3D Video Recorder

Stephan Würmlin[1] Edouard Lamboray[1] Oliver G. Staadt[2] Markus H. Gross[1]

[1]Computer Science Department
Swiss Federal Institute of Technology (ETH)
Zurich, Switzerland
{wuermlin, lamboray, grossm}@inf.ethz.ch

[2]Computer Science Department
University of California, Davis
Davis, CA
staadt@cs.ucdavis.edu

Abstract

We present the 3D Video Recorder, a system capable of recording, processing, and playing three–dimensional video from multiple points of view. We first record 2D video streams from several synchronized digital video cameras and store pre-processed images to disk. An off-line processing stage converts these images into a time–varying three–dimensional hierarchical point–based data structure and stores this 3D video to disk. We show how we can trade–off 3D video quality with processing performance and devise efficient compression and coding schemes for our novel 3D video representation. A typical sequence is encoded at less than 7 megabit per second at a frame rate of 8.5 frames per second. The 3D video player decodes and renders 3D videos from hard–disk in real–time, providing interaction features known from common video cassette recorders, like variable–speed forward and reverse, and slow motion. 3D video playback can be enhanced with novel 3D video effects such as freeze–and–rotate and arbitrary scaling. The player builds upon point–based rendering techniques and is thus capable of rendering high–quality images in real–time. Finally, we demonstrate the 3D Video Recorder on multiple real–life video sequences.

1 Introduction

Conventional two–dimensional video is a mature technology in both professional environments and home entertainment. A multitude of analog and digital video formats is available today, tailored to the demands and requirements of different user groups. Efficient coding schemes have been developed for various target bit rates, ranging from less than 100 kilobit per second for video conferencing applications to several megabit per second for broadcast quality TV. All of these technologies, however, have in common that they are only capable of capturing temporal changes of scenes and objects. Spatial variations, i.e., altering the viewpoint of the user, are not possible at playback. 3D video captures dynamics and motion of the scene during recording, while providing the user with a possibility to change the viewpoint during playback.

Spatio-temporal effects (e.g., *freeze–and–rotate*) have been demonstrated in numerous recent feature films. However, these effects can only be realized by employing a large number of still cameras, and involve a considerable amount of manual editing (http://www.mvfx.com/). Typically, input data is processed off–line in various ways to create a plenitude of stunning effects. A three-dimensional video system could capture and process this data without manual intervention in shorter time and at lesser cost.

The approach presented in this paper is a generalization of spatio–temporal or 3D video. We present algorithms and data structures for acquisition and construction of view-independent time–varying video. Each 3D video frame is stored in a three–dimensional point–based data structure which allows for direct rendering. We present a 3D video player supporting multi-viewport rendering of pre–recorded dynamic scenes in real–time. Moreover, we adapt standard video cassette recorder interaction features, such as variable-speed forward and reverse and slow motion, and introduce novel 3D video effects such as freeze–and–rotate, or arbitrary scaling.

The architecture of the 3D Video Recorder reflects two main performance constraints of any video recorder: Acquisition of input data as well as playback of 3D video have to be accomplished in real–time. However, processing 3D video from acquired image data can be carried out off-line without real-time constraints. After the acquisition of multiple video streams, we reconstruct 3D information for every pixel in the input images and store this representation using a hierarchical, point–based data structure. Although our framework does not build upon a specific reconstruction scheme, we currently use a shape–from–silhouette method to extract positional and normal information. Other reconstruction schemes, such as voxel coloring, multi-baseline stereo, or active vision systems [4], can be used instead. In general, our framework is able to build a 3D video including fore- and background. The prototype presented in this paper, however, aims at generating 3D video from objects, i.e. humans, and not from scenes. The 3D video player decodes the compressed 3D video progressively from disk and renders the 3D video object employing efficient point-based rendering schemes.

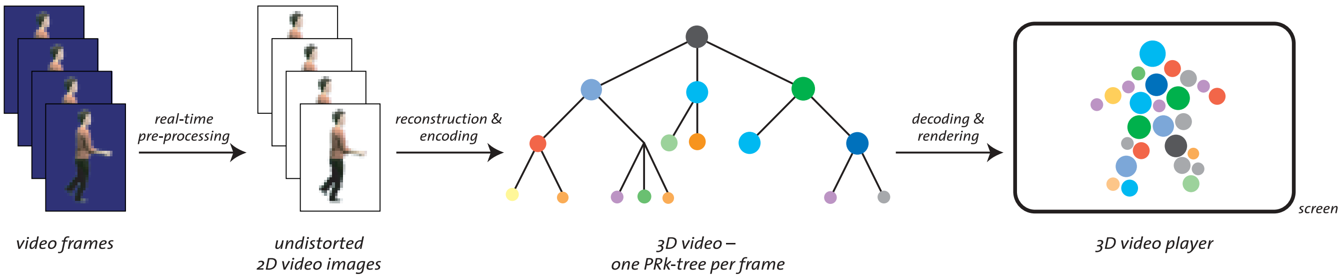


Figure 1: System Architecture of the 3D Video Recorder. The pipeline illustrates the process for one frame.

The main contributions of this paper can be summarized as follows:

- A general framework for 3D video recording comprising acquisition, processing, and rendering of time–varying three–dimensional scenes.
- An efficient processing pipeline to reconstruct view–independent 3D video with a user–controlled trade-off between processing time and 3D video quality.
- Efficient algorithms for encoding and compressing the resulting progressive 3D video.
- A full-featured 3D video player built upon a real-time point-based rendering framework.

1.1 Previous Work

View–dependent 3D video representations and systems are still an emerging technology, and various approaches have been developed in the past.

Kanade et al. [11] and Narayanan et al. [18] employ a triangular texture–mapped mesh representation. Multi-baseline stereo techniques extract depth maps from a fixed set of camera positions, which requires significant off-line processing. At rendering time, depth information from one or more cameras closest to the new virtual view are used to construct the mesh. An edge-collapse algorithm is used for mesh simplification. For the *EyeVision* system, based on Vedula [30], a large number of video cameras, distributed in a football stadium, are used for capturing sports events from different views. Replays are constructed from interpolated views, both in space and time.

Mulligan and Daniilidis [17] presented an approach utilizing trinocular stereo depth maps. Their system is used in a tele-immersive conference where advanced off-line processing and storage as well as compression issues have not been addressed. Pollard and Hayes [21] utilize depth map representations to render real scenes from new viewpoints by morphing live video streams. Pixel correspondences are extracted from video using silhouette–edge information. This representation, however, can suffer from inconsistencies between different views.

