Capturing Geometry in Real-Time Using a Tracked Microsoft Kinect

Daniel Tenedorio, Marlena Fecho, Jorge Schwartzhaupt, Robert Pardridge, James Lue, Jürgen P. Schulze

University of California, San Diego

(a) Photograph (b) Interactive Scanning Preview (c) Textured Triangle Mesh

Figure 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 during the scanning process to allow the user to find and fill holes in real time.

ABSTRACT

We investigate the suitability of the Microsoft Kinect device for capturing real-world objects and places. Our new geometry scanning system permits the user to obtain detailed triangle models of non-moving objects with a tracked Kinect. 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. We demonstrate the ability of our system to acquire 3D models at human scale with a prototype implementation in the StarCAVE, a virtual reality environment at the University of California, San Diego. We designed the capturing algorithm to support the scanning of large areas, provided that accurate tracking is available.

1. INTRODUCTION

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 endeavor to construct high-resolution models with texture-mapped primitives using inexpensive, off-the-shelf 3D scanning technology.

Most range scanners record data from a single stationary view. To build a complete 3D model, the user must move a 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, as shown in Fig. (1).

We propose the use of a tracked Microsoft Kinect device to perform these steps and 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. The goal of this article is to present our prototype implementation, which makes several novel contributions to the science of 3D model acquisition, and use it to analyze the suitability of the Kinect for 3D scanning.

In Sections 2 and 3, we perform a rigorous analysis of the Kinect's capabilities in the context of a general 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 4, 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 5, we demonstrate that it is possible to perform triangle meshing and texturing from the Kinect's RGB camera in real-time. In Section 6, 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 present ideas for future research in Sections 7 and 8.

2. PREVIOUS WORK

Computer vision scientists have developed techniques to infer structure from motion using multiple images of a non-moving scene from a single moving camera.[17,23] 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 off-line 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.[18]

On the other hand, range scanning systems based on projected structured-light patterns can be efficient and robust.[11,12,15] 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.[13,20,22,26] 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.[24] A third class of alignment methods involves placing markers on the subject and using their images to transform scans relative to each other.[8] There are commercial 3D scanning systems on the market based on this approach.[2] These systems can be expensive, and require the user to intrusively place markers in the scene prior to scanning.

In contrast to these approaches, our proposed system uses a tracker to record the real-time pose of the Kinect, allowing us to align the range scans into a global model efficiently and accurately. The user is free to move the scanner at any speed, and the system records scans without the need for putting any additional objects into the scene, even if the scanned subject lacks texture or contains planar or cylindrical surfaces.

3. 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. It 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. 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.

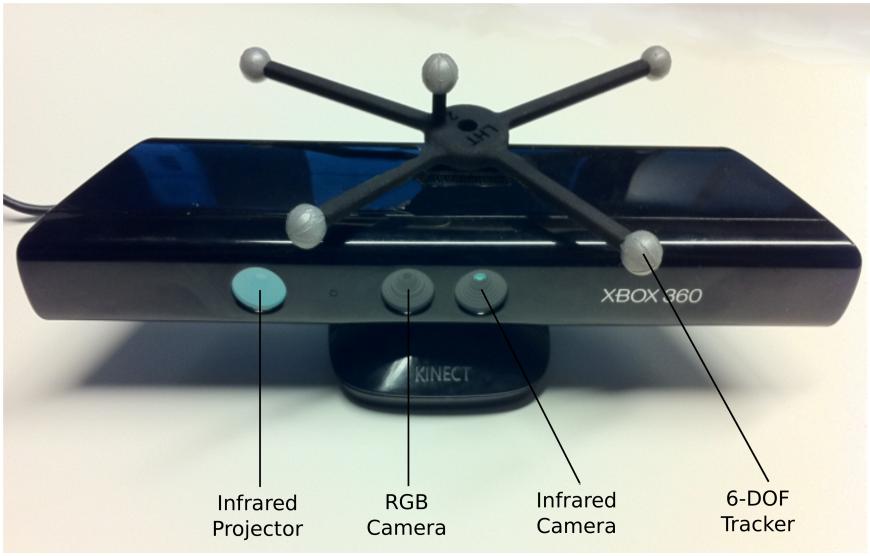


Figure 2: The projector on the left casts a structured pattern of infrared dots onto the scanned object. The infrared camera on the right creates a map of depth values using the dots' projected locations. The middle RGB camera records standard video frames. The tracking sensor obtains the Kinect's real-time camera pose.

