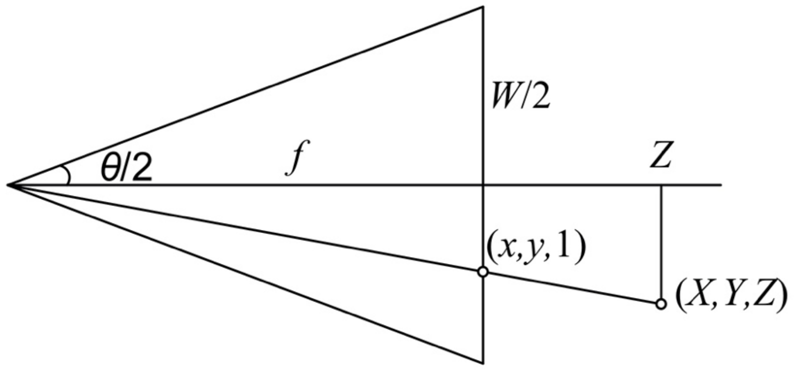
CSC528 Assignment #1.

Name: Kavana Manvi Krishnamurthy
Student ID: 2158984

Problem 1: CCD to Camera Transformation
Consider a perfect perspective projection camera with focal length 24 mm and a CCD array of size 16 mm × 12 mm, containing 500 × 500 pixels.

Field of View (FOV) is defined as the angle between two points at opposite edges of the image (CCD array), either horizontally or vertically. Thus there are two FOVs, one horizontal and one vertical. The FOV is twice the angle between the optical axis and one edge of the image.

i. Give a general expression for computing horizontal FOV from focal length and image width.



The above figure represents the relationship between the sensor width W, the focal length f and the field of view θ. Given by the formula:

$$\tan\frac{\theta}{2} = \frac{W}{2f}$$

Or field of view θ is given by:

$$\theta = 2 \cdot tan^{-1}\left[\frac{W}{2f}\right]$$

ii. Compute the horizontal FOV and vertical FOV of the given camera.

Focal length (f) = 24 mm
Sensor width (W) = 16 mm
Sensor height (H) = 12 mm
Resolution = 500 × 500 pixels

Horizontal FOV:

$$\theta_{Horizontal} = 2 \cdot tan^{-1}\left[\frac{W}{2f}\right]$$

$$\theta_{Horizontal} = 2 \cdot tan^{-1}\left[\frac{16}{2*(24)}\right]$$

$$\theta_{Horizontal} = 2 \cdot tan^{-1}[0.333]$$

$$\theta_{Horizontal} = 2 * 18.3^{\circ}$$

$$\theta_{Horizontal} = 36.6^{\circ}$$

Horizontal FOV:

$$\theta_{Vertical} = 2 \cdot tan^{-1}\left[\frac{W}{2f}\right]$$

$$\theta_{Vertical} = 2 \cdot tan^{-1}\left[\frac{12}{2*(24)}\right]$$

$$\theta_{Vertical} = 2 \cdot tan^{-1}[0.25]$$

$$\theta_{Vertical} = 2 * 14^{\circ}$$

$$\theta_{Vertical} = 28^{\circ}$$

iii. Comment on how FOV affects resolution in an image.

Resolution basically refers to how much detail an image can show. It is usually measured by the number of pixels. Field of View (FOV), on the other hand, is the angle that defines how much of the scene the camera can see from one edge to the other.
When you shrink the FOV, the camera captures a smaller part of the scene, but in doing so, it can show more detail in that area because all the pixels are focused on a smaller region. This is like zooming in to get a clearer view of something specific. Widening the FOV does the opposite—it lets the camera see more of the scene at once, but each part of that scene takes up fewer pixels, so the image ends up with less detail per object. It's like zooming out: you see more, but each thing looks smaller and blurrier.
In short:
Wider FOV = More coverage, but lower detail per object.
Narrower FOV = Less coverage, but higher detail per object.

Problem 2: Exercise 2.2 from Szelinski book (2D transform editor)
Feel free to use any existing code/libraries you wish, in whatever language you like.

Ex 2.2: 2D transform editor Write a program that lets you interactively create a set of rectangles and then modify their "pose" (2D transform). You should implement the following steps:
1. Open an empty window ("canvas").
2. Shift drag (rubber-band) to create a new rectangle.
3. Select the deformation mode (motion model): translation, rigid, similarity, affine, or perspective.
4. Drag any corner of the outline to change its transformation.
This exercise should be built on a set of pixel coordinate and transformation classes, either implemented by yourself or from a software library. Persistence of the created representation

(save and load) should also be supported (for each rectangle, save its transformation).

