# notebook

April 30, 2018

## Contents

**1 Information** **1**

**2 Build a simple Virtual world** **1**

**3 Show the virtual world** **3**

**4 Camera Model** **4**

**5 Triangulation** **6**

# 1 Information

- So this pdf will basically follow the matlab notebook and each section will be linked to the corresponding section in the matlab notebook.
- This will just commentate on the code given, not add any extra code
- This is being done as I can easily manipulate an org file as it's not too difficult to mess with formatting
- Throughout this file I'll probably explain a lot about the matlab code since it's foreign.

# 2 Build a simple Virtual world

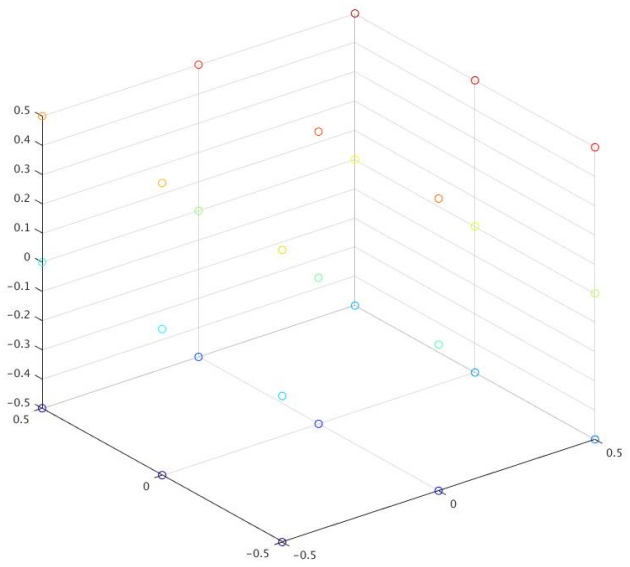
1. Add something into the world
   - so this creates a function called `create_points()`. `create_points()` starts off by creating a 3dmesh grid with values -5,0,5
     - A mesh grid in this case seems create a 3d array for X, Y, and Z such that the shift of values for the values for X shifts along the X axis (so `X(1,:,:)` has all the variations), the values for Y shifts along the Y axis (so `Y(:,1,:)` has all the variation), and

the value for the Z axis changes on the Z axis (so `Z(:,:,1)` has all the variation)

- The last thing is that we get the jet of the length of points (which is just $3^3 = 27$)
  - Jet returns some kind of color-map with the specified length
- Nothing too interesting happening here

2. Plot the points

- So the code in the above section generates this graph here




- 
- which we can see is just a 3x3 grid with the colors given by the color map
- The code itself just explains a scatter plot of the X Y and Z variables defined above with the colors applied

3. Set up a pair of Cameras

- So this code sets up 2 cameras, both with the exact same function just not abstracted out. The only difference is the $\alpha$ variable given
- this beta only shows up in the position of the camera, and the 3rd position does not rely on $\alpha$ at all so it can only effect the first two values of the array.
- We can also notice that $\sin(\frac{\pi}{3}) = \cos(\frac{\pi}{6})$ which means that the only difference between the cameras is that the first two positions of the

camera are swapped, since the only difference is that one is cos(alpha) vs sin(alpha) (which swap with each other when $\pi/6$ turns into $\pi/3$)

- Therefor the could be improved in this particular instance by instead writing
- Other values on the camera struct are the following
  - up
    - This is the vector for the camera up direction
  - Target
    - This is the point in which the camera is looking (which in this case is [0 0 0] which is the origin)
  - `focal_length`
    - this is how far out the rectangle the camera projects is
  - `film_width`
    - this will be used to get the width of the rectangle of the rectangle that the camera projects with respect to the focal length
  - `film_height`
    - Like width but for height

```
for fn = fieldnames(cam2)'
cam2.(fn{1}) = cam1.(fn{1});
end
cam2.position([1 2]) = cam2.position([2 1])
```

  - of course a more generic function that took $\beta$ and $\alpha$ would be better and allow for more kinds of behavior

4. Plot camera
   - the `camera_coordinate_system` is rather interesting here because we can note for the two cameras we are using zcam is really just (- cam.position) since both cameras have a [0 0 0] for target, which means. The rest of the function does math that is already commented in the file.
   - The next section after this just goes through with details of trying to get the rectangle that camera sees
   - it uses details like `film_width` and `film_length` to determine the corners of the rectangle the camera projects as it looks towards it's target (in this case the origin)

# 3 Show the virtual world

- The first section shows the output values of the camera, which the differences were already discussed in Set up a pair of Cameras

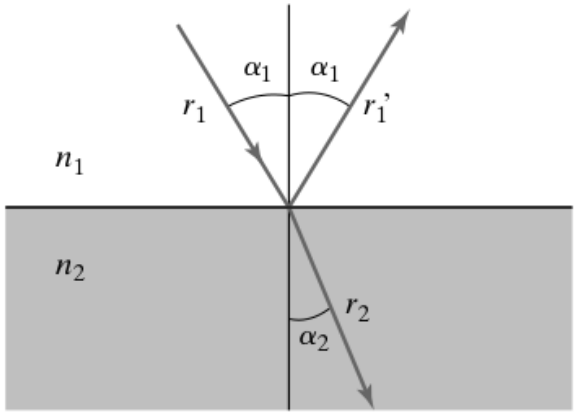
- The second and third section just plot the camera and show the side by side of where they are located