The image-based visual hull (IBVH) system, introduced by Matusik et al. [15], is taking advantage of epipolar geometry [6] to build an LDI [28] representation. Color information is determined using nearest-neighbor interpolation. Thus, depth–color alignment cannot be guaranteed. While the resulting LDI is neither progressive nor view–independent, it allows for free scaling and freezing. Due to the fact that it is a real–time system, free spatio-temporal effects (e.g., freeze–and–continue) cannot be accomplished. The polyhedral visual hull [14] also builds upon an epipolar geometry scheme for constructing a triangular representation of an object. While this enables viewpoint–independent rendering, the limitations of mesh-based methods persist: The potentially improper alignment of geometry and texture, and the non-trivial realization of progressive streaming of dynamically changing polygonal meshes.

Other approaches are based on volumetric object representations. Moezzi et al. [16] create view–independent 3D digital video from multiple video sequences. The dynamic information from each video is fit into a voxel model and combined with a pre-generated static model of the scene. Yemez and Schmitt [32] introduce an octree-based particle representation of 3D objects, which they use for multilevel object modeling, storage, and progressive transmission. However, they transform the octree particles into a triangle mesh before rendering and do not address compression issues.

Rusinkiewicz and Levoy presented Streaming QSplat [26], a view-dependent progressive transmission technique for a multi–resolution rendering system, which is based on a bounding sphere hierarchy data structure and splat rendering [25]. Their framework is efficient for streaming and rendering large but static data sets.

1.2 Overview

The following section describes the overall architecture of the 3D Video Recorder. Section 3 presents recording and preprocessing of the 2D video input streams. Section 4 describes the different steps comprising the 3D video processing pipeline, including the hierarchical data representation of 3D video and compression. The 3D video player is introduced in Section 5. Our prototype acquisition environment is presented in Section 6, followed by a discussion of experimental results in Section 7. Finally, Section 8 concludes the paper and discusses some ideas for future work.

2 System Architecture

A schematic overview of the 3D Video Recorder is depicted in Figure 1. As opposed to a traditional 2D home video recorder comprising only two stages—recording and playback—the 3D Video Recorder features an additional stage: processing. It is clear that recording and playing need to be carried out in real–time. There are no hard real–time constraints for the off-line processing stage. According to Deering [5], spending 60x more time on off-line processing than on on-line decompression is still acceptable. A 3D–replay application in a broadcasting studio has stronger time constraints, since the replay needs to be available for broadcast only 10–20 seconds after the live action. The ultimate goal of a 3D video recorder is to process 3D information within these time limits.

The key elements of the 3D Video Recorder stages can be summarized as follows:

- **2D Video Recording.** (Section 3) Video images from multiple cameras are streamed to disk in real–time. The images are undistorted and segmented on-the-fly.
- **3D Video Processing.** (Section 4) The key task of the processing stage is the reconstruction of 3D positional information for every foreground pixel of the input images. Optionally, we can calculate a surface normal for every 3D point. Since we employ a shape–from–silhouette algorithm whose performance largely depends on the approximation of the silhouette contours, the user can control the accuracy of the approximation to trade–off performance and quality. The reconstructed 3D points from all cameras are merged in a hierarchical data structure, the PR*k*-tree, which is a variant of an octree. This tree is encoded, compressed, and stored to disk.
- **3D Video Playing.** (Section 5) The decoded sequence is displayed using a point–based rendering framework. We use efficient splatting schemes for rendering a continuous surface of the 3D video object. The 3D video player incorporates features known from 2D video such as play, pause, fast–forward, fast–reverse, and slow motion. Moreover, it also integrates novel video effects like arbitrary scaling, panning, tilting, and freeze–and–rotate.

3 2D Video Recording

Recording multiple synchronized live video streams requires preprocessing and file streaming techniques that run at interactive rates. Since we need to store several NTSC-resolution video sequences, we have to reduce the data volume during acquisition. For that purpose, we did not consider lossy image or video compression schemes, such as JPEG or MPEG, because we did not want to compromise the 3D video reconstruction by introducing compression artifacts in the 2D video sequence. As an alternative, we carry out image segmentation, which is necessary for separating the foreground object from the background. Based on the segmented images, we only record the area-of-interest for each frame. This results in storing a rectangular subimage which contains the complete view of the object, together with offset position and dimensions of the area-of-interest. For typical sequences, we can thereby reduce the size of the 2D video stream by 50%. Off-line entropy coding of the 2D video sequences additionally results in 30-50% compression gain.

The employed segmentation method is built on traditional chroma–keying [31, 29]. We can classify an object in front of a blue background as foreground if

$$B_b \le a_2 \cdot G_b , \qquad (1)$$

where a_2 is an arbitrary constant between $0.5 \le a_2 \le 1.5$, and B_b and G_b are the blue and the green chroma channels of the video pixel, respectively. Note that we only need to consider a binary decision for separating foreground and background pixels. We do not need to calculate an α-value, as in traditional matting. One problem arising from this approach is color spill near object silhouettes [29], i.e., the reflection of background color on the foreground object due to increasing specularity near grazing angles. This leads to poor image quality when merging the points together from multiple cameras as discussed in Section 4.2. Since we oversample the object in a multi–camera setup, we solve this problem by simply shrinking the mask by a few pixels for reconstruction.

Another issue that leads to severe artifacts when merging 3D points from multiple cameras is also tackled at this stage: The use of lenses with short focal length results in video frames with heavy distortions. A 4.5 millimeters lens, for example, can lead to displacements of up to 20 pixels in the periphery of the image. Our point-merging framework relies on the assumption of point correspondences between frames from different cameras. This assumption does not hold anymore when the video images are distorted. Thus, we need to undistort the image before further processing. We employ the Open Computer Vision Library (http://sf.net/projects/opencvlibrary/) for undistortion, which is based on a pre–computed displacement look–up table and thus reasonably fast.

4 3D Video Processing

Once the 2D video sequences are recorded, we start the 3D video generation. Since our 3D video framework builds upon point–based modeling and rendering, we do not need connectivity information for rendering. The input data, i.e. video images, do not provide meaningful neighborhood information, since adjacent pixels on the image plane are not necessarily adjacent on the surface of the sampled object. Since we associate a 3D point with every 2D pixel, 3D video processing provides a one–to–one mapping between the color of 2D pixels in the input images and the corresponding 3D points. Consequently, we can avoid interpolation and alignment artifacts introduced by mesh–based representations. At the end of the processing stage, the 3D points are encoded, compressed and streamed to disk.