<u>Code in python:</u>

```
import cv2
import numpy as np
import json

WINDOW_TITLE = "2D Transform Editor"
RECTANGLE_DIMENSIONS = (120, 80)

class EditableRectangle:
    def __init__(self, position, transformation_type='translation'):
        self.position = np.array(position, dtype=np.float32)
        self.width, self.height = RECTANGLE_DIMENSIONS
        self.transformation_type = transformation_type
        self.transformation_matrix = np.eye(3, dtype=np.float32)
        self.translate(position[0], position[1])

    def get_corners(self):
        half_width, half_height = self.width / 2, self.height / 2
        corners = np.array([
            [-half_width, -half_height, 1],
            [ half_width, -half_height, 1],
            [ half_width,  half_height, 1],
            [-half_width,  half_height, 1],
        ]).T
        transformed_corners = self.transformation_matrix @ corners
        return (transformed_corners[:2] / transformed_corners[2]).T.astype(np.float32)

    def translate(self, dx, dy):
        translation_matrix = np.array([
            [1, 0, dx],
            [0, 1, dy],
            [0, 0, 1]
        ], dtype=np.float32)
        self.transformation_matrix = translation_matrix @ self.transformation_matrix

    def apply_transformation(self, matrix):
        if matrix.shape == (2, 3):
            matrix = np.vstack([matrix, [0, 0, 1]])
        self.transformation_matrix = matrix @ self.transformation_matrix

    def to_dict(self):
        return {
            'transformation_matrix': self.transformation_matrix.tolist(),
            'width': self.width,
            'height': self.height
        }

    @staticmethod
    def from_dict(data):
        rectangle = EditableRectangle((0, 0))
        rectangle.transformation_matrix = np.array(data['transformation_matrix'], dtype=np.float32)
        rectangle.width = data['width']
        rectangle.height = data['height']
        return rectangle

def convert_to_homogeneous(matrix):
    return np.vstack([matrix, [0, 0, 1]])

# Global Variables
rectangles_list = []
dragging_active = False
active_rectangle = None
active_corner = None
previous_point = None
current_mode = 'translation'
MODES = ['translation', 'rigid', 'similarity', 'affine', 'perspective']

# Rubber-band for rectangle creation
rubber_band_active = False
start_point = None
end_point = None

def find_nearest_corner(rectangle, point):
    corners = rectangle.get_corners()
    for idx, corner in enumerate(corners):
        if np.linalg.norm(corner - point) < 20:
            return idx
    return None

def mouse_event_handler(event, x, y, flags, param):
    global dragging_active, active_rectangle, active_corner, previous_point
    global rubber_band_active, start_point, end_point

    if event == cv2.EVENT_LBUTTONDOWN:
        if flags & cv2.EVENT_FLAG_SHIFTKEY:
            rubber_band_active = True
            start_point = (x, y)
            end_point = (x, y)
        else:
            for rectangle in reversed(rectangles_list):
                corner_idx = find_nearest_corner(rectangle, np.array([x, y], dtype=np.float32))
                if corner_idx is not None:
                    active_rectangle = rectangle
                    active_corner = corner_idx
                    dragging_active = True
                    previous_point = np.array([x, y], dtype=np.float32)
                    break

    elif event == cv2.EVENT_MOUSEMOVE:
        if rubber_band_active:
            end_point = (x, y)
        elif dragging_active and active_rectangle is not None:
            current_point = np.array([x, y], dtype=np.float32)
            old_corners = active_rectangle.get_corners()
            new_corners = old_corners.copy()
            new_corners[active_corner] = current_point

            if current_mode == 'translation':
                dx, dy = current_point - previous_point
                active_rectangle.translate(dx, dy)

            elif current_mode == 'rigid':
                src = old_corners[:2].astype(np.float32)
                dst = new_corners[:2].astype(np.float32)
                src_center = np.mean(src, axis=0)
                dst_center = np.mean(dst, axis=0)
                src_centered = src - src_center
                dst_centered = dst - dst_center
                H = src_centered.T @ dst_centered
                U, _, Vt = np.linalg.svd(H)
                R = Vt.T @ U.T
                if np.linalg.det(R) < 0:
                    Vt[-1, :] *= -1
                    R = Vt.T @ U.T
                matrix = np.eye(3, dtype=np.float32)
                matrix[:2, :2] = R
                matrix[:2, 2] = dst_center - R @ src_center
                active_rectangle.apply_transformation(matrix)

            elif current_mode == 'similarity':
                src = old_corners[:3].astype(np.float32)
                dst = new_corners[:3].astype(np.float32)
                src_center = np.mean(src, axis=0)
                dst_center = np.mean(dst, axis=0)

                src_centered = src - src_center
                dst_centered = dst - dst_center

                matrix, _ = cv2.estimateAffinePartial2D(src_centered, dst_centered, method=cv2.LMEDS)
                if matrix is not None:
