Technical Report

Pattern Recognition and Image Processing Group
Institute of Computer Aided Automation
Vienna University of Technology
Favoritenstr. 9/1832
A-1040 Vienna AUSTRIA
Phone: +43 (1) 58801-18351
Fax: +43 (1) 58801-18392
E-mail: tos@prip.tuwien.ac.at
URL: http://www.prip.tuwien.ac.at/

PRIP-TR-064 October 6, 2000

Shape from Silhouette

Srdan Tosovic

Abstract

An algorithm for automatic construction of a 3D model of an object is presented. The construction is based on a sequence of images of the object taken from different viewpoints. The object's silhouette is the only feature which is extracted from an input image. Images are acquired by rotating the object on a turntable in front of a stationary camera. The algorithm uses an octree representation of the model, and builds this model incrementally, by performing limited processing of all input images for each level of the octree. Beginning from the root node at the level 0 a rough model of the object is obtained quickly and is refined as the processed level of the octree increases. Results of the algorithm developed are presented for both synthetic and real input images.

Contents

List of Figures

List of Tables

1 Introduction

Shape from Silhouette is a method of automatic construction of a 3D model of an object based on a sequence of images of the object taken from multiple views, in which the object's silhouette represents the only interesting feature of an image [Sze93, Pot87]. The object's silhouette in each input image corresponds to a conic volume in the object real-world space (Figure 1). A 3D model of the object can be built by intersecting the conic volumes from all views.

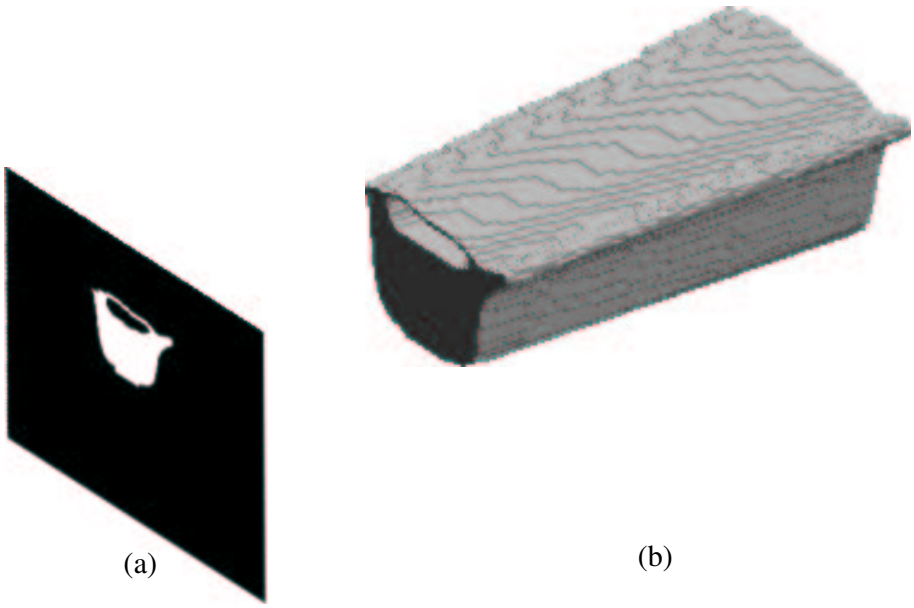


Figure 1: Image silhouette (a) and the corresponding conic volume (b)

Shape from Silhouette is a computationally simple algorithm — it employs only basic matrix operations for all transformations — and it requires only a camera as equipment, so it can be used to obtain a quick initial model of an object which can then be refined by other methods. It can be applied on objects of arbitrary shapes, including objects with certain concavities (like a handle of a cup), as long as the concavities are visible from at least one input view. It can also be used to estimate the volume of an object.

There have been many works on construction of 3D models of objects from multiple views. Baker [Bak77] used silhouettes of an object rotating on a turntable to construct a wire-frame model of the object. Martin and Aggarwal [MA83] constructed volume segment models from orthographic projection of silhouettes. Chien and Aggarwal [CA83] constructed an object's octree model from its three orthographic projections. Veenstra and Ahuja [VA86] extended this approach to thirteen standard orthographic views. Potmesil [Pot87] created octree models using arbitrary views and perspective projection. For each of the views he constructs an octree representing the corresponding conic volume (Figure 1) and then intersects all octrees. In contrast to this, Szeliski [Sze93] first creates a low resolution octree model quickly and then refines this model iteratively, by intersecting each new silhouette with the already existing model. The last two approaches project an

octree node into the image plane to perform the intersection between the octree node and the object's silhouette. Srivastava and Ahuja [SA90] in contrast, perform the intersections in 3D-space. The work of Szeliski [Sze93] was used as a base for the approach presented in this report.

To acquire images from multiple views, we put an object on a turntable which rotates in front of a stationary camera. For 3D models of objects octree [CH88] representation is used, which has several advantages: for a typical solid object it is an efficient representation, because of a large degree of coherence between neighboring volume elements (voxels), which means that a large piece of an object can be represented by a single octree node. Another advantage is the ease of performing geometrical transformations on a node, because they only need to be performed on the node's vertices. The disadvantage of octree models is that they digitize the space by representing it through cubes whose resolution depend on the maximal octree depth and therefore cannot have smooth surfaces.

The implementation of the *Shape from Silhouette* method presented in this paper was done as part of the project *Computer Aided Classification of Ceramics* [Cer], which is performed by Pattern Recognition and Image Processing Group at the Institute of Computer Aided Automation at the Vienna University of Technology in cooperation with the Institute of Classical Archaeology at the University of Vienna, with the goal of automating the documentation process of archaeological sherds.

This document is structured as follows: Section 1.1 gives an overview of possible 3D model representations. Section 1.2 is a short review of octree representation. Section 2 describes the acquisition system and gives an overview of the calibration method used. Section 3 describes the implemented approach in detail, Section 4 presents the results, followed by the conclusion in Section 5.

Appendix A goes into implementation details on the programming language level and Appendix B is the user's manual for the implemented task.

1.1 Types of Model Representation

There are many different model representations in computer vision and computer graphics used. Here we will mention only the most important ones. Surface-based representations describe the surface of an object as a set of simple approximating patches, like planar or quadric patches [Nal93]. Generalized cylinder representation [Shi87] defines a volume by a curved axis and a cross-section function at each point of the axis. Overlapping sphere representation [OB79] describes a volume as a set of arbitrarily located and sized spheres. Approaches such as these are efficient in representing a specific set of shapes but they are not flexible enough to describe arbitrary solid objects. Two of the most commonly used representations for solid volumes are boundary representation (B-Rep) and constructive solid geometry (CSG) [Shi87]. The B-Rep method describes an object as a volume enclosed by a set of surface elements, typically sections of planes and quadratic surfaces such as spheres, cylinders and cones.The CSG method uses volume elements rather than surface elements to describe an object. Typical volume elements are blocks, spheres, cylinders, cones and prisms. These elements are combined by set operations into the modeled object. The B-Rep and CSG method suffer from quadratic growth of elemental operations as the complexity of the modeled object increases.

1.2 Octree Model Representation

An octree [CH88] is a tree-formed data structure used to represent 3-dimensional objects. Each node of an octree represents a cube subset of a 3-dimensional volume. A node of an octree which represents a 3D object is said to be:

- *black*, if the corresponding cube lies completely within the object
- *white*, if the corresponding cube lies completely within the background, i.e., has no intersection with the object
- *gray*, if the corresponding cube is a boundary cube, i.e., belongs partly to the object and partly to the background. In this case the node is divided into 8 child nodes (octants) representing 8 equally sized subcubes of the original cube

All leaf nodes are either black or white and all intermediate nodes are gray. An example of a simple 3D object and the corresponding octree is shown in Figure 2.

An octree described above contains binary information in the leaf nodes and therefore it is called a binary octree, and it is suitable for representation of 3D objects where the shape of the object is the only object property that needs to be modeled by the octree. Non-binary octrees can contain other information in the leaf nodes, e.g., the cube color in RGB-space. For the *Shape from Silhouette* algorithm presented, a binary octree model is sufficient to represent 3D objects, and in the remainder of this paper the term *octree* will always refer to a binary octree.

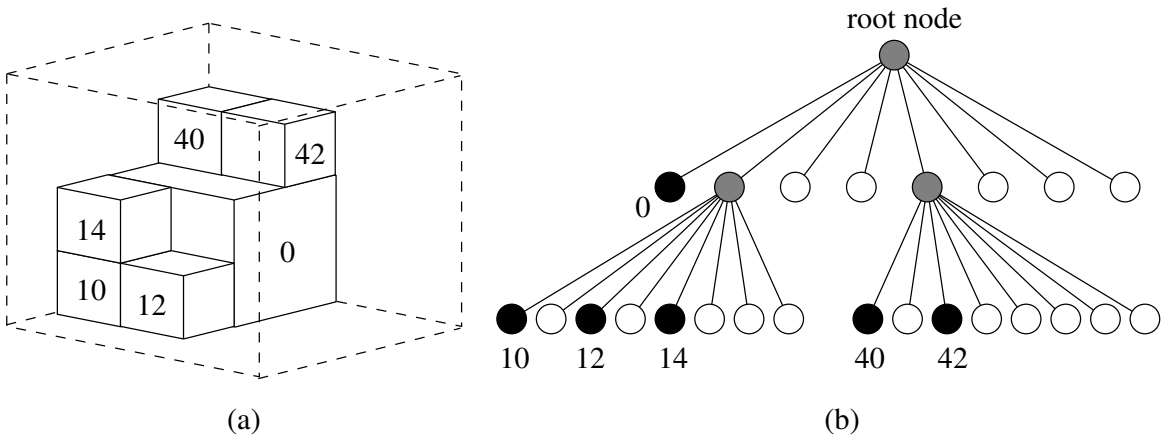


Figure 2: A simple object (a) and the corresponding octree (b)

