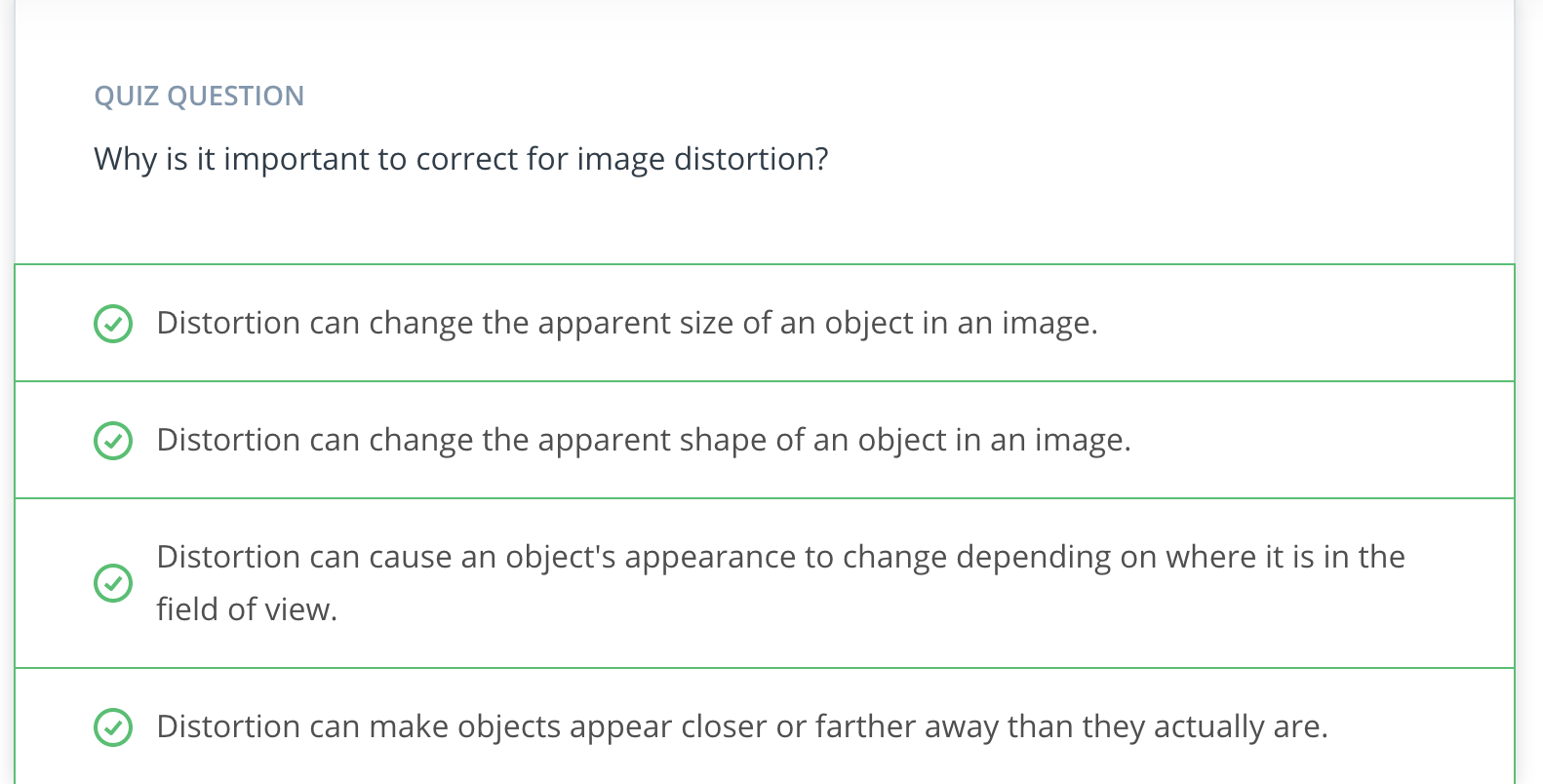
**Distortion Correction**

This uses a visual to convey the big idea - all about distortion and that is what this lesson is about so the students understand the motivation for the lesson.