                    T1 = np.array([[1, 0, -src_center[0]],
                                   [0, 1, -src_center[1]],
                                   [0, 0, 1]], dtype=np.float32)
                    T2 = np.array([[1, 0, dst_center[0]],
                                   [0, 1, dst_center[1]],
                                   [0, 0, 1]], dtype=np.float32)
                    H = T2 @ convert_to_homogeneous(matrix) @ T1
                    active_rectangle.apply_transformation(H)

            elif current_mode == 'affine':
                indices = [(active_corner + i) % 4 for i in range(3)]
                src = old_corners[indices].astype(np.float32)
                dst = new_corners[indices].astype(np.float32)

                src_center = np.mean(src, axis=0)
                dst_center = np.mean(dst, axis=0)

                src_centered = src - src_center
                dst_centered = dst - dst_center

                matrix = cv2.getAffineTransform(src_centered, dst_centered)
                T1 = np.array([[1, 0, -src_center[0]],
                               [0, 1, -src_center[1]],
                               [0, 0, 1]], dtype=np.float32)
                T2 = np.array([[1, 0, dst_center[0]],
                               [0, 1, dst_center[1]],
                               [0, 0, 1]], dtype=np.float32)
                H = T2 @ convert_to_homogeneous(matrix) @ T1
                active_rectangle.apply_transformation(H)

            elif current_mode == 'perspective':
                src = old_corners.astype(np.float32)
                dst = new_corners.astype(np.float32)

                src_center = np.mean(src, axis=0)
                dst_center = np.mean(dst, axis=0)

                src_centered = src - src_center
                dst_centered = dst - dst_center

                matrix = cv2.getPerspectiveTransform(src_centered, dst_centered)
                T1 = np.array([[1, 0, -src_center[0]],
                               [0, 1, -src_center[1]],
                               [0, 0, 1]], dtype=np.float32)
                T2 = np.array([[1, 0, dst_center[0]],
                               [0, 1, dst_center[1]],
                               [0, 0, 1]], dtype=np.float32)
                H = T2 @ matrix @ T1
                active_rectangle.transformation_matrix = H @ active_rectangle.transformation_matrix

            previous_point = current_point

        elif event == cv2.EVENT_LBUTTONUP:
            if rubber_band_active:
            rubber_band_active = False
            x1, y1 = start_point
            x2, y2 = end_point
            center = ((x1 + x2) / 2, (y1 + y2) / 2)
            width = abs(x2 - x1)
            height = abs(y2 - y1)
            if width > 5 and height > 5:
                rectangle = EditableRectangle(center, transformation_type=current_mode)
                rectangle.width = width
                rectangle.height = height
                rectangles_list.append(rectangle)
            start_point = None
            end_point = None
            dragging_active = False
            active_rectangle = None
            active_corner = None

def render():
        canvas = np.ones((600, 800, 3), dtype=np.uint8) * 255
        for rectangle in rectangles_list:
            corners = rectangle.get_corners().astype(np.int32)
            cv2.polylines(canvas, [corners], isClosed=True, color=(0, 0, 255), thickness=2)
            for pt in corners:
            cv2.circle(canvas, pt, 5, (255, 0, 0), -1)
        if rubber_band_active and start_point and end_point:
            cv2.rectangle(canvas, start_point, end_point, (0, 255, 0), 1)
        cv2.putText(canvas, f"Mode: {current_mode}", (10, 25), cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0, 0, 0), 2)
        return canvas

def save_rectangles_data(filename="rectangles.json"):
        data = [rect.to_dict() for rect in rectangles_list]
        with open(filename, 'w') as file:
            json.dump(data, file)
        print("Saved to", filename)

def load_rectangles_data(filename="rectangles.json"):
        global rectangles_list
        with open(filename, 'r') as file:
            data = json.load(file)
            rectangles_list = [EditableRectangle.from_dict(d) for d in data]
        print("Loaded from", filename)

def start():
        global current_mode
        cv2.namedWindow(WINDOW_TITLE)
        cv2.setMouseCallback(WINDOW_TITLE, mouse_event_handler)

        print("Controls:")
        print("  Shift + Drag: Create rectangle")
        print("  Drag corner: Transform rectangle")
        print("  1-5: Switch mode (1=Translation, 2=Rigid, 3=Similarity, 4=Affine, 5=Perspective)")
        print("  s: Save, l: Load, q: Quit")

        while True:
            canvas = render()
            cv2.imshow(WINDOW_TITLE, canvas)
            key = cv2.waitKey(1) & 0xFF

            if key == ord('q'):
            break
            elif key == ord('s'):
            save_rectangles_data()
            elif key == ord('l'):
            load_rectangles_data()
            elif key in [ord(str(i)) for i in range(1, 6)]:
            current_mode = MODES[int(chr(key)) - 1]

        cv2.destroyAllWindows()

if __name__ == '__main__':
        start()
```

