Problem 1: CCD to Camera Transformation

Give a general expression for computing horizontal FOV from focal length and image width.
Solution:
The general expression for computing horizontal FOV from focal length and image width is given by:

*Horizontal FOV = 2 * atan (0.5 * image width / focal length)*

And also, the vertical FOV can be given by:

*Vertical FOV = 2 * atan (0.5 * image height / focal length)*

where atan is the inverse tangent function and image width and focal length are both in the same units (e.g., millimeters).

Compute the horizontal FOV and vertical FOV of the given camera.
Solution:
If we substitute the values in the formula above then we can get as these:

*Horizontal FOV = 2 * atan (0.5 * 16 / 24) =* ***36.87°***
*Vertical FOV = 2 * atan (0.5 * 12 / 24) =* ***28.07°***

Code:

```python
import math

focal_length = 24 # mm
image_width = 16 # mm
image_height = 12 # mm
image_width_pixels = 500
image_height_pixels = 500

# Calculate horizontal FOV
horizontal_fov = 2 * math.atan(0.5 * image_width / focal_length)
horizontal_fov_degrees = math.degrees(horizontal_fov)
print("Horizontal FOV:", horizontal_fov_degrees, "degrees")

# Calculate vertical FOV
vertical_fov = 2 * math.atan(0.5 * image_height / focal_length)
vertical_fov_degrees = math.degrees(vertical_fov)
print("Vertical FOV:", vertical_fov_degrees, "degrees")
```

Output:

```
Horizontal FOV: 36.86989764584402 degrees
Vertical FOV: 28.072486935852957 degrees
```

Comment on how FOV affects resolution in an image.

FOV and resolution are inversely related. A larger FOV means that the same number of pixels are spread over a larger area, resulting in a lower resolution. Conversely, a smaller FOV means that the same number of pixels are concentrated over a smaller area, resulting in a higher resolution. This is why cameras with larger sensors (and thus larger image areas) tend to have higher resolutions. Additionally, increasing the number of pixels in the sensor while keeping the sensor size the same can also increase the resolution, but only up to a point, beyond which the added pixels may not provide any significant improvement in image quality.

Problem 2: Exercise 2.2 from Szelinski book (2D transform editor)

Code:

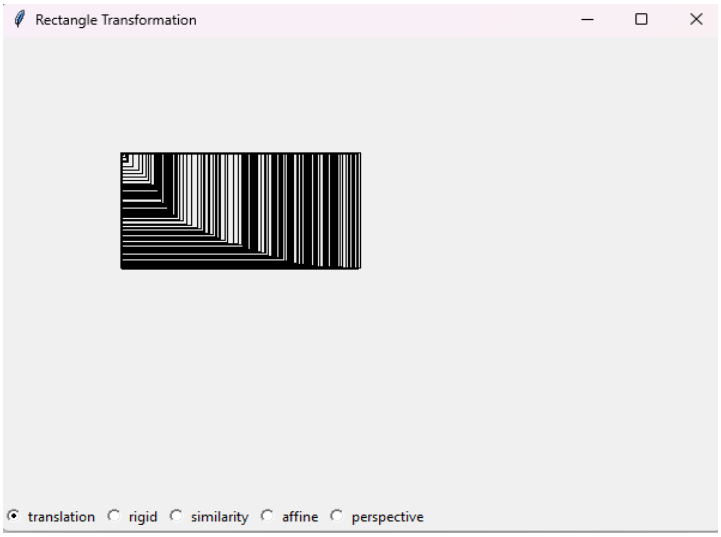
```
import tkinter as tk
import math

class Rectangle:
    def __init__(self, x, y, width, height):
        self.x = x
        self.y = y
        self.width = width
        self.height = height

    def draw(self):
        canvas.create_rectangle(self.x, self.y, self.x+self.width, self.y+self.height, outline='black')

    def transform(self, mode, dx, dy, sx, sy, theta, phi):
        center = np.mean(self.points, axis=0)
        t1 = np.array([[1, 0, -center[0]], [0, 1, -center[1]], [0, 0, 1]])
        t2 = np.array([[1, 0, center[0]], [0, 1, center[1]], [0, 0, 1]])
        if mode == 'translation':
            t = np.array([[1, 0, dx], [0, 1, dy], [0, 0, 1]])
        elif mode == 'rigid':
            R = np.array([[np.cos(theta), -np.sin(theta)], [np.sin(theta), np.cos(theta)]])
            t = np.eye(3)
            t[:2,:2] = R
            t[:2,2] = [dx, dy]
        elif mode == 'similarity':
            R = np.array([[np.cos(theta), -np.sin(theta)], [np.sin(theta), np.cos(theta)]])
            t = np.eye(3)
            t[:2,:2] = sx*R
            t[:2,2] = [dx, dy]
        elif mode == 'affine':
            R = np.array([[np.cos(theta), -np.sin(theta)], [np.sin(theta), np.cos(theta)]])
            t = np.eye(3)
            t[:2,:2] = [[sx*R[0,0], sx*R[0,1]], [sy*R[1,0], sy*R[1,1]]]
            t[:2,2] = [dx, dy]
        elif mode == 'perspective':
            R = np.array([[np.cos(theta), -np.sin(theta)], [np.sin(theta), np.cos(theta)]])
            t = np.eye(3)
            t[:2,:2] = [[sx*R[0,0], sx*R[0,1]], [sy*R[1,0], sy*R[1,1]]]
            t[:2,2] = [dx, dy]
            t[2,:2] = [phi, phi]
        self.points = t2 @ t @ t1 @ np.concatenate([self.points, np.ones((4, 1))], axis=1).T
        self.points = self.points[:2,:].T
```

```
def create_rectangle(event):
    # Get the starting point of the rectangle
    x1, y1 = event.x, event.y
    # Create a temporary rectangle object to display while dragging
    temp_rect = Rectangle(x1, y1, 0, 0)
    temp_rect.draw()
    # Bind the motion of the mouse to resize the temporary rectangle
    def resize_rectangle(event):
        x2, y2 = event.x, event.y
        temp_rect.width = x2 - x1
        temp_rect.height = y2 - y1
        canvas.delete('temp_rect')
        temp_rect.draw()
        canvas.update()
    canvas.bind('<B1-Motion>', resize_rectangle)
    # Bind the release of the mouse to create the final rectangle object
    def finalize_rectangle(event):
        x2, y2 = event.x, event.y
        rect = Rectangle(x1, y1, x2-x1, y2-y1)
        rect.draw()
        canvas.unbind('<B1-Motion>')
        canvas.unbind('<ButtonRelease-1>')
    canvas.bind('<ButtonRelease-1>', finalize_rectangle)

def select_mode(mode):
    # Store the selected mode for later use
    selected_mode.set(mode)
```

