3D Reconstruction using Voxel Coloring

Koen van de Sande and Rein van den Boomgaard
Informatics Institute
University of Amsterdam
The Netherlands

October 9, 2004

UNIVERSITEIT VAN AMSTERDAM

1 Introduction

During a Bachelors project [8] a practical setup for 3D reconstruction using the voxel coloring algorithm [10] was developed. The voxel coloring algorithm takes a number of photos with known camera position as input and outputs a 3D reconstruction consisting of a regular grid of cube-shaped voxels (see figure 1). Voxels are 'volume elements' in analogy with pixels.

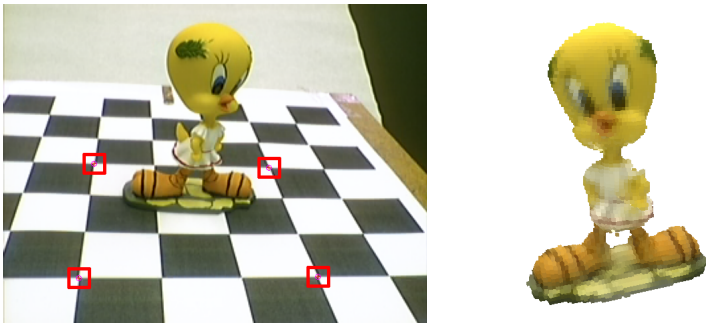

Figure 1: On the left an input image for voxel coloring is shown, with 4 marked reference points (red squares) which need to be selected by the user manually in every image to determine camera position and orientation. On the right a voxel reconstruction created using voxel coloring. The images show a figure called *Tweety* which is trademarked and copyrighted by Warner Bros.

The practical setup used consisted of a cheap webcam, a tripod and a home-built Meccano turntable. The setup could be used to take photos from different sides of an object automatically. We are going to perform 3D reconstruction using voxel coloring as well, but without using the turntable as it was shown in [8] that it isn't really needed.

During this assignment we are going to look at the different steps involved in 3D reconstruction with voxel coloring. Intuitively, the first step is to acquire images of the object we want to reconstruct (section 3). The second step is to determine the real-world position and orientation of the camera, as that information is required by the voxel coloring algorithm. We will cover the perspective camera model and camera calibration in section 5. After several in-

termediate steps[1] we will cover the actual voxel coloring. After voxel coloring is complete one wants to view the reconstruction made; for this purpose an OpenGL voxel viewer for Windows is available[2].

2 Roadmap

This assignment will span 3 lab sessions. The schedule for these lab sessions is as follows:

- **Week 1**. Image acquisition (section 3) and camera calibration (section 5). You should take pictures of an object suitable for voxel coloring and have performed extrinsic calibration on them.
  Hand-ins. You have to send us the code of your Matlab image acquisition function and answer the questions on the noisiness of the images acquired and on the intrinsic calibration.

- **Week 2**. Combine your images and calibration to form a voxel coloring dataset, implement the `projectVoxel` function and perform voxel coloring on your dataset (section 8). After that you can work on implementing consistency checks.
  Hand-ins. You have to send us the Matlab code of your `projectVoxel` function, your implementation of the original consistency check and a few screenshots of a reconstruction based on your own images.

- **Week 3**. Finish your implementation of the original and adaptive consistency checks, look at the effect of background removal on reconstruction quality and write your report.
  Hand-ins. Your report and your dataset(s).

The final result of this assignment will be a written report. Your grade will be based primarily on this report.

3 Step 1: Image acquisition

The Image Acquisition Toolbox for Matlab has been installed on all computers in the lab room and there are five webcams available, which should be plug and play. The Image Acquisition (IMAQ) toolbox will allow you to easily access the webcam from Matlab, independant of model or brand of webcam (though there is some configuration needed).

Matlab functions from the IMAQ toolbox you are going to need are:

- `imaqhelp [function]`, which can be used to obtain help about *function*. Use this to get specific help on IMAQ functions.

- `imaqhwinfo`, which returns information about image acquisition hardware on the system. Use this function to get the name of your image device (only works if the webcam is connected). Supply the device name to obtain specific information about a device, like the supported formats in the structure returned.

[1] These steps include background removal on your images and implementing several pieces of the voxel coloring algorithm yourself using MatLab inside the Voxel Coloring Framework [9]

[2] You are invited to port it to other operating systems if you have time to spare. The source can be found on the VCF website.

- `videoinput`, which can be used to obtain a handle to the webcam. Use the third parameter (format) to select resolution and color model. Leaving out this parameter will select the cameras default resolution.

- `preview`, which can be used to see a live video feed from the camera; for testing purposes.

- `getsnapshot`, which takes a photo using a webcam handle.

- `delete(handle)`, which you should use to release the handle to your webcam after you're done with it; if you do not release the handle then you may have to restart Matlab and/or disconnect the webcam.

Depending on the format of the output of your webcam, you may have to convert the result of `getsnapshot` to the RGB color model[3].

You can write your images to disk using the Matlab `imwrite` function. We suggest that you use the Windows Bitmap (.BMP) file format, as this is a lossless format supported by the camera toolbox we are going to use (section 5). Please do not use JPEG or any other lossy format, because we are going to edit these files later on as well: resaving multiple times using lossy compression will introduce artefacts. As a naming convention we suggest `serie1.bmp`, `serie2.bmp`, `serie3.bmp` ...`serie99.bmp`. Please replace `serie` with something else and make sure that the numbering of your images is continous.

Question. Compare multiple images taken of the same scene: how noisy are your images? Is there a great variation of the lighting in your image? How can you improve your image quality? What is the standard deviation of your images in the RGB channels?

Once you are able to capture images using a webcam, you should read the next section on the perspective camera model. If there is no webcam available at the moment, then read the next section as well; to do the tutorial of the camera toolbox (see section 5) you do not yet need your own images.

4 Perspective camera model

Now that we are able to capture images, we just need to know where the camera was positioned in every image and then we have all the input we need for voxel coloring. To retrieve camera position and orientation we will use the perspective camera model.