Output:

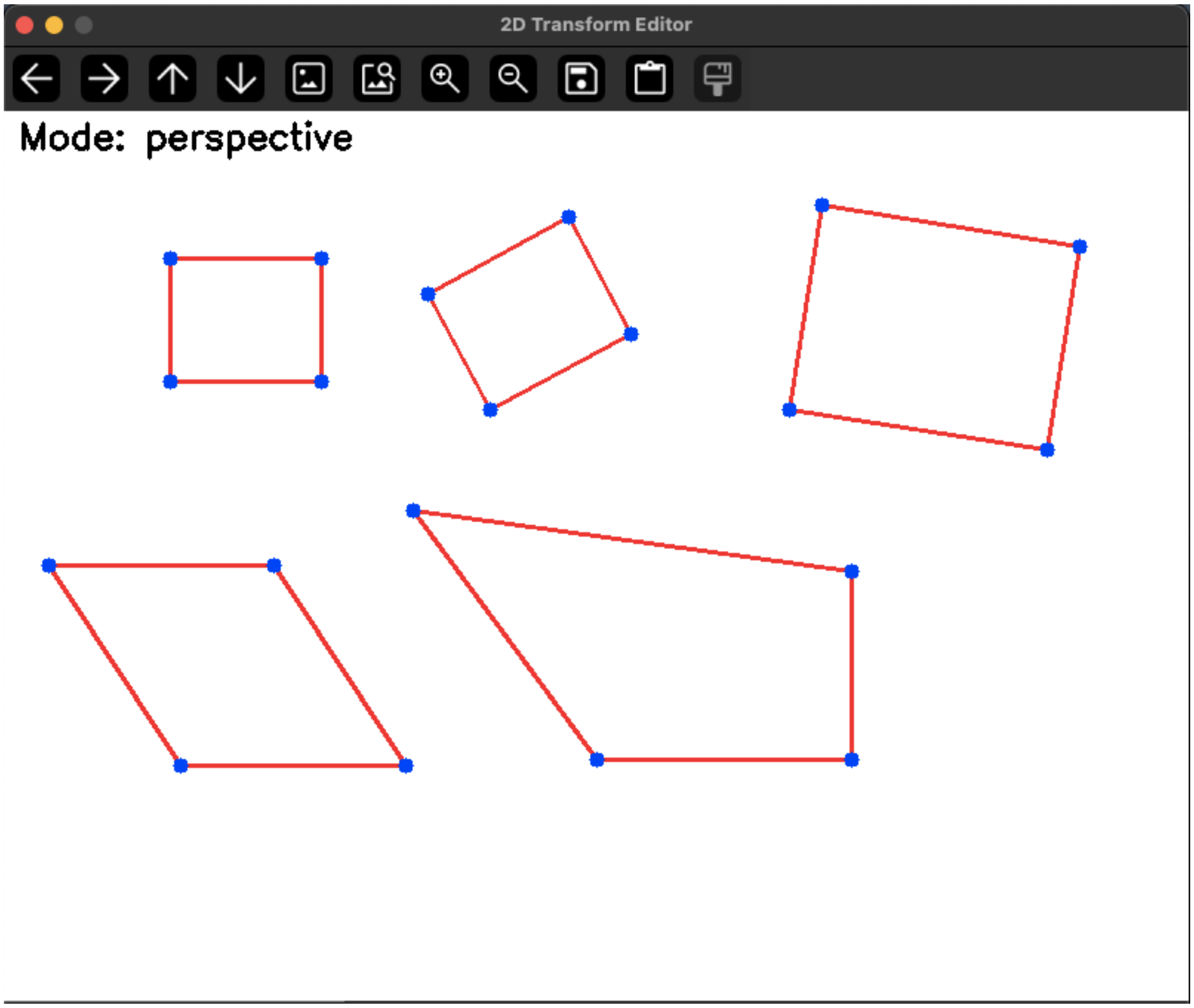
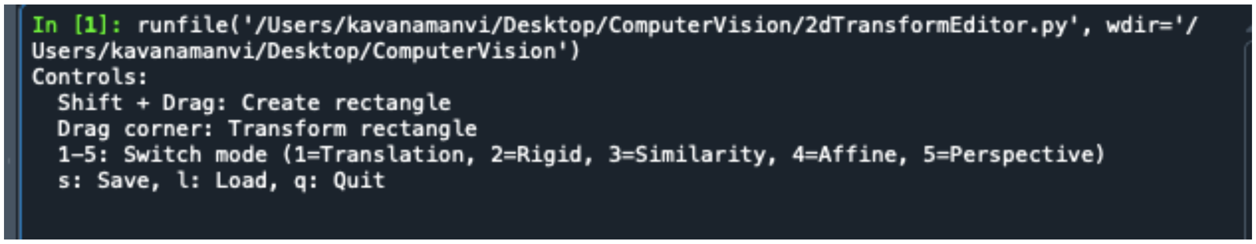


Figure left to right: Shift + drag translation, rigid, similarity, affine and perspective.



```
In [1]: runfile('/Users/kavanamanvi/Desktop/ComputerVision/2dTransformEditor.py', wdir='/
Users/kavanamanvi/Desktop/ComputerVision')
Controls:
  Shift + Drag: Create rectangle
  Drag corner: Transform rectangle
  1-5: Switch mode (1=Translation, 2=Rigid, 3=Similarity, 4=Affine, 5=Perspective)
  s: Save, l: Load, q: Quit
```

Shift+ drag to create a rectangle
Translation: rectangle is moved when you click and drag corner while the shape, size, or orientation remains the same.