3.1 Kinect Hardware

Our system uses data from both the color and depth cameras in the Kinect (see Fig. (2)). The infrared projector illuminates the scene with a structured dot pattern, which the infrared camera 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.

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 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.[4]

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.

3.2 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. We suspect that this is because the pulsed light used for tracking is not bright enough to wash out the Kinect's patterns.

3.3 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.[16] ARToolkit's list of supported cameras does not include the Kinect, but we were able to modify an existing configuration for a VGA webcam 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 marker's 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. The ARToolkit approach will allow us to scan much larger areas than our optical tracking system covers.

3.4 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.[6] 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:

$$\Pi^{-1}(\lambda\,x') = \Pi^{-1}\,\Pi X_0$$
$$\Pi^{-1}(\lambda\,x') = X_0$$

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:

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

4. REDUNDANT DATA ELIMINATION

Our real-time scanning system accumulates data rapidly as it aligns 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,[21] 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. We address these issues with the following redundant point-elimination algorithms and data structures.

4.1 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.[9,25] 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 m^3 and the points stored are no closer together than r m, then it is likely that $x \leq (\frac{b}{r})^2$ if we assume that each bin contains a 2D surface of points. We examine an interesting way to enforce this assumption in the next subsection.

4.2 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 frame rates, 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)). 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. (4).

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.

4.3 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.

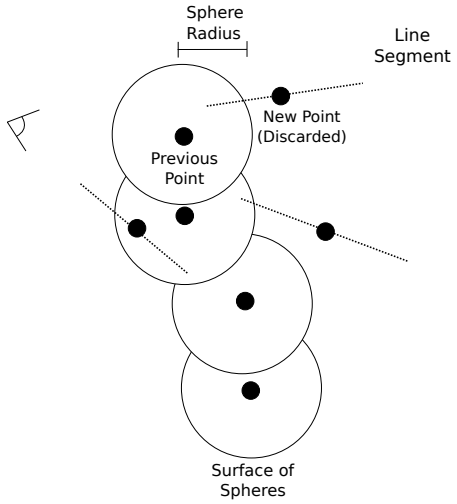


Figure 3: For each new 3D point, we perform an intersection test between spheres about nearby points and a short line segment in the view direction. When the test fails, we merge the new point into the old one, preventing a buildup of multiple layers due to noise or miscalibration. A surface of spheres 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 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 4: 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 $\|L\|$.

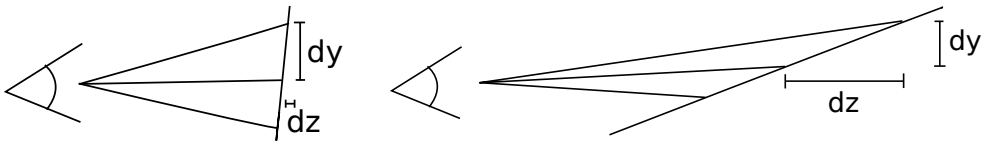


Figure 5: The system performs filtering by setting a maximum allowed ratio of change in depth (dz) to change in tangent direction (dx or dy). Surfaces that are perpendicular to the Kinect have a relatively low ratio, whereas more parallel surfaces have a higher ratio.