4.1 3D Reconstruction

For every pixel used in the 3D video we have to compute a 3D point. Thus, we need a method which projects a pixel from two–dimensional image space into three–dimensional object space. In our implementation we utilize a variant of the image–based visual hulls (IBVH) [15]. Originally, depth and normal information are calculated for each pixel for a *desired view* different from the reference views of the cameras. In our approach, we carry out the reconstruction for each of the *camera views*. This ensures a one–to–one mapping between depth information and texture color information. The IBVH method computes a volume known as the visual hull, which is the maximal volume consistent with all input silhouettes [12]. The visual hull is computed in such a manner that quantization and aliasing artifacts imposed by volumetric approaches are removed.

For computing image–based visual hulls we determine a depth interval for 3D rays originating from the camera's center–of–projection and going through every pixel. Each ray is projected onto the image planes of the other cameras. Finally, we intersect the projected ray with all silhouette contour edges and project the resulting interval back onto the original 3D ray. The performance bottleneck is the 2D line–line intersection in the image planes which depends on the number of silhouette contour edges. We devise a novel multi–scale contour extraction method which leads to a user–controllable trade–off between performance and resulting image quality. Note that the contour extraction is computed on the original silhouette mask rather than on the shrunken silhouette as explained in Section 3.

Contour Extraction. Our contour extraction algorithm marches the pixels of silhouette boundaries and stores edge endpoints. A user–controlled parameter determines the accuracy of a contour and is a threshold for permitted direction changes along the approximation of a silhouette edge. Hence, larger thresholds result in coarser approximations of the edge and, thus, a smaller number of contour edges.

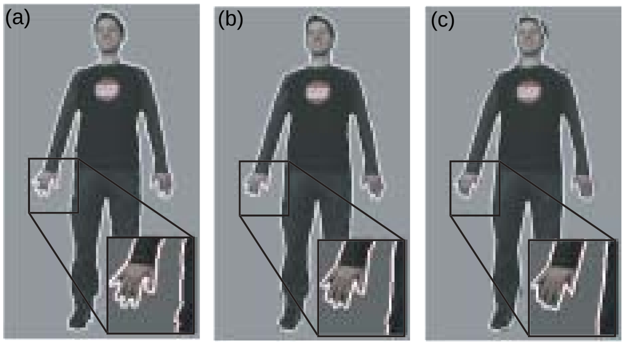


Figure 2: Different scales in the contour extraction. (a) an exact contour with 2548 points, (b) an approximation with 221 points (threshold=5), and (c) an approximation with 94 points (threshold=15).

One special property of our extraction method is that it approximates the silhouette very well along curved boundaries, while only few edges are required for straight regions. The threshold used for describing a person's silhouette is typically in the range of 0, which results in an exact contour, to 15, which is leading to a total number of edges between 2500 and 50. For a more precise discussion of the performance–quality trade-off, see Section 7.

4.2 PR*k* –trees

The generation of 3D points for every camera view still leads to a view–dependent 3D video. The support of arbitrary viewpoints, however, requires a view–independent representation. We therefore merge all 3D information in a PR*k*–tree, which is a hierarchical point–based data structure. This data structure is a variant of the point–region quadtree (PR quadtree) [27], which is very well suited for 2D point representations. The PR quadtree forms a regular decomposition of planar regions, which associates data points with quadrants. Leaf nodes are either empty or contain a single point and its coordinate. A straightforward extension for 3D points in volumetric regions is the PR octree. The main disadvantage of the PR quadtree/octree is that the maximum decomposition level largely depends on the distance between points in the same spatial region. If two points are very close, the decomposition can be very deep. This can be avoided by employing bucketing strategies, viewing leaf nodes as buckets with capacity c. Nodes are only decomposed further if the bucket contains more than c points.

Another approach for controlling the number of decompositions at each level consists in subdividing a node by an integer factor of s in each spatial direction. In the 3D case, this results in $k = s^3$ child nodes at the next level. For $s = 2$, our representation forms a PR*8*–tree, which is identical to the PR octree. By varying s, we can balance the depth of the tree for a desired number of points. Figure 3 depicts a simple 2D example of a PR*9*–tree decomposition.

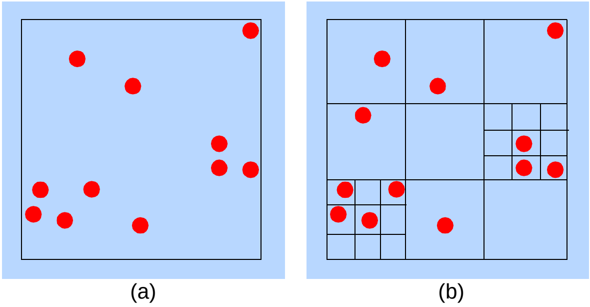


Figure 3: An example of a 2D two-level PR9–tree (s=3). Each node is decomposed into nine child nodes. a: Distribution of points in the plane at level zero. b: Decomposition of the PR9–tree at level two.

Table 1 lists the minimum, maximum, and average depths of a PR*k*–tree for real data. For this case, $s = 3$ is a good choice. The maximum depth for $s = 2$ is too large. On the other hand, the tree for $s = 4$ becomes too wide with already more than 16 million nodes at level four. Note that the maximum depth of the tree is not fixed, but depends on the sampling density of the points.

Table 1: Empirical depths of a PRk–tree for ~100,000 points. The PR27–tree (s=3) provides a good balance between number of leaf nodes and average depth.

TREE DEPTH	S = 2	S = 3	S=4
minimum	6	4	4
maximum	12	9	7
average	7	6	4

In principle, each leaf node in the PR*k*–tree stores coordinate, color, and normal information of a single point. Internal nodes of the PR*k*–tree store averaged data from their child nodes, as well as their number of children and pointers to the corresponding child objects.

A 3D PR*k*–tree represents a regular partition of a volume. If the total volume is a cube with side length s_0, the side length s_d of a partition at depth d of the PR*k*–tree equals

$$s_d = \frac{s_0}{(\sqrt[3]{k})^d}. \qquad (2)$$

Given the dimensions and configuration of the acquisition and reconstruction environment, we can compute a limit depth d_{lim}, for which the dimensions of the partitions fall below the initial camera precision. Thus, we can prune all tree nodes which are located at depth $d \geq d_{lim}$. Because of the averaged representation contained in the inner nodes, which might become leaf nodes after pruning, our strategy is similar to a bucketing strategy.

From the distance between object and camera, and from the camera resolution, we conclude that an area of approximately 3×3 mm^2 projects into one camera pixel. For our acquisition environment (see Section 6) with size $s_0 \approx 2$ m and typical configuration with $k = 27$, we have $d_{lim} = 7$.