Rigid: Original shape and size of rectangle remains the same but you can rotate it and translate it.

Affine: Parallel lines remain the same, but you can do shearing, scaling, rotaion and translation.

Similarity: Translation and rotation are like rigid, but you can do scaling.

Perspective: Applies full projective transformation, allowing for effects like 3d tilt or foreshortening.

Problem 3: Exercises from Forsyth book

i. 2.8.1 #1

What shapes can the shadow of a sphere take if it is cast on a plane, and the source is a point source?

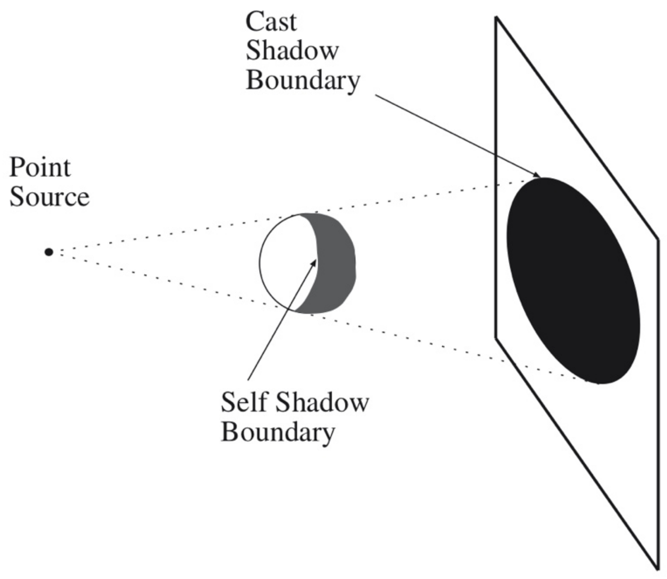


Figure 2.6. Shadows cast by point sources on planes are relatively simple. Self shadow boundaries occur when the surface turns away from the light and cast shadow boundaries occur when a distant surface occludes the view of the source.

As shown in the figure, if the plane is parallel to the base of the sphere and the light source is right in front of the sphere, then it causes a circle-shaped shadow on the plane. This is called the Cast shadow boundary. The source of light is almost like a tangent to the surface of the sphere; the surface turning away from the point light sources causes a self-shadow boundary. If the point source of light is coming from an angle, the shadow cast could be shaped like an ellipse. In extreme scenarios it could be like a hyperbola or a parabola.

ii. 2.8.1 #9

If one looks across a large bay during the daytime, it is often hard to distinguish the mountains on the opposite side. However, near sunset, they become clearly visible. This phenomenon is related to the scattering of light by the atmosphere — a large volume of air is actually a source of scattered light.
Explain what is happening.
We have modeled air as a vacuum and asserted that no energy is lost along a straight line in a vacuum. Use your explanation to give an estimate of the kind of length scales over which that model (treating air as a vacuum) is acceptable.