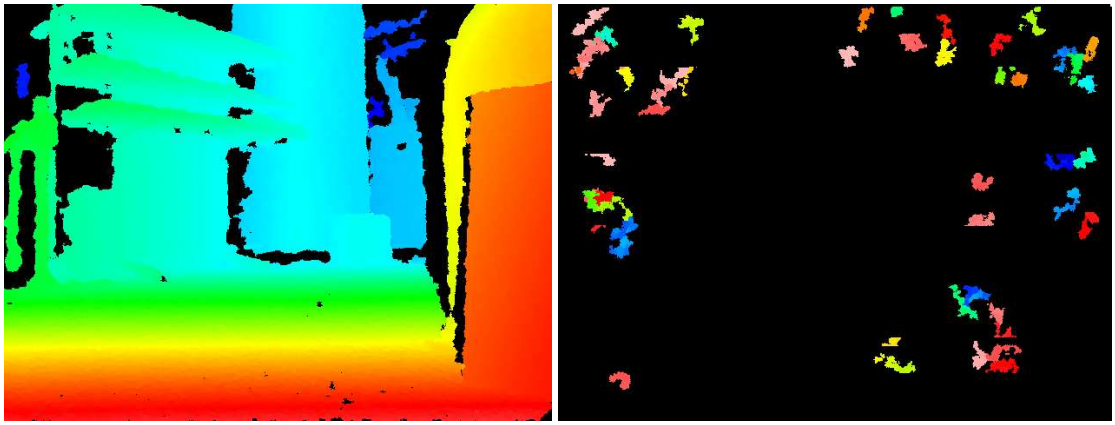


Figure 6: The left image is a visualization of the depth map from a Kinect held at proper range, using example code from libfreenect.[6] The right image uses the same software to illustrate how the depth map becomes corrupted when the scanned object is too close. Our high depth-gradient filter successfully removes these artifacts before processing the depth map each frame.

Fig. (5) shows how the algorithm detects surfaces of high depth-gradient. 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. (6).

We note that Rusinkiewicz, Hall-Holt, and Levoy have proposed an alternative offline outlier elimination method based on removal of skinny triangles.[21] 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.

5. 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 off-line meshing using a software package like Meshlab.[3] 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.

5.1 Ball-Pivoting

Bernardini, et al. proposed the original "Ball-Pivoting Algorithm for Surface Reconstruction".[10] 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

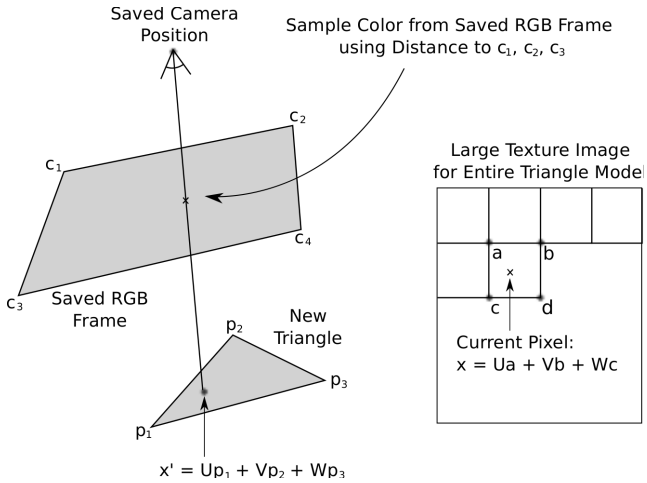


Figure 7: The system sets color values for a square-shaped portion of the texture image for each new triangle. It begins by calculating barycentric coordinates (U, V, W) for each texture pixel with respect to three of the four of the square's corners (a, b, c, d). It then calculates an interpolated position 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.

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 used the ball-pivoting implementation from the VCG library, the same library that powers Meshlab,[3] using a ball-pivoting radius equal to three times the sphere radius described in the above `AddPoint` algorithm.

5.2 Marching Cubes

As an alternative triangle meshing approach, we integrated nVidia's CUDA implementation of the Marching Cubes algorithm.[5,19] 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.[7] 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.

5.3 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.

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. (7) illustrates the process of sampling from the RGB planes to fill the pixels in the texture.

6. USER INTERFACE AND STARCAVE DISPLAY

6.1 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.[13,20,22,26] We find that in addition to producing more robust pose estimation, the system is capable of calculating

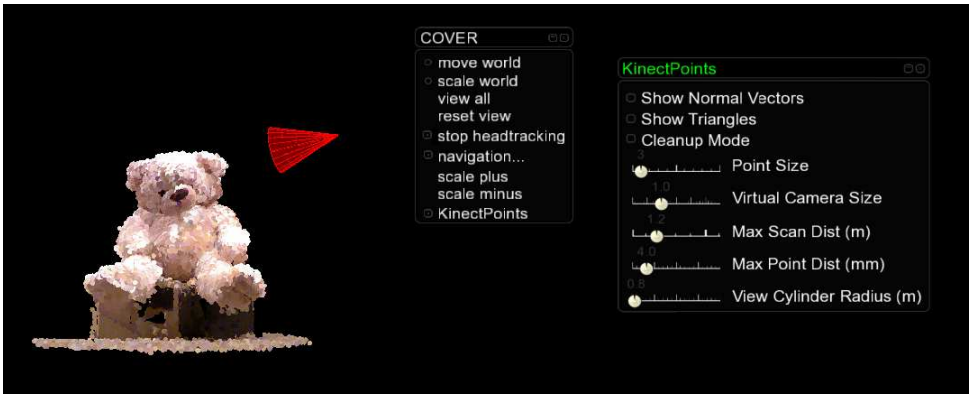


Figure 8: This screenshot illustrates how the tracker permits development of useful user-interface features. The cone visualizes the Kinect's pose relative to the scanned model at all times. The color of the cone indicates the scanning mode: red for scanning on, purple for scanning off, yellow for cleanup 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.