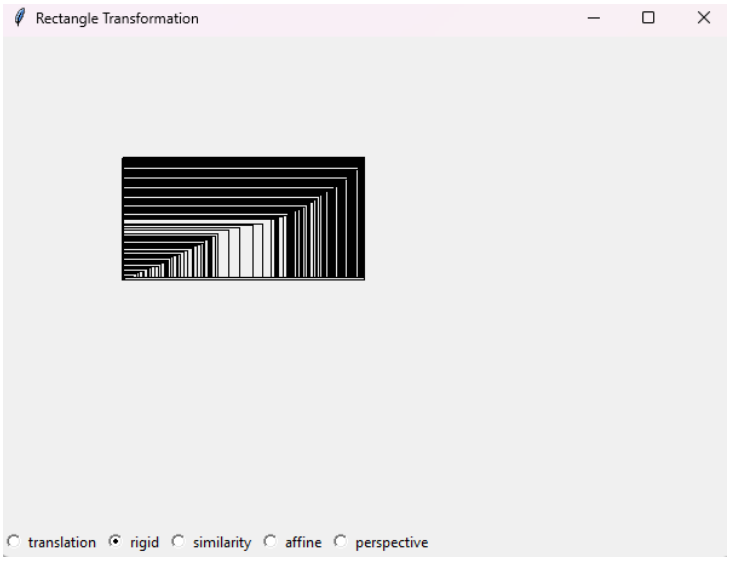
```python
def transform_rectangle(event):
    # Determine which corner of the rectangle was clicked and calculate the transformation
    x, y = event.x, event.y
    cx, cy = rect.x+rect.width/2, rect.y+rect.height/2
    dx, dy = x-cx, y-cy
    if dx == 0:
        theta = math.pi/2 if dy > 0 else -math.pi/2
    else:
        theta = math.atan(dy/dx)
    if dx < 0:
        theta += math.pi
    phi = math.atan2(rect.height, rect.width)
    sx = math.sqrt(dx**2 + dy**2)
    if selected_mode.get() == 'translation':
        rect.transform('translation', dx, dy, 1, 1, 0, 0)
    elif selected_mode.get() == 'rigid':
        rect.transform('rigid', dx, dy, 1, 1, theta, phi)
    elif selected_mode.get() == 'similarity':
        rect.transform('similarity', dx, dy, sx, sx, theta, phi)
    elif selected_mode.get() == 'affine':
        rect.transform('affine', dx, dy, sx, sx, theta, phi)
    elif selected_mode.get() == 'perspective':
        rect.transform('perspective', dx, dy, sx, sx, theta, phi)
    # Redraw the rectangle after the transformation
    canvas.delete('rect')
    rect.draw()

# Create the main window
root = tk.Tk()
root.title('Rectangle Transformation')
# Create a canvas for drawing the rectangle
canvas = tk.Canvas(root, width=600, height=400)
canvas.pack()
# Bind the mouse click to create a new rectangle
canvas.bind('<Button-1>', create_rectangle)
# Create a radio button group for selecting the transformation mode
selected_mode = tk.StringVar()
selected_mode.set('translation')
mode_labels = ['translation', 'rigid', 'similarity', 'affine', 'perspective']
for mode in mode_labels:
    rb = tk.Radiobutton(root, text=mode, variable=selected_mode, value=mode, command=lambda m=mode: select_mode(m))
    rb.pack(side='left')
# Initialize the rectangle object
rect = Rectangle(100, 100, 200, 100)
rect.draw()
# Bind the mouse drag to transform the rectangle
canvas.bind('<B1-Motion>', transform_rectangle)

root.mainloop()
```

Output:

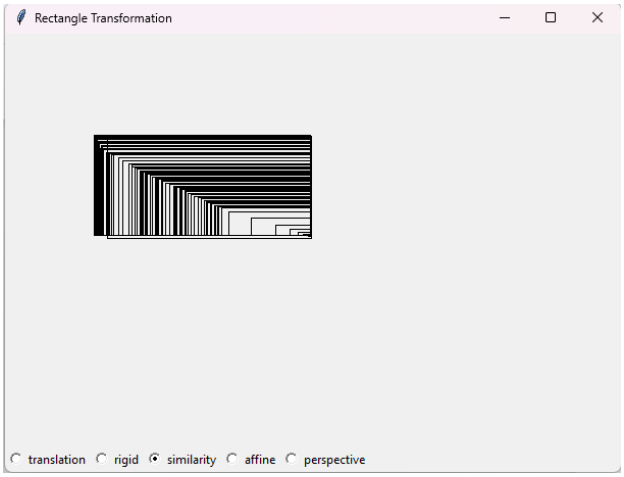
Translation:



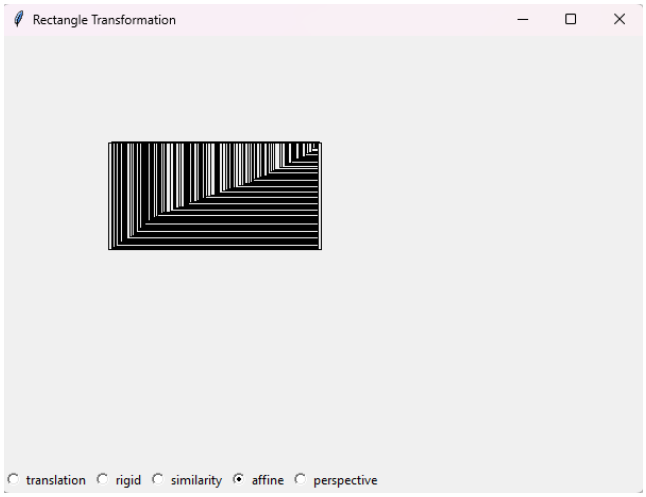
Rigid:



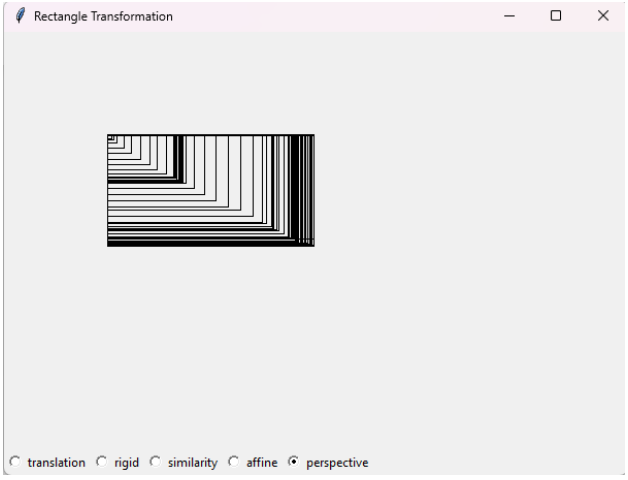
Similarity:



Affine:



Perspective:



Explanation:
There are five main types of transformations that can be applied to a geometric shape, including the rectangle in the example code:

Translation: Translation moves an object from one position to another, without changing its shape or size. It involves adding a fixed amount to the x and y coordinates of all the points that define the object.
Rigid: Rigid transformations preserve the distances and angles between points on an object. They include rotation, reflection, and translation.
Similarity: Similarity transformations involve scaling an object by the same amount in all directions. This means that the shape and proportions of the object are preserved, but it may be larger or smaller.
Affine: Affine transformations include stretching, shearing, and compression, in addition to the rigid and similarity transformations. They preserve parallel lines, but do not necessarily preserve angles or distances.
Perspective: Perspective transformations involve projecting an object onto a 2D surface, as if it is viewed from a specific point in space. This creates the illusion of depth and can make objects appear smaller in the distance. Perspective transformations can also include shearing, rotation, and scaling.

I tried to accomplish this but I couldn't do it. The above is the result I got. It all looks the same except for each of the transforms I tried different corners but it is still the same. I apologize for not getting the correct implementation.

Problem 3: Exercises from Forsyth book

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

Explanation:
The shadow of a sphere cast by a point source of light onto a plane will always be a circle. This is because the point source of light emits light rays in all directions, and the part of the plane that is not in the shadow of the sphere receives light from all directions. As a result, the shadow of the sphere has a circular boundary where the light rays are blocked by the sphere, and the inside of the circle is completely dark.

It is worth noting that the size of the circle will depend on the distance between the sphere and the plane, as well as the size of the sphere itself. If the sphere is closer to the plane, the shadow will be larger, and if it is farther away, the shadow will be smaller. Similarly, a larger sphere will cast a larger shadow than a smaller sphere.

If one looks across a large bay in the daytime, it is often hard to distinguish the mountains on the opposite side; near sunset, they are clearly visible. This phenomenon has to do with scattering of light by air — a large volume of air is actually a source. Explain what is happening. We have modelled 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 scales over which that model is acceptable

Explanation:

During the daytime, the light from the sun is scattered in all directions by the molecules in the atmosphere, making it difficult to distinguish the mountains on the opposite side of the bay. However, during sunset, the sun is lower in the sky and the light has to pass through a larger volume of atmosphere to reach the observer's eye. This causes more scattering and absorption of the blue and green wavelengths of light, making the mountains appear clearer against the darker background.