4.3 Encoding and Compression

As elucidated in the previous section, the processing of each frame of the initial 2D image data leads to a point cloud organized in a PR*k*-tree. These trees, which represent the reconstructed 3D video frames, need to be stored using a space efficient and progressive representation. In order to achieve a progressive encoding, we traverse the tree in a breadth-first manner. Hence, we first encode the upper-level nodes, which represent an averaged representation of the corresponding subtree. As suggested in [5], a succinct storage of the 3D representation can be achieved by considering separately the different data types it contains. We distinguish between the connectivity of the PR*k*-tree, which needs to be encoded without loss, and the position of the points, color and normal information, where the number of allocated bits can be traded against visual quality and lossy encoding is acceptable.

Note that in our coder design, we also need to trade-off storage requirements, and hence coding complexity, against complexity at the decoding step, which needs to be feasible in real-time. We focus on coding strategies which provide reasonable data reduction at low decoding complexity.

The following paragraphs describe the encoding of each separate data stream:

Connectivity. In the field of tree representation, a lot of research is limited to binary trees [2]. Benoit *et al.* classify trees of higher degrees into two categories: ordinal trees, where the children of each node are simply ordered, and cardinal trees, where each node has k positions, each of which may have a reference to a child. According to this terminology, a PR*k*-tree belongs to the family of cardinal trees. Jacobson developed a simple, but asymptotically optimal scheme for coding the connectivity of ordinal trees [10]. Benoit *et al.* extended this representation to both ordinal and cardinal trees and additionally support basic navigation in constant time. The information–theoretic lower bound for the connectivity encoding of a PR*k*-tree containing n nodes and up to k children per node, n being large with respect to k, can be stated as

$$(\lg k + \lg e)n \text{ bits,} \tag{3}$$

where $\lg$ denotes the logarithm in base 2.

The algorithm presented in [2] requires

$$(\lceil \lg k \rceil + \lg \lceil e \rceil)n + O(n) \text{ bits.} \tag{4}$$

In our current application, we do not exploit the additional features of elaborated tree coding strategies, as those presented in [2]. Hence, we use 2 bits per node for encoding the raw connectivity of the PR*k*-tree. Given a node n_i, we write a 1 for every child node of n_i and a 0 to terminate the sequence. Thus, we follow Jacobson's algorithm from [10].

Position approximation. In order to achieve a complete cardinal tree encoding, we need to encode additionally the indices of the child nodes. This can be achieved by using $\lg\lceil k \rceil$ bits per child node. A better compression ratio can be achieved by using an adaptive encoding of the index c_i of node n_i. In that case, the number of bits spent on c_i is

$$\lg\lceil k \rceil \text{ for } i = 0 \quad \text{and} \quad \lg\lceil k - c_{i-1} - 1 \rceil \text{ for } 1 \leq i < k. \tag{5}$$

Note that both variants of these representations of cardinal trees are bounded by

$$(\lceil \lg k \rceil + \lg 2)n \text{ bits.} \tag{6}$$

Using this information only, $\bar{p}_i$, the centre of the volume described by node n_i, can be used as an approximation of the correct position p_i of the point primitive represented by node n_i.

Position refinement. However, the position approximation we deduce by just considering the connectivity of the cardinal tree is not precise enough. The position can further be refined by encoding the error $\Delta p_i = p_i - \bar{p}_i$ using a Laplace quantizer. We achieved good results using an 8-level quantizer. The trade-offs of this part of the encoding scheme are discussed in Section 7 and shown in Figure 8.

Leaf flag. We introduce one additional bit per node which indicates if the current node is a leaf node. This information is redundant, but it helps to simplify the progressive decoding process. We refer to Section 5.1 for further explanations.

Comparison with mesh coding. Most effort in the field of geometry compression has been concentrated on the compression of triangular meshes [5, 9, 24, 7]. State-of-the art multi–resolution geometry coders for triangular meshes achieve a compression performance between 3 and 5 bits per triangle for connectivity and 10-20 bits per vertex for its coordinates, depending on the model and the initial quantization [1, 3, 19]. Our tree-based geometry coding scheme uses a total of 17 bits per node, which is of the same order-of-magnitude, and which can be further improved by entropy coding. Yet, a tree is a more redundant data structure than a mesh. Hence, for a given 3D object, the number of nodes in its tree representation is larger than the number of vertices in its mesh representation. In practice, the number of nodes is approximately equal to 1.1-1.4 times the number of vertices.

In both cases, additional bits need to be spent on normal and color information. An extensive analysis of 3D object compression with respect to geometry, color and normal information was done by Deering [5].

Color. The hierarchical data representation cannot easily be exploited for color coding. In case of high frequency textures, the color values of two neighboring points can be completely different. Hence we decided to use a straightforward quantization scheme for the color coding, and, at this point, do not yet exploit eventual redundancies in the PR*k*-tree representation. The colors are encoded in YUV format, and we achieved visually appealing results using 6 bits for Y and 3 bits for U and V respectively. During decoding however, we need to transform the color into RGB space for rendering purposes. Storing the color information already in RGB format would simplify the decoding process, but the compression ratio, respectively the reconstruction quality would decrease. By using twice as much bits for the Y component than for the U and V components respectively, we follow the recommendation of the well established 4:2:2 format in traditional 2D video coding.

Normals. The normal vectors are encoded using quantized spherical coordinates. We normalize the vectors before encoding and then allocate 4 bits for each of the two angles. In general, the use of 2 bytes for normal quantization is recommended [5, 25], however, the quality of our current normal computation is not exploiting a larger bit budget than 1 byte per node.

Table 2: Memory requirements for one PR27 node.

NAME	DATA TYPE	RAW [BITS]	COMPRESSED [BITS]
position	float[3]	$3 \cdot 32$	$3 + 3 + 3$
color	char[3]	$3 \cdot 8$	$6 + 3 + 3$
normal	float[3]	$3 \cdot 32$	8
numberOfChildren	unsigned char	8	$2 + 1 + \lceil \lg 27 \rceil$
children	*PRkNode	$27 \cdot 32$	
Total		1088	37

Table 2 summarizes the storage requirements for the different data types per node and compares them to the initial data size. For the lossless encoding of the connectivity of the PR*k*-tree, we use a scheme coming close to the information theoretic bound. The indicated values for the remaining data types are those which provided us with visually appealing results.

Inter-frame coherence. Consecutive frames in a 3D video sequence contain a lot of redundant information, i.e., regions of the object remaining almost static, or, changes which can be efficiently encoded using temporal prediction and motion compensation algorithms. These techniques are commonly used in 2D video compression. However, the efficient computation of 3D scene flows is non-trivial. Previous efforts into this direction predict for our current prototype configuration 30 seconds of computation time for the scene flow per frame [30]. Nevertheless, exploiting inter-frame coherence based on the analysis of 3D displacements of all points, together with color and normal encoding exploiting the hierarchical data structure, can certainly improve the compression performance of our coding scheme.