2 Acquisition System

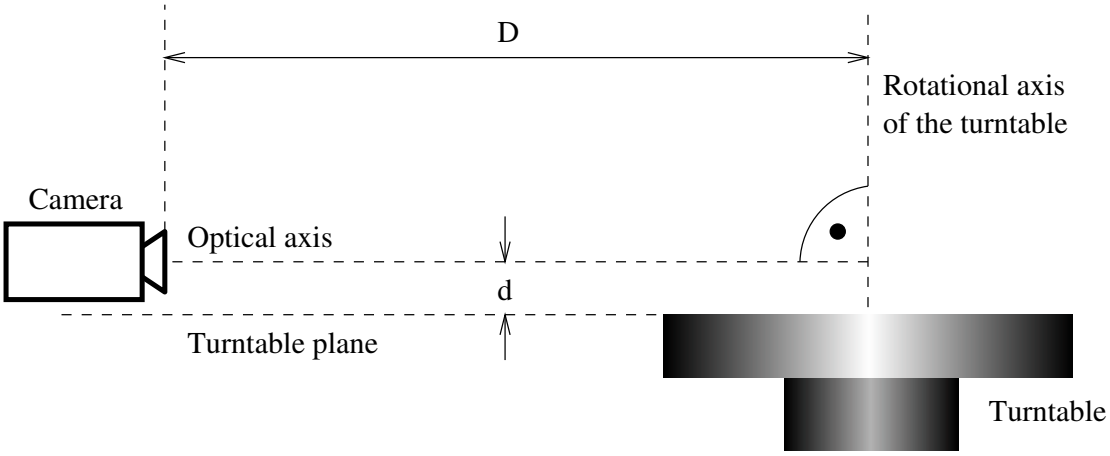
This section describes the acquisition system used during the implementation of the *Shape from Silhouette* algorithm presented and also gives an overview of the method used for its calibration.

2.1 Description

The acquisition system [KT00] consists of the following devices:

- a monochrome CCD-camera with a focal length of 16 mm and a resolution of 768x576 pixels
- a turntable with a diameter of 50 cm, whose desired position can be specified with an accuracy of 0.05 degrees

The distance between the camera and the turntable is approx. 120 cm and it is estimated by the calibration algorithm. Figure 3 depicts the geometrical setup of the camera and the turntable.



$D \approx 120$ cm, $d \approx 3$ mm (ideally 0)

Figure 3: Geometrical setup of the camera and the turntable

Another important issue is the illumination of the object observed. The object should be clearly distinguishable from the background, independent from the object's shape or the type of its surface. For that reason backlighting [HS91] is used. A large (approx. 50x40 cm) rectangular lamp is put behind the turntable (as seen from the camera). In addition, a white piece of paper, larger than the lamp, is put right in front of the lamp, in order to make the light more diffuse. The whole system is protected against the ambient light by a thick black curtain. Figure 4 shows the real acquisition system used — in the lower left corner is the camera and on the right side in the middle is the test object (in this case an archeological sherd) on the turntable, in front of the paper covering the lamp, which cannot be seen in the figure.

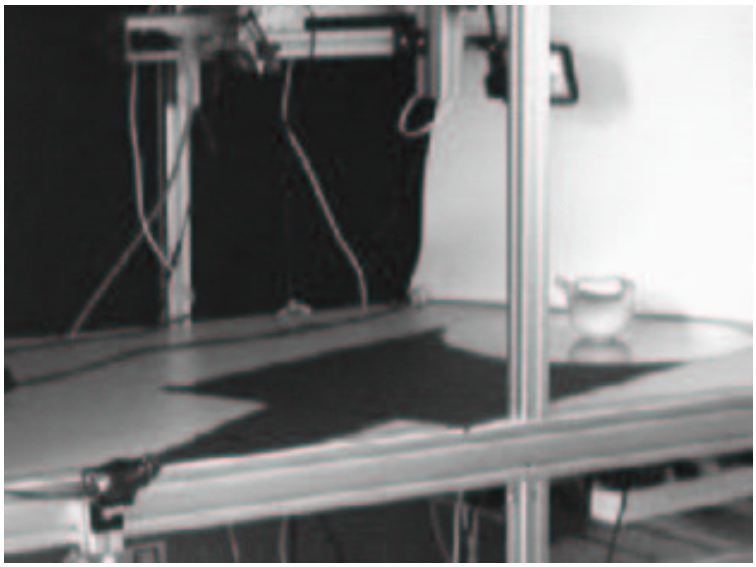


Figure 4: The acquisition system

2.2 Turntable Calibration

The calibration method used was exclusively developed for the *Shape from Silhouette* algorithm presented and it is described in detail in [Tos99] and [KT00]. Here we give a short overview only.

The calibration is based on tracking the peak and the baseline of a cone with known dimensions (diameter of the base 160 mm, height 80 mm) while it turns on the turntable. The output is a 4×4 matrix which transforms the image coordinate system into the object coordinate system if the points in each system are written in their homogeneous form [Nal93]. The origin of the image coordinate system is taken to be in the center of the image and the origin of the object coordinate system at the intersection of the rotational axis of the turntable and the plane of its surface, with the y axis being the rotation axis. Figure 5 illustrates the orientation of the two coordinate systems to one another.

The detection of the peak and the base of the cone is based on the linear Hough transform [DH72], with the assumption that the base of the cone is projected into a straight line in the image coordinate system. For that reason the following assumptions were made for the calibration algorithm:

- pinhole camera model [Nal93]
- there is no lens distortion
- image center point lies exactly in the center of an image
- the optical axis of the camera is orthogonal to the rotational axis of the turntable and its distance to the plane of the turntable's surface (d in Figure 3) is negligible (ideally 0)

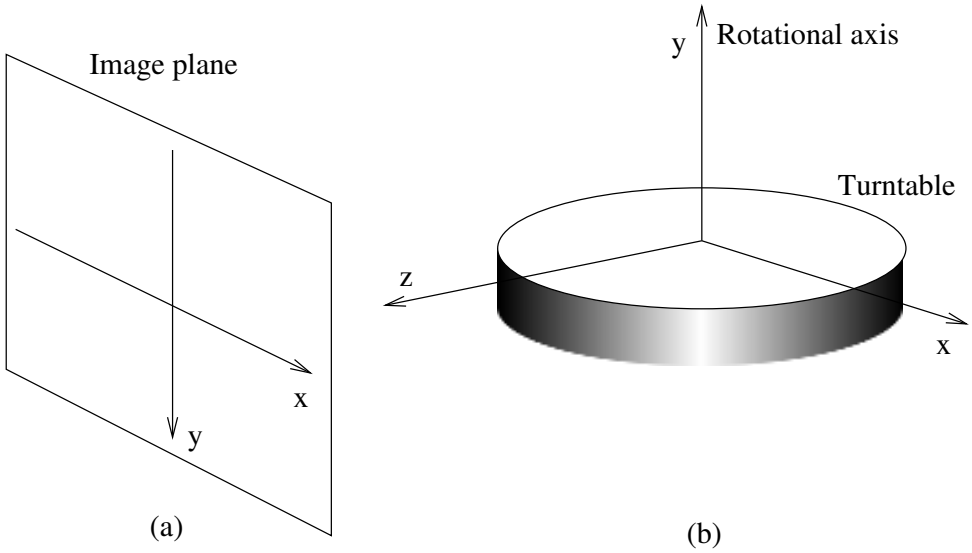


Figure 5: Image- (a) and object (b) coordinate system

3 Building a 3D Model of an Object

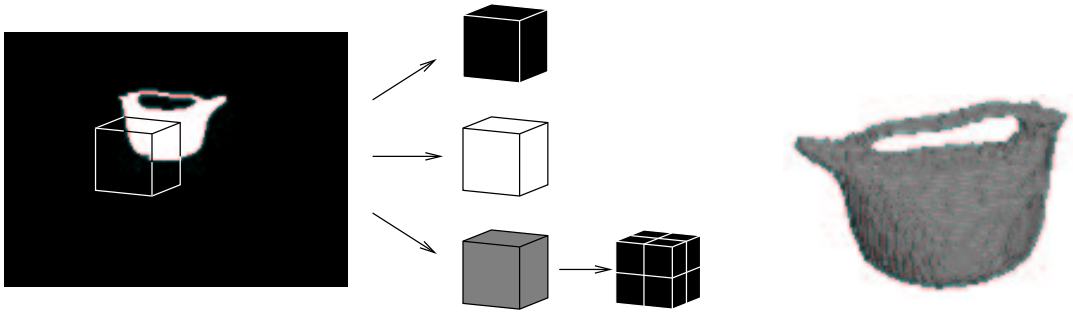
This section describes the presented *Shape from Silhouette* algorithm in detail. Section 3.1 gives an overview of the algorithm, without going into details on all of its steps. Section 3.2 describes the method of extracting an object's silhouette from an input image, Section 3.3 the projection of an octree node into the image plane and Section 3.4 is a description of the intersection test of the projected octree node with the object's image silhouette.

3.1 Overview

The algorithm described in this report builds up a 3D model of an object in the following way: first, all input images are transformed into binary images where a "black" pixel belongs to the object observed and a "white" one to the background[1] (Figure 6a). Then, the initial octree is created with a single root node (Figure 6b) representing the whole object space, which will be "carved out" corresponding to the shape of the object observed. Then, the octree is processed in a level-by-level manner: starting from level 0 (with root

(a) Extraction of object's silhouette

(b) Initial octree



(c) Octree node projection

(d) Intersection test result

(e) Final octree

Figure 6: Algorithm overview

node as the only node), all octree nodes of the current level marked as "black", i.e., belonging to the object, are projected into the first input image (Figure 6c) and tested for

[1] in the implementation, *black* means background and *white* means object, but it is more intuitive to describe an object pixel as "black" and a background pixel as "white"

intersection with the object's image silhouette. Depending on the result of the intersection test, a node can remain to be "black", it can be marked as "white" (belonging to the background) or in case it belongs partly to the object and partly to the background, it is marked as "grey" and divided into 8 black child nodes of the next higher level (Figure 6d). The remainder of the black nodes of the current level is then projected into the next input image where the procedure of intersection testing with the object's silhouette is repeated. Once all input images have been processed for the current octree level, the current level is incremented by one and the whole procedure (starting from the projection of the black nodes of the current level into the first input image) is repeated, until the maximal octree level has been reached. The remaining octree after the processing of the last level is the final 3D model of the object (Figure 6e).