**Distortion**

Image distortion occurs when a camera looks at 3D objects in the real world and transforms them into a 2D image; this transformation isn’t perfect. Distortion actually changes what the shape and size of these 3D objects appear to be. So, the first step in analyzing camera images, is to undo this distortion so that you can get correct and useful information out of them.



**Types of Distortion**

Real cameras use curved lenses to form an image, and light rays often bend a little too much or too little at the edges of these lenses. This creates an effect that distorts the edges of images, so that lines or objects appear more or less curved than they actually are. This is called **radial distortion**, and it’s the most common type of distortion.

Another type of distortion, is **tangential distortion**. This occurs when a camera’s lens is not aligned perfectly parallel to the imaging plane, where the camera film or sensor is. This makes an image look tilted so that some objects appear farther away or closer than they actually are.

**Distortion Coefficients and Correction**

There are three coefficients needed to correct for **radial distortion**: **k1**, **k2**, and **k3**. To correct the appearance of radially distorted points in an image, one can use a correction formula.

In the following equations, (x, y)(*x*,*y*) is a point in a distorted image. To undistort these points, OpenCV calculates **r**, which is the known distance between a point in an undistorted (corrected) image (x\_{corrected}, y\_{corrected})(*xcorrected*​,*ycorrected*​) and the center of the image distortion, which is often the center of that image (x\_c, y\_c)(*xc*​,*yc*​). This center point (x\_c, y\_c)(*xc*​,*yc*​) is sometimes referred to as the distortion center. These points are pictured below.

Note: The distortion coefficient **k3** is required to accurately reflect major radial distortion (like in wide angle lenses). However, for minor radial distortion, which most regular camera lenses have, k3 has a value close to or equal to zero and is negligible. So, in OpenCV, you can choose to ignore this coefficient; this is why it appears at the end of the distortion values array: [k1, k2, p1, p2, k3]. In this course, we will use it in all calibration calculations so that our calculations apply to a wider variety of lenses (wider, like wide angle, haha) and can correct for both minor and major radial distortion.

A picture containing drawing

Description automatically generated

Points in a distorted and undistorted (corrected) image. The point (x, y) is a single point in a distorted image and (x\_corrected, y\_corrected) is where that point will appear in the undistorted (corrected) image.

x\_{distorted} = x\_{ideal} (1 + k\_1r^2 + k\_2r^4 + k\_3r^6)*xdistorted*​=*xideal*​(1+*k*1​*r*2+*k*2​*r*4+*k*3​*r*6)y\_{distorted} = y\_{ideal} (1 + k\_1r^2 + k\_2r^4 + k\_3r^6)*ydistorted*​=*yideal*​(1+*k*1​*r*2+*k*2​*r*4+*k*3​*r*6)

###### **Radial distortion correction.**

There are two more coefficients that account for **tangential distortion**: **p1** and **p2**, and this distortion can be corrected using a different correction formula.

x\_{corrected} = x + [2p\_1xy + p\_2(r^2 + 2x^2)]*xcorrected*​=*x*+[2*p*1​*xy*+*p*2​(*r*2+2*x*2)]y\_{corrected} = y + [p\_1(r^2 + 2y^2) + 2p\_2xy]*ycorrected*​=*y*+[*p*1​(*r*2+2*y*2)+2*p*2​*xy*]

###### **Tangential distortion correction.**

A screenshot of a social media post

Description automatically generated

**Finding Corners**

In this exercise, you'll use the OpenCV functions findChessboardCorners() and drawChessboardCorners() to automatically find and draw corners in an image of a chessboard pattern.

