CV - Report Assignment 2

GitHub repository: https://github.com/rickdott/mcv
Video: https://drive.google.com/file/d/1o7w_0VJTgxRpTLenDMJddx3FFiwFI4xL/view?usp=share_link

Background subtraction

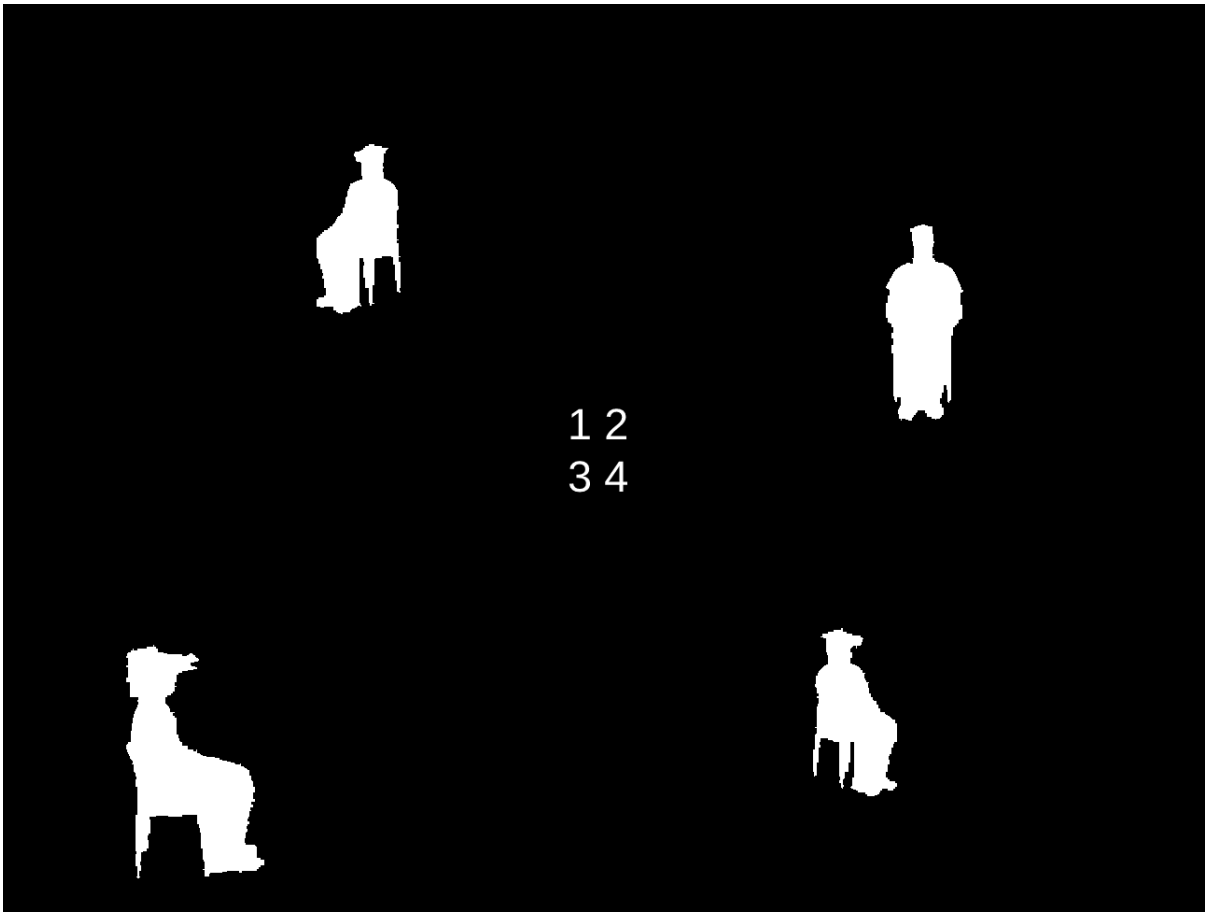
When I was starting with the assignment, I only wanted to get something that worked, quickly. I turned to OpenCV's `BackgroundSubtractorKNN` class. This turned out to work so well that it seemed disrespectful of my own time to create a marginally better approach. Instead I chose to focus more on the rest of the assignment, and complete choice tasks that I found interesting.

I train the model on the background video with a learning rate of 0.1, changing this did not make a discernable difference in quality of the subtraction. I ensured that the model does not learn when applying the subtractor.