3.2 Extraction of the Silhouette of an Object

A reliable extraction of the object's silhouette from an input image is of crucial importance for obtaining an accurate 3D model of an object. If the background brightness is not uniform, i.e., if there are parts of the background which are brighter than the others, and especially if there are parts with similar brightness like the object observed, the silhouette extraction can be a difficult task. For that reason, in addition to all images of the object taken from different viewpoints, an image of the turntable (Figure 7b) is taken, without any object on it. Then, the absolute difference between this image and an input image is built, which creates an image (Figure 7c) with a uniform background and a high contrast between the object and the background. Then, a simple thresholding is used to create a binary image (Figure 7d) where pixels with the value 1 represent the object's silhouette and those with value 0 the background. The threshold value is user definable. Figure 7 illustrates the process of silhouette extraction described above.

3.3 Projection of an Octree Node into the Image Plane

We define the following coordinate systems relevant for the projection of an octree node into the image plane:

- *octree coordinate system*: rooted at the intersection of the rotational axis of the turntable and its rotational plane. For the first input image, it is identical to the object coordinate system and it rotates with the object observed
- *object coordinate system*: rooted at the same point as the octree coordinate system, but it is static, i.e., it doesn't rotate with the object. The y axis is the rotation axis of the turntable and the z axis is orthogonal to the image plane
- *image coordinate system*: lies in the image plane and it is rooted at the image center point

Units of the octree- and object coordinate system are millimeters and units of the image coordinate system pixels. Figure 8 depicts these coordinate systems and their relative positions to one another.

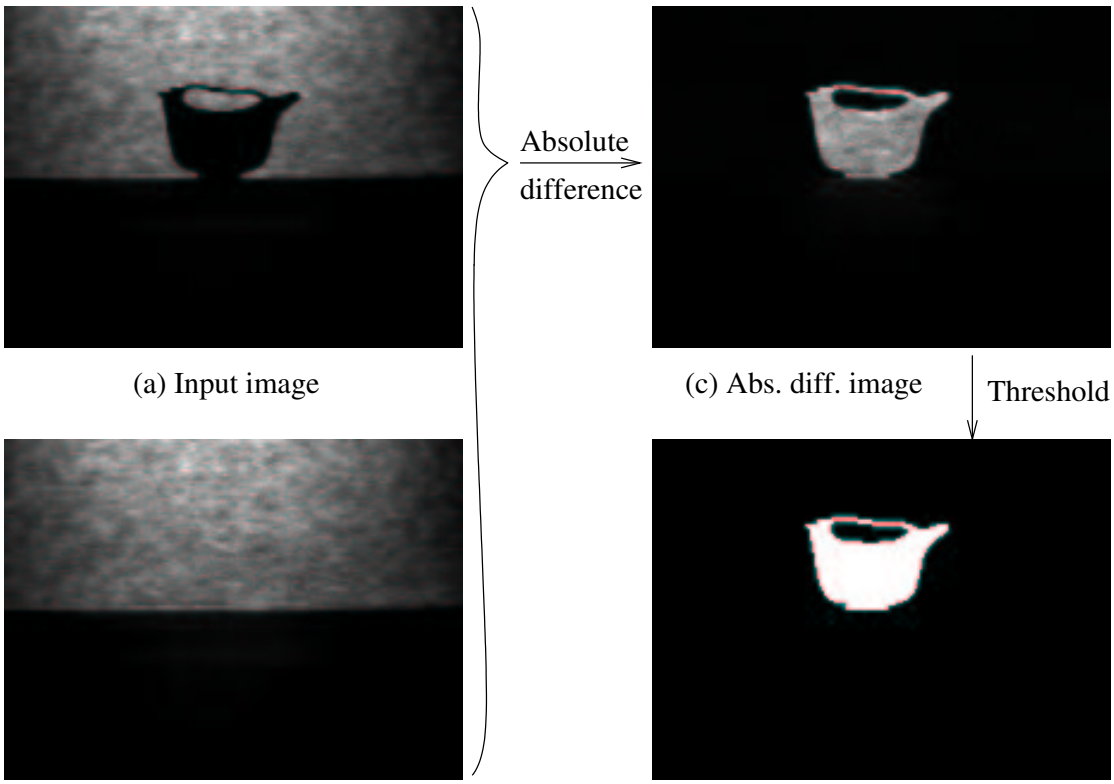


Figure 7: Extraction of an object's silhouette

An octree node is projected into the image plane by projecting all of its eight vertices. Octree coordinates of a vertex can be determined by the path from the root node of the octree to the node the vertex belongs to. We need to transform the octree coordinates of a vertex into image coordinates. This transformation is performed in three steps, each of which, if we use homogeneous coordinates [Nal93] to represent points in all three coordinate systems, can be described by a 4×4 or 4×3 transformation matrix.

1. transformation of octree coordinate system into object coordinate system. This transformation can be described by the rotation angle α (see Figure 8) around the y axis, represented by the transformation matrix R_α

2. transformation of object coordinate system into image coordinate system. The parameters of this transformation are represented by the matrix T, calculated by the calibration algorithm, described in Section 2.2

3. correction (scaling) of the image x and y coordinates, based on the value of the "imaginary" image z coordinate, as calculated in the steps 1 and 2. The corresponding transformation matrix is named S_z

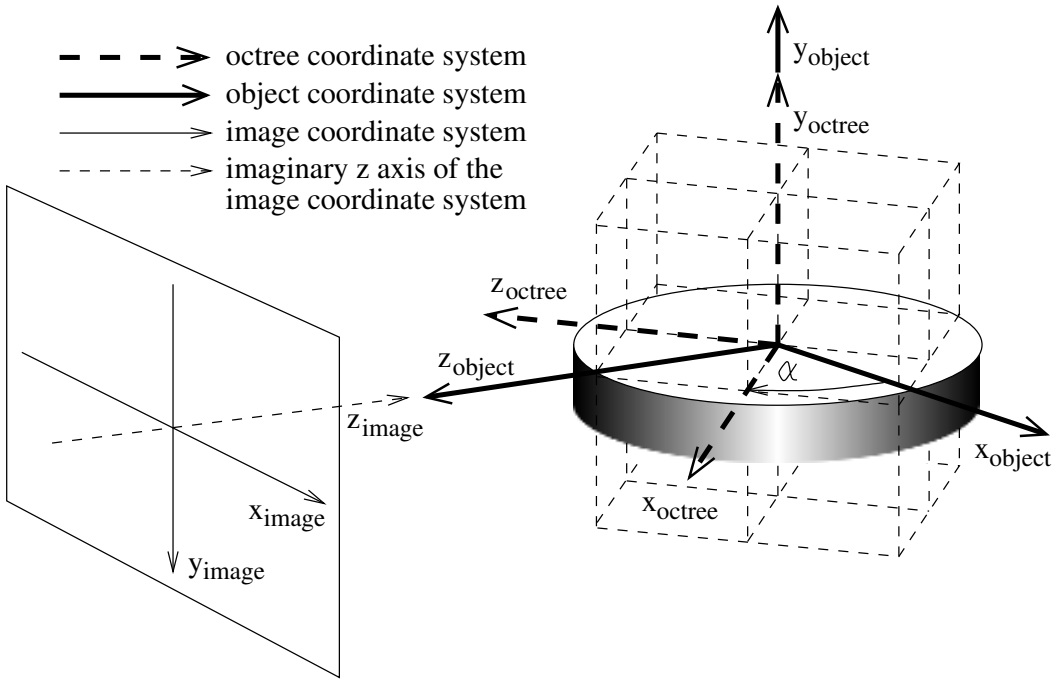


Figure 8: The octree-, object- and image coordinate system

For a given viewing angle α (Figure 8), the transformation matrix R_α (step 1) has the following form:

$$R_\alpha = \begin{pmatrix} \cos\alpha & 0 & -\sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ \sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The transformation matrix T (step 2) calculated by the calibration algorithm is independent from the viewing angle and generally looks as follows:

$$T = \begin{pmatrix} r_1 & r_2 & r_3 & t_1 \\ r_4 & r_5 & r_6 & t_2 \\ r_7 & r_8 & r_9 & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

r_1–r_9 are rotation and t_1–t_3 translation parameters of the transformation between the object and image coordinate systems.

The scaling matrix S_z (step 3) depends on the result of the transformations from the first two steps, i.e., there will be different scaling matrices for different 3D points in the object coordinate system. The reason is that the transformation matrix T gives the correct image x and y coordinates of a point in object space only if the z coordinate (in object coordinate system) equals zero, because the calibration algorithm actually gives the transformation parameters between image coordinates in the image (x–y) plane and the object coordinates in the x–y plane of the object coordinate space. For all other points

a correction has to be made, which is a simple scaling by a factor k_z, whose calculation will be described later. The scaling matrix S_z then looks like:

$$S_z = \begin{pmatrix} k_z & 0 & 0 & 0 \\ 0 & k_z & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The octree coordinates are denoted with $(x_{oct}, y_{oct}, z_{oct})$, the object coordinates with $(x_{obj}, y_{obj}, z_{obj})$ and the image coordinates with (x_{img}, y_{img}). With $(x_{img0}, y_{img0}, z_{img0})$ we denote the 3-dimensional image coordinates with the imaginary z axis, before performing the step 3 (correction of the image x and y coordinates). Using the homogeneous form [Nal93] of these coordinates, the transformations described above can be expressed as follows:

$$\begin{pmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ 1 \end{pmatrix} = R_\alpha \begin{pmatrix} x_{oct} \\ y_{oct} \\ z_{oct} \\ 1 \end{pmatrix}, \begin{pmatrix} x_{img0} \\ y_{img0} \\ z_{img0} \\ 1 \end{pmatrix} = T \begin{pmatrix} x_{obj} \\ y_{obj} \\ z_{obj} \\ 1 \end{pmatrix} \Longrightarrow \begin{pmatrix} x_{img0} \\ y_{img0} \\ z_{img0} \\ 1 \end{pmatrix} = TR_\alpha \begin{pmatrix} x_{oct} \\ y_{oct} \\ z_{oct} \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} x_{img} \\ y_{img} \\ 1 \end{pmatrix} = S_z \begin{pmatrix} x_{img0} \\ y_{img0} \\ z_{img0} \\ 1 \end{pmatrix} = \begin{pmatrix} k_z x_{img0} \\ k_z y_{img0} \\ 1 \end{pmatrix}$$

We still need to explain how the factor k_z is calculated. This process is illustrated in Figure 9. A point with z coordinate of zero in the object coordinate space (like P in Figure 9) multiplied with the transformation matrix T will have the "imaginary" z coordinate in the image coordinate system with the value of t_3 (from the transformation matrix T). That means, if the value of z_{img0} equals t_3, we do not perform any correction of x_{img0} and y_{img0}, i.e., k_z equals 1. In all other cases k_z has the value by which we need to scale the vector $(x_{img0}, y_{img0}, z_{img0})$ (the point P_0 in Figure 9) in order to bring z_{img0} to the value of t_3. It follows that k_z equals t_3/z_{img0}.