To learn more about both of those functions, you can have a look at the OpenCV documentation here: [**cv2.findChessboardCorners()**](http://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#cv2.findChessboardCorners) and [**cv2.drawChessboardCorners()**](http://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#cv2.drawChessboardCorners).

Applying these two functions to a sample image, you'll get a result like this:

A close up of a tiled wall

Description automatically generated

In the following exercise, your job is simple. Count the number of corners in any given row and enter that value in nx. Similarly, count the number of corners in a given column and store that in ny. Keep in mind that "corners" are only points where two black and two white squares intersect, in other words, only count inside corners, not outside corners.

If you'd like to test the above image (before drawing the corners) on your own machine, download it [**here**](https://s3-us-west-1.amazonaws.com/udacity-selfdrivingcar/files/Advanced_Lane_Finding_Images/calibration_test.png).

You can count these in the image above - there are eight inside corners for x (nx=8) and six inside corners for y (ny=6).

## Note Regarding Corner Coordinates

Since the origin corner is (0,0,0) the final corner is (6,4,0) relative to this corner rather than (7,5,0).

**Examples of Useful Code**

Converting an image, imported by cv2 or the glob API, to grayscale:

gray = cv2.cvtColor(img,cv2.COLOR\_BGR2GRAY)

Note: If you are reading in an image using mpimg.imread() this will read in an **RGB** image and you should convert to grayscale using cv2.COLOR\_RGB2GRAY, but if you are using cv2.imread() or the glob API, as happens in this video example, this will read in a **BGR** image and you should convert to grayscale using cv2.COLOR\_BGR2GRAY. We'll learn more about color conversions later on in this lesson, but please keep this in mind as you write your own code and look at code examples.

Finding chessboard corners (for an 8x6 board):

ret, corners = cv2.findChessboardCorners(gray, (8,6), **None**)

Drawing detected corners on an image:

img = cv2.drawChessboardCorners(img, (8,6), corners, ret)

Camera calibration, given object points, image points, and the **shape of the grayscale image**:

ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, gray.shape[::-1], **None**, **None**)

Undistorting a test image:

dst = cv2.undistort(img, mtx, dist, **None**, mtx)

### A note on image shape

The shape of the image, which is passed into the **calibrateCamera** function, is just the height and width of the image. One way to retrieve these values is by retrieving them from the **grayscale image shape** array gray.shape[::-1]. This returns the image width and height in pixel values like (1280, 960).

Another way to retrieve the image shape, is to get them directly from the color image by retrieving the first two values in the color image shape array using img.shape[1::-1]. This code snippet asks for just the first two values in the shape array, and reverses them. Note that in our case we are working with a greyscale image, so we only have 2 dimensions (color images have three, height, width, and depth), so this is not necessary.

It's important to use an entire grayscale image shape or the first two values of a color image shape. This is because the entire shape of a color image will include a third value -- the number of color channels -- in addition to the height and width of the image. For example the shape array of a color image might be (960, 1280, 3), which are the pixel height and width of an image (960, 1280) and a third value (3) that represents the three color channels in the color image which you'll learn more about later, and if you try to pass these three values into the calibrateCamera function, you'll get an error.

A picture containing photo, tiled, large, person

Description automatically generated

Here, you'll get a chance to try camera calibration and distortion correction for yourself!

There are two main steps to this process: use chessboard images to obtain image points and object points, and then use the OpenCV functions cv2.calibrateCamera() and cv2.undistort() to compute the calibration and undistortion.

Unfortunately, we can't perform the extraction of object points and image points in the browser quiz editor, so we provide these for you in the quiz below.

Try computing the calibration and undistortion in the exercise below, and if you want to play with extracting object points and image points yourself, fork the Jupyter notebook and images in [**this repository**](https://github.com/udacity/CarND-Camera-Calibration). You can also download the [**distortion pickle file**](https://s3-us-west-1.amazonaws.com/udacity-selfdrivingcar/files/Advanced_Lane_Finding_Images/correct_for_distortion/wide_dist_pickle.p) and [**test image**](https://s3-us-west-1.amazonaws.com/udacity-selfdrivingcar/files/Advanced_Lane_Finding_Images/correct_for_distortion/test_image.png) used in the below quiz if you'd like to run the below code on your own machine.

