Images To 3D

Erocr

February 27, 2026

Abstract

The goal of this project is to develop a program that create a 3d model from some images. You take some photos of an object, and then, the program should generate a 3d mesh of the object .

1 Use case description

This part will describe how to use this program.

1.1 Requirements

If you want this program to create the 3d model correctly, you have to be careful at some details.

1.1.1 The light

The light is something really useful to understand the 3D in an image. However, analyse it is pretty complicated. In fact, we have to know where does the light source come from, all the characteristics of this light source, and even how does the object reflect this light. And all this information are really hard to get. So, we decide to ignore the light. We prefer have less light disturbance possible. In order that this program execute at his best, you should try to put a uniform light on the objects.

1.1.2 The germ

The algorithm used to create the voxel use a germ. In other words, we take a first cube in the 3D object, and then we add cube per cube until we have something really similar to the object. The problem is on the choice of the first element. We decided to chose it via a reference. The reference is an image that you have to put on the object. We shall be able to see this image in all the photos taken. This image has also another utility. It permits to know what is the relative orientation of the camera.

1.1.3 The file format

The program will output a .vox file. It is a file format that stores voxels. You can then view the result, or transform in another file format with https://imagetostl.com/view-vox-online.

2 Technical description

2.1 Image preprocessing

2.1.1 Removing light

We want to remove the light, and to have in image as if there was a complete uniform light. We will suppose that it is lighted with a white light, with no reflections. We will approximate the effect of light on a surface as a scalar multiplication. So, to erase that scalar, we will normalize the color, and then multiply it by 255. **The problems:** - Two different colors can be considered as same. - If the light source is not white, it can be completely unuseful

2.2 Find the seed

This part will search the reference image in the image, and then will deduct the camera position from the reference.

2.2.1 find black things

The reference image is formed by black triangles. The first part is to determine with a threshold where is black. We take only the pixels which color has a norm below the threshold. So we get a grid, with ones where the pixel is almost black, and zeros otherwise.

2.2.2 Find contour

A lot of algorithms could be used to do that.

A first one is with convolution matrices. We take the matrix (0 1 0) (1 -1 1) (0 1 0) The convolution gives us numbers above one if they are in the contour and 0 or less if they are not. We have to be careful because this solution take a lot of time. If we want to find something better, this page look pretty interesting: `https://www.imageprocessingplace.com/downloads_V3/root_downloads/tutorials/contour_tracing_Abeer_George_Ghuneim/alg.html`

Another idea is to use the Suzuki's algorithm : `https://theailearner.com/tag/suzuki-contour-algorithm-opencv`

2.2.3 Contours to polygons

To transform the contours in polygon, we use the Douglas-Peucker algorithm. We take two random points in the contour (A and B). Then, we search the point the most far to the line (C). If C is below a threshold, we remove all the points between A and B. If it is above, we call recursively the algorithm on A and C, and on C and B.

2.2.4 Find the reference

Now, we have a list of polygons. But how to find the reference ? We chose the reference to simplify as possible this part.

We draw inspiration from the qr-code detection algorithms.

2.2.5 Find the camera orientation

This is the hard part.

Idea 1: homography :

Here some leads on where to look: - homography: `https://en.wikipedia.org/wiki/Homography_(computer_vision)` - `https://nulldog.com/camera-pose-estimation-from-homography-of-4-points`

Here a quick explanation on how it works: We have the following equation

$$\lambda P T \vec{x} = \vec{p}$$

P the projection matrix
T the camera transformation matrix
$\vec{x}$ the point in 3D space
λ the zooming factor
$\vec{p}$ the position on the image (in 2D space)

If we develop it, we get this equation

$$\lambda \begin{pmatrix} f & 0 & C_x \\ 0 & f & C_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{1,1} & r_{2,1} & r_{3,1} & t_x \\ r_{1,2} & r_{2,2} & r_{3,2} & t_y \\ r_{1,3} & r_{2,3} & r_{3,4} & t_z \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x_s \\ y_s \\ s \end{pmatrix}$$

To simplify the problem, we take the spatial coordinates so that z = 0. So, we get the following equation

$$\lambda \begin{pmatrix} f & 0 & C_x \\ 0 & f & C_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{1,1} & r_{2,1} & t_x \\ r_{1,2} & r_{2,2} & t_y \\ r_{1,3} & r_{2,3} & t_z \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x_s \\ y_s \\ s \end{pmatrix}$$

Let H be λPT. H is a square 3x3 matrix ! With the homography method, we can compute this H matrix (called the homography matrix).