To summarize, an octree node is projected into the image plane in the following way: as a preprocessing step, the matrices T and R_α are multiplied for all possible view angles α, and the resulting matrices of these multiplications are stored in a lookup table. This is done before any processing of octree nodes starts. Once it starts, all vertices of the current node are projected into the image plane by multiplying their octree coordinates with the matrix from the lookup table corresponding to the current view angle and then multiplying the result with the appropriate scaling matrix S_z.

3.4 Silhouette Intersection Test

The result of the projection of an octree node into the image plane are image coordinates of all of the vertices of the node's corresponding cube. In the general case, the projection of a node looks like a hexagon, as depicted in Figure 10(a). To find the hexagon corresponding to the eight projected vertices is a costly task, because it requires to determine which points are inside and which outside the hexagon, and there can be hundreds of thousands

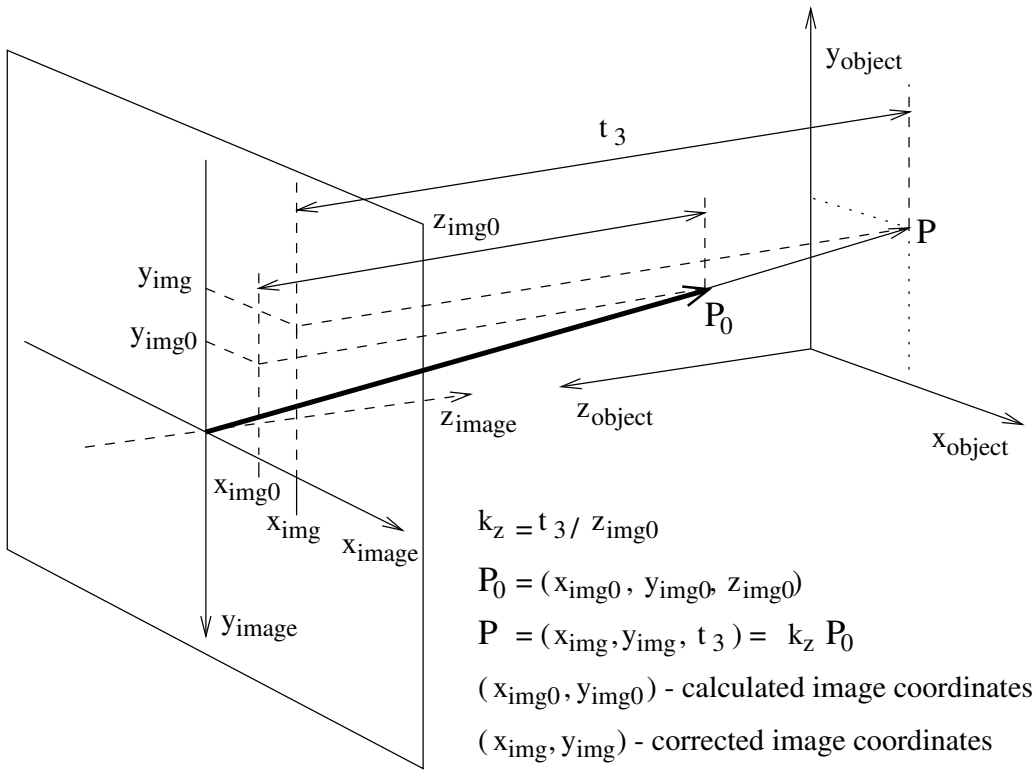


Figure 9: Correction of the image coordinates

of octree nodes that need to be processed. It is much simpler (and therefore faster) to compare the bounding box of the eight points. Figure 10 shows a projected octree node and the corresponding bounding box. The bounding box is tested for intersection with

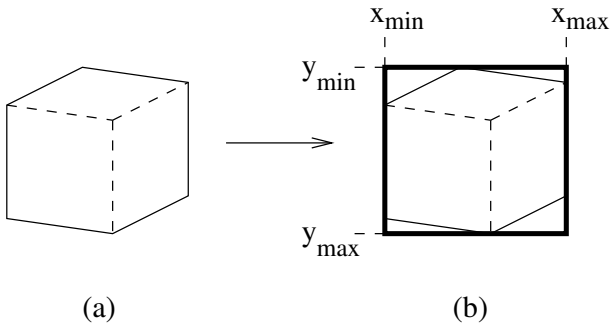
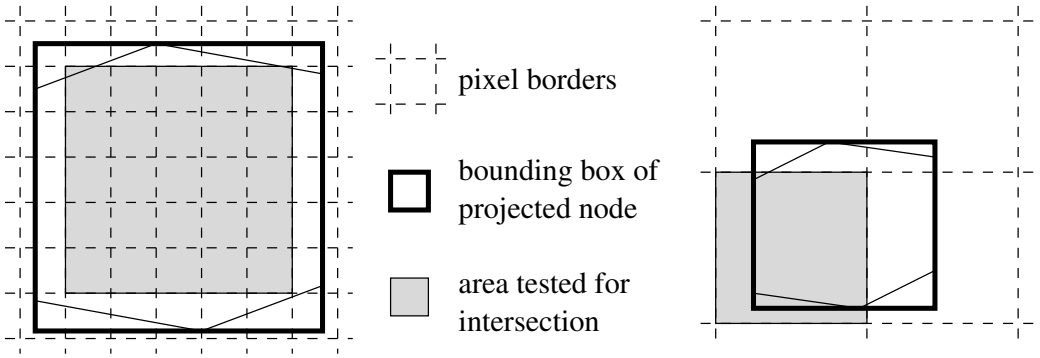


Figure 10: Projection of a node (a) and its bounding box (b)

the object's silhouette in the current input (binary) image. All image pixels within the bounding box are checked for their color, whether they are black or white. The output of the intersection testing procedure is percentage of the black pixels of the bounding box, i.e., percentage of pixels belonging to the object. If this percentage is equal or higher than a user definable threshold for black nodes, the node is marked as black. If the

percentage is smaller than or equal with a user definable threshold for white nodes, the node is marked as white. Otherwise, the node is marked as gray and it is divided into eight child nodes representing eight subcubes of finer resolution.

One more issue needs to be mentioned. The calculated image coordinates of the cube's vertices can lie between two image pixels, and a pixel is the smallest testable unit for intersection testing. Which pixels are considered to be "within" the bounding box? Figure 11 illustrates our answer to this question. We decided to test only pixels that lie completely within the bounding box (Figure 11a), because that way the number of pixels that need to be tested is smaller than if we tested all pixels that are at least partly covered by the bounding box and it also makes sense to exclude the pixels at the border of the bounding box, because most of them do not lie within the hexagon approximated by the bounding box. In the special case if there are no pixels that lie completely within the bounding box (Figure 11b) the pixel closest to the center of the bounding box is checked for the color.



(a) at least one pixel completely within bounding box

(b) no pixels completely within bounding box

Figure 11: Selection of pixels for the intersection test

4 Results

This section presents the tests performed on the implemented *Shape from Silhouette* algorithm and analyzes their results. Section 4.1 describes the tests with synthetic 3D objects and Section 4.2 with real objects. Section 4.3 analyses the results of tests with both synthetic and real objects.

4.1 Synthetic Objects

Here we describe the tests with two synthetic 3D objects: a sphere and a cone (Figure 12). Both objects are rotational-symmetric, so if we assume that they rotate around their axis of symmetry, we need to create only one synthetic input image for each object, because the projection of the object into the image plane will always look the same, independent on the viewing angle. We assume we have such a virtual acquisition system where:

- a millimeter in the x–y plane of the object coordinate system (see Figure 5) corresponds to exactly one pixel in the image plane
- x and y axes of the object coordinate system are parallel to the x and y axes of the image coordinate system

Under these assumptions the transformation matrix describing the transformation between the two systems is very simple, containing only 0's and ± 1's as rotational parameters (r_1–r_9 of the transformation matrix T, see Section 3.3). The translational parameters (t_1–t_3 of the matrix T in Section 3.3) can be chosen freely in a certain range. We assume t_1 and t_2 to be 0, which means that the origin of the object coordinate systems projects exactly into the origin of the image coordinate system, and we set t_3 to 1000, which means that the distance between the virtual camera and the rotation axis of the object is 1000 millimeters.