Entropy coding. After the encoding of a given frame, we get six separate bit streams, describing connectivity, position approximation, position refinement, leaf flags, colors and normals. Each of these bit streams is further compressed by an entropy coder. In our current implementation, we use the range coder provided by Schindler (http://www.compressconsult.com/rangecoder). Range coding is similar to arithmetic coding [23], but is about twice as fast for a negligible decrease in compression performance [13]. The main difference lies in the renormalization step, which is bitwise in classical arithmetic coding and bytewise in range coding.

File format. Currently, we save the different bit streams into separate files. For each frame and bit stream, we store the compressed bit stream, preceded by a frame identifier and the compressed bit stream length in bytes. This format allows for navigation in the sequence and does not impose any restrictions on the order in which the frames are decoded.

5 3D Video Playing

The 3D video player is the final stage of the 3D Video Recorder framework. It decodes 3D video sequences from disk and renders individual points by employing a point–based splatting technique. Progressive rendering is implemented not only by splatting individual points at leaf nodes, but also by using *averaged points* from intermediate tree nodes. Frames and quality levels are controlled by user interaction and desired frame rate.

5.1 Decoding

During playback, the 3D video player requests a frame f_{target} at quality level q, where q is the maximum depth of the returned tree data structure. The main parts of the decoding process are described in the remainder of this section.

File positioning. In a first step, the decoder needs to position the input file at the correct frame. This is achieved by reading the frame header information and thus retrieving the current frame number f_i and the length of the bit stream l_i. If f_i is different from f_{target}, the decoder advances l_i bytes and reads the header information of the next frame as long as the requested frame has not been reached. During the first pass, the decoder builds a look-up table containing (f_i, l_i) tuples. This table allows for backward navigation and simplifies forward navigation in subsequent passes.

Progressive symbol load. Once the input file is at the right position, the first part of the bit stream is read and entropy decoded into a buffer B. The number of bytes which are initially read depend on quality level q and on bit stream length l_i.

During the actual decoding process, decoder symbols are retrieved from B. If B does not contain enough symbols to reach the requested quality level q, the next set of symbols can be progressively read from the file into B.

Since the PR*k*–tree is traversed in breadth-first order during encoding, we retrieve the information for the top nodes by decoding the first part of the bit stream.

Point Primitive Decoding. The decoder does not need to return a complete tree data structure, but only the decoded point primitives. Hence, the parent-child relations from the initial data structure are only temporarily stored. In fact, the currently decoded node n_i needs to know its parent node for setting the bounds of the volume it represents. This volume is determined by the parent's volume and by the structure of the cardinal tree, i.e., by n_i's index c_i. This information is necessary for correctly decoding the position of the point primitive contained in n_i.

For playback at quality level q, we need to decode the set of significant nodes for quality level q. A node $n_{i,d}$, located at depth d in the tree, is significant for quality level q if $d = q$ or if $n_{i,d}$ is a leaf node with $d < q$.

Since our connectivity encoding only allows us to determine if $n_{i,d}$ is a leaf node during decoding of level $d + 1$, we use the additional leaf flag for taking this decision already at level d.

Based on the results of decoding the connectivity and of the leaf flag bit streams, we determine whether $n_{i,d}$ is significant with respect to q. If this is the case, we further proceed with the complete decoding of $n_{i,d}$ and write its position, normal, and color information into the vertex array. The cartesian coordinates of the normal vectors are retrieved from a precomputed look-up table and the YUV color components additionally need to be transformed into RGB space.

5.2 Point Based Rendering

We employ two different splatting schemes for point–based rendering. With the first scheme we can achieve pure rendering frame rates up to 20 frames per second. The second scheme, which provides higher quality rendering, still ensures interactive frame rates up to 5 frames per second. We use the computed surface normals for optional per–point reshading.

Similar to QSplat [25], the first splatting scheme uses OpenGL GL_POINTS as splatting primitive. After decoding the 3D video we directly transfer the corresponding points from disk into an *OpenGL vertex array.* The vertex array is then rendered using the OpenGL pipeline. The shape of the splatting primitive is very important for the resulting visual quality of the rendered object. By using GL_POINTS, we are restricted to either square (non anti-aliased points) or circular primitives (anti-aliased points). Unfortunately, these splats are nothing more than "fat pixels" on the screen. Thus, the splats cannot adjust to the shape of the object's silhouette in an optimal way. This is a limitation of the OpenGL point primitive. By using other graphics APIs such as Microsoft's DirectX, one can overcome this restriction. DirectX provides a *point sprite* primitive whose projected shape depends on the direction of the normal, which can point in arbitrary directions. Jaggy edges arising with fixed splat shapes could so be avoided. Note that the use of GL_POINTs enables us to easily merge 3D video with conventional geometry-based objects and environments by depth compositing.

The second scheme utilizes the EWA surface splatting approach from Zwicker et al. [34], which is based on a screen space formulation of the elliptical weighted average (EWA) filter adapted for irregular point samples. EWA surface splatting provides us with high–quality images and renders approximately 250,000 anti–aliased points per second. Recently, Ren et al. [22] introduced object space EWA surface splatting which implements the EWA filter as a two–pass rendering algorithm using programmable vertex and pixel shader architectures. This hardware–accelerated approach achieves rendering performance of up to 3 million points per second and is well suited for future integration into our 3D video framework.

5.3 Interaction

As described in Section 5.1, we can randomly access and decode individual frames of the 3D video. Furthermore, every frame can be retrieved at different quality levels, reaching from a coarse approximation to a very detailed representation. During normal playback, we read frame after frame and control the quality level such that the player adheres to the frame-rate of the original 2D video sequence. The quality level is thus determined by the decoder's performance and rendering complexity. When the sequence is paused, the decoder can completely decode the current frame at the highest quality level.

Fast-forward is implemented by decoding only a coarse representation, by using larger frame increments, or by a combination of both. Fast-reverse is realized in a corresponding way, we just need to decrement the requested frame number instead of incrementing it in between frames.

Slow motion can simply be realized by slowed-down playback, i.e., by decoding higher quality levels than the player supports in real-time. High-quality slow motion, however, requires additional point-based shape interpolation between consecutive frames, or the use of high-speed cameras.

The 3D video player implements a virtual trackball and hence arbitrary navigation and scaling is possible and follows the popular interaction metaphors from other graphics renderers.

Special effects, such as freeze–and–rotate, can easily be achieved by playing a sequence, pausing, rotating the view–point, and continuing playback again. In case the system is used for editing a 2D video from a 3D video sequence, the virtual camera path and the frame increments can be configured in a script file.