We will start our discussion with the perspective camera model, which corresponds to an ideal pinhole camera. In figure 2 we have illustrated our model[4]. The origin O is the pinhole of the camera. Point P (with coordinates (X, Y, Z)) lies on a real-world object and is projected onto the image plane I as point P'. Point C is called the image center or principal point and is the point where the optical axis intersects the image plane.

We will assume we know the distance between the pinhole O of the camera and the principal point C; this is the focal distance f of the camera. If we know the real-world coordinates X, Y and Z of point P, then it is possible to calculate the image coordinates x and y of the point

[3]The Philips webcam, for example, outputs the YCbCr color model and you can use the `ycbcr2rgb` function to convert an image to RGB

[4]The real situation is 3D, but since it is the same for X and Y, we decided to show it as a 2D scene where the vertical axis can be either, as that is more instructive.

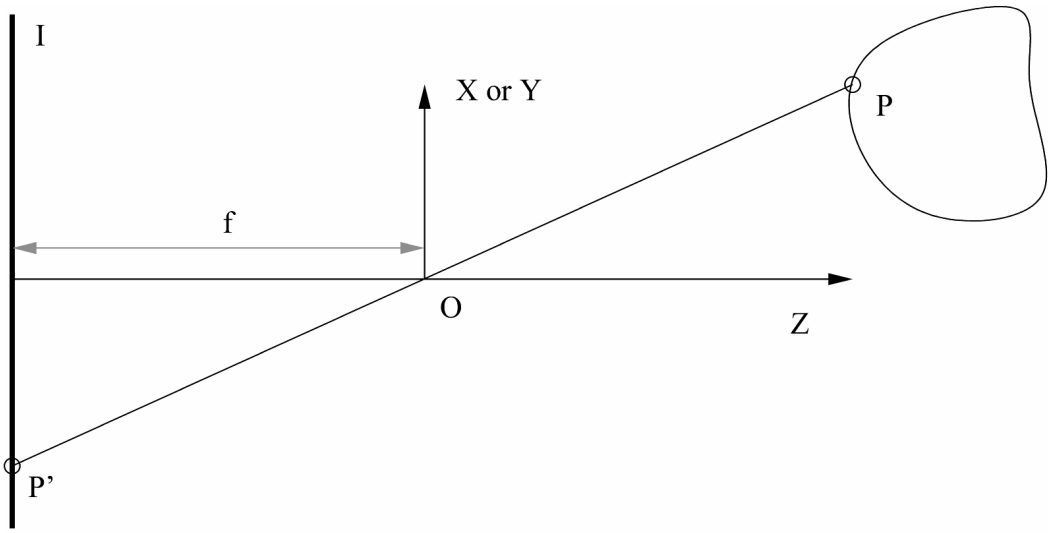

Figure 2: A schematic version of the perspective camera model. The origin O is the camera pinhole; point P is an object point with its Z component along the horizontal axis (also known as the optical axis); the vertical axis can be its X or Y component. I is the image plane which is perpendicular to the horizontal optical axis and point P' is the projection of P onto the image plane I. f and its double-arrowed line illustrate that the distance between the pinhole O and the principal point C is the focal distance f.

P' projected onto the image plane I:

$$\vec{x} = \begin{cases} x = f\frac{X}{Z} \\ \\ y = f\frac{Y}{Z} \end{cases}$$

We switch to homogenous coordinates so we can write the above equations in matrix form. Homogenous coordinates are explained in [4]. Basically an extra scaling coordinate w is added to a point and all other coordinates x, y, z need to be divided by w to get normal 3D coordinates out of a 4D homogenous vector. We use the symbol $\sim$ for equality between homogenous coordinates in the sense that they are equal if all vector elements are equal if the vector is normalised[5].

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \sim \begin{pmatrix} x' \\ y' \\ w' \end{pmatrix} = \begin{pmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

4.1 Modelling real cameras

Now we extend our model to make it more suitable for real-world usage. First of all, pixels are never exactly square in a real camera, so we add two factors k and l (expressed in $\frac{pixels}{cm}$) to account for this:

[5] A homogenous vector is normalised if the w coordinate is 1

$$\vec{x} = \begin{cases} x = kf\frac{X}{Z} \\ \\ y = lf\frac{Y}{Z} \end{cases}$$

Instead of writing kf and lf all the time, we will combine these factors into single terms f_x and f_y.

Another feature of real cameras is that the origin of the coordinate system generally does not lie at the principal point/image center C, but in or near one of the image corners. We introduce u_0 and v_0 to translate the principal point to the right location:

$$\vec{x} = \begin{cases} x = f_x\frac{X}{Z} + u_0 \\ \\ y = f_y\frac{Y}{Z} + v_0 \end{cases}$$

in matrix form this becomes:

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \sim \begin{pmatrix} f_x & 0 & u_0 & 0 \\ 0 & f_y & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

Another possible quirk in real-world cameras is that the angle between the x and y axes is not equal to 90 degrees. This is called *skew* and occurs when the camera CCD is not perpendicular to the optical axis (the lens and CCD are not properly aligned). Equations for skew are derived in appendix A of [12]. It introduces an extra non-zero element αf_x at position (1, 2) in our matrix and an additional term inside f_y. Since our cameras had virtually no skew, we will not discuss it here as it does not fundamentally change the rest of our discussion. The matrix we have derived will from now on be known as the intrinsic camera matrix $\mathbf{M_{int}}$ as it contains the intrinsic camera parameters (excluding distortion).

4.2 Separating camera frame and world frame

Currently our world frame has its origin at the pinhole of the camera, as it is equal to the camera frame. This is rather inflexible: we do not want the origin of our world coordinate system to lie at the pinhole. We introduce an intermediate rigid body transformation $\mathbf{M_{ext}}$ which converts from the world frame to the camera frame:

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \sim \begin{pmatrix} f_x & 0 & u_0 & 0 \\ 0 & f_y & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = \mathbf{M_{int}}\mathbf{M_{ext}}\vec{X}$$

When one inverts $\mathbf{M_{ext}}$ (so it converts from the camera to the world frame) and multiplies with the origin O, then one gets the camera position in world coordinates, which is equal to the point (t_x, t_y, t_z). The rotational part of the inverse rigid body transformation specifies the orientation of the camera.