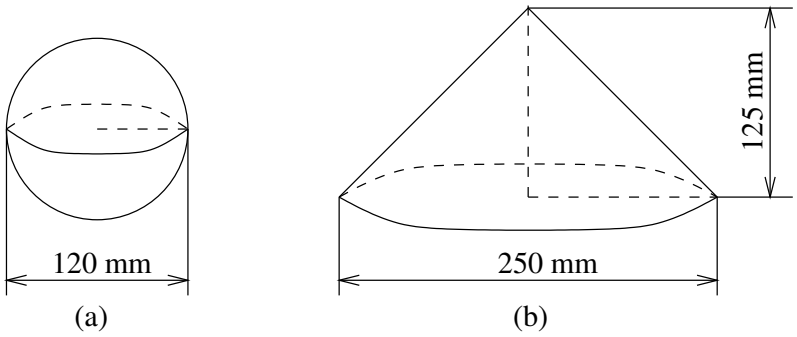


Figure 12: Synthetic 3D objects: a sphere (a) and a cone (b)

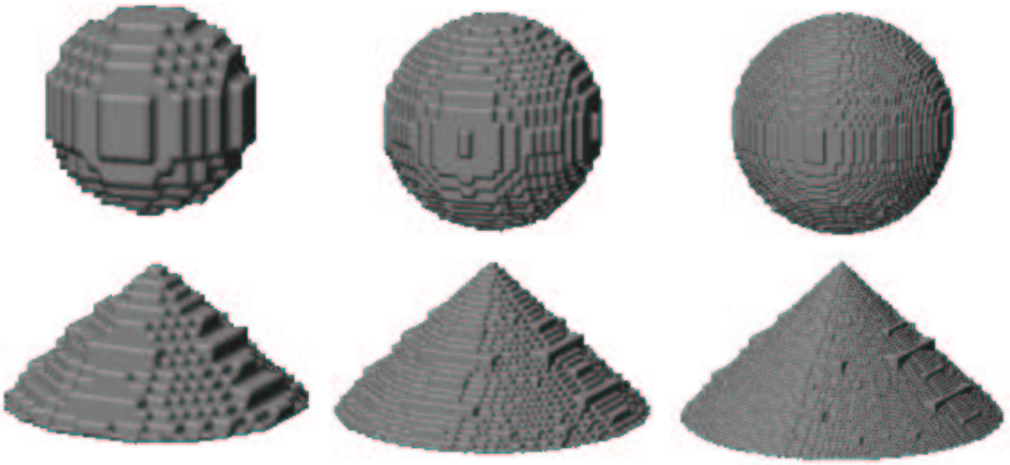
Under the assumptions described above, we can construct perfect synthetic input images for the sphere and the cone in Figure 12, i.e., the transformation between the virtual image and object coordinate systems will be described by our synthetic transformation matrix T without any errors. With the results of the tests of these synthetic images we

can estimate the accuracy of the *Shape from Silhouette* algorithm implemented, without any influence of camera calibration error, which is unavoidable for a real acquisition system. Figure 13 shows the synthetic input images corresponding to the sphere and the cone in Figure 12.

(a) (b)

Figure 13: Synthetic input image for the sphere (a) and the cone (b)

In the first sequence of tests we build a 3D model of an object based on 36 different views, with an angle of 10° between two consequent views, in 64^3, 128^3 and 256^3 voxel resolutions. Figure 14 shows the reconstructed 3D models of our synthetic objects for each of these resolutions and Table 1 compares their analytic and the computed dimensions and volume.



64^3 voxels 128^3 voxels 256^3 voxels

Figure 14: Constructed models of synthetic objects in different voxel resolutions

In the second test sequence we vary the number of views used to build a 3D model of an object. First we use 4, then 12, then 36 and then 72 input images (with an angle of 90°, 30°, 10° and 5° between two consequent views, respectively). The constructed 3D

object	dimensions	voxel size	volume
analytic sphere	120 x 120 x 120 mm	—	904 779 mm^3
sphere 64^3 voxels	112 x 112 x 112 mm	4 mm	733 184 mm^3 (-18.96%)
sphere 128^3 voxels	120 x 116 x 120 mm	2 mm	823 296 mm^3 (-9.01%)
sphere 256^3 voxels	120 x 120 x 120 mm	1 mm	883 712 mm^3 (-2.33%)
analytic cone	250 x 125 x 250 mm	—	2 045 308 mm^3
cone 64^3 voxels	224 x 112 x 224 mm	4 mm	1 802 240 mm^3 (-11.88%)
cone 128^3 voxels	240 x 120 x 240 mm	2 mm	1 943 040 mm^3 (-5.00%)
cone 256^3 voxels	248 x 124 x 248 mm	1 mm	2 037 056 mm^3 (-0.40%)

Table 1: Comparison of analytic and calculated dimensions (1)

models are shown in Figure 15 and their analytic and computed dimensions and volume compared in Table 2.

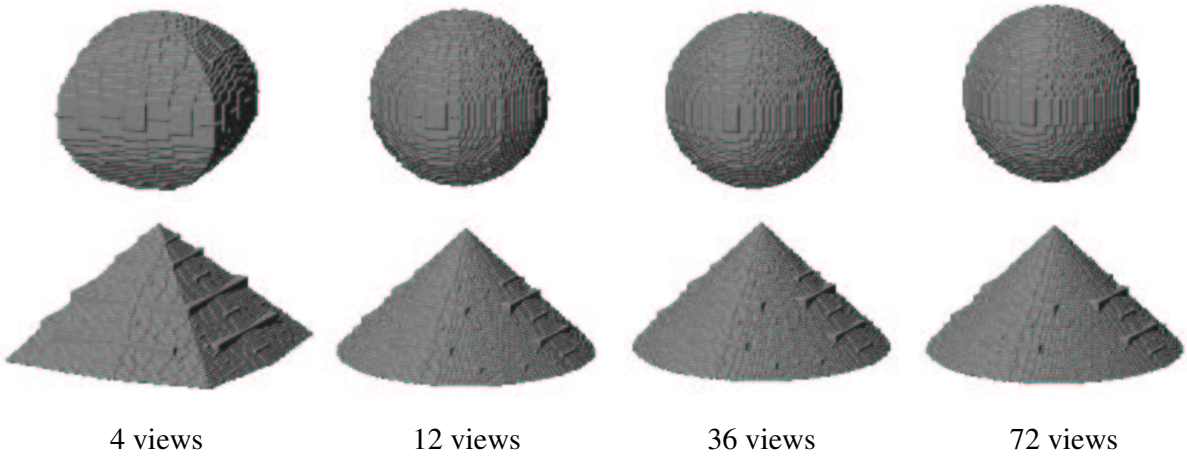


Figure 15: Constructed models of synthetic objects with different number of views

Table 3 summarizes some performance-related statistics for the tests described above. All test were done on a Pentium II 450 MHz machine with 256 MB of RAM.

4.2 Real Objects

For tests with real objects we used a metal cuboid and two ceramic pots (Figure 16). The acquisition system was calibrated by using the turntable calibration method described in the section 2.2. The calculated distance between the camera and the rotation axis of the turntable (D in Figure 3) was 118.0 cm.

Just like with synthetic objects, two sequences of tests were performed: one with a varying voxel resolution of the final octree and one with a varying number of views taken as input.

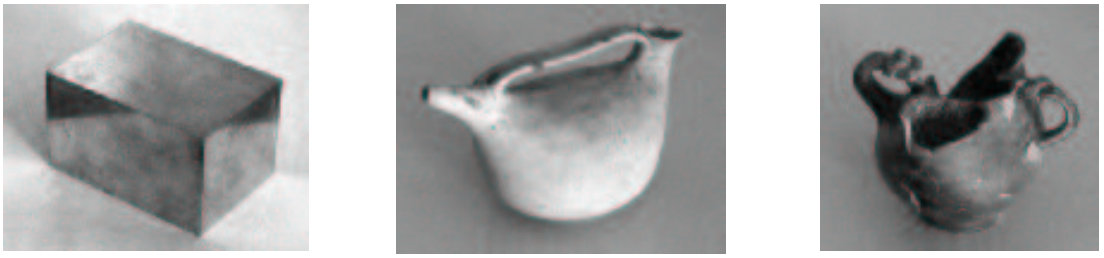
In the first test sequence all octrees were built using 36 views. Figure 17 shows the constructed 3D models of the objects in Figure 16 with octree resolution of 64^3, 128^3 and

object	dimensions	voxel size	volume
analytic sphere	120 x 120 x 120 mm	—	904 779 mm^3
sphere 4 views	120 x 120 x 120 mm	1 mm	1 071 072 mm^3 (+18.38%)
sphere 12 views	120 x 120 x 120 mm	1 mm	895 616 mm^3 (-1.01%)
sphere 36 views	120 x 120 x 120 mm	1 mm	883 712 mm^3 (-2.33%)
sphere 72 views	120 x 120 x 120 mm	1 mm	882 432 mm^3 (-2.47%)
analytic cone	250 x 125 x 250 mm	—	2 045 308 mm^3
cone 4 views	248 x 124 x 248 mm	1 mm	2 364 368 mm^3 (+15.60%)
cone 12 views	248 x 124 x 248 mm	1 mm	2 048 256 mm^3 (+0.01%)
cone 36 views	248 x 124 x 248 mm	1 mm	2 037 056 mm^3 (-0.40%)
cone 72 views	248 x 124 x 248 mm	1 mm	2 028 384 mm^3 (-0.83%)

Table 2: Comparison of analytic and calculated dimensions (2)

object	number of processed octree nodes	total CPU time
sphere 64^3 voxels	40 633	5.24 sec
sphere 128^3 voxels	178 281	14.75 sec
sphere 256^3 voxels	360 401	34.55 sec
sphere 4 views	81 681	8.92 sec
sphere 12 views	179 089	14.26 sec
sphere 36 views	360 401	34.55 sec
sphere 72 views	573 265	80.96 sec
cone 64^3 voxels	45 097	5.90 sec
cone 128^3 voxels	205 289	18.31 sec
cone 256^3 voxels	395 721	39.74 sec
cone 4 views	111 569	9.88 sec
cone 12 views	209 393	16.04 sec
cone 36 views	395 721	39.74 sec
cone 72 views	618 409	95.00 sec

Table 3: Octree statistics for reconstruction of synthetic objects



(a) (b) (c)

Figure 16: Real objects: a metal cuboid (a) and two ceramic pots (b) and (c)

256^3 voxels. Table 4 gives some octree statistics and compares the size of the real object, i.e., its maximal width, height and depth with the maximal width, height and depth of the reconstructed 3D model.

In the second test sequence we built three octree models for each object in Figure 16, by using 4, 12 and 36 input views, with uniform angle between two consequent views, with octree resolution of 256^3 voxels. Figure 18 shows the 3D models constructed and Table 5 gives the octree statistics and and the size comparison between the real objects and their constructed models.