I did apply some post-processing, first I remove the shadows that the background subtractor automatically detects. Then I apply the morphological closing operation (first dilate, then erode) using a 3×3 rectangular kernel to denoise the edges of the foreground without losing much detail. Finally I find the biggest remaining contour by aera in the image using `cv.FindContours()` and return it drawn onto an empty mask.

Processing (left = before, right = after)

Foreground images



Building the voxel model

Before any voxels get built, I calculate the lookup tables in parallel, using Python's `multiprocessing` package. The tables are created from a `np.meshgrid` with a range that puts the object in the center of the image, while remaining a cube. I chose a 1500mm cube for this.

The resolution of this cube can be set at the top of `VoxelReconstructor.py`. 100 is a tractable size, taking ~5s for the first frame. The voxel grid is then projected onto 2d using the camera intrinsics and extrinsics, which have been loaded from file.

The last thing to be done is to reverse the direction, since we want to be able to optimize our code. For this we need a dictionary like:

$$(x, y) \mapsto [(x_1, y_1, z_1), (x_2, y_2, z_2), \ldots]$$

Then, when the next frame is requested (`VoxelReconstructor.next_frame() → next_voxels, next_colors`), the VoxelReconstructor requests the changed pixels since last frame (using `cv.bitwise_xor()`). It loops over the changed pixels, keeping track of how often a voxel occurs in a central dictionary with default value of 0. After all the voxels have been checked, the reconstructor adds all of those voxels with counter equal to the amount of views. It also takes the mean color of the four foreground pixels and sends it to the visualization code. Other (partially visible) voxels get reset.

Using this approach saves a lot of time after the first frame, since only those voxels that are changed are reconsidered.

Extrinsic parameters

These were calculated using `cv.PerspectiveTransform` and linearly interpolating between corners.

```
Cam 1 rotation matrix
[[ 0.70823744 -0.70532204  0.03034065]
 [ 0.09328297  0.1360957   0.9862942 ]
```

```
 [-0.69978427 -0.69570021  0.1621826 ]]
Cam 1 translation vector
[[ 230.51272516]
 [ 662.0067211 ]
 [4437.31867197]]

Cam 2 rotation matrix
[[ 0.99909612 -0.04231083  0.00409153]
 [-0.00395415  0.00333008  0.99998664]
 [-0.04232389 -0.99909895  0.00315976]]
Cam 2 translation vector
[[-227.7597725 ]
 [1459.64341369]
 [3621.93324418]]

Cam 3 rotation matrix
[[-0.22503211  0.97412709  0.02090379]
 [-0.07724367 -0.0392223   0.99624045]
 [ 0.9712847   0.2225714   0.08407143]]
Cam 3 translation vector
[[-726.10634303]
 [1173.13381205]
 [2505.89977107]]

Cam 4 rotation matrix
[[ 0.6358178   0.7718369  -0.00187632]
 [-0.1080839   0.09144301  0.98992729]
 [ 0.76423399 -0.6292106   0.14156424]]
Cam 4 translation vector
[[-915.26287473]
 [ 849.37526561]
 [4046.8868366 ]]
```