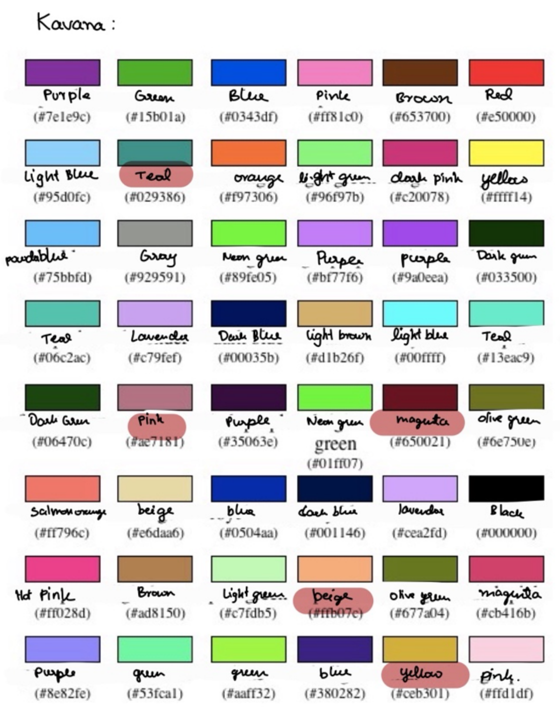
There is a phenomenon called Rayleigh scattering. Basically it means when the sun rays hit the atmospjhere, it scatters due to contact with gas molecules like nitrogen, oxygen and dust present in the air. This causes light to scatter. During the day time in bay area,blue light which has shorter wavelength is scattered more by the particles present in air. This makes it appear the sky blue and same color is reflected by water. So everything looks blue ish. It's hard to see the mountains far off because of low contrast. But later during sun set, the longer end of sun's visible light spectrum which is red/orange is scattered more and hence the sky and water body appears red ish. Because of high contrast between the mountains and the sky we can see it more clearly. If we have snowy mountains, they reflect more light. Changes in temperature and humidity between midday and sunset can also affect visibility. For instance, cooler temperatures improve how far you can see, while higher humidity—especially across a bay—can reduce the maximum line of sight. For someone around 5'6" to 6' tall, the horizon across a flat bay is just over 3 miles away. However, because the mountains rise higher than the horizon, you can see them from much farther away. The idea of air as a vacuum works well for everyday situations, especially when the weather's clear. For example, from a tall building on a clear day, you could easily snap photos of mountains up to 100 miles away. But it's hard to say exactly how far this model holds up because it really depends on the environment. For instance, if there's fog, the model doesn't work for even a mile, but it's fine on a clear day.Overall, the vacuum model works better over short distances. Several factors can affect how well it holds up, like humidity, air pollution, temperature, and how much light is around, as well as whether it's daytime or near sunset. When the air is dry, clear, and cool—especially as the sun is setting—the vacuum model works well over distances of several miles, all the way to the horizon. But when the air is humid, polluted, hot, or filled with light scatter, the model really only applies for shorter distances—like just a few feet or inches—depending on how bad the conditions are.

iii. 3.7 #1 (Then read https://blog.xkcd.com/2010/05/03/color-survey-results/)

Sit down with a friend and a packet of coloured papers, and compare the colour names that you use. You will need a large packet of papers — one can very often get collections of coloured swatches for paint, or for the Pantone colour system, very cheaply.
The best names to try are **basic colour names** — the terms *red, pink, orange, yellow, green, blue, purple, brown, white, gray,* and *black*, which (with a small number of other terms) have remarkable canonical properties that apply widely across different languages [?, ?, ?].
You will find it surprisingly easy to disagree on which colours should be called *blue* and which *green*, for example.

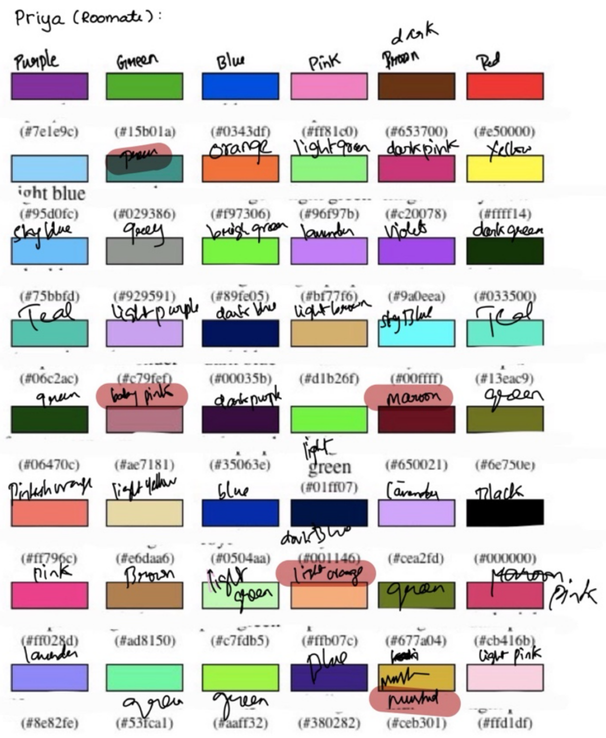
I did the color naming survey with my roommate Priya, who is also a girl. We were surprisingly on the same page for most of the colors. We didn't have much trouble distinguishing between basic color names like red, blue, green, purple, brown, white, black, and gray. Even the more subtle differences between shades like light gray vs silver, or navy vs royal blue, felt pretty intuitive to us. We could both easily agree on which ones were pink vs peach, and which were yellow vs gold.
The only real debate came up when we looked at teal. Priya insisted it leaned more towards green, while I saw it as closer to blue. It was one of those borderline colors that kind of lives in between the two categories, and depending on the lighting or context, you can really see it swing either way. It made me realize how fuzzy the boundaries between certain colors can be.
We also noticed that some shades didn't feel like they belonged strictly to any one color name. For example, some shades of brown looked kind of orange-y, and some blues had a bit of a purple tint to them. It was interesting how the names we chose were influenced not just by the color itself, but by what it reminded us of — for instance, a dusty pink and we ended up calling it "baby pink" even though that's not a basic color term.