If you run into any *errors* as you run your code, please refer to the **Examples of Useful Code** section in the previous video and make sure that your code syntax matches up!

# Challenge: Calibrate your own camera!

If you're up for a challenge, go ahead and try these steps on your own camera images. Just print out a [**chessboard pattern**](http://docs.opencv.org/2.4/_downloads/pattern.png), stick it to a flat surface, and take 20 or 30 pictures of it. Make sure to take pictures of the pattern over the entire field of view of your camera, particularly near the edges.

To extract object points and image points you can check out the Jupyter notebook in [**this repository**](https://github.com/udacity/CarND-Camera-Calibration). If you're following along with the code in the notebook, be sure you set the right size for your chessboard, the link above is to a 9 x 6 pattern, while before we were using an 8 x 6.

Here is one example answer:

**def** **cal\_undistort**(img, objpoints, imgpoints):

ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, img.shape[1:], **None**, **None**)

undist = cv2.undistort(img, mtx, dist, **None**, mtx)

**return** undist

Note that the quiz is loading this image as channels first, so we use img.shape[1:] to get the height and width. This should be adjusted as necessary, such as is seen on the previous page for a grayscale image.

**Calculating Lane Curvature**

Self-driving cars need to be told the correct steering angle to turn, left or right. You can calculate this angle if you know a few things about the speed and dynamics of the car and how much the lane is curving.

One way to calculate the curvature of a lane line, is to fit a 2nd degree polynomial to that line, and from this you can easily extract useful information.

For a lane line that is close to vertical, you can fit a line using this formula: **f(y) = Ay^2 + By + C**, where A, B, and C are coefficients.

A gives you the curvature of the lane line, B gives you the heading or direction that the line is pointing, and C gives you the position of the line based on how far away it is from the very left of an image (y = 0).

**Perspective transform**

A perspective transform maps the points in a given image to different, desired, image points with a new perspective. The perspective transform you’ll be most interested in is a bird’s-eye view transform that let’s us view a lane from above; this will be useful for calculating the lane curvature later on. Aside from creating a bird’s eye view representation of an image, a perspective transform can also be used for all kinds of different view points.

Ultimately, we use the perspective transform to get a top-down view in order to see the curvature of the lines better.

**Examples of Useful Code**

Compute the perspective transform, M, given source and destination points:

M = cv2.getPerspectiveTransform(src, dst)

Compute the inverse perspective transform:

Minv = cv2.getPerspectiveTransform(dst, src)

Warp an image using the perspective transform, M:

warped = cv2.warpPerspective(img, M, img\_size, flags=cv2.INTER\_LINEAR)

Note: When you apply a perspective transform, choosing four source points manually, as we did in this video, is often not the best option. There are many other ways to select source points. For example, many perspective transform algorithms will programmatically detect four source points in an image based on edge or corner detection and analyzing attributes like color and surrounding pixels.

A screenshot of a social media post

Description automatically generated

# Undistort and Transform Perspective

A picture containing object, different, person

Description automatically generated

Here's a tricky quiz for you! You have now seen how to find corners, calibrate your camera, undistort an image, and apply a perspective transform. Now it's your chance to perform all these steps on an image. In the last quiz you calibrated the camera, so here I'm giving you the camera matrix, mtx, and the distortion coefficients dist to start with.