4.3 Distortion

Besides real-world camera properties discussed thus far, there is also camera *distortion*. Most cameras tend to have radial distortion due to the shape of the lenses used. This distortion is typically done in 2D as it causes a certain pixel displacement due to irregular shape of the lense or the CCD chip. We will assume we have a distortion model which can apply a distortion to coordinates x and y and that coordinates can be undistorted again as well. If our distortion model could only go one way, then it would have been poorly chosen.

The distortion used by the Bouguet toolbox is the following:

$$x_d = x + k_1 r^2 x + k_2 r^4 x + 2k_3 xy + k_4(r^2 + 2x^2) + k_5 r^6 x$$
$$y_d = y + k_1 r^2 y + k_2 r^4 y + k_3(r^2 + 2y^2) + 2k_4 xy + k_5 r^6 y$$

with $r = x^2 + y^2$ and k_1 through k_5 the distortion parameters.

Bouguet applies this distortion after the extrinsic matrix has been applied ($\mathbf{M_{ext}}\vec{X}$), then the coordinates are flattened into 2 dimensions by doing the perspective part of the projection (dividing by z). On the remaining x and y the above equations are applied and then the 'remaining' part of the intrinsic matrix on x_d and y_d instead of x and y.

4.4 Determining extrinsic parameters from four planar points

We now move to one of the crucial parts of our camera calibration: we want to derive matrix $\mathbf{M_{ext}}$ (which can be converted to camera position and orientation) from four planar points with known correspondence between real-world coordinates and image coordinates. Since we know the points are planar, we choose them to lie in the $Z = 0$ plane in the real world[6]. We will presume our image coordinates have been undistorted already so our equations become:

$$\begin{pmatrix} x_i \\ y_i \\ 1 \end{pmatrix} \sim \begin{pmatrix} f_x & 0 & u_0 & 0 \\ 0 & f_y & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X_i \\ Y_i \\ 0 \\ 1 \end{pmatrix}$$

We have the above equation for all four reference points i: $i \in \{1, 2, 3, 4\}$. Since the third element of $\vec{X}$ is 0 for all points, we can leave out this element and the third column of $\mathbf{M_{int}}$ since their product will always be 0 after the matrix multiplication:

$$\begin{pmatrix} x_i \\ y_i \\ 1 \end{pmatrix} \sim \begin{pmatrix} f_x & 0 & u_0 & 0 \\ 0 & f_y & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & t_x \\ r_{21} & r_{22} & t_y \\ r_{31} & r_{32} & t_z \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X_i \\ Y_i \\ 1 \end{pmatrix}$$

The homogenous factor 1 of $\vec{X}$ will become the fourth element after this vector has been multiplied by $\mathbf{M_{ext}}$ and will always become 1 since $\mathbf{M_{ext}}$ is a rigid body transformation. If we look at $\mathbf{M_{int}}$, we see that this 1 isn't actually used anywhere (the fourth column only contains zeros), so we can safely leave out this term as well:

$$\begin{pmatrix} x_i \\ y_i \\ 1 \end{pmatrix} \sim \begin{pmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & t_x \\ r_{21} & r_{22} & t_y \\ r_{31} & r_{32} & t_z \end{pmatrix} \begin{pmatrix} X_i \\ Y_i \\ 1 \end{pmatrix}$$

[6]We do not lose generality here as it is easy to create a rigid body transformation which moves a known plane into the $Z = 0$ plane. This transformation can then be multiplied by our final result to get $\mathbf{M_{ext}}$

With the fourth column of $\mathbf{M_{int}}$ gone, this matrix has now become invertible. We will call the coordinates with $\mathbf{M_{int}}$ removed x' and y'. You might wonder now how it is possible to invert a perspective projection, as depth information is lost in such projections. There is a simple response to that: the projection isn't invertible at all; we have used a clever trick: the homogenous coordinate. The homogenous factor is a scaling factor, effectively allowing our point to represent the line of all 3D points that project onto our 2D point. After the projection is removed, this scaling factor is still 1 but if you change it (and x' and y' as well corresponding to the rules of homogenous coordinates) you get other points on that line.

$$\begin{pmatrix} x'_i \\ y'_i \\ 1 \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & t_x \\ r_{21} & r_{22} & t_y \\ r_{31} & r_{32} & t_z \end{pmatrix} \begin{pmatrix} X_i \\ Y_i \\ 1 \end{pmatrix} = \mathbf{H} \begin{pmatrix} X_i \\ Y_i \\ 1 \end{pmatrix}$$

We now have a correspondence between two planes (a real-world plane and the image plane). A mapping between two planes is known as a *plane to plane homography* and can be described by a 3x3 matrix $\mathbf{H}$. In the equation above we are dealing with a plane to plane homography so our matrix is this $\mathbf{H}$. Calculating a homography is a solved problem. We have 4 points with 2 coordinate-correspondences each; this gives us 8 equations. The homography matrix $\mathbf{H}$ has 9 elements, but it has only 8 degrees of freedom since the scale of the matrix does not matter[7]. We have to write out our 8 equations to see they are lineair for elements of $\mathbf{H}$, so we can lose the homogenous scaling factor. After converting these equations to matrix form and splitting off the elements of $\mathbf{H}$ into a separate vector h we get:

$$\begin{pmatrix} X_1 & Y_1 & 1 & 0 & 0 & 0 & -X_1x_1 & -Y_1x_1 & -x_1 \\ 0 & 0 & 0 & X_1 & Y_1 & 1 & -X_1y_1 & -Y_1y_1 & -y_1 \\ X_2 & Y_2 & 1 & 0 & 0 & 0 & -X_2x_2 & -Y_2x_2 & -x_2 \\ 0 & 0 & 0 & X_2 & Y_2 & 1 & -X_2y_2 & -Y_2y_2 & -y_2 \\ X_3 & Y_3 & 1 & 0 & 0 & 0 & -X_3x_3 & -Y_3x_3 & -x_3 \\ 0 & 0 & 0 & X_3 & Y_3 & 1 & -X_3y_3 & -Y_3y_3 & -y_3 \\ X_4 & Y_4 & 1 & 0 & 0 & 0 & -X_4x_4 & -Y_4x_4 & -x_4 \\ 0 & 0 & 0 & X_4 & Y_4 & 1 & -X_4y_4 & -Y_4y_4 & -y_4 \end{pmatrix} \begin{pmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{pmatrix} = 0$$

Linear algebra tells us that h can be calculated from the above equation[8], minimizing the error in reprojecting our points. Since we have 4 points, the non-trivial solution is uniquely defined. If you have more points available, then you can add them as extra rows to matrix $\mathbf{A}$.