It also made me think about how color perception isn't the same for everyone. For example, people with color blindness — especially red-green color blindness, which is the most common — might not see the same differences we do between colors like red and brown or green and yellow. That could totally change how someone labels a color. It's kind of wild to think that what looks clearly green to one person might look more like a dull yellow or even gray to someone else. If I had done this survey with someone who was color blind, I'm sure our answers would've been completely different, especially for those colors that already sit on the edge between two categories.



Kavana :

Purple (#7e1e9c)	Green (#15b01a)	Blue (#0343df)	Pink (#ff81c0)	Brown (#653700)	Red (#e50000)
Light Blue (#95d0fc)	Teal (#029386)	orange (#f97306)	light green (#96f97b)	dark pink (#c20078)	yellow (#ffff14)
powder blue (#75bbfd)	Gray (#929591)	Neon green (#89fe05)	Purple (#bf77f6)	purple (#9a0eea)	Dark green (#033500)
Teal (#06c2ac)	Lavender (#c79fef)	Dark Blue (#00035b)	light brown (#d1b26f)	light blue (#00ffff)	Teal (#13eac9)
Dark Green (#06470c)	Pink (#ae7181)	Purple (#35063e)	Neon green green (#01ff07)	magenta (#650021)	olive green (#6e750e)
salmon orange (#ff796c)	beige (#e6daa6)	blue (#0504aa)	dark blue (#001146)	lavender (#cea2fd)	Black (#000000)
Hot Pink (#ff028d)	Brown (#ad8150)	Light green (#c7fdb5)	beige (#ffb07c)	olive green (#677a04)	magenta (#cb416b)
Purple (#8e82fe)	green (#53fca1)	green (#aaff32)	blue (#380282)	yellow (#ceb301)	pink. (#ffd1df)



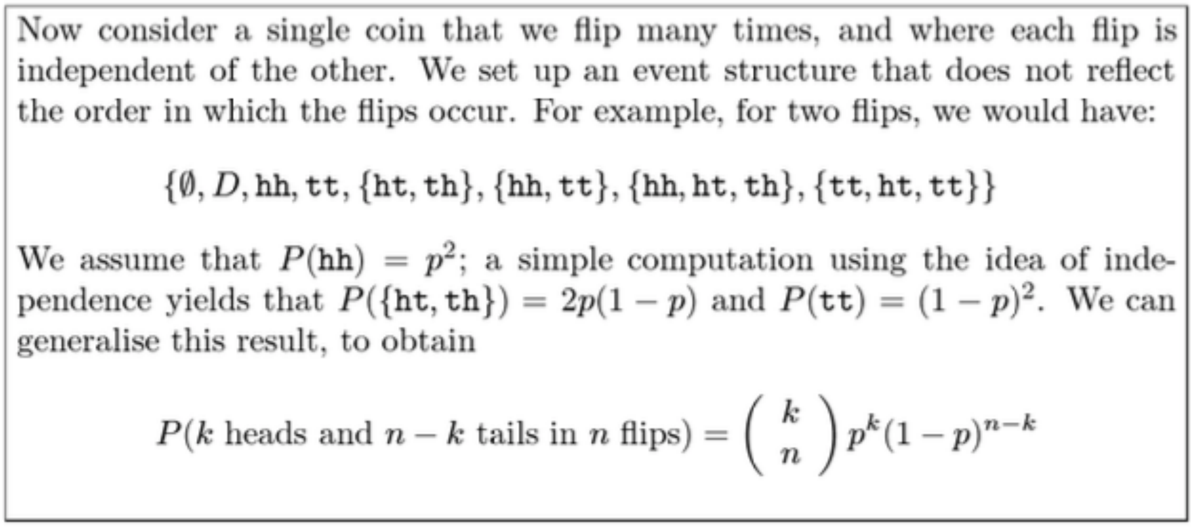
Priya (Roomate):

Purple (#7e1e9c)	Green (#15b01a)	Blue (#0343df)	Pink (#ff81c0)	dark Brown (#653700)	Red (#e50000)
ight blue (#95d0fc)	green (#029386)	orange (#f97306)	light green (#96f97b)	dark pink (#c20078)	Yellow (#ffff14)
sky blue (#75bbfd)	grey (#929591)	bright green (#89fe05)	lavender (#bf77f6)	violet (#9a0eea)	dark green (#033500)
Teal (#06c2ac)	light purple (#c79fef)	dark blue (#00035b)	light brown (#d1b26f)	sky Blue (#00ffff)	Teal (#13eac9)
green (#06470c)	baby pink (#ae7181)	dark purple (#35063e)	light green (#01ff07)	Maroon (#650021)	green (#6e750e)
Pinkish orange (#ff796c)	light yellow (#e6daa6)	blue (#0504aa)	dark Blue (#001146)	Lavender (#cea2fd)	Black (#000000)
pink (#ff028d)	Brown (#ad8150)	light green (#c7fdb5)	light orange (#ffb07c)	green (#677a04)	Maroon Pink (#cb416b)
lavender (#8e82fe)	green (#53fca1)	green (#aaff32)	blue (#380282)	mustard (#ceb301)	light pink (#ffd1df)

iv. Chapter 6 exercises, #5 (this has little to do with CV, but is very useful to understand)

A careless study of Example 10 often results in quite muddled reasoning, of the following form:
"I have bet on heads successfully ten times, therefore I should bet on tails next."
Explain why this muddled reasoning — which has its own name, *the gambler's fallacy* in some circles, *anti-chance* in others — is flawed.