Finally, Figure 19 shows the reconstructed 3D models of the two pots from three sides. For these models octree of resolution of 256^3 voxels was built, based on input images from 36 views.

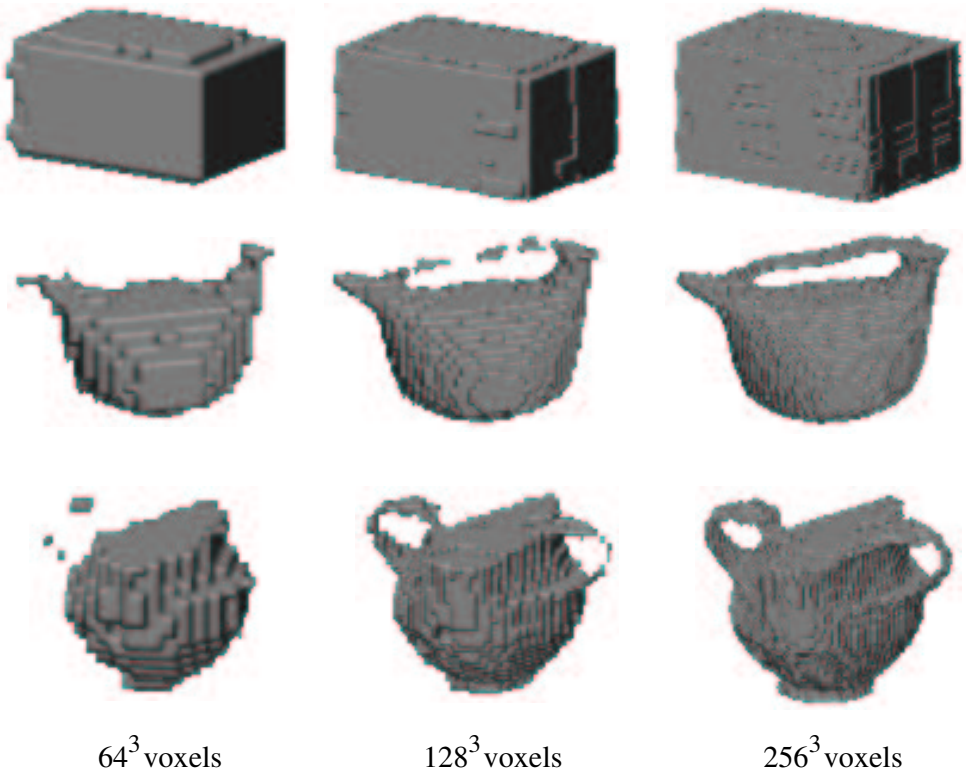


Figure 17: Constructed models of real objects in different voxel resolutions

4.3 Analysis

The results with both synthetic and real input data show that there is a certain minimal octree resolution required to obtain an accurate model of an object, especially for highly detailed objects, like the two pots used for tests with real images. To increase the octree resolution to 512^3 would not improve the results of our tests, because the projection of a single voxels would be less than half the pixels size. Concerning the number of input views used for obtaining a model of an object, it turned out that beginning from 12 views, the constructed model doesn't change a lot — in our tests the octrees built from 12 views

object	dimensions	voxel size	# nodes	CPU time
cuboid real	100.2 x 60.1 x 70.3 mm	—	—	—
cuboid 64^3 voxels	103.1 x 60.9 x 75.0 mm	2.34 mm	44 337	8.61 sec
cuboid 128^3 voxels	107.8 x 60.9 x 77.3 mm	1.17 mm	190 105	20.91 sec
cuboid 256^3 voxels	109.0 x 62.1 x 77.3 mm	0.59 mm	310 649	36.10 sec
pot 1 real	141.2 x 93.7 x 84.8 mm	—	—	—
pot 1 64^3 voxels	137.5 x 87.5 x 81.2 mm	3.12 mm	58 713	8.08 sec
pot 1 128^3 voxels	140.6 x 93.8 x 84.4 mm	1.56 mm	277 169	22.77 sec
pot 1 256^3 voxels	145.3 x 93.8 x 85.9 mm	0.78 mm	857 753	67.28 sec
pot 2 real	114.2 x 87.4 x 114.6 mm	—	—	—
pot 2 64^3 voxels	100.0 x 87.5 x 106.2 mm	3.12 mm	47 849	8.14 sec
pot 2 128^3 voxels	115.6 x 87.5 x 112.5 mm	1.56 mm	231 921	21.36 sec
pot 2 256^3 voxels	117.2 x 89.1 x 114.1 mm	0.78 mm	726 689	65.10 sec

Table 4: Octree statistics for real data with varying octree resolution

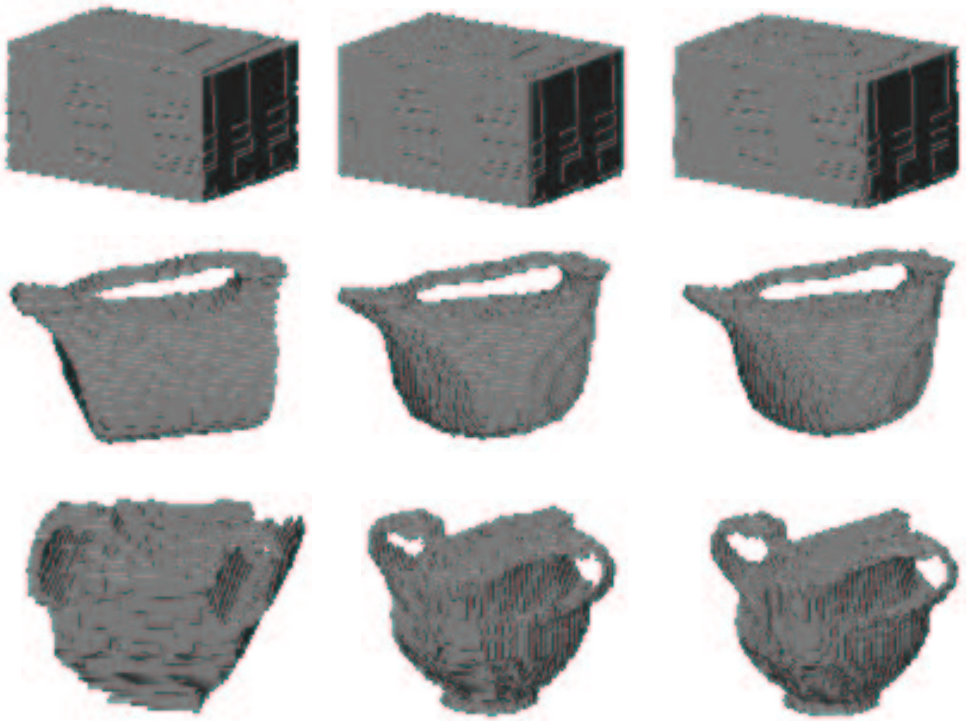


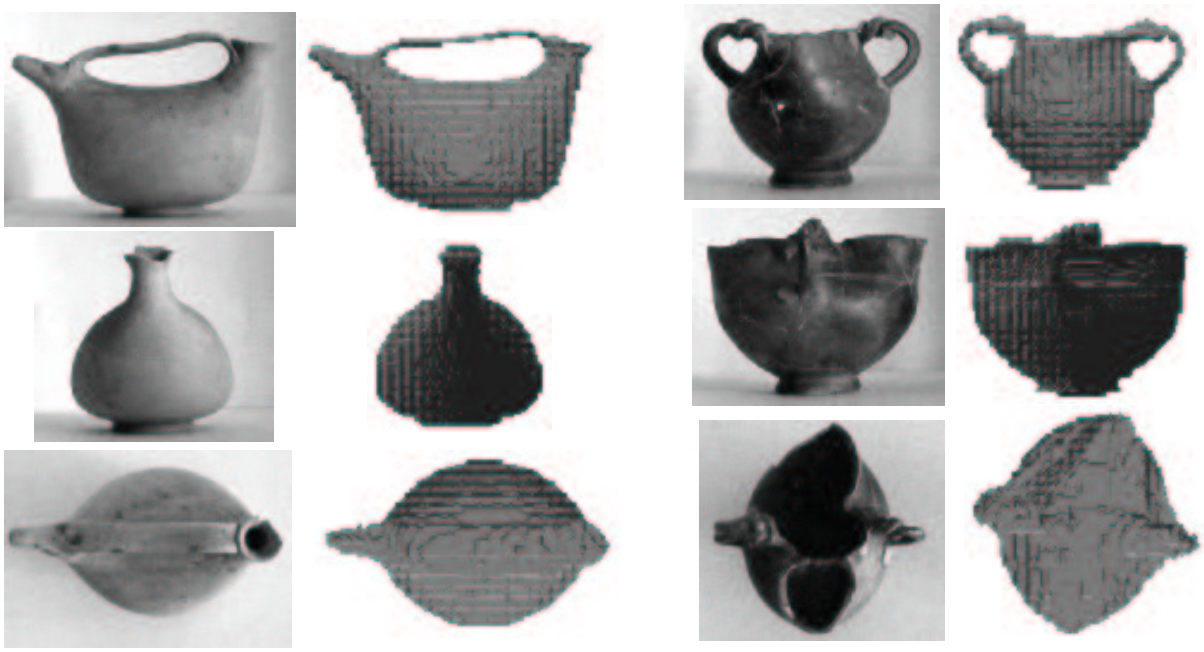
Figure 18: Constructed models of real objects with different number of views

were almost the same as the ones built from 36 views, except that they took much less time to construct.