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. (8) 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. Pressing a different button on the same controller triggers an undo feature, removing all points scanned within the past several frames from the model. 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.

6.2 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.[14] 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. 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.

7. 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.

(a) ARToolkit

(b) Tracking System

(c) Photograph

Figure 9: 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.

7.1 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. (9). Note that the camera must be able to see at least one marker in each image for ARToolkit to work. In Fig. (10), we juxtapose a photograph of our stuffed bear with some closeup renderings of generated triangle models.

7.2 Redundant Data Elimination Algorithms

We used a world space of ± 10 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 over a world size of $\pm w$ m^3 with bin size of b m^3 uses $2\left(\frac{w}{b}\right)^3$ bytes of RAM, as shown in Fig. (11). 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.

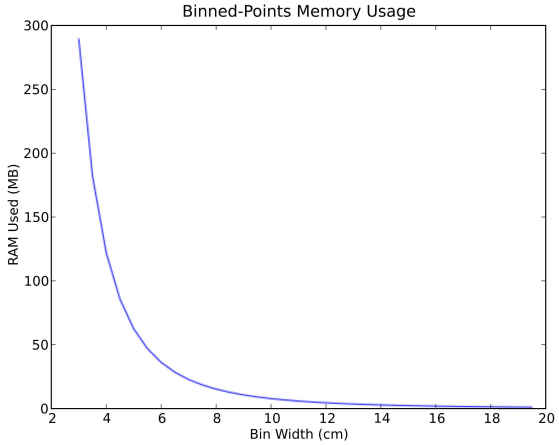
For purposes of comparison with previous work,[20] we implemented a feature wherein we initialize a small voxel grid within each active bin. This feature uses a reasonable amount of memory because it does not waste space on empty bins. We implement the same intersection test of a line segment parallel 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. (11), we discover that the average processing time per bin no longer rises with the bin width, as expected.

(a) Rendering 1

(b) Rendering 2

(c) Photograph

Figure 10: Comparison of a photograph a stuffed bear with two renderings of textured triangle meshes obtained using our scanning system. In Rendering 1, 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 Rendering 2, borders between triangles are more visible and there are some artifacts from the bear's fur reflecting different brightnesses at different times.



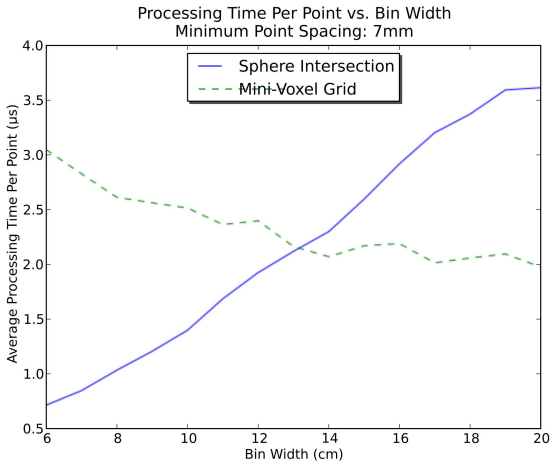


Figure 11: The first graph shows how memory usage is inversely proportional to the bin size. In the other graph, we see that as the bins get larger, the average number of points in each bin's list becomes longer. Since we have to examine them all before adding a new point to any bin, the processing time 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 $2-3\,\mu$m. The latter approach does not consume unreasonably large amounts of RAM because it does not waste space allocating voxels for empty bins.