6 Prototype System

We built a prototype acquisition and reconstruction environment with six digital cameras — two Point Grey Research Dragonfly and four SONY DFW–V500 cameras — allowing to generate 3D video from approximately 160 degrees. Both camera types are equipped with 640×480 CCD imaging sensors. We use C– and CS–mount lenses with focal lengths between 2.8mm and 6mm. For calibrating the six cameras, we employed the Caltec *camera calibration toolbox* (http://www.vision.caltech.edu/bouguetj/calib_doc/), which is based on [8, 33]. Each of the six cameras is connected via FireWire to a 1.2 GHz Athlon Linux-based PC System, where the 2D video is recorded. The camera acquisition software is based on the linux1394 project (http://sf.net/projects/linux1394/) and the libdc1394 digital camera control library for Linux (http://sf.net/projects/libdc1394/). Although both camera types are capable of capturing unsynchronized images at 30 frames per second, external synchronization decreases the capture rate to 10 frames per second. The DFW-V500 cameras deliver YUV 4:2:2 images which are directly converted to RGB. The Dragonflys deliver 8-bit Bayer tiled images and thus RGB color interpolation is needed. For this purpose, we either use bilinear interpolation or an edge sensing algorithm. Note that we carry out chroma–keying in the RGB color space. The stored image sequences are processed on a dual processor machine with two AMD AthlonMP 1600+ CPUs where the 3D video is generated. The 3D video renderer runs on a 1.8 GHz Pentium4 machine equipped with an nVidia GeForce3 Ti200 graphics accelerator. The physical setup of the acquisition environment is depicted in Figure 4.

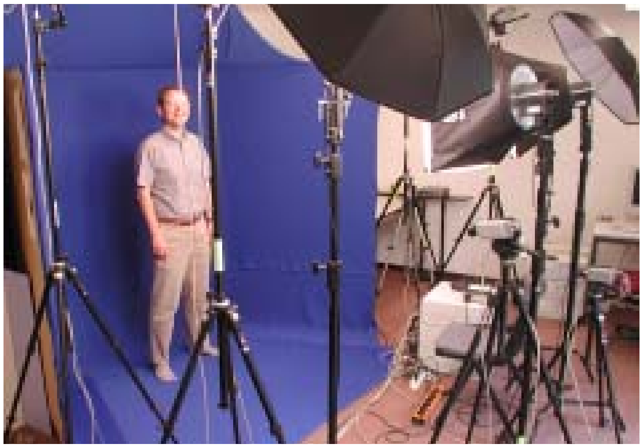


Figure 4: Physical setup of the prototype system.

7 Experimental Results

Figure 5 a) and b) show some example images from 3D video sequences recorded in our prototype system. Note that we currently build 3D videos using all cameras and a contour approximation threshold of 1. Unfortunately, due to the physical dimensions of the prototype system, the reconstruction frustum is rather small and does not allow for large movements. As discussed in Section 4.2, we encode PR27-trees down to depth 6 and position errors down to depth 5. Each frame leads to approximately 56k tree nodes and 48k significant point primitives for quality level 6.

Visual Quality. The small number of contours can lead to artifacts in regions occluded by all reference images, especially visible between the legs and under the arms. Furthermore, the normals (see flat–shaded images in Figure 5c) from the 3D reconstruction method are not very precise. We use a normal fairing algorithm [20] to optimize the surface normals. It turned out that this approach did not provide us with better normals since the quality of the underlying surface representation is also not too good (see depth map in Figure 5d). Figure 6 shows comparative frames rendered with the simple splatting scheme based on OpenGL point primitives (a) and using EWA surface splatting (b). Note the anti–aliasing provided by surface splatting, which is especially visible on the logo of the shirt. Figure 8 shows identical frames, except that frame (a) is rendered with refinement of the point primitives' position and frame (b) only uses the position approximation resulting from the PR*k*-tree structure. In frame (b), the position errors are well noticeable at the contours of the 3D video object, and are especially disturbing during playback. Our results show that an accurate encoding of the point primitives' position, at least down to a certain tree depth, is essential for visually appealing images.

Timings. The timings of our multi-threaded implementation of the 3D video processing on a dual processor machine are as follows:

- Contour extraction: ~30 milliseconds per 3D video frame
- 3D reconstruction: ~25 seconds per 3D video frame
- Encoding: ~1 second per 3D video frame.

It appears that the 3D reconstruction is the most expensive part. However, while using less accurate contours (see Figure 7), the performance of the 3D reconstruction can be improved by a factor of five.

Compression. The bit rates for the different data streams of typical sequences are listed in Table 3. Note that in our prototype implementation, we encode the 12 color bits per node on 2 bytes, and we use 6 bits per node for the position approximation, instead of the required $\lg\lceil 27\rceil = 5$ bits for PR27-trees. Furthermore, we point the reader's attention to the fact that the final entropy coding step, reduces the total data volume by almost 60%. The framework allows us to encode 3D video sequences of humans at a total bit rate of less than 7 megabit per second, the sequence running with 8.5 frames per second in normal playback. This represents an average of 14 bits per node of the PR27-tree and leads to a total size of typically less than 30 megabytes for a sequence of 30 seconds. Compared to the memory requirements of the complete data structure (see Table 2), we achieve a compression ratio of 64:1. Recording the same sequence with 6 cameras at 8.5 frames per second would lead to approximately 2 megabit per second in consumer video quality (MPEG-1) or 5 megabit per second in broadcast quality TV (MPEG-2). However, 6 separate MPEG sequences would only include temporal correlation in between frames from different cameras, but no spatial correlation, as in our 3D video format.

Table 3: Bit rates in megabit per second for 3D video streams recorded at 8.5 frames per second.

Type of information	before entropy coding [Mbps]	after entropy coding [Mbps]
Connectivity	0.95	0.37
Position approximation	2.96	1.43
Position refinement	0.61	0.61
Leaf flag	0.48	0.06
Color	7.62	2.92
Normals	3.81	1.30
Total	16.4	6.7

8 Conclusions and Future Work

The 3D Video Recorder is a powerful framework for generating three–dimensional video. Our 3D video concept is founded on point primitives which are stored in a hierarchical data structure. The 3D video player decodes and displays the representation from disk in real–time and provides interaction features like fast–forward and fast–reverse, as well as arbitrary scaling and rotating. Limitations include the quality of the underlying surface representation and the precision of the reconstructed normals. We plan to optimize the quality by employing other 3D reconstruction methods and by using more than six cameras. Furthermore, photometric calibration of the cameras is needed for our point–merging framework which would improve the texture quality. We expect better compression performance by exploiting inter–frame coherence and by devising adaptive bit allocation schemes for the lossy parts of the 3D video encoding. Future work will also include the integration of the hardware–accelerated EWA surface splatting and view-dependent decoding into the 3D video player.