The results with synthetic data, where we had a perfect transformation matrix, showed that the error in the dimensions of the model lies within or is slightly higher than the error

object	dimensions	voxel size	# nodes	CPU time
cuboid real	100.2 x 60.1 x 70.3 mm	—	—	—
cuboid 4 views	109.0 x 62.1 x 77.3 mm	0.59 mm	84 065	9.76 sec
cuboid 12 views	109.0 x 62.1 x 77.3 mm	0.59 mm	161 625	15.15 sec
cuboid 36 views	109.0 x 62.1 x 77.3 mm	0.59 mm	310 649	36.10 sec
pot 1 real	141.2 x 93.7 x 84.8 mm	—	—	—
pot 1 4 views	148.4 x 93.8 x 89.1 mm	0.78 mm	275 177	12.30 sec
pot 1 12 views	145.3 x 93.8 x 89.1 mm	0.78 mm	486 585	23.47 sec
pot 1 36 views	145.3 x 93.8 x 85.9 mm	0.78 mm	857 753	67.28 sec
pot 2 real	114.2 x 87.4 x 114.6 mm	—	—	—
pot 2 4 views	118.8 x 89.1 x 115.6 mm	0.78 mm	257 897	12.57 sec
pot 2 12 views	117.2 x 89.1 x 114.1 mm	0.78 mm	424 201	23.32 sec
pot 2 36 views	117.2 x 89.1 x 114.1 mm	0.78 mm	726 689	65.10 sec

Table 5: Octree statistics for real data with varying number of views



(a) (b)

Figure 19: Reconstructed 3D models of the pot 1 (a) and pot 2 (b)

introduced through the minimal voxel size. The error with real data depends additionally on the accuracy of the calibration algorithm,

The results also showed that the algorithm works much better with oval objects, i.e., with objects that do not have completely flat surfaces or sharp edges.

5 Conclusion

In this report an implementation of *Shape from Silhouette* method was presented, which creates a 3D model of on object from images of the object taken from different viewpoints. It showed to be a simple and fast algorithm, which is able to reconstruct models of arbitrarily shaped objects, as long as they do not have too many hidden concavities, i.e., concavities not visible in any of the input images. The algorithm is simple, because it employs only simple matrix operations for all the transformations and it is fast, because even for highly detailed objects, a high resolution octree (256^3 voxels) and a high number of input views (36), the computational time hardly exceeded 1 minute. Already for a smaller number of views (12) the constructed models were very similar to the ones constructed from 36 views and they took less than 25 seconds of computational time.

Except for the general drawback of any *Shape from Silhouette* algorithm, that certain concavities of an object cannot be reconstructed independent from the number of input images and where they were taken from, the implementation presented has some other drawbacks which could be lifted. The first one refers more to the calibration algorithm, which makes many simplifying assumptions about the acquisition system, the one about the optical axis of the camera lying exactly in the turntable plane being the most questionable one, because it never really holds. However, it showed to be a very good approximation which greatly simplifies the calibration algorithm. A more severe drawback is that the *Shape from Silhouette* algorithm presented uses "too much" knowledge about the calibration — for projection of the nodes into the image plane it uses the fact that the image plane and the x–y plane of the object coordinate system are completely parallel, making the algorithm unusable for any other setup of a camera and a turntable, where the camera, for example, looks at the turntable from the angle of 45°. A better approach would be to give the parameters of transformation between object- and camera coordinate system as output of the calibration, and pass the intrinsic camera parameters as additional input to our *Shape from Silhouette* algorithm, which then can be used for a transformation between the camera and the object coordinate system. Another thing that could be improved is the method of acquiring binary images with object's silhouette from a real-world input image. In our approach we use simple user definable thresholding. An ideal threshold value can differ from image to image, therefore adaptive thresholding would be a better idea. Another possibility would be to use edge detection for silhouette extraction instead of thresholding.

References

[Bak77] H. Baker. Three-dimensional modelling. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, pages 649–655, 1977.

[CA83] C. H. Chien and J. K. Aggarwal. Volume/surface octrees for the representation of three-dimensional objects. *Computer Vision, Graphics, and Image Processing*, 36:100–113, 1983.

[Cer] http://www.prip.tuwien.ac.at/Research/ArcheologicalSherds/.

[CH88] H. H. Chen and T. S. Huang. A survey of construction and manipulation of octrees. *Computer Vision, Graphics, and Image Processing*, 43:409–431, 1988.

[DH72] R. O. Duda and P. E. Hart. Use of the Hough transformation to detect lines and curves in pictures. *Communications of the ACM*, 15(1):11–15, January 1972.

[HS91] R. M. Haralick and L. G. Shapiro. Glossary of computer vision terms. *Pattern Recognition*, 24(1):69–93, 1991.

[Kho] http://www.khoral.com.

[KT00] M. Kampel and S. Tosovic. Turntable calibration for automatic 3D-reconstruction. In *Applications of 3D-Imaging and Graph-based Modelling, Proceedings of the 24th Workshop of the Austrian Association for Pattern Recognition (ÖAGM)*, pages 25–31, 2000.

[Lin] http://www.linux.org.

[MA83] W. N. Martin and J. K. Aggarwal. Volumetric description of objects from multiple views. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-5(2):150–158, 1983.

[Nal93] V. S. Nalwa. *A Guided Tour Of Computer Vision*. Addison-Wesley, 1993.

[OB79] J. O'Rourke and N. Badler. Decomposition of three-dimensional objects into spheres. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(3):295–305, 1979.

[Pot87] M. Potmesil. Generating octree models of 3D objects from their silhouettes in a sequence of images. *Computer Vision, Graphics, and Image Processing*, 40:1–29, 1987.

[RH] http://www.redhat.com.

[SA90] S. K. Srivastava and N. Ahuja. Octree generation from object silhouettes in perspective views. *Computer Vision, Graphics, and Image Processing*, 49:68–84, 1990.

[Shi87] Y. Shirai. *Three-Dimensional Computer Vision*. Springer-Verlag, 1987.

[Sze93] R. Szeliski. Rapid octree construction from image sequences. *CVGIP: Image Understanding*, 58(1):23–32, July 1993.

[Tos99] S. Tosovic. Lineare Hough-Transformation und Drehtellerkalibrierung. Technical Report PRIP-TR-59, Institute of Computer Aided Automation, Pattern Recognition and Image Processing Group, Vienna University of Technology, Austria, 1999.

[VA86] J. Veenstra and N. Ahuja. Efficient octree generation from silhouettes. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 537–542, 1986.

A Implementation Details

A.1 Environment

The System

The *Shape from Silhouette* algorithm presented was implemented using *Khoros 2001* [Kho] programming environment, on a 450 MHz *Pentium II* machine with 256 MB of RAM running *Linux* [Lin] operating system with kernel version 2.2.14 (*Red Hat* [RH] Linux distribution 6.2) and C-compiler *gcc* version egcs-2.91.66.

The Program

A new *Khoros*-toolbox named `tos_silhouette` was created, with a single *kroutine* named `silhouette`, the executable program file. All source files for the program were written in C.

A.2 Implementation

This part of the section describes the implementation of the program on the programming language level. First we will describe the data structure used to represent an octree, followed by description of most important C-routines.

Octree Data Structure

The data structure `tNode`, used for a single octree node, looks as follows:

```
typedef struct t_octree_node {     /***    single octree node     ***/
  int                   type;      /* 'B' or 'W' or 'G'             */
  double                cx;        /* x, y and z octree coordinates */
  double                cy;        /* coordinates of the center of  */
  double                cz;        /* the node                      */
  struct t_octree_node *octant[8]; /* pointers to child nodes       */
  struct t_octree_node *nextNode;  /* next node at the same level   */
} tNode;
```

`type` of the node can be `'B'` (black node), `'W'` (white node) or `'G'` (gray node). `cx`, `cy` and `cz`, are coordinates of the center of the node in octree coordinate space (see Figure 8). `octant` is an array of pointers to child nodes of this node, in case of a black or white node all these pointers have value of `NULL`. `nextNode` is a pointer to the next node at the same level. It is used to build a linear data structure for nodes at the same level, so processing of nodes at the same level can be done without traversing the octree. The whole octree is represented by a single pointer to the root node:

```
tNode          *rootNode;
```

The average number of nodes created for an object during the testing of the program was around 530 000, sometimes being near 1 000 000, with multiple accesses per node. For

that reason additional data structures were created, in order to improve the performance of allocating memory for a node and accessing it.

To speed up memory allocation for a node, the allocation is done for 50 000 nodes at once. A data structure called `tNodePool` is holder for the allocated nodes:

```
typedef struct t_node_pool {       /***  container for nodes  ***/
  int                 length;      /* number of allocated nodes */
  int                 count;       /* number of occupied nodes  */
  tNode              *nodes;       /* list of allocated nodes   */
  struct t_node_pool *nextPool;    /* pointer to next container */
} tNodePool;
```

If the number of occupied nodes in a node pool reaches 50 000, a new node pool is created and attached to the previous pool as `nextPool`.

To speed up the per-level access to the octree nodes, a data structure `tLevel` was created, which contains all relevant information for a single level:

```
typedef struct t_level {           /*** single octree level data  ***/
  tNode  *nodes;                   /* list of nodes at this level   */
  int     nodeCount;               /* number of nodes at this level */
  double  nodeSize;                /* size of a node at this level  */
  double  cpuTime;                 /* CPU time used for this level  */
} tLevel;
```

`nodes` points to a node allocated in `tNodePool`. The nodes of the same level are concatenated through the `nextNode` field of the `tNode` structure.

Finally, all these structures are summarized in the data structure `tOctree`:

```
typedef struct t_octree {          /***      all octree data      ***/
  tNodePool *nodePool;             /* pointer to the node container */
  tLevel     level[MAX_POSSIBLE_OCTREE_DEPTH];
                                   /* array of octree levels        */
  int        maxDepth;             /* maximal possible octree depth */
} tOctree;
```

The global variable `octree` is the final holder for all octree data:

```
tOctree         octree;
```

The access to nodes of the level i is made through `octree.level[`i`].nodes`. Root node can be referred to as `rootNode`, which is the same as `octree.level[0].nodes`.

C-Routines

The source files for the program are `silhouette.h`, `silhouette.c` and `utilities.c`. `silhouette.c` contains the main routine, `run_silhouette` and all other routines are in the file `utilities.c`. The most important routines are `processNode`, which does all the necessary processing of a single octree node, including projection of the node into

the image plane and decision whether the node should be marked as black, white or gray, and `testBoundingBox`, which gives the percentage of image pixels within the node's projection that belong to the silhouette of the observed object. This percentage is the base for deciding the color of the octree node.