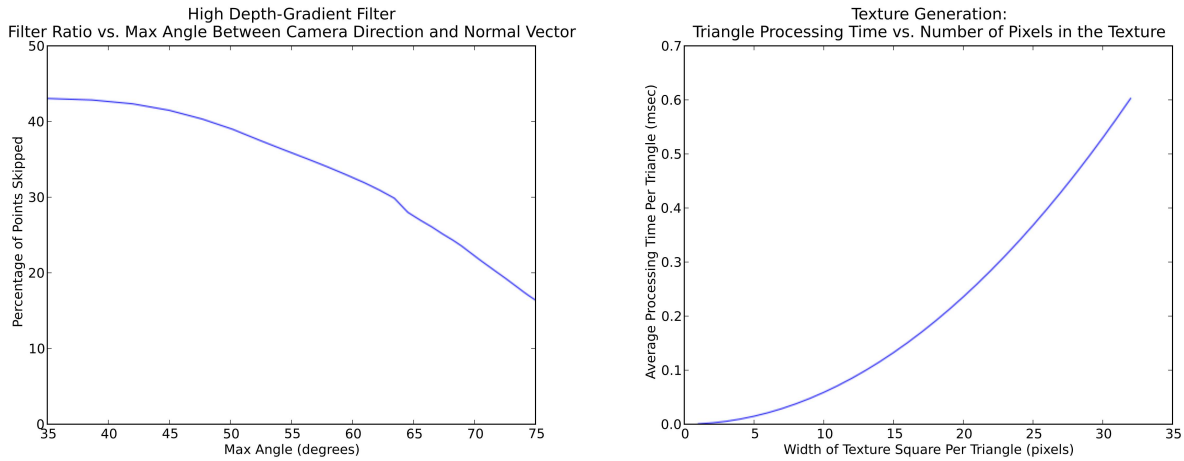


Figure 12: The graph on the left shows how many points our high depth-gradient filter removes as a function of the maximum angle permitted between the camera direction and surface normal vector. Adjusting this parameter allows the operator to tune how "picky" the system is for adding new points into the model. In the graph on the right, we see that 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.

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. (12). We found that a maximum angle of about 52 degrees works best for our application. On average, the system uses 18.658 msec per frame to compute the filter mask.

7.3 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[7] 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.

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 µsec, as shown in Fig. (12). 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.

8. 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.

8.1 Limitations

Problems arise 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. The Kinect's RGB camera also 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.

High-resolution texture generation tends to amplify the effects of miscalibration and noise in the Kinect and tracking system. Unlike 3D reconstruction systems that use the range scans or RGB images for pose estimation,[13,17,20,22,23,26] the tracking-based system must extrinsically match range scans to each other. This can cause visible discontinuities in the texture image on triangle boundaries. Interpolating colors from several recent RGB frames can address this issue, but may produce a blurry texture image in the case of noisy pose estimation.

8.2 Future Work

Our prototype scanning application for the StarCAVE shows that we can capture static 3D objects at reasonably high quality with a hand-held, tracked Kinect, while viewing the meshed and textured result in real-time. After creating the prototype using the highly accurate, commercial tracking system in the StarCAVE, we integrated support for tracking using ARToolkit markers. Our software is designed to capture areas much larger than the StarCAVE, so that in the future we can scan archaeological excavation sites and other large areas.

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.

An interesting low-cost idea would be for the user to scatter ARToolkit markers about the scene, then rely completely on ARToolkit to provide the camera pose. 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. 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.

REFERENCES

[1] AR Tracking GmbH.
http://ar-tracking.eu.

[2] Creaform 3D: Handyscan 3D Scanners.
http://www.creaform3d.com.

[3] MeshLab.
http://meshlab.sourceforge.net.

[4] Nicolas Burrus' Kinect Calibration.
http://nicolas.burrus.name/index.php/Research/KinectCalibration.

[5] NVIDIA's CUDA Implementation of Marching Cubes.
http://developer.download.nvidia.com/compute/cuda/1_1/
Website/Graphics_Interop.html.

[6] OpenKinect.
http://openkinect.org.

[7] The Mobile Robot Programming Toolkit - Randomized Sample Consensus C++ Examples.
http://www.mrpt.org/RANSAC_C++_examples.

[8] Brett Allen and Brian Curless. The space of human body shapes : reconstruction and parameterization from range scans. *Engineering*, 2003.

[9] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] Richard A Newcombe and Andrew J Davison. Live Dense Reconstruction with a Single Moving Camera. *Structure*, 2010.

[19] Timothy S. Newman and Hong Yi. A survey of the marching cubes algorithm. *Computers & Graphics, Vol. 30, Iss. 5*, 2006.

[20] 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.

[21] Szymon Rusinkiewicz, Olaf Hall-Holt, and Marc Levoy. Real-time 3D model acquisition. *ACM Transactions on Graphics*, 21(3), July 2002.

[22] 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.

[23] 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.

[24] Oliver Wulf and Bernardo Wagner. FAST 3D SCANNING METHODS FOR LASER MEASUREMENT SYSTEMS. *Scanning*, (section 2):3–8, 2003.

[25] 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.

[26] 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.