Acknowledgements

We would like to thank Stefan Hösli, Nicky Kern, Christoph Niederberger, and Lior Wehrli for implementing parts of the system; Martin Näf for producing the video; Mark Pauly and Matthias Zwicker for the point rendering engine and for proofreading the paper. Many thanks to all members of the blue-c team for many fruitful discussions. This work has been funded by ETH Zurich as a “Polyprojekt”.

References

[1] C. L. Bajaj, V. Pascucci, and G. Zhuang. “Progressive Compression and Transmission of Arbitrary Triangular Meshes.” In *Proceedings Visualization 99*, pages 307–316. IEEE Computer Society Press, 1999.

[2] D. Benoit, E. D. Demaine, J. I. Munro, and V. Raman. “Representing trees of higher degree.” In *Proceedings of the 6th International Workshop on Algorithms and Data Structures 99*, Lecture Notes in Computer Science 1663, pages 169–180. Springer-Verlag, 1999.

[3] D. Cohen-Or, D. Levin, and O. Remez. “Progressive Compression of Arbitrary Triangular Meshes.” In *Proceedings Visualization 99*, pages 67–72. IEEE Computer Society Press, 1999.

[4] B. Curless and S. Seitz. “3D photography.” Course Notes. ACM SIGGRAPH 2000, 2000.

[5] M. F. Deering. “Geometry compression.” In *Proceedings of SIGGRAPH 95*, pages 13–20. ACM SIGGRAPH, Addison Wesley, 1995.

[6] O. Faugeras. *Three-dimensional Computer Vision: A Geometric Viewpoint*. MIT Press, 1993.

[7] S. Gumhold and W. Straßer. “Real time compression of triangle mesh connectivity.” In *Proceedings of SIGGRAPH 98*, pages 133–140. ACM SIGGRAPH, Addison Wesley, 1998.

[8] J. Heikkila and O. Silven. “A four-step camera calibration procedure with implicit image correction.” In *Proceedings of the International Conference on Computer Vision and Pattern Recognition 97*. IEEE Computer Society Press, 1997.

[9] H. Hoppe. “Progressive meshes.” In *Proceedings of SIGGRAPH 96*, pages 99–108. ACM SIGGRAPH, Addison Wesley, 1996.

[10] G. Jacobson. “Space-efficient Static Trees and Graphs.” In *30th Annual Symposium on Foundations of Computer Science*, pages 549–554. IEEE, 1989.

[11] T. Kanade, P. Rander, and P. Narayanan. “Virtualized reality: Constructing virtual worlds from real scenes.” In *IEEE MultiMedia*, Vol.4, No.1, Jan.-Mar. 1997, pages 43–54, 1997.

[12] A. Laurentini. “The visual hull concept for silhouette-based image understanding.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(2):150–162, 1994.

[13] G. Martin. “Range encoding: an algorithm for removing redundancy from a digitised message.” In *Video & Data Recoding Conference, Southampton*, 1979.

[14] W. Matusik, C. Buehler, and L. McMillan. “Polyhedral visual hulls for real-time rendering.” In *Proceedings of Twelfth Eurographics Workshop on Rendering*, pages 115–125, 2001.

[15] W. Matusik, C. Buehler, R. Raskar, S. J. Gortler, and L. McMillan. “Image-based visual hulls.” In *Proceedings of SIGGRAPH 2000*, pages 369–374. ACM Press / ACM SIGGRAPH / New York, 2000.

[16] S. Moezzi, A. Katkere, D. Y. Kuramura, and R. Jain. “Immersive Video.” In *Proceedings of the 1996 Virtual Reality Annual International Symposium*, pages 17–24. IEEE Computer Society Press, 1996.

[17] J. Mulligan and K. Daniilidis. “View-independent scene acquisition for tele-presence.” In *Proceedings of the International Symposium on Augmented Reality*, pages 105–108, 2000.

[18] P. J. Narayanan, P. Rander, and T. Kanade. “Constructing virtual worlds using dense stereo.” In *Proceedings of the International Conference on Computer Vision ICCV 98*, pages 3–10, 1998.

[19] R. Pajarola and J. Rossignac. “Squeeze: Fast and Progressive Decompression of Triangle Meshes.” In *Proceedings of Computer Graphics International*, pages 173–182. IEEE Computer Society Press, 2000.

[20] M. Pauly and M. Gross. “Spectral processing of point-sampled geometry.” In *Proceedings of SIGGRAPH 2001*, pages 379–386. ACM Press / ACM SIGGRAPH, 2001.

[21] S. Pollard and S. Hayes. “View synthesis by edge transfer with application to the generation of immersive video objects.” In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, pages 91–98. ACM Press / ACM SIGGRAPH, New York, 1998.

[22] L. Ren, H. Pfister, and M. Zwicker. “Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering.” In *Proceedings of Eurographics 2002*, COMPUTER GRAPHICS Forum, Conference Issue, 2002. to appear.

[23] J. Rissanen and G. G. Langdon Jr. “Arithmetic coding.” *IBM Journal of Research and Development*, 23(2):149–162, 1979.

[24] J. Rossignac. “Edgebreaker: Connectivity compression for triangle meshes.” *IEEE Transactions on Visualization and Computer Graphics*, 5(1):47–61, January - March 1999. ISSN 1077-2626.

[25] S. Rusinkiewicz and M. Levoy. “QSplat: A multiresolution point rendering system for large meshes.” In *Proceedings of SIGGRAPH 2000*, pages 343–352. ACM Press / ACM SIGGRAPH, New York, 2000.

[26] S. Rusinkiewicz and M. Levoy. “Streaming QSplat: A Viewer for Networked Visualization of Large, Dense Models.” In *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, pages 63–68. ACM, 2001.

[27] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[28] J. W. Shade, S. J. Gortler, L. He, and R. Szeliski. “Layered depth images.” In *Proceedings of SIGGRAPH 98*, pages 231–242. ACM SIGGRAPH, Addison Wesley, 1998.

[29] A. R. Smith and J. F. Blinn. “Blue screen matting.” In *Proceedings of SIGGRAPH 96*, pages 259–268. ACM SIGGRAPH, Addison Wesley, 1996.