The main routine (`run_silhouette` in `silhouette.c`), written in pseudo-C code, looks as follows:

```
int run_silhouette(void)
{
   /* initialize global variables */

   /* parse the command line input parameters */

   /* parse transformation matrix input file and build the
      inverse matrix of the input transformation matrix */

   /* determine minimal voxel size and maximal octree depth */

   /* create initial octree, with a single root black node,
      centered at (0, 0, 0) */

   /* in case of real (grey scale) input images, extract
      objects silhouette from every input image, i.e.,
      create binary images */

   /* calculate the final transformation matrix for every input
      image, which is equal the inverted input transformation
      matrix multiplied with the viewing angle of the input image
      relative to the viewing angle of the first input image
      and store the result in an array, and store the results
      in an array */

   /* loop through octree levels: */
   for (currentLevel = 0;
        currentLevel <= octree.maxDepth;
        ++currentLevel)
   {
      /* ... */
      /* loop through all input images for the current level: */
      for (angleCounter = angleCounterStart;
           angleCounter < angleCounterEnd;
           angleCounter += angleCounterStep)
      {
         /* process each node at this level: */
         hPtr = octree.level[currentLevel].nodes;
         while (hPtr != NULL)
```

```
        {
          if (hPtr->type == 'B')
            processNode(hPtr, currentLevel);
          /* ... */
          hPtr = hPtr->nextNode;
        }
      }
   }

   /* build a 3D output object based on the octree */

   /* clean up resources and exit */
}
```

The routines `processNode` and `testBoundingBox` from `utilities.c` look as follows:

```
void processNode(tNode *thisNode, int thisNodeLevel)
{
   /* project the vertices into the image plane, by multiplying
      their octree coordinates with the final transformation
      matrix for the current input image and then correcting
      the calculated x and y image coordinates */

   /* find the minimum and maximum of x and y image coordinates
      of the eight projected vertices of the node and store
      them in localMinX, localMaxX, localMinY and localMaxY */

   /* test the bounding box of the projection of the node: */
   fractionBlack = testBoundingBox(localMinX, localMaxX,
                                   localMinY, localMaxY);

   /* depending on the test result, determine the color of the node: */
   if (fractionBlack >= blackThreshold)
   {
     thisNode->type = 'B';
   }
   else if (fractionBlack <= whiteThreshold)
   {
     thisNode->type = 'W';
   }
   else if (thisNodeLevel == octree.maxDepth)
   {
     if (fractionBlack < 0.5)
        thisNode->type = 'W';
     else
        thisNode->type = 'B';
```

```
   }
   else
   {
     thisNode->type = 'G';
     /* create eight child nodes of this node and mark them as black */
   }
   return;
}


double testBoundingBox(double xMin, double xMax,
                       double yMin, double yMax)
{
   /* determine the range of pixels that lie completely
      within the bounding box: */
      ixMin = (int) (xMin+1.49999);
      ixMax = (int) (xMax-0.5);
      iyMin = (int) (yMin+1.49999);
      iyMax = (int) (yMax-0.5);

   /* count the black and white pixels within this range:
      countB and countW */

   /* calculate the fraction of the black pixels: */
   fractionB = ((double) countB) / ((double) (countB+countW));

   /* in case there are no pixels within the range, just
      check the color of the pixels nearest to the center
      of the bounding box -- if it's black, set
      fractionB to 1.0, otherwise to 0.0 */

   return fractionB;
}
```

B User's Manual

B.1 Requirements

You have to have a working *Khoros 2001* [Kho] environment on your system. Make sure that the following toolboxes are installed: `datamanip`, `design` and `matrix`.

B.2 Installation

The installation procedure follows the rules of installing a new toolbox on a *Khoros 2001* [Kho] conform system. We assume that the distribution file `silhouette.tar.gz` is downloaded or copied to the directory `<toolbox-distrib-dir>` and that the toolbox should be installed in the directory `<toolbox-install-dir>`. The distribution file `silhouette.tar.gz` can be found at the following URL:
`http://www.prip.tuwien.ac.at/~tos/silhouette.tar.gz`

1. Copy the distribution file `silhouette.tar.gz` to the installation directory:
   ```
   cd <toolbox-install-dir>
   cp <toolbox-distrib-dir>/silhouette.tar.gz .
   ```

2. Uncompress the distribution file:
   ```
   gunzip silhouette.tar.gz
   tar xf silhouette.tar
   ```
   in the installation directory

3. Add the following line at the end of your `$HOME/.kri/KP2001/Toolboxes` file:
   `TOS_SILHOUETTE:<toolbox-install-dir>/tos_silhouette`

4. Go to the `objects` directory of the toolbox:
   `cd <toolbox-install-dir>/tos_silhouette/objects`

5. Generate the makefiles:
   ```
   kgenmake -recreate
   kmake Makefiles
   ```

6. Build and install the program file:
   ```
   kmake install
   ```

7. Optionally, to save some disk space, you can type:
   ```
   kmake clean
   rm <toolbox-install-dir>/silhouette.tar.gz
   ```
   The first command removes the object files created during the build of the program file and the second removes the distribution file from the installation directory.

The program named `silhouette` is now ready for use. It can be accessed through `cantata` — in the '`Glyph`' menu select the submenu '`Shape from...`', then its submenu '`Silhouette`' and then the entry '`Shape from Silhouette`' or you can start program directly by typing

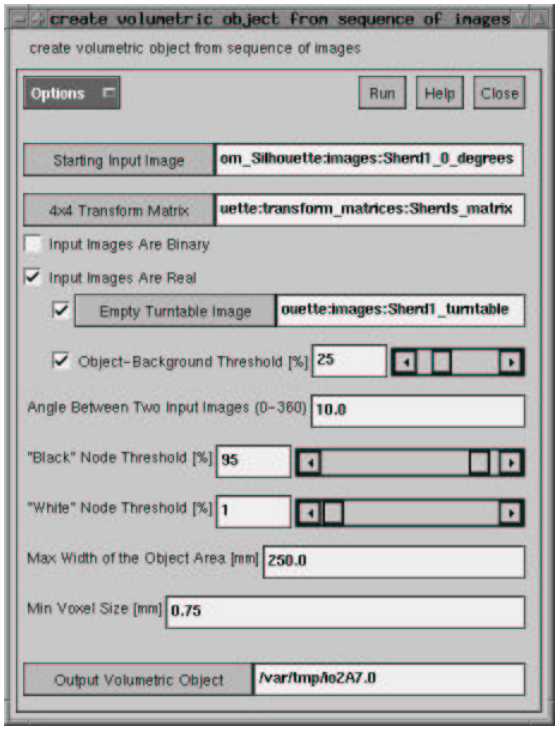
`<`*toolbox-install-dir*`>/bin/<`*conf-name*`>/silhouette -gui &`
`<`*conf-name*`>` depends on the system you have, for example, it can be `linux2.2.14`. Sample `cantata` workspaces for `silhouette` can be found under alias name `Shape_from_Silhouette:workspaces:<`*workspace-name*`>`.

B.3 Uninstallation

To remove the program and all its belonging files, do the following:

1. Remove the following line from your `$HOME/.kri/KP2001/Toolboxes` file:
   `TOS_SILHOUETTE:<`*toolbox-install-dir*`>/tos_silhouette`
2. Delete the toolbox directory and all of its subdirectories:
   `cd <`*toolbox-install-dir*`>`
   `rm -fR tos_silhouette`

B.4 Manual Page




```
PROGRAM NAME
     silhouette - create volumetric object from sequence of
     images

DESCRIPTION
     silhouette is a task that creates a 3D model of an object
```

```
     from sequence of images taken from multiple views. It is an
     implementation of Shape from Silhouette method, which takes
     the image silhouette of the observed object as the only
     interesting feature of an input image. The 3D model of the
     object is built based on the shapes of the object in all
     input images.


     The basic input for the task are:

     - the input images, specified by the path of the starting
       input image, whose name has to be a 4-digit number (plus
       the extension). An example of the name for an input
       image would be 1500.pgm

     - the transformation matrix which describes the
       transformation from the image coordinate system into
       object coordinate system. A calibration task, that can
       produce such a matrix as output, is TosCalib

     - viewing angle for each input image. It must be encoded
       in the names of input images, through a 4-digit number.
       An input image taken from 0 degrees has to have a name
       0000 (plus the extension), and a one taken from
       75 degrees 0750



     Output of the program is a 3D kdf object representing the
     contructed model of the object. It can be visualized using
     the Khoros taks gisosurface, followed by render. In gisosur-
     face, the contouring level has to be set to 1.


REQUIRED ARGUMENTS
     -i1    type: infile
            desc: first of the sequence of input images

     -i2    type: infile
            desc: image-to-realworld 4x4 transformation matrix

     -angle type: double
            desc: angle between two consequent input images
            (0-360)
            bounds: 0 < [-angle] < 360
```

```
-blackthres
       type: double
       desc: object node threshold [%]
       bounds: 0 < [-blackthres] < 100

-whitethres
       type: double
       desc: background node threshold [%]
       bounds: 0 < [-whitethres] < 100

-owidth
       type: double
       desc: maximal width of the object area [mm]
       bounds: value > 0.0

-vsize type: double
       desc: minimal voxel size [mm]
       bounds: value > 0.0

-o     type: outfile
       desc: output volumetric object

Mutually Exclusive Group; you must specify ONE of:

-bin   type: flag
       desc: input images are already binary

OR

     Mutually Inclusive Group; you must specify ALL of:

-real  type: flag
       desc: input images are real-world images

     AND

-i3    type: infile
       desc: turntable image, with no object on it
       default: {none}

     AND

-thres type: double
       desc: object-background threshold [%]
```

```
            default: 50
            bounds: 0 < [-thres] < 100


EXAMPLES
     none

SEE ALSO
     TOS_CALIB::TosCalib, GEOMETRY::gisosurface, GEOMETRY::render

RESTRICTIONS
     none

REFERENCES
COPYRIGHT
     Copyright (C) 1993 - 2000, Khoral Research, Inc., ("KRI").
     All rights reserved.  See $BOOTSTRAP/repos/license/License
     or run klicense.
```