StarCAVE, a third-generation CAVE and virtual reality OptIPortal. *Future Generation Computer Systems*, 25(2):169–178, February 2009.
- [14] O. Hall-Holt and S. Rusinkiewicz. Stripe boundary codes for real-time structured-light range scanning of moving objects. *Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001*, pages 359–366, 2001.
- [15] H. Kato and M. Billinghurst. Marker tracking and HMD calibration for a video-based augmented reality conferencing system. *Proceedings 2nd IEEE and ACM International Workshop on Augmented Reality (IWAR'99)*, pages 85–94, 1999.
- [16] J J Koenderink and a J van Doorn. Affine structure from motion. *Journal of the Optical Society of America. A, Optics and image science*, 8(2):377–85, March 1991.
- [17] Richard A Newcombe and Andrew J Davison. Live Dense Reconstruction with a Single Moving Camera. *Structure*, 2010.
- [18] Timothy S. Newman and Hong Yi. A survey of the marching cubes algorithm. *Computers & Graphics, Vol. 30, Iss. 5*, 2006.
- [19] S. Rusinkiewicz and M. Levoy. Efficient variants of the ICP algorithm. *Proceedings Third International Conference on 3-D Digital Imaging and Modeling*, pages 145–152, 2001.
- [20] Szymon Rusinkiewicz, Olaf Hall-Holt, and Marc Levoy. Real-time 3D model acquisition. *ACM Transactions on Graphics*, 21(3), July 2002.
- [21] D.a. Simon, M. Hebert, and T. Kanade. Real-time 3-D pose estimation using a high-speed range sensor. *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, 2(1):2235–2241, 1994.
- [22] Noah Snavely, Steven M. Seitz, and Richard Szeliski. Modeling the World from Internet Photo Collections. *International Journal of Computer Vision*, 80(2):189–210, December 2007.
- [23] Oliver Wulf and Bernardo Wagner. FAST 3D SCANNING METHODS FOR LASER MEASUREMENT SYSTEMS. *Scanning*, (section 2):3–8, 2003.

- [24] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. Real-time KD-tree construction on graphics hardware. *ACM Transactions on Graphics*, 27(5):1, December 2008.
- [25] T. Zinsser, J. Schmidt, and H. Niemann. A refined ICP algorithm for robust 3-D correspondence estimation. *Proceedings 2003 International Conference on Image Processing (Cat. No.03CH37429)*, pages II-695–8, 2003.