Each coin flip is its own thing, so just because you got 10 heads in a row doesn't mean the next one is more likely to be tails.Each event is independent, is what I would saya. It's kind of a misconception to think the next flip will be tails just because of the streak. But, yeah, getting 10 heads in a row is super unlikely—only about 1 in 1,024. Honestly, if I were the gambler, I'd probably bet on tails too.


Now consider a single coin that we flip many times, and where each flip is independent of the other. We set up an event structure that does not reflect the order in which the flips occur. For example, for two flips, we would have:

$$\{\emptyset, D, \mathtt{hh}, \mathtt{tt}, \{\mathtt{ht}, \mathtt{th}\}, \{\mathtt{hh}, \mathtt{tt}\}, \{\mathtt{hh}, \mathtt{ht}, \mathtt{th}\}, \{\mathtt{tt}, \mathtt{ht}, \mathtt{tt}\}\}$$

We assume that $P(\mathtt{hh}) = p^2$; a simple computation using the idea of independence yields that $P(\{\mathtt{ht}, \mathtt{th}\}) = 2p(1-p)$ and $P(\mathtt{tt}) = (1-p)^2$. We can generalise this result, to obtain

$$P(k \text{ heads and } n-k \text{ tails in } n \text{ flips}) = \left(\begin{array}{c} k \\ n \end{array} \right) p^k (1-p)^{n-k}$$

Example 6.10: *The probability of various frequencies in repeated coin flips*

Problem 4: Computer Vision Concepts
Briefly explain the following concepts

i. Perspective projection

Imagine turning a 3D world into a flat image—like snapping a photo. That's perspective projection. It is a method used to map 3D objects onto a 2D image plane, mimicking how the human eye perceives depth. Objects that are farther from the viewer appear smaller, and parallel lines (like train tracks) appear to converge as they extend into the distance. This projection creates a realistic sense of depth and scale in images and is fundamental to computer vision and graphics.

ii. Vanishing point

The vanishing point is where all those parallel lines (like road edges or railroad tracks) seem to meet when they stretch off into the distance. The vanishing point is the location in a perspective projection where parallel lines appear to converge. It represents the point at which an object recedes infinitely into the distance. Depending on the orientation of the objects and viewpoint, a scene can have one or more vanishing points. This concept is key in understanding linear perspective and is often used in 3D reconstruction and image interpretation.

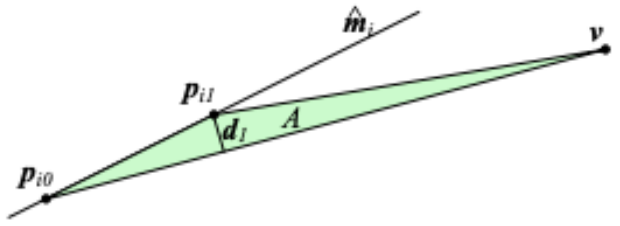


Figure 4.46 Triple product of the line segments endpoints p_{i0} and p_{i1} and the vanishing point v. The area A is proportional to the perpendicular distance d_1 and the distance between the other endpoint p_{i0} and the vanishing point.

iii. Stereopsis

Stereopsis is the process of perceiving depth by combining two slightly different images from each eye. The human visual system uses this binocular disparity to judge distances between objects. In computer vision, this concept is applied in stereo imaging systems, where two cameras simulate human eyes to extract depth information from a scene. Since each eye sees the world from a slightly different angle, our brains compare those two images and say, "Okay, this thing is close, and that thing's far." It's how we get that 3D vision without even thinking about it—like built-in AR.

iv. Optical flow

Optical flow describes the apparent motion of objects within a visual scene caused by the relative motion between the observer (camera) and the scene. It is typically calculated by analyzing changes in pixel intensities across frames in a video sequence. Optical flow is used in motion tracking, video stabilization, object detection, and autonomous navigation.

v. Parallax

Parallax is the apparent shift in the position of an object when viewed from different perspectives. It's more noticeable for objects that are closer to the viewer and less for those farther away. In computer vision, parallax is used to estimate depth and 3D structure from multiple viewpoints, especially in stereo vision systems and structure-from-motion techniques.

Put all your work into a single file: output images; text responses; and program code. Submit using the dropbox in D2L.