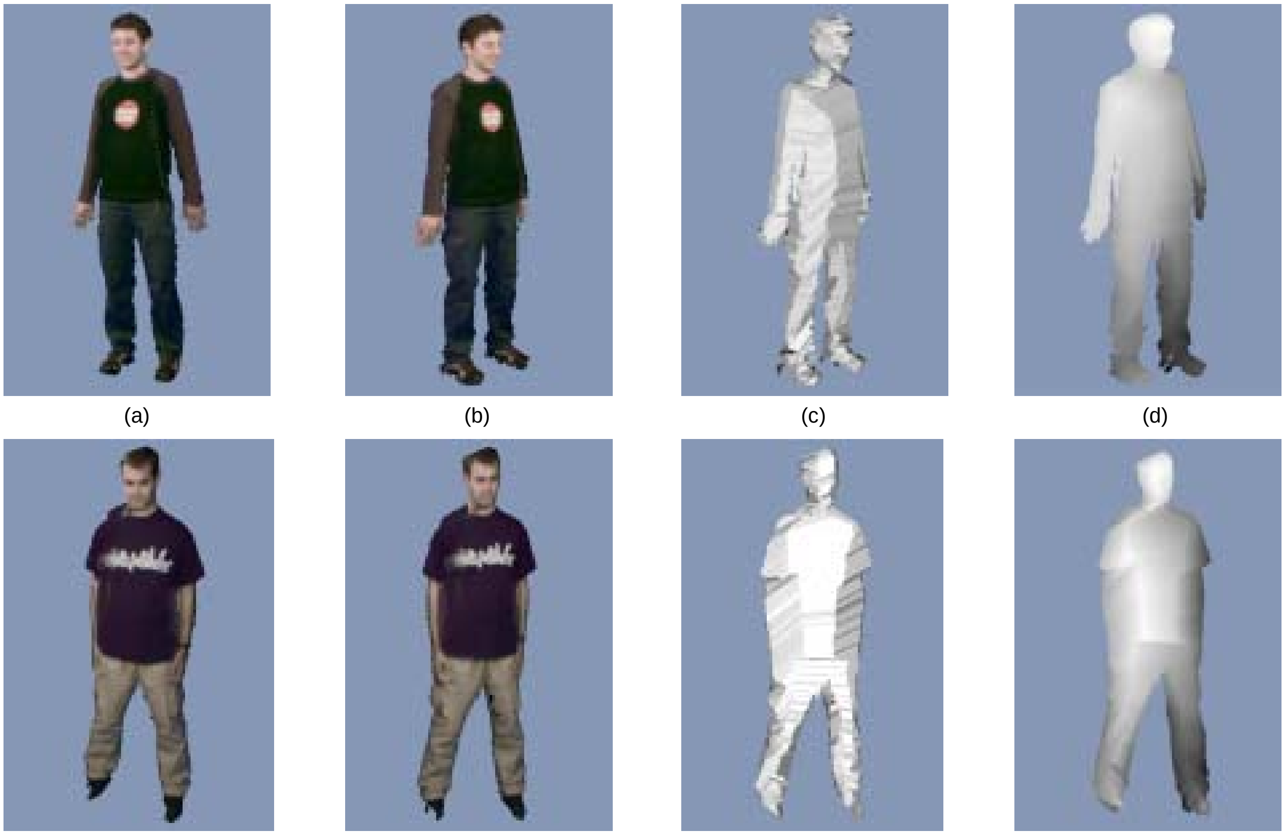
[30] S. Vedula. *Image Based Spatio-Temporal Modeling and View Interpolation of Dynamic Events*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 2001.

[31] P. Vlahos. *Comprehensive Electronic Compositing System*. U.S. Patent 4,100,569, July 11 1978.

[32] Y. Yemez and F. Schmitt. “Progressive Multilevel Meshes from Octree Particles.” In *Proceedings of the 2nd International Conference on 3-D Imaging and Modeling*, pages 290–299. IEEE Computer Society Press, 1999.

[33] Z. Zhang. “Flexible camera calibration by viewing a plane from unknown orientations.” In *Proceedings of the 7th International Conference on Computer Vision 99*, pages 666–673. IEEE Computer Society Press, 1999.

[34] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. “Surface splatting.” In *Proceedings of SIGGRAPH 2001*, pages 371–378. ACM Press / ACM SIGGRAPH, New York, 2001.



(a) (b) (c) (d)

Figure 5: Images from 3D video sequences. (a) and (b) show surface splatted views from the 3D video. (c) reconstructed normals by flat–shading the view from (b) without colors. (d) is the corresponding depth map.

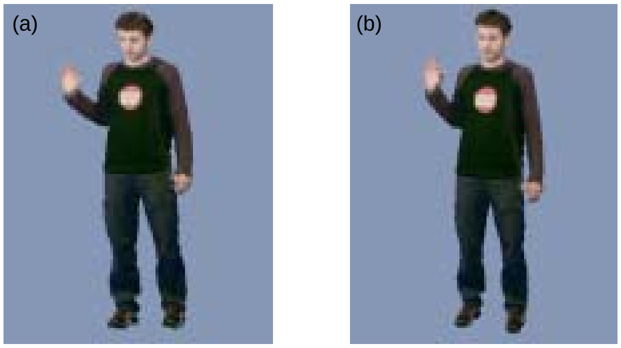


Figure 6: Different splatting schemes. (a) shows an image rendered using the simple splatting scheme. b) image rendered with surface splatting.

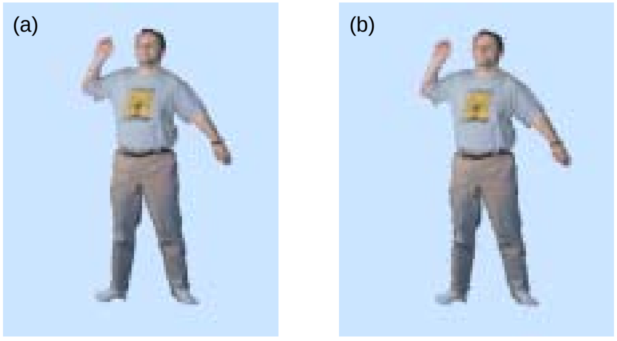


Figure 8: Necessity of accurate position encoding. (a) with, and (b) without position refinement.

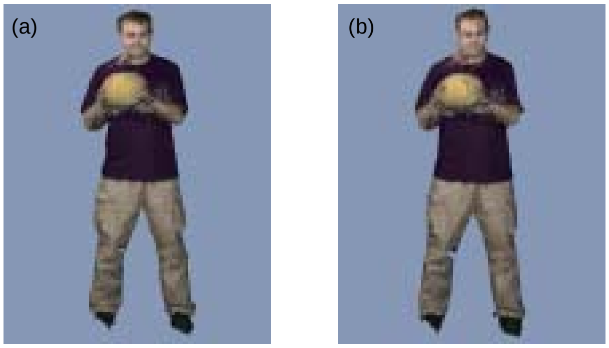


Figure 7: Quality–performance trade-off. Both images are rendered using the simple splatting scheme. (a) shows an image reconstructed with a contour extraction threshold of 1. (b) shows the same image reconstructed with threshold 15.