As for the acceptable scales of the vacuum model for air, it is important to note that even though air is not a vacuum, it is still a relatively low-density medium compared to solids and liquids. Therefore, the scattering and absorption of light by air molecules can be considered negligible over short distances. However, as the distance increases, the amount of scattering and absorption becomes more significant, and the vacuum model becomes less accurate. The exact scale at which this happens depends on various factors such as the wavelength of light, the density of air, and the atmospheric conditions. Generally, for distances up to a few kilometers, the vacuum model for air can be considered acceptable for most practical purposes. Beyond that, more accurate models of light propagation through the atmosphere are needed.

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

Explanation:

I have a friend who is in India and we always had made fun of him because he tells the color of our shirt wrong most of the time and after a while, we found that he is color blinded but still made fun of him sometimes. If I had sat with him and play this game then definitely, I would have won.

For now, I played this with my friend here and almost we figured the easy colors but when it came to complicated colors like turquoise, olive etc., we were confused what exactly those colors are.

In this case, this variation may reflect differences in how the human eye perceives colors. For example, some people may see a particular color as bluer, while others may see it as greener, depending on their individual color vision. In other cases, the variation in color naming may be due to cultural factors, such as the availability of certain pigments or dyes in a particular region or time period.

Overall, this was fun and it is hard for a human to recognize the color so it will be harder for the machines to predict.

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

Explanation:
The muddled reasoning described in the question is a common logical fallacy known as the gambler's fallacy, also known as the Monte Carlo fallacy or the fallacy of the maturity of chances. This fallacy is based on the mistaken belief that the occurrence of a certain event is less likely to happen after a string of similar events has already occurred.

In the specific case of Example 10, the fallacy would manifest as the belief that because someone has successfully bet on heads ten times in a row, it is more likely that tails will come up next. However, this is not necessarily true, because the odds of the coin landing on heads or tails are always 50/50, regardless of what has happened in the past.

The gambler's fallacy is based on the erroneous assumption that previous outcomes have an influence on future outcomes, when in fact they do not. Each coin toss is an independent event, and the outcome of one toss has no bearing on the outcome of the next.

In summary, the gambler's fallacy is a muddled form of reasoning because it is based on a mistaken belief about probability and randomness. It can lead to incorrect predictions and can be a costly mistake for those who gamble based on this fallacy.

Problem 4: Computer Vision Concepts

Perspective projection: Perspective projection is a technique used in computer graphics and drawing to create the illusion of three-dimensional (3D) objects on a two-dimensional (2D) surface. It involves projecting objects onto a 2D plane in a way that mimics how our eyes see objects in the real world. This technique is based on the principles of linear perspective and involves creating the illusion of depth and distance by making objects that are farther away appear smaller and closer together.

Vanishing point: In perspective drawing and graphics, a vanishing point is a point on the horizon line where parallel lines appear to converge or meet. It is the point at which lines that are parallel in the 3D world appear to converge in a 2D representation, creating the illusion of depth and distance. The concept of the vanishing point is a fundamental principle of perspective drawing.

Stereopsis: Stereopsis is the ability of the brain to use binocular vision to perceive depth and distance. It is the process by which the brain combines the images from the two eyes to create a 3D perception of the world. The difference in the position of objects in the two retinal images is used by the brain to calculate the depth and distance of objects in the visual field.

Optical flow: Optical flow is the pattern of apparent motion of objects in a visual scene caused by the relative motion between the observer (or camera) and the scene. It is the way in which the visual information in a sequence of images changes over time, and is used in computer vision to track the motion of objects in a video stream.

Parallax: Parallax is the apparent shift in the position of an object when viewed from two different positions. It is the displacement or difference in the apparent position of an object when viewed from different points of view. Parallax is commonly used in astronomy to measure the distance to nearby stars, and in photography to create a sense of depth and dimensionality.