Choice tasks

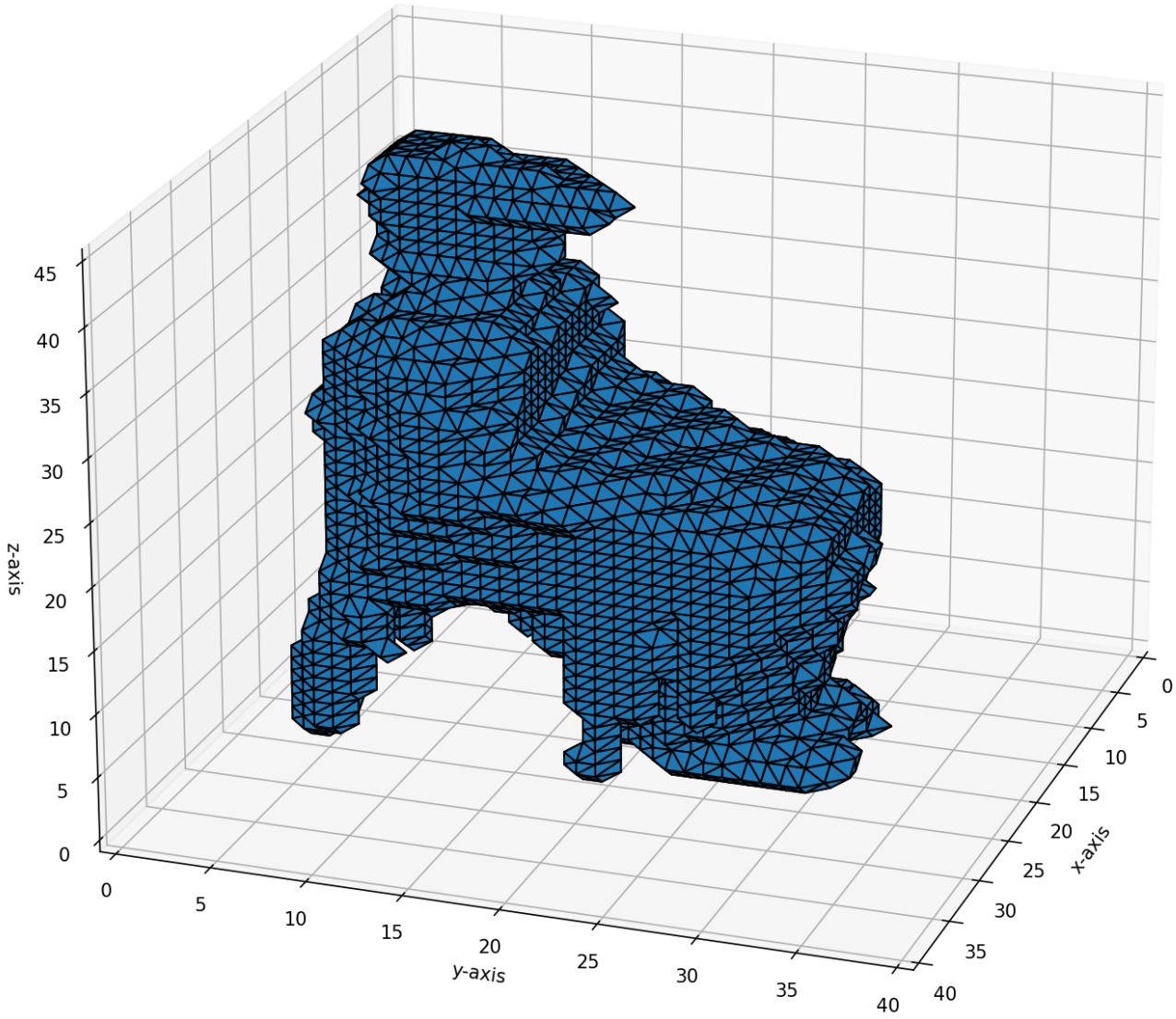
Coloring the voxel model

I colored the voxel model by taking the average color of the foreground pixel in all four views. I thought about implementing that the voxel would be colored using the view of the closest camera, but I did not implement this because it would hinder me in the next assignment.

Implementing the surface mesh

In `2_marching_cubes.py` I used the given implementation of marching cubes to display a mesh version of the object. The only thing that I had to do was to convert the list of voxels to a 50×50×50 numpy array consisting of 0's and 1's. A 1 signifies that a voxel is 'on'. I tried this approach first since I used marching cubes in Unity for a procedurally generated cave system in a school project and that implementation used

this format.



Optimizing the construction of the voxel model

As described above, I use a method that only updates those voxels that correspond to changed foreground pixels, using `cv.bitwise_xor()`. This means that after the first frame, only a few pixels have to be updated. Advancing to the next frame using the 'G' key is very fast.

Amount of changed pixels for the first four frames

```
1 Changed pixels: 9468
2 Changed pixels: 11558
3 Changed pixels: 17627
4 Changed pixels: 9296
1 Changed pixels: 12
2 Changed pixels: 9
3 Changed pixels: 64
4 Changed pixels: 19
1 Changed pixels: 11
2 Changed pixels: 10
3 Changed pixels: 55
4 Changed pixels: 3
1 Changed pixels: 11
2 Changed pixels: 7
```

```
3 Changed pixels: 73
4 Changed pixels: 23
```

Speeding up creation of the look-up table

I used Python's `multiprocessing` module to calculate the look-up table in four simultaneous processes. When testing this using a naive (for-loop) implementation, the time taken went from 41 seconds to 16 seconds. After optimizing the code, the difference is less noticeable but definitely still present.

In the following, the number at the start is the index of the `VoxelCam`, you can see that the order is different depending on when the process finishes.

```
1 Calculating table
2 Calculating table
3 Calculating table
4 Calculating table
4 Projecting points
3 Projecting points
1 Projecting points
2 Projecting points
2 Wrapping up
4 Wrapping up
3 Wrapping up
1 Wrapping up
```