We can reconstruct $\mathbf{M_{ext}}$ from $\mathbf{H}$ as column 1 and 2 are the same and define the base vectors for the X and Y axes. The Z base vector can be calculated from the first two by taking their cross product. The translational part of $\mathbf{M_{ext}}$ is equal to the 3rd column of $\mathbf{H}$. Our matrix is completed by giving the 4th row their default values for a rigid body transformation: 0, 0, 0 and 1.

$\mathbf{M_{ext}}$ is not necessarily a rigid body transformation now, as it is possible for the base vectors of the X and Y axes to have a length different from 1 (they are slightly scaled then). This happens because the reference points are generally not exact: they are noisy.

[7] This is due to homogenous coordinates; if you divide all matrix elements by a certain factor a, then all elements of the resulting vector will also be divided by this a. As the resulting vector is homogenous, the division by a is gone after normalisation.

[8] A solution where h has length 1 that is, to prevent trivial solutions such as the null vector

Bouguet solves this by taking the $\mathbf{M_{ext}}$ we have derived as an initial guess for the real $\mathbf{M_{ext}}$. He then converts the rotational part to Rodrigues coordinates[9]. He then performs a gradient descent to minimize the reprojection error in the reference points on the 3 Rodrigues coordinates and the 3 elements of the translation vector. The Rodrigues coordinates can only describe rotations without scaling and ensure that the final version of matrix $\mathbf{M_{ext}}$ is indeed a rigid body transformation.

5 Step 2: Camera calibration

We are now going to actually perform the calibration using the model described in the previous section. Because implementing this yourself would take too much time, we are going to use Bouguets camera toolbox:

- Go to the Bouguet website [1] and download his toolbox.
- Extract the toolbox (preferably on your account as you won't have to reinstall it all the time then) and add the root folder of this toolbox to your Matlab paths (see the File menu and then 'Set path...'). This will allow you to run the calibration toolbox from any folder instead of just the folder where the toolbox is located.
- Go to the `calibration_pattern` subfolder in the toolbox and print the `pattern.pdf` file. Measure the size of the squares on your printout; they are typically around 30mm, but you should use the exact size you measure later on during the calibration.
- You can start the toolbox by running `calib_gui`.
- In appendix A (taken from [8]) we describe some additional details of the toolbox.
- On the website of the Camera calibration toolbox a detailed example is available. Go to [1] and look for *First calibration example - Corner extraction, calibration, additional tools*.

5.1 Intrinsic camera calibration

To be able to retrieve camera position and orientation from a single image later, an intrinsic calibration will have to be performed first on images of the checkerboard pattern you printed shown in different orientations. Take several images of the calibration pattern in different orientations, but make sure that you keep the paper absolutely flat! The pattern should be a placed in a single plane. Before taking pictures, please read the rest of this section as there are various hints and tips to help you take images usable for voxel coloring.

After you've performed the intrinsic calibration, you can save your results to disk using the 'Save' button in the GUI.

Question. How often should this intrinsic calibration be performed for a single camera? When do you need to redo it? How big is the pixel error?

[9]Rodrigues coordinates describe a rotation with just 3 parameters: an unit vector (2 parameters) and a rotation angle around this vector (1 parameter). See [3] for an introduction to Rodrigues coordinates.

5.2 Extrinsic camera calibration

During the extrinsic calibration step, you are going to determine the camera position from 4 reference points (as described in section 4.4). In every image of the object, you have to select 4 planar reference points. In principle any four planar points will do, but this makes it harder to process a lot of images because then you also have to specify how these points are related inside the plane. Instead, a Matlab script is available to let you process multiple images after one another. In the images you will have to select the exact same reference points in every image, in the exact same order. Moreover, these points together have to form a rectangle (on the calibration pattern grid of course). So if the object obscures one of your reference points, then the image is not usable! We suggest that you acquire about 10 good images[10].

When taking images, you should already know which reference points you are going to use, so you can see if an image is usable!

The Matlab function `calibrate_sequence` (included in `vcf_package.zip` in the `calibration` folder) can process series of images and will allow you to select the reference points in all images. This function is also available in the camera toolbox, but there you have to enter the name for every image you want to process and re-enter parameters all the time. As long as your files are numbered consecutively and you have saved your calibration results[11] using the 'Save' button in the toolbox in the same folder as the images you are going to process, there should be no problems.

The parameters of `calibrate_sequence` are:

- `prefix`: the filenames of all your images, excluding number and file extension.
- `suffix`: the file extension of your images.
- `startNumber`: the number of the first image in the sequence to be processed.
- `dX` and `dY`: the size, in millimeters, of the rectangle you are going to select in every image. If your calibration pattern has squares sized 30mm and the rectangle is 3 by 4 squars, then `dX` would be 90 and `dY` 120. When selecting points, the first point you select defines the origin, the line between the first and second point will be the X-axis and the Y-axis will lie between the first and fourth point.
- `wintx` and `winty`: the size of the area around the points you select which will be scanned for corners (using the corner finder of the Bouguet toolbox). Typical values for webcam resolutions are around 8 (giving a window size of 17x17), for digital camera resolutions they are around 20 (giving a window size of 41x41).

If you want to skip an image (not all 4 reference points are available), then close all Figure windows of Matlab and the image will be skipped. For all suitable images a `.CPCC` file will be created which contains all intrinsic parameters (which is the same for all images) and all extrinsic parameters for that image.

5.3 Construct a Voxel Coloring dataset out of your images

Create a folder with all your bitmap images and the `.CPCC` files; give this folder a meaningful name, for example `rose_digitaal`. In the parent folder of this folder (the folder which contains

[10] You can acquire more images, as you can always leave out images if you have to many

[11] `Calib_results.m` contains these results

the `rose_digitaal` folder) create a textfile named `dataset_rose.dataset`. On the first line of this file place the name of the folder with all content, on the second line add the prefix text of all your image filenames (for `series40.bmp`, this would be `series`) and on the third line include the number of the image that should be processed.

An example file would be:

```
rose_digitaal
series
40
```

Recommended: Make a backup of your data folder now. Later on you might need to edit this file to remove the background and losing the original images can be a problem as you won't be able to compare them then for your report.

Optionally, you can now convert all `.BMP` files into the PNG file format as that is a compressed lossless file format supported by the Voxel Coloring Framework [9]. We have provided the script `bmp2png.m` to convert all files with the `.BMP` extension in the current folder to the PNG format.

6 Voxel Coloring

Voxel coloring is an algorithm to reconstruct the 3D shape from a number of input photographs with known camera locations. It was originally introduced by Seitz and Dyer in [10]. The reconstruction consists of a regular grid of cube-shaped voxels. Voxels are 'volume elements' in analogy with pixels. Like a pixel, every voxel is associated with a color. See figure 3 for an example of a voxel representation. More information about these representations can be found in [4].

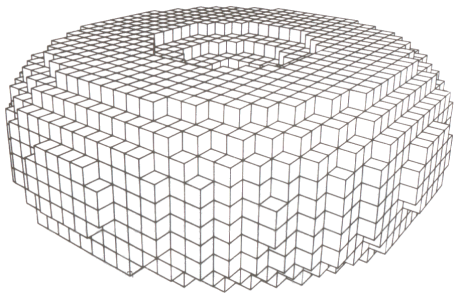

Figure 3: Example of a voxel representation. Picture taken from [4].

The problem addressed by voxel coloring is the assignment of colors to points in a 3D volume as to be consistent with the input photographs. If a single point has the same color in all images from which it is visible, then it should be given that color. If the colors do not match, then there probably is no point in the 3D volume there and the voxel should be removed. This basic principle underlying voxel coloring is illustrated in figure 4.

When rendering the voxel reconstruction from the photo viewpoints, it should be 'identical' to the input photographs (because of limitations of the voxel representation used, they are not always identical). When a reconstruction matches with the input images, it is *photo-consistent*. However, a photo-consistent reconstruction is not unique, as is illustrated in figure 5. The

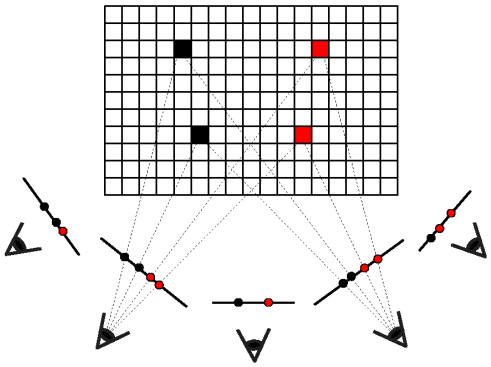

Figure 4: Given a set of input images and a voxel space, we want to assign colors to voxels in a way that is consistent with all images. Picture taken from [11].

different types of ambiguity are explored in [11].

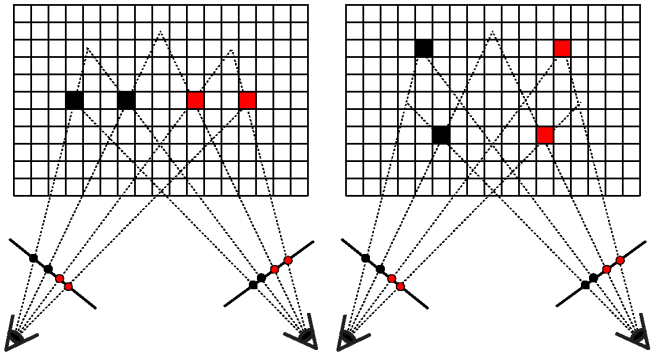

Figure 5: Both voxel colorings appear to be identical from these two viewpoints but have no voxels in common. Picture taken from [11].

The *photo-hull*, introduced in [5], is the union of all photo-consistent shapes and is thus the maximally photo-consistent shape. Contrary to a photo-consistent reconstruction, the photo-hull is uniquely defined and happens to be the reconstruction created by voxel coloring (this is shown in [5]). If we apply voxel coloring to the case of figure 5 now, then the reconstruction is uniquely defined as is shown in figure 6.

The working of voxel coloring relies heavily on the ability to compare colors between different images. These colors are only the same if we assume a Lambertian reflection model, which says that all objects in the scene reflect the light equally in all direction. If we do not assume a Lambertian model, then the amount of light reflected would depend on the camera angle being used, which means that a single point in space can have multiple colors.

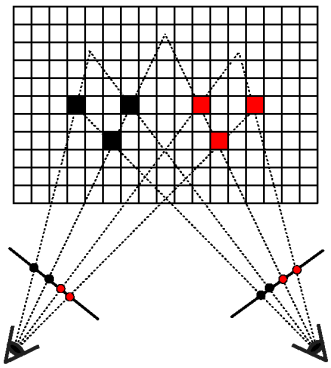

Figure 6: Voxel coloring using the photo-hull. Every voxel has the same color in every reconstruction in which it is contained. Picture taken from [11].

6.1 Occlusion handling

In order to handle occlusion properly, the voxel coloring algorithm traverses the voxel space in a special order. Voxels closer to the camera are visited first. This ensures that a voxel cannot be occluded by an unvisited voxel (since an occluding voxel must be closer to the camera, it has already been visited).

Across all input images, this is known as the *ordinal visibility constraint* (see [10]), which ensures that for scene points P and Q, if P occludes Q, there is some metric which says that $\|P\|$ is smaller than $\|Q\|$ across all input images (thus if P occludes Q, then there are no images possible in which Q is occluded by P).

The ordinal visibility constraint is satisfied when no scene point is contained within the convex hull of camera centers. In case a pinhole camera is assumed, this camera center is equal to the pinhole of the camera. The metric above can be taken as the distance to the to the convex hull around the camera centers.

In figure 7 two practical camera setups are shown which satisfy the ordinal visibility constraint. Note that the constraint implies that having two cameras on exactly opposite sides is not allowed. Because of this, not all sides of an object can be reconstructed. There are generalizations of voxel coloring, such as Space Carving [5] and Generalized Voxel Coloring [2], which can both handle arbitrary camera positions.

7 Voxel Coloring algorithm

Using the principles described in the previous section, we can now construct the basic voxel coloring algorithm. This algorithm assumes that all voxels V are traversed in an order which satisfies the ordinal visibility constraint (described in section 6.1).

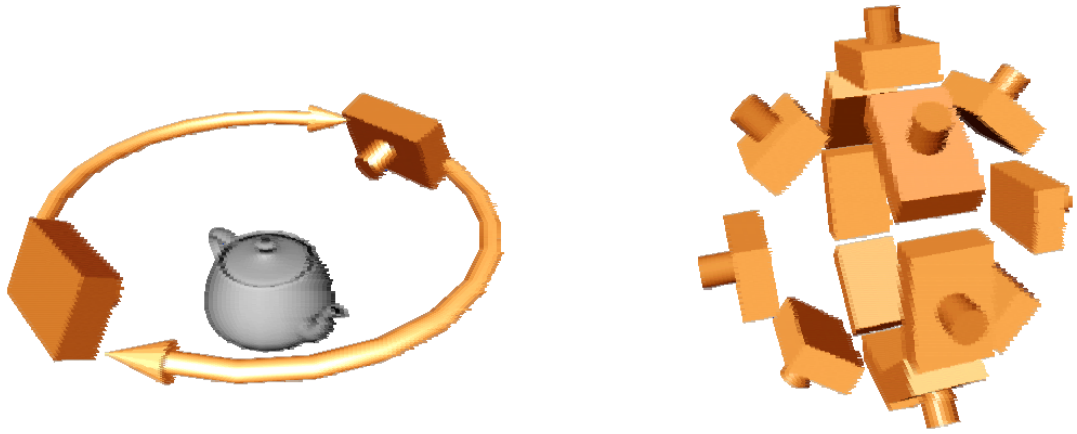

Figure 7: Camera configurations that satisfy the ordinal visibility constraint. Picture taken from [11].

```
for every voxel V do
  pixels ← ∅
  for all images I do
    pixels ← pixels ∪ selectUnmarked(projectVoxel(V, I))
  end for
  consistent ← consistencyCheck(pixels)
  if pixels ≠ ∅ and consistent then
    colorVoxel(V, mean(pixels))
    mark(pixels)
  else
    carveVoxel(V)
  end if
end for
```

Every voxel is projected onto the input images, and all pixels that have not yet been marked (as done) are collected. If there are no pixels the voxel projects onto, then the voxel is carved. If the pixel collection is consistently colored, then the voxel is accepted and given the mean color of the collection. The pixels are then marked (to indicate they are done). If the collection does not have a consistent color, then the voxel is carved.

A common optimization is to use a *sprite projection* when projecting a voxel onto an image: all vertices of the voxel are projected and a bounding box around the projected points is used to approximate the shape of the actual voxel projection. Constructing a bounding box around eight projected points is much faster than scan-converting the actual projection, while being more accurate than a simple point projection (which only projects the center of the voxel).

8 Step 3: Performing Voxel Coloring

You should download and extract `vcf_package.zip` from the labsite if you have not done so already. This package contains a precompiled version of the Voxel Coloring Framework modified for this assignment. From the voxel coloring algorithm described above, the `projectVoxel` and `consistencyCheck` functions are not available: you have to implement these yourself[12].

8.1 Using your own dataset

You should look at the file `your_reconstruction.m` which contains the voxel coloring configuration. You should change the `dataset` variable to your own and correct the path to the Voxel Viewer.

For your dataset you can specify the initial voxel volume using the `vcs.minimum` and `vcs.maximum` variables, which specify the lower and upper bounds on the X, Y, Z values in the voxel volume. These values are defined in real-world coordinates in millimeters so with knowledge of the rectangle size of your extrinsic calibration you should not have any problem setting these properly, except for perhaps the height (Z-axis).

With `vcs.resolution` you can specify the number of voxels in the X,Y,Z direction. Using low resolutions is recommended when parts of voxel coloring are implemented in Matlab (sizes in the order of 20-30 give reasonable runtimes).

Question Why should you use cube-shaped voxels instead of non-cubic voxels like $1x2x2$?

The `parameters` variable controls which consistency check is used. When set to `[0 45]` it will use the original consistency check with a threshold of 45. You can use this one when testing the `projectVoxel` function you have to implement. When you want to use your own Matlab consistency check, you should set it to `[4 0]`.

8.2 projectVoxel

You should implement the projectVoxel function in the file `voxelcolor_matlab_projectvoxel.m`. The parameter `p` contains *several* points using homogenous coordinates[13]. Inside the parameter `calibration` you get the calibration settings for the image; this 22-element vector contains: *fc1,fc2,cc1,cc2,alphac,kc1,kc2,kc3,kc4,kc5,R11,R12,R13,R21,R22,R23,R31,R32,R33,T1,T2,T3* where kc1 through kc5 are distortion parameters and alphac is not used by default.

After you have implemented this function you can try your own reconstruction for the first time! Include the implementation of this function in an appendix of your report.

Note. If your code manages to crash the Voxel Coloring DLL, then you might get an error 'Camera not assigned' the next time you try to run your reconstruction. You can resolve this by typing `clear voxelcolor` in the Matlab prompt; this will forcefully unload the crashed instance of the Voxel Coloring DLL and your reconstruction will work again.

[12] You must write the functions asked for in Matlab

[13] Because the overhead of calling a Matlab function is large compared to the cost of the actual computation, the points in an entire layer of voxels are supplied at the same time, introducing just n calls for n layers in the reconstruction instead of several million calls if all points are projected one by one.

8.3 consistencyCheck

In a file called `voxelcolor_matlab_consistencycheck.m` you should place the implementation of your own consistency check. The parameters your consistency check needs will either have to be hardcoded or passed through Matlab global variables[14].

The consistency checks you have to implement are the original and the adaptive consistency check. The original consistency is the one which was included in the first voxel coloring article by Seitz and Dyer [10]. Read either Van de Sande [8] or Verstraeten [12] for details on these consistency checks.

Experiment with different parameters for the consistency checks and compare them in your report. You have to include the code of the consistency checks in an appendix of your report.

8.4 Background removal

You may have noticed that the image background surrounding the object you are trying to reconstruct is causing artefacts. You can improve results by replacing these background parts in your input images with black (RGB (0,0,0)). Compare reconstructions with and without background removal in your report.

Question. Is it better to remove too much background (e.g. remove some object parts as well) or is it better to leave some of the background in? Why?

Optional. Evaluate the consistency checks against your dataset with background included and with the background removed. Do you notice differences in performance (program runtime, reconstruction results)?

[14] A line with (`global parameters` might allow access to the global parameters variable from your configuration script, but this might not be reliable if this variable does not exist

9 Checklist

Your report should contain the following:

- Code of your image acquisition function.
- An evaluation of the noisiness of the images you have acquired (question)
- Comment on the intrinsic calibration (question)
- Show parts of your dataset (calibration images and object images) and report their noisiness
- Why use cube-shaped voxels (question)
- When showing any reconstruction, be sure mention resolution, consistency check, real-world voxel size.
- A comparison of the original to the adaptive consistency check.
- A comparison of reconstruction results without and with background removed (also includes a question on background removal)
- *optional* Compare the consistency checks against (no) background removal: what are the best combinations?
- Code with documented implementation of `projectVoxel`, the original consistency check and the adaptive consistency check.

We would like to get a copy of your dataset with background and without background (if applicable), the `.dataset` file and the reconstruction script(s), send this by e-mail in a ZIP file.

A Camera calibration toolbox

We have used the Camera Calibration Toolbox for Matlab [1] in our 3D reconstruction setup. With this toolbox, we can determine camera properties, position and orientation. The camera properties, which will be referred to as intrinsic parameters, do not change between different images. The camera position and orientation do change between images and are commonly referred to as the extrinsic parameters.

A.1 Intrinsic parameters

The first step in calibrating the camera is to take numerous pictures of a checkerboard pattern (see figure 8) and use these to determine the intrinsic parameters of the camera. These intrinsic parameters include focal distance, principal point, skew and numerous radial distortion parameters.

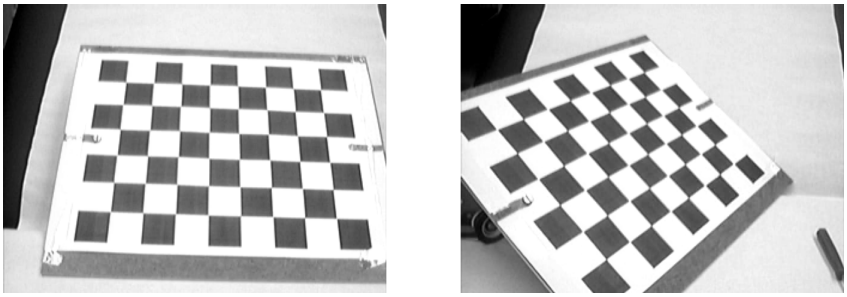

Figure 8: Two calibration images with the checkerboard pattern

First, the toolbox asks for the size of the squares in the calibration pattern in millimeters. By having a correspondence between the squares and a real-world metric (millimeters), the calibration toolbox can express the intrinsic parameters in millimeters, where applicable.

In every picture with a checkerboard pattern, the user has to select 4 corners which together form a rectangle (see figure 9). After selecting these points, the toolbox will automatically count the number of squares inside the selected rectangle. Should the toolbox fail to count the number of squares automatically, it will consult the user. With knowledge of the number of squares, the corners inside the selected rectangle can be estimated by interpolating the 4 points selected. An edge detector is used on the estimated corners to determine the exact locations of the corners. This is illustrated in figure 10.

After all calibration images have been processed, the toolbox can perform the calibration. It makes a first estimation of the intrinsic parameters and then performs an iterative gradient search in order to minimize the error over all the detected corners in the calibration images.

The intrinsic parameters are now known, but the toolbox offers additional functionality to analyze calibration results and improve them further, if needed. We will not discuss the analysis features of the toolbox here; see the tutorials in [1].

Please note that the intrinsic camera parameters change if the camera changes focus (because the focal distance changes), so be sure to set your camera to manual focus to get correct results.

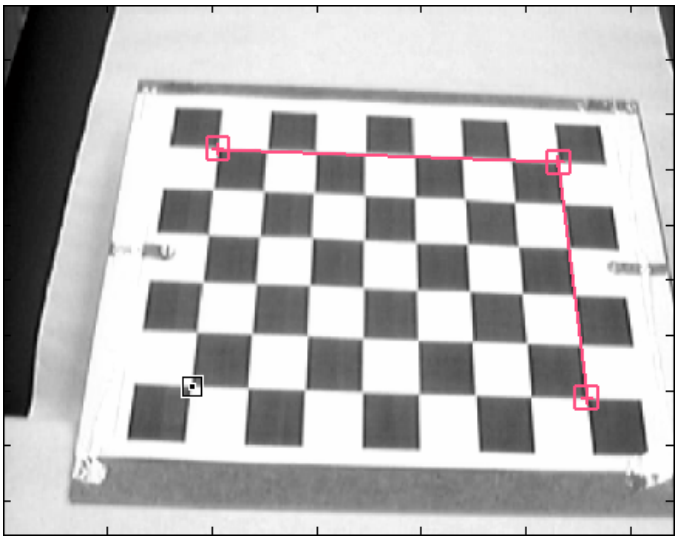

Figure 9: Three corners of the calibration rectangle have been selected and the cursor is over the fourth corner

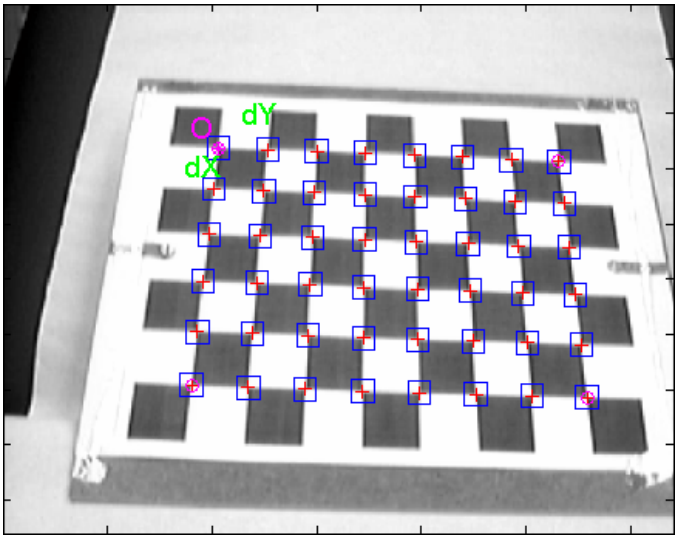

Figure 10: All corners inside the calibration rectangle have been extracted (red crosses). The blue squares indicate the area where the edge detector operated.

A.2 Extrinsic parameters

For voxel coloring, we have to know the position and orientation of the camera. Once the intrinsic calibration is complete, the toolbox offers a function to determine the extrinsic parameters for an image. These extrinsic parameters together form a rigid body transformation, which is made up of a rotation and a translation, giving the orientation and position of the camera.

To determine the extrinsic parameters for an image, the user has to select four reference points which together form a planar rectangle (see figure 11). These four points have to be selected in the exact same order in every image, which ensures that all images have the same coordinate system. In section 4.4 we have shown that it is possible to determine the extrinsic parameters from just 4 planar points.

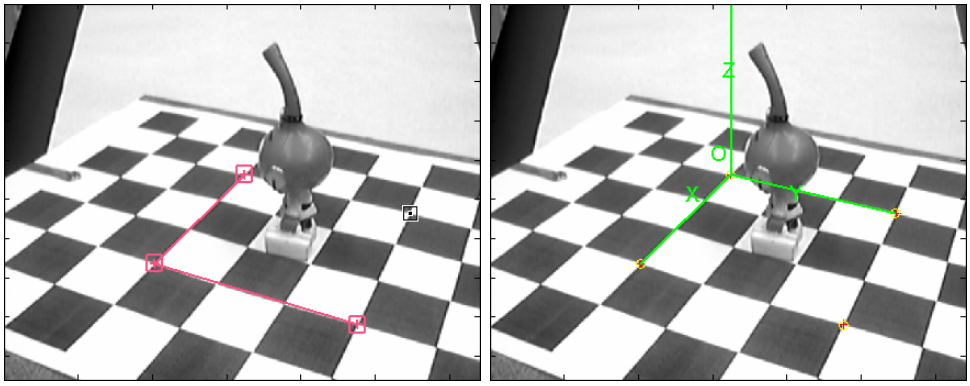

Figure 11: On the left four reference points are being selected. On the right the coordinate system extracted is shown. The red crosses indicate the points selected and the yellow circles indicate the reprojected points using the extrinsic parameters.

After selecting the four points, the user has to enter the width and height of the selected rectangle. If it consists of 3 by 3 squares measuring 30x30mm each, then this is 90mm by 90mm. We thus treat the entire rectangle as a single 'square'. We do this because the corners inside the rectangle will generally be obscured by an object (see figure 11), and we do not want the toolbox to use these points (as this gives strange results). After entering the real-world sizes the toolbox can determine the extrinsic parameters.

If one wants to know the position of the camera, then one has to invert the rigid body transformation matrix (which can be constructed from the extrinsic parameters) and extract the translation, as that is the camera position.

References

[1] Jean-Yves Bouguet. *Camera calibration toolbox for Matlab.* `http://www.vision.caltech.edu/bouguetj/calib_doc`

[2] Bruce Culbertson, Tom Malzbender and Greg Slabaugh. *Generalized Voxel Coloring.* In Proceedings of the ICCV Workshop, Vision Algorithms Theory and Practice, Springer-Verlag Lecture Notes in Computer Science 1883, pages 100-115, 1999.

[3] Laura Downes and Alex Berg. *CS184: Computing rotations in 3D.* `http://www.cs.berkeley.edu/~ug/slide/pipeline/assignments/as5/rotation.html`

[4] J.D. Foley, A. van Dam, S.K. Feiner and J.F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesly Publishing Co, section 12.6, 1990.

[5] Kiriakos N. Kutulakos and Steven M. Seitz. *A theory of shape by space carving.* In International Journal of Computer Vision, 38(3): pages 198-218, 2000.

[6] *Phidgets Inc. - Unique USB Interfaces* `http://www.phidgets.com`

[7] Andrew C. Prock and Charles R. Dyer. *Towards real-time voxel coloring.* In Proceedings of the DARPA Image Understanding Workshop, pages 315-321, 1998.

[8] Koen E.A. van de Sande. *A Practical Setup for Voxel Coloring using off-the-shelf Components.* Bachelor Thesis, University of Amsterdam, June 2004.
Available from `http://voxelcoloring.sourceforge.net`.

[9] Koen E.A. van de Sande. *Voxel Coloring Framework.* `http://voxelcoloring.sourceforge.net`

[10] Steven M. Seitz and Charles R. Dyer. *Photorealistic scene reconstruction by voxel coloring.* In Proceedings of the Computer Vision and Pattern Recognition Conference, pages 1067-1073, 1997.

[11] Steven M. Seitz and Charles R. Dyer. *Photorealistic scene reconstruction by voxel coloring.* In International Journal of Computer Vision, 35(2): pages 151-173, 1999.

[12] Thomas Verstraeten. *3D Scene Reconstruction using Voxel Coloring.* Master Thesis, University of Amsterdam, August 2003.