Your goal is to generate output like the image shown above. To do that, you need to write a function that takes your distorted image as input and completes the following steps:

* Undistort the image using cv2.undistort() with mtx and dist
* Convert to grayscale
* Find the chessboard corners
* Draw corners
* Define 4 source points (the outer 4 corners detected in the chessboard pattern)
* Define 4 destination points (must be listed in the same order as src points!)
* Use cv2.getPerspectiveTransform() to get M, the transform matrix
* use cv2.warpPerspective() to apply M and warp your image to a top-down view

**HINT:** Source points are the x and y pixel values of any four corners on your chessboard, you can extract these from the corners array output from cv2.findChessboardCorners(). Your destination points are the x and y pixel values of where you want those four corners to be mapped to in the output image.

If you run into any errors as you run your code, please refer to the **Examples of Useful Code** section in the previous video and make sure that your code syntax matches up! For this example, please also refer back to the examples in the Calibrating Your Camera video. You can also download the [**distortion pickle file**](https://s3-us-west-1.amazonaws.com/udacity-selfdrivingcar/files/Advanced_Lane_Finding_Images/correct_for_distortion/wide_dist_pickle.p) and [**test image**](https://s3-us-west-1.amazonaws.com/udacity-selfdrivingcar/files/Advanced_Lane_Finding_Images/correct_for_distortion/test_image2.png) used in the below quiz if you'd like to run the below code on your own machine.

# Solution to Undistort and Transform quiz:

That was a tricky quiz!

Let's take a closer look at the pieces involved. These steps will all be useful when you get to the project at the end of this section.

First off, we defined a function for you, corners\_unwarp(). The function accepts an image and our previously calculated values for the camera matrix, mtx, and distortion coefficients, dist.

Then you had to implement the function.

Here's how I implemented mine:

*# Define a function that takes an image, number of x and y points,*

*# camera matrix and distortion coefficients*

**def** **corners\_unwarp**(img, nx, ny, mtx, dist):

*# Use the OpenCV undistort() function to remove distortion*

undist = cv2.undistort(img, mtx, dist, **None**, mtx)

*# Convert undistorted image to grayscale*

gray = cv2.cvtColor(undist, cv2.COLOR\_BGR2GRAY)

*# Search for corners in the grayscaled image*

ret, corners = cv2.findChessboardCorners(gray, (nx, ny), **None**)

**if** ret == **True**:

*# If we found corners, draw them! (just for fun)*

cv2.drawChessboardCorners(undist, (nx, ny), corners, ret)

*# Choose offset from image corners to plot detected corners*

*# This should be chosen to present the result at the proper aspect ratio*

*# My choice of 100 pixels is not exact, but close enough for our purpose here*

offset = 100 *# offset for dst points*

*# Grab the image shape*

img\_size = (gray.shape[1], gray.shape[0])

*# For source points I'm grabbing the outer four detected corners*

src = np.float32([corners[0], corners[nx-1], corners[-1], corners[-nx]])

*# For destination points, I'm arbitrarily choosing some points to be*

*# a nice fit for displaying our warped result*

*# again, not exact, but close enough for our purposes*

dst = np.float32([[offset, offset], [img\_size[0]-offset, offset],

[img\_size[0]-offset, img\_size[1]-offset],

[offset, img\_size[1]-offset]])

*# Given src and dst points, calculate the perspective transform matrix*

M = cv2.getPerspectiveTransform(src, dst)

*# Warp the image using OpenCV warpPerspective()*

warped = cv2.warpPerspective(undist, M, img\_size)

*# Return the resulting image and matrix*

**return** warped, M

## Output

Here's the output you needed to achieve:

A picture containing object, different, person

Description automatically generated

**Coming Up Next**

A picture containing airplane

Description automatically generated

Halfway There!

Nice work so far! You've learned a lot about how cameras work, the distortions they cause, and how to correct for that distortion. You've also learned about perspective transformation - a great method to help you see a view from a different angle than the original image.

While you'll still be using the earlier concepts later on, we'll be shifting into color and gradient thresholds, a way of determining where certain objects occur in images that have already been undistorted and perspective transformed. From there, you'll learn how to identify the lane lines themselves.