This matrix is all we need. In fact, we can use it to project the voxels into an image.

Idea 2: Gradient descent :

We want the projection matrix. So a matrix who looks like

$$A := \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix}$$

Here some notations:
We note the points in 3D space of the reference in homogeneous form $v_1, v_2, \ldots, v_n$
With $\forall i \leq n, v_i = \begin{pmatrix} x_i \\ y_i \\ z_i \\ 1 \end{pmatrix}$
The points in 2D space (after projection) are $u_1, u_2, \ldots, u_n$
We want A so that $\forall i \leq n, Av_i = \begin{pmatrix} s_i \\ t_i \\ d_i \end{pmatrix}$ and $d_i u_i = \begin{pmatrix} s_i \\ t_i \end{pmatrix}$
Le $u_i = \begin{pmatrix} u_{i,1} \\ u_{i,2} \end{pmatrix}$
Let the loss be

$$\begin{aligned} \mathcal{L}(A) &= \sum_{0<k\leq n} \big(s_k - u_{k,1} d_k\big)^2 + \big(t_k - u_{k,2} d_k\big)^2 \\ &= \sum_{0<k\leq n} \big(a_{11}x_k + a_{12}y_k + a_{13}z_k + a_{14} - u_{k,1}(a_{31}x_k + a_{32}y_k + a_{33}z_k + a_{34})\big)^2 \\ &+ \sum_{0<k\leq n} \big((a_{21}x_k + a_{22}y_k + a_{23}z_k + a_{24} - u_{k,2}(a_{31}x_k + a_{32}y_k + a_{33}z_k + a_{34})\big)^2 \end{aligned} \tag{1}$$

Our goal is now to find the matrix A which minimize $\mathcal{L}$. Let's derivate this stuff

$$\forall i \in \{1,2\}, \frac{\partial \mathcal{L}}{\partial a_{ij}}(A) = 2\sum_{k\leq n}(a_{i1}x_k + a_{i2}y_k + a_{i3}z_k + a_{i4} - u_{k,1}(a_{31}x_k + a_{32}y_k + a_{33}z_k + a_{34})) \times v_{i,k}$$

for i=3, we get

$$\frac{\partial \mathcal{L}}{\partial a_{3j}}(A) = 2\sum_{k\leq n}(a_{i1}x_k + a_{i2}y_k + a_{i3}z_k + a_{i4} - u_{k,1}(a_{31}x_k + a_{32}y_k + a_{33}z_k + a_{34})) \times v_{i,j} \times (-u_{i,1} - u_{i,2})$$

And so, we get a linear system for the variables a_{ij} ! So, we can use the gaussian elimination to solve this equations.

2.2.6 Conclusion

Finally, the seed position is just under the cube. It is pretty easy to find a valid position.

2.3 The germ algorithm

Once we chose the seed, we will place cubes until we have something pretty similar to the target.

2.3.1 choice of the next cube

At each iteration, we will chose a cube from all the neighbors of a known cube. For each cube, we will compute a likelihood. We will then chose the cube that maximize the likelihood.

2.3.2 Likelihood

To compute the likelihood of a cube, we start by finding the pixels it affects on each image. To do so, we project the cube with the projection matrices of each image we found earlier. We have to be aware of the fact that some other cubes could be in front, so we have to project all the cubes to remove the non visible parts. We will take the mean / median / mean by image / idk. This will be the color that would take the cube if we chose it.
Then, we can compute the likelihood. It is the sum of the square differences.

$$\mathcal{L}(p) = \sum_{im \in images} \sum_{px \in affected(im)} (col(px) - col(p))^2$$

2.3.3 When do we stop the algrithm ?

For now I don't know, so we stop it after a certain number of iterations, gived by the user.