# 4 Camera Model

1. Euclidean Transformation matrix
   - So this transformation deals with the reflection when light hits a surface at an angle
   - in the case of the code this surface can be seen as the origin, and the vector given back could be seen as $r_1$' on the slides, which we can tell since the code returns [R; -origin*R] where R = [xcam(:), ycam(:), zcam(:)], which we can visualize the change as the following from camera1

```
>> [xcam;ycam;zcam;origin]

ans =

   -0.5000    0.8660         0
    0.4330    0.2500   -0.8660
   -0.7500   -0.4330   -0.5000
    3.7500    2.1651    2.5000

>> ExtrinsicsMtx(cam1)

ans =

   -0.5000    0.4330   -0.7500
    0.8660    0.2500   -0.4330
         0   -0.8660   -0.5000
   -0.0000         0    5.0000
```

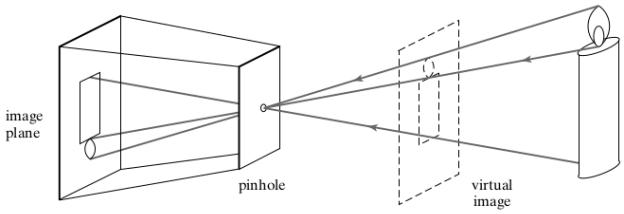
   – if we ignore the bottom rows of both matrices, we are just doing a diagonal flip down the center of the matrix, essentially achieving the change from $r_1$ to $r_1$' in the image below




*

  - As for the bottom row of each matrix, we can see that origin is the starting location of camera1, and the new location is what I'm guessing is the image is being projected from [0 0 5] which can be seen as the new origin of the camera instead of the original origin.

2. Camera Calibration matrix

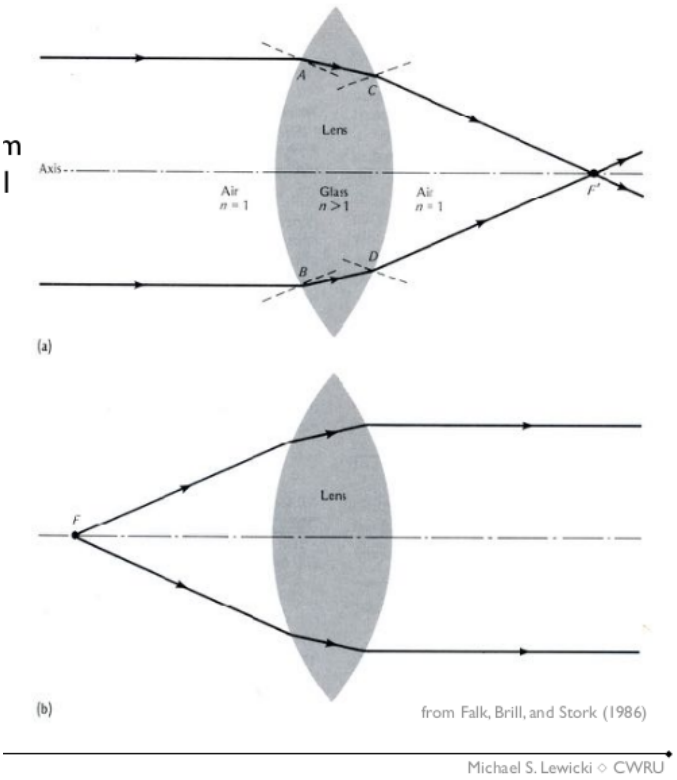



- 
- The slide on page 4 describes viewing an image through a pinhole and seeing the inverse image.
- In our case, the pinhole would be our camera lens, and the outlining sketch of the candle (virtual image) would correspond to the rectangle drawn previously that depends on the `focal_length` and the height and width.
- I can't visualize exactly the transformation of the diagram to the code here, but I do notice there is 4 0s in the returned matrix, which I think has to do with wasting light which is described on slide 5. it would make sense that some information would thus be wasted with this model of the camera.

3. Camera Matrix

- The previous two come together, and we should note the names of the functions, one is extrinsic and the other is intrinsic, which we can see in the way they view the world. the first one views it as a

reflection on the surface and the other views it as lights converging on it, which seem to reflect the image on slide 7




4. Generate the Image Pair
    - it seems here we do some math to construct the 2d projection of the 3d image, I'm not exactly sure of the math that is happening here that dictates the placement of the points
    - the second section in this file is the set-Color function which does some math to colorize each point differently
    - After this we just plot both cameras, and see that the structure looks like our original structure from both images (scroll back up to look through camera 2, and look through camera 1, you can tell they look very similar) just at different angles from our original two cameras.

# 5 Triangulation

- Since we have 2 2d projection we can try to recover the 3d information
- Really this can be done by solving a linear equation from the images given for every point in common

1. Linear Triangulation method

    - `triangulationOnePoint` basically seems to solve the triangulation of two points given to it from the corresponding images, though doing some transformations with the 2 vectors gathered from the cameras in the last section.
    - From here we just map this triangulation over all the shared points between the two images
    - Note that nothing in this algorithm is really specialized for 2 images, one can generalize this to n-images by sending in 2 lists, 1 of the points and 1 of the 2d matrices of the camera, essentially just doing this equation with more matrices.
    - Finally we plot the 3d structure and see that it isn't too different from the original 3d image that is at the start of this document
    - The final section just adds some noise into the image, and as one can see, the noise doesn't really hurt the representation of the structure much, and it